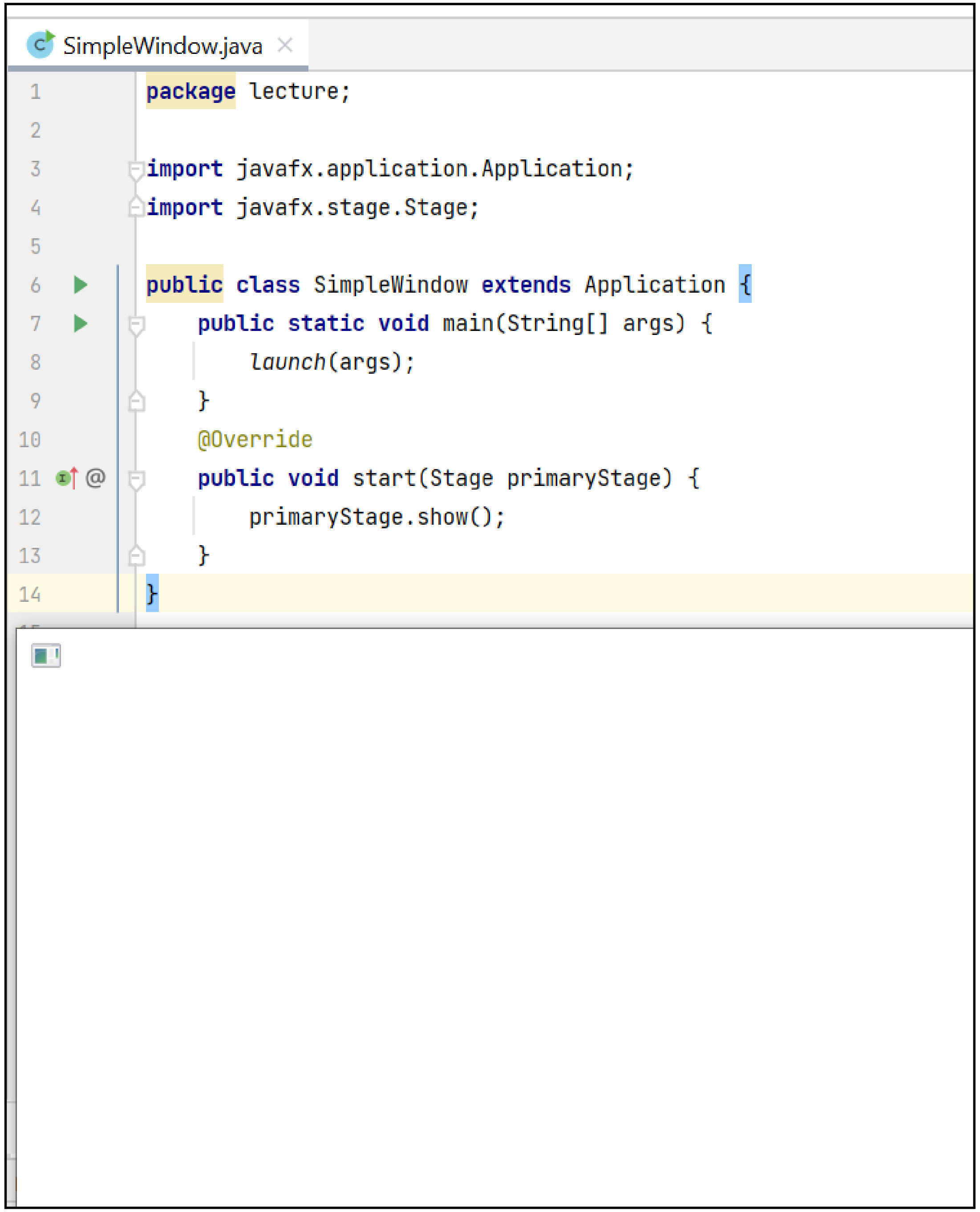
CSC1120 – Data Structures and Graphical Interfaces

Week 2: JavaFX Intro

Graphical Interfaces

- Up to this point we have been interacting with the user via the console and text-based operations
- But most computers make use of some type of graphical interfaces
- For this course, to create graphical interfaces, we are going to be using JavaFX

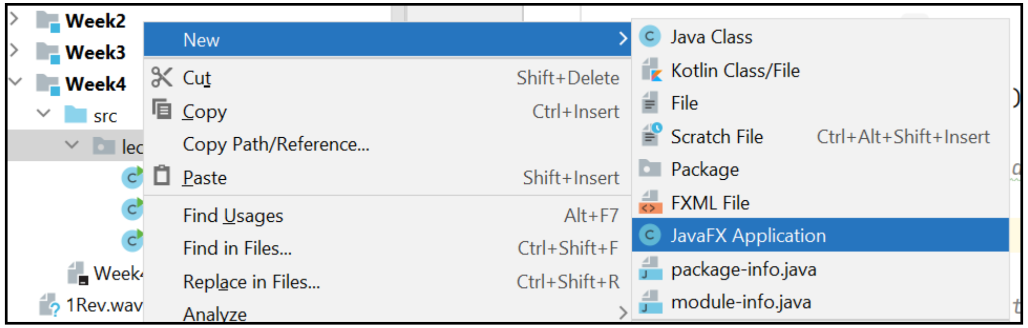


```java
package lecture;

import javafx.application.Application;
import javafx.stage.Stage;

public class SimpleWindow extends Application {
    public static void main(String[] args) {
        launch(args);
    }
    @Override
    public void start(Stage primaryStage) {
        primaryStage.show();
    }
}
```

JavaFX

- JavaFX is a Java library for creating GUI programs
- There are other GUI libraries like Swing (AWT), but it is an older and mainly kept around for backward compatibility
  - **Note that many of the JavaFX and Swing libraries have similar names. So be careful in IntelliJ when importing libraries that you are importing the right one.**
- JavaFX is not included with the base Java by default, so you must install it manually
  - https://csse.msoe.us/csc1120/javafx/#installation-instructions

SimpleWindow.java



- Let’s start by creating a simple GUI program
- If you have installed and configured JavaFX, when you go to create a new class, IntelliJ will offer the option to create a JavaFX class
- A JavaFX Class is slightly different from the normal classes we have been building

Application Class

- JavaFX programs extend the Application class
- The Application class is an abstract class in JavaFX
    - The Application class has a single abstract method called start() which must be implemented
- We still have a main(), but all it does is call launch(args);
    - launch(args) will call start and pass in a default Stage object
- Note that we could omit the main() method and IDEs like IntelliJ will insert it during compilation for us

```
import javafx.application.Application;
import javafx.stage.Stage;

public class SimpleWindow extends Application {

  public static void main(String[] args) {
    launch(args);
  }

  @Override
  public void start(Stage stage) {

  }
}
```

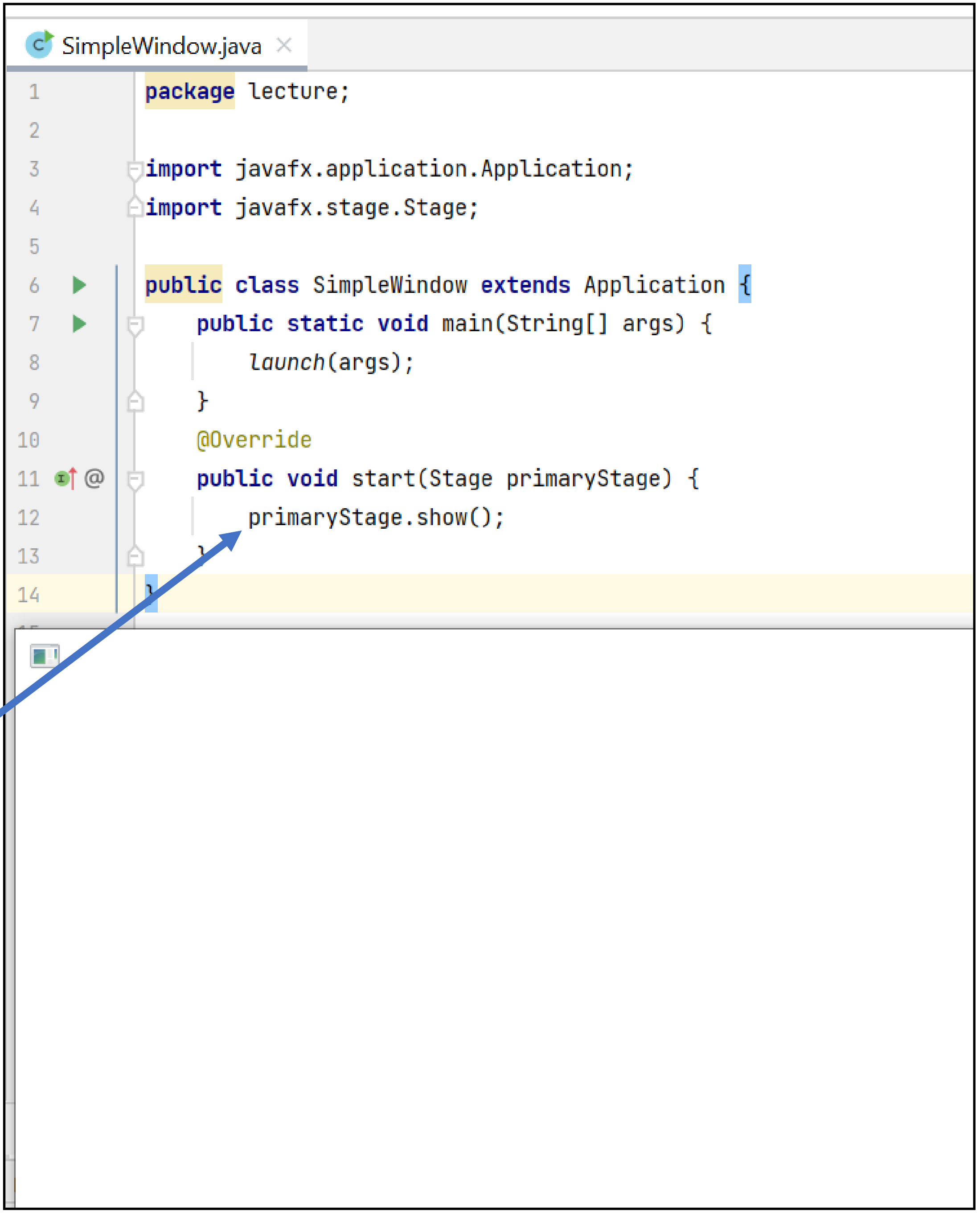
Application Class

- The method start(Stage stage), has a stage object passed into it by launch()
    - You do not have to do anything to create this stage object, it is passed in by default
- We also have imports for Application and Stage
- As we add more components to our program, we will need to import them

```
import javafx.application.Application;
import javafx.stage.Stage;

public class SimpleWindow extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage stage) {

    }
}
```

Stage

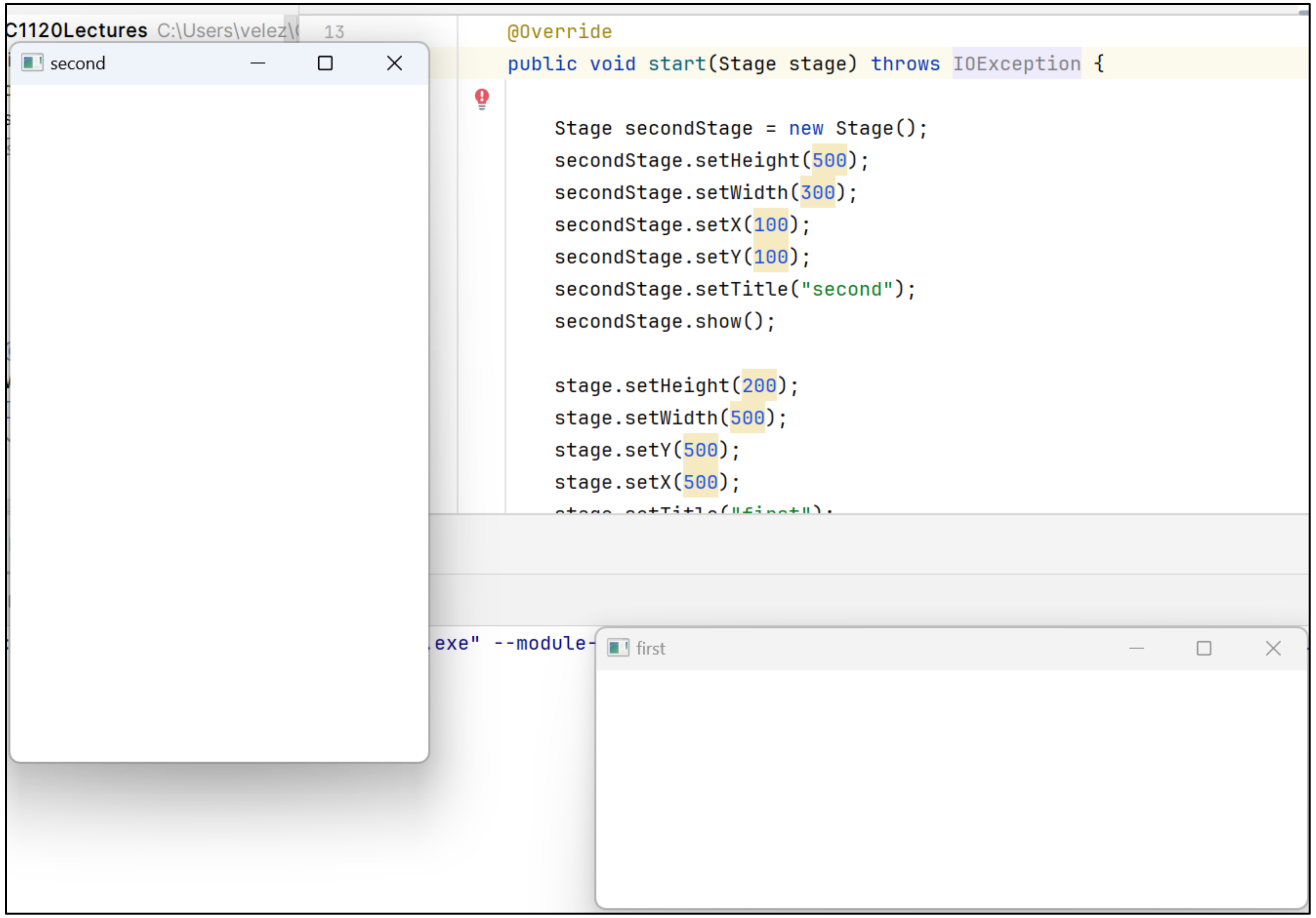
- The Stage object passed into start represents the window that will hold our GUI program
  - You can create and launch additional stage objects, which will create additional windows
- From this point, we can run our program, but nothing shows up
- If we want our window to show something, we must call show() on the stage object
- An empty window will pop up when we call show() because we haven't added anything to the stage



```java
package lecture;

import javafx.application.Application;
import javafx.stage.Stage;

public class SimpleWindow extends Application {
    public static void main(String[] args) {
        launch(args);
    }
    @Override
    public void start(Stage primaryStage) {
        primaryStage.show();
    }
}
```

Stage attributes

- When creating GUI components there is a lot of room for customization
- This includes changing aspects of the stage
- Some common attributes we can change are the width, height, title, and x and y coordinates



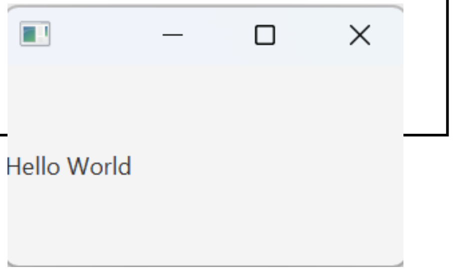
```
@Override
public void start(Stage stage) throws IOException {

    Stage secondStage = new Stage();
    secondStage.setHeight(500);
    secondStage.setWidth(300);
    secondStage.setX(100);
    secondStage.setY(100);
    secondStage.setTitle("second");
    secondStage.show();

    stage.setHeight(200);
    stage.setWidth(500);
    stage.setY(500);
    stage.setX(500);
    stage.setTitle("first");
```

```
.exe" --module-
```

Label

- The first thing we can add to our Window is a Label
  - A Label is a component that displays text
- To create a Label, we first must import it
- Be careful with importing JavaFX elements
  - IntelliJ likes to default to importing Swing (AWT) versions of these components
- **YOU SHOULD NEVER HAVE AN IMPORT FROM java.awt IN YOUR JAVAFX PROGRAMS**
  - It causes strange bugs that are hard to debug
- After we create a Label, to display it, we must first add it to a Scene

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.stage.Stage;
//You should never have any imports with awt
//import java.awt.*;

import java.io.IOException;

public class BasicWindow extends Application {
   public static void main(String[] args) {
      launch(args);
   }
   @Override
   public void start(Stage stage) throws IOException {
      Label l1 = new Label("Hello World");
      //200 and 100 are the width and height
      Scene scene = new Scene(l1,200,100);
      stage.setScene(scene);
      stage.show();
   }
}
```

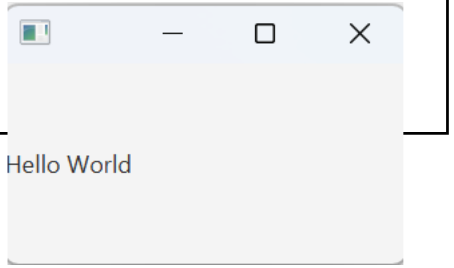


Hello World

Scene

- A Scene is a collection of elements that is displayed on a Stage
- Instead of placing elements one at a time into a Stage, we can place them into Scenes that can be swap them into and out of the Stage
- We can relate a Scene to the different scenes in a play
  - For different acts, the furniture or props for that scene change
- Or we can think of a Scene in terms of video games
  - A game can have different scenes such as the home scene, game over scene, scenes for various levels, etc

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.stage.Stage;
//You should never have any imports with awt
//import java.awt.*;

import java.io.IOException;

public class BasicWindow extends Application {
   public static void main(String[] args) {
      launch(args);
   }
   @Override
   public void start(Stage stage) throws IOException {
      Label l1 = new Label("Hello World");
      //200 and 100 are the width and height
      Scene scene = new Scene(l1,200,100);
      stage.setScene(scene);
      stage.show();
   }
}
```

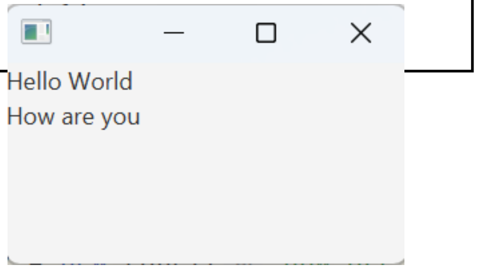


Hello World

Layout (Pane)

- If we wanted to add more than a single element to our scene, we could first add a layout (Pane) to the Scene and then add elements to the layout
- A layout is a container that can hold numerous components and even other layouts
- It also specifies how the components are arranged relative to one another

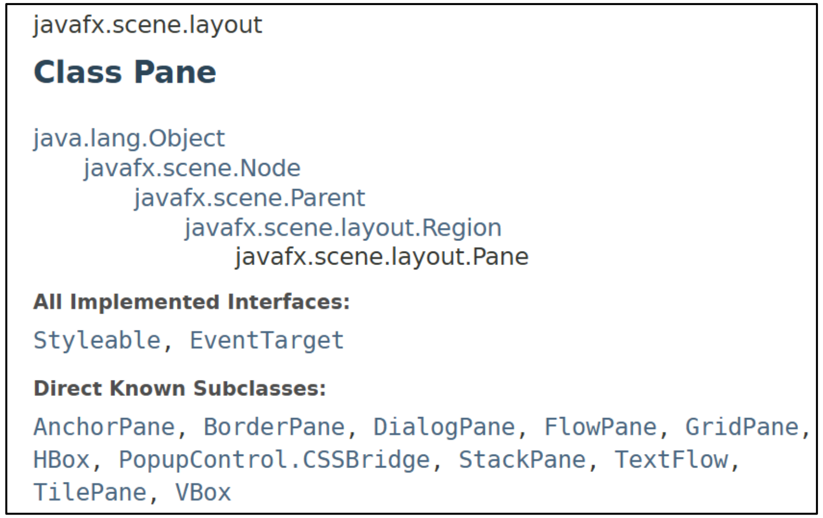
```
  @Override
  public void start(Stage stage) throws IOException {
    Pane root = new VBox();
    Label l1 = new Label("Hello World");
    Label l2 = new Label("How are you");
    root.getChildren().addAll(l1,l2);

    //200 and 100 are the width and height
    Scene scene = new Scene(root,200,100);
    stage.setScene(scene);
    stage.show();
  }
}
```



```
Hello World
How are you
```

Pane

- Usually, we don’t make a base Pane; we make one of its subclasses
- The sub-classes of Pane include Vbox, HBox, FlowPane, BorderPane, and GridPane
  - These subclasses differ in how they arrange components
- For example, a VBox arranges components vertically, one on top of another



javafx.scene.layout

Class Pane

java.lang.Object
    javafx.scene.Node
        javafx.scene.Parent
            javafx.scene.layout.Region
                javafx.scene.layout.Pane

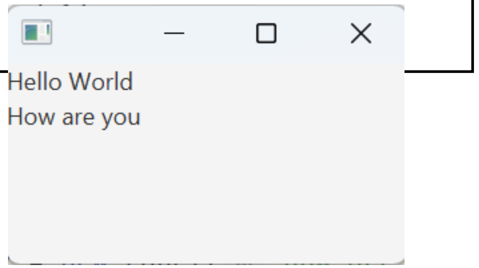
All Implemented Interfaces:

`Styleable, EventTarget`

Direct Known Subclasses:

`AnchorPane, BorderPane, DialogPane, FlowPane, GridPane, HBox, PopupControl.CSSBridge, StackPane, TextFlow, TilePane, VBox`

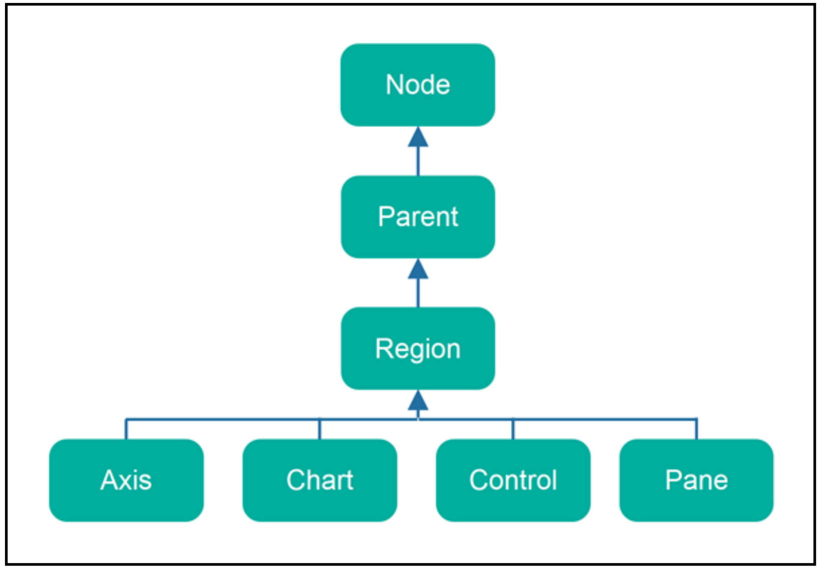
```
  @Override
  public void start(Stage stage) throws IOException {
    Pane root = new VBox();
    Label l1 = new Label("Hello World");
    Label l2 = new Label("How are you");
    root.getChildren().addAll(l1,l2);

    //200 and 100 are the width and height
    Scene scene = new Scene(root,200,100);
    stage.setScene(scene);
    stage.show();
  }
}
```

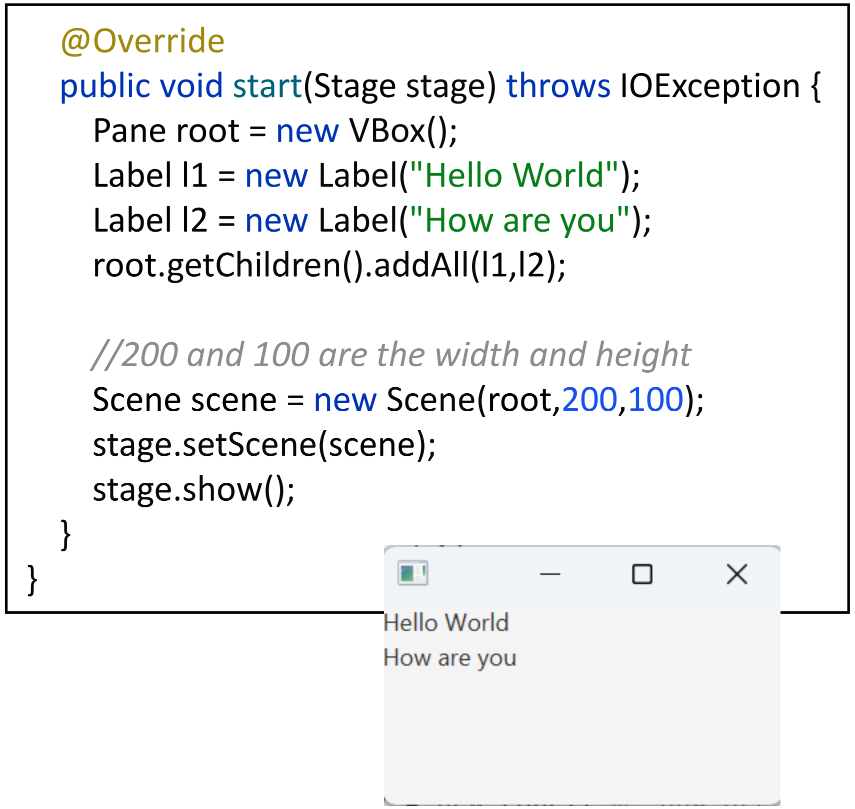


Hello World
How are you

getChildren() and Node

- To add an element to a Pane, we first call getChildren(), which returns a list of Node objects within the Pane
- Like Exceptions, JavaFX has an extensive Class hierarchy
- In this hierarchy, the Node Class is one of the most important classes in JavaFX that almost all other classes extend from it
- Anything that is a Node in JavaFX, which is most things, can be added to the list of Nodes in Pane



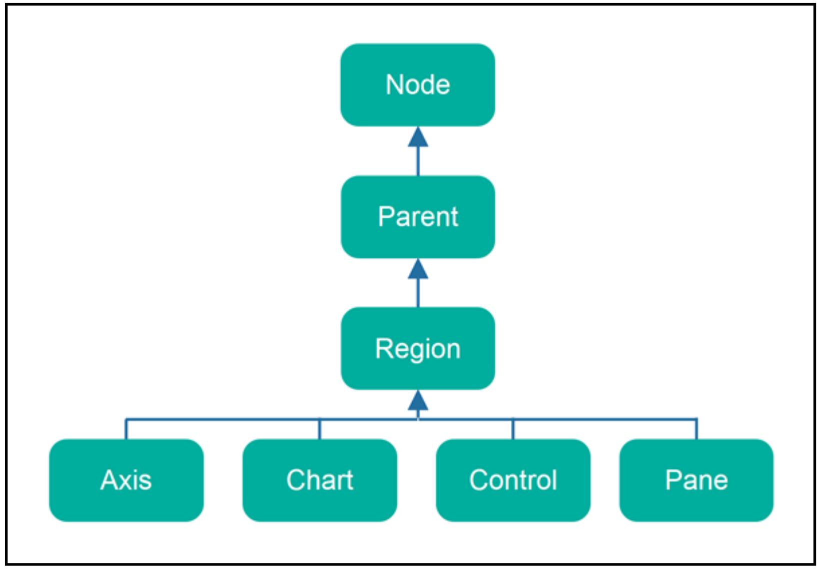


```
    @Override
    public void start(Stage stage) throws IOException {
        Pane root = new VBox();
        Label l1 = new Label("Hello World");
        Label l2 = new Label("How are you");
        root.getChildren().addAll(l1,l2);

        //200 and 100 are the width and height
        Scene scene = new Scene(root,200,100);
        stage.setScene(scene);
        stage.show();
    }
}
```

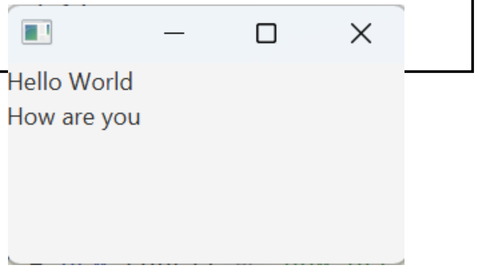
```
Hello World
How are you
```

add() and addAll()

- To add to getChildren(), we can call either add() to add a single component, or addAll() to add multiple components,
- Note that the order in which you add components matters



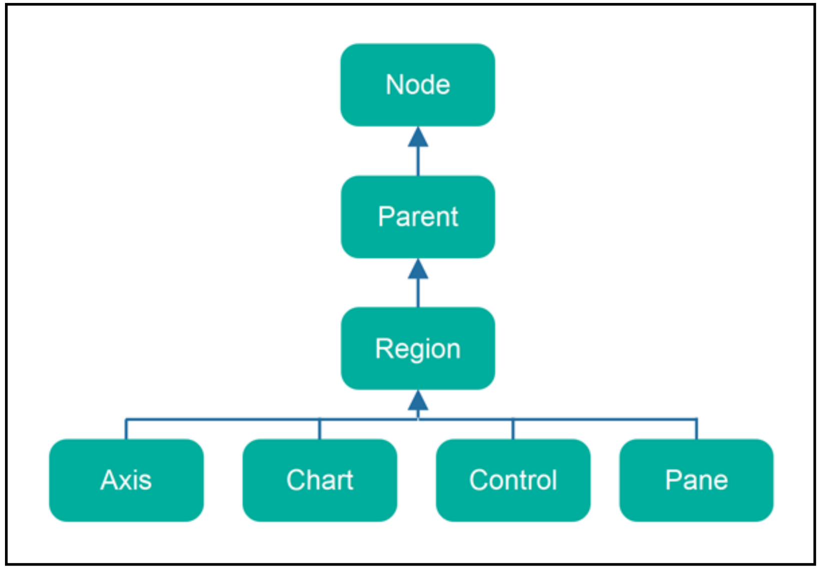
```
    @Override
    public void start(Stage stage) throws IOException {
        Pane root = new VBox();
        Label l1 = new Label("Hello World");
        Label l2 = new Label("How are you");
        root.getChildren().addAll(l1,l2);

        //200 and 100 are the width and height
        Scene scene = new Scene(root,200,100);
        stage.setScene(scene);
        stage.show();
    }
}
```

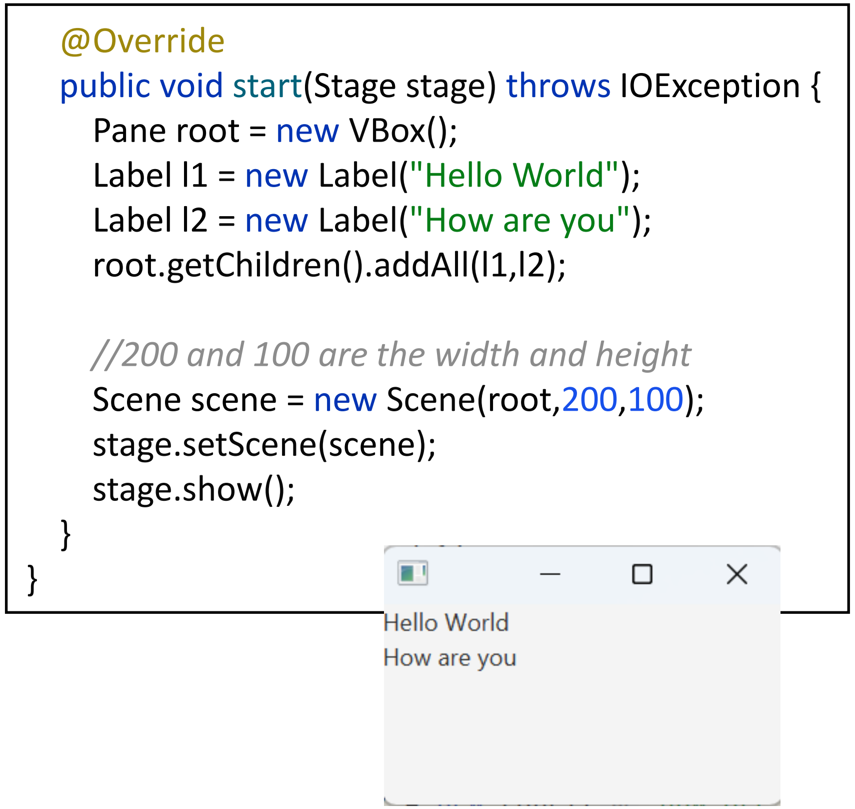


Hello World
How are you

Control Class

- In the example shown, we are adding a Label, which is a subclass of Control, to the Pane
- The Control Class represents components that you can put into your GUI that a user can interact with
- Other sub-classes of the Control Class are Button, TextField, Menu, ScrollPane, RadioButton, CheckBox, and many more
- Because Control and Pane Class are sub-classes of Node, they can also be added to a Pane





```
    @Override
    public void start(Stage stage) throws IOException {
        Pane root = new VBox();
        Label l1 = new Label("Hello World");
        Label l2 = new Label("How are you");
        root.getChildren().addAll(l1,l2);

        //200 and 100 are the width and height
        Scene scene = new Scene(root,200,100);
        stage.setScene(scene);
        stage.show();
    }
}
```

```
Hello World
How are you
```

Nesting Layouts

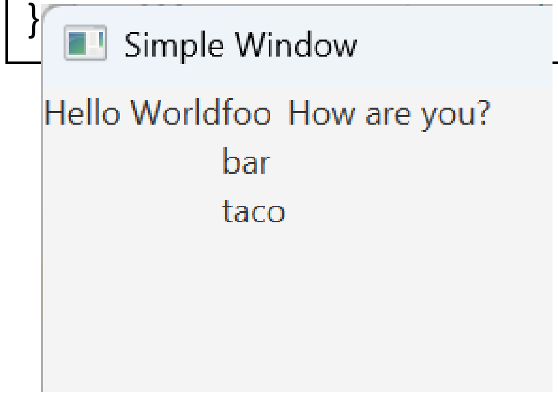
- As mentioned already, the Pane Class represents different types of layouts
- It’s also a Node and can be added to another Pane
- What happens if you add one Pane into another (nesting)?

```
@Override
public void start(Stage primaryStage) {
   primaryStage.setTitle("Simple Window");
   primaryStage.setX(0);
   primaryStage.setY(0);

   Pane root;
   root = new HBox();
   VBox innerPane = new VBox();

   Label l1 = new Label("Hello World");
   Label l2 = new Label("How are you?");
   innerPane.getChildren().addAll(new Label("foo"),
         new Label("bar"), new Label("taco"));

   root.getChildren().addAll(l1, innerPane, l2);
   Scene scene = new Scene(root, 600, 400);
   primaryStage.setScene(scene);
   primaryStage.show();
}
```

Nesting Layouts

- As mentioned already, the Pane Class represents different types of layouts
- It’s also a Node and can be added to another Pane
- What happens if you add one Pane into another (nesting)?
- We get two labels to the left and the right of a lists of labels arranged vertically
- What is going on here?

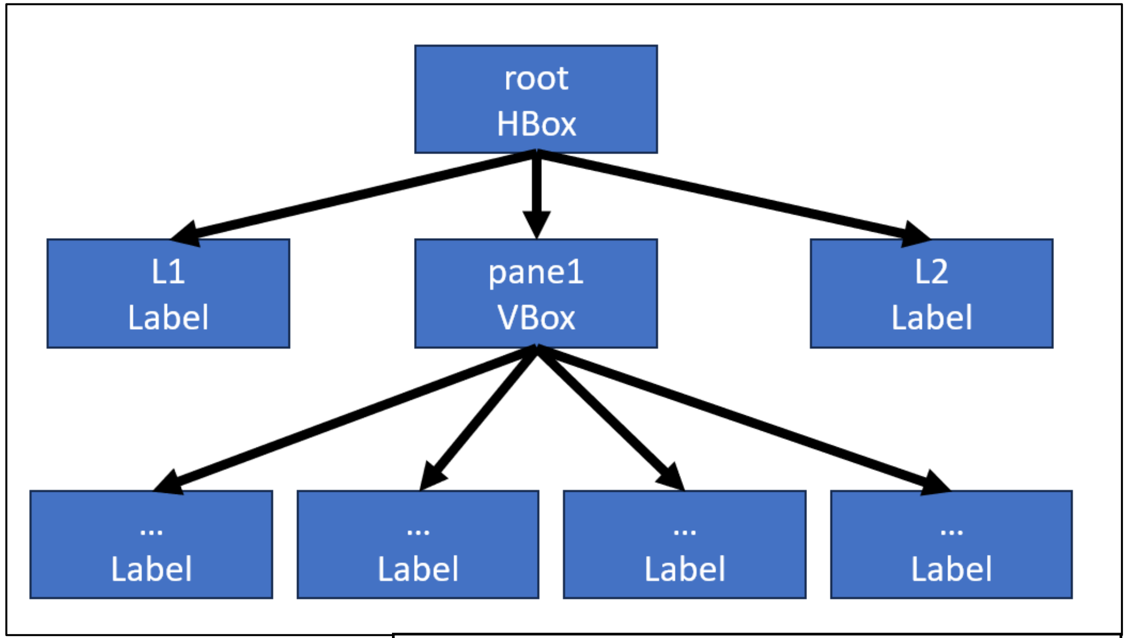
```
@Override
public void start(Stage primaryStage) {
   primaryStage.setTitle("Simple Window");
   primaryStage.setX(0);
   primaryStage.setY(0);

   Pane root;
   root = new HBox();
   VBox innerPane = new VBox();

   Label l1 = new Label("Hello World");
   Label l2 = new Label("How are you?");
   innerPane.getChildren().addAll(new Label("foo"),
         new Label("bar"), new Label("taco"));

   root.getChildren().addAll(l1, innerPane, l2);
   Scene scene = new Scene(root, 600, 400);
   primaryStage.setScene(scene);
   primaryStage.show();
}
```



```
Hello Worldfoo  How are you?
           bar
           taco
```

Scene Graph

- The elements in a Scene are organized in a hierarchical tree called the Scene Graph
- The object we pass into the Scene will become the **root** of the graph
- The **root** must be a subclass of the Parent Class
    - Both Control and Pane are subclasses of Parent
- The other objects, aside from the **root**, are called nodes
- Objects directly added a node obey the rules of that node



root HBox

L1 Label

pane1 VBox

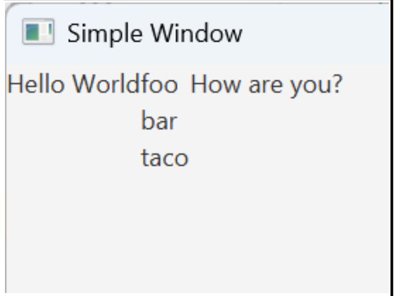
L2 Label

… Label

… Label

… Label

… Label

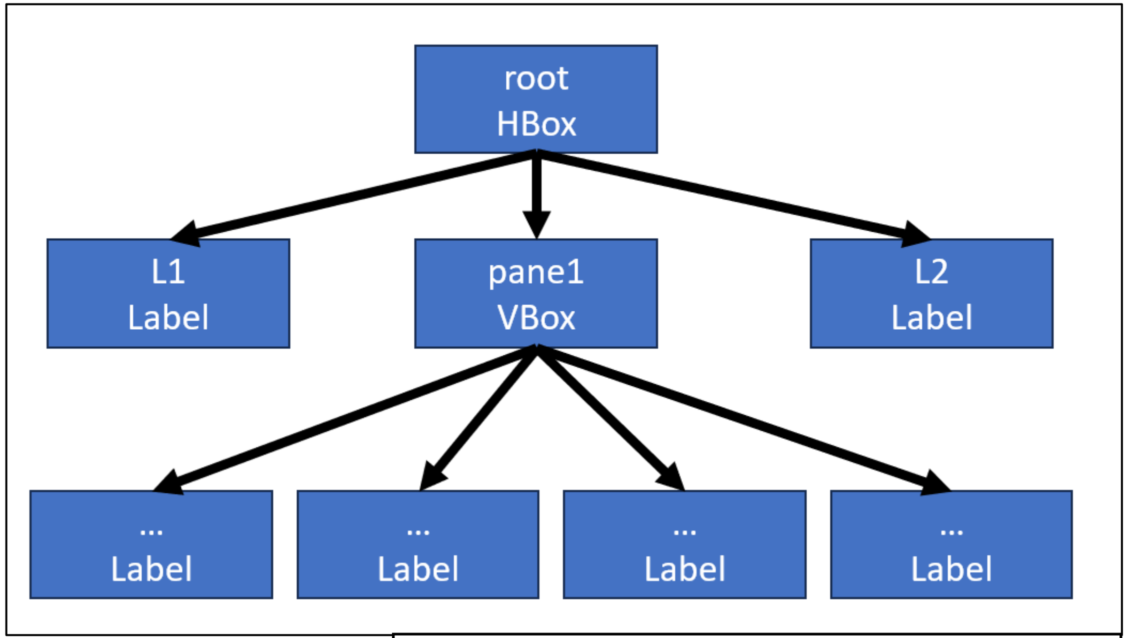


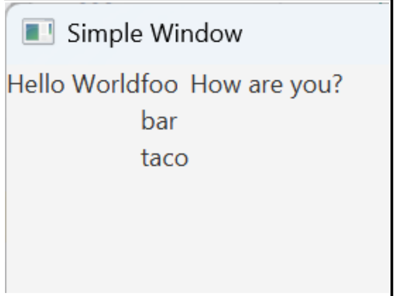
```
Hello Worldfoo  How are you?
          bar
          taco
```

```
@Override
public void start(Stage primaryStage) {
   primaryStage.setTitle("Simple Window");
   primaryStage.setX(0);
   primaryStage.setY(0);

   Pane root;
   root = new HBox();
   VBox innerPane = new VBox();

   Label l1 = new Label("Hello World");
   Label l2 = new Label("How are you?");
   innerPane.getChildren().addAll(new Label("foo"),
         new Label("bar"), new Label("taco"));

   root.getChildren().addAll(l1, innerPane, l2);
   Scene scene = new Scene(root, 600, 400);
   primaryStage.setScene(scene);
   primaryStage.show();
}
```

Scene Graph

- The root in this example is an HBox, so all the objects directly added to the root are arranged from left to right
  - This includes the VBox
- But the VBox is also container and has its own rules for arranging things
  - Everything added to the VBox is arranged vertically





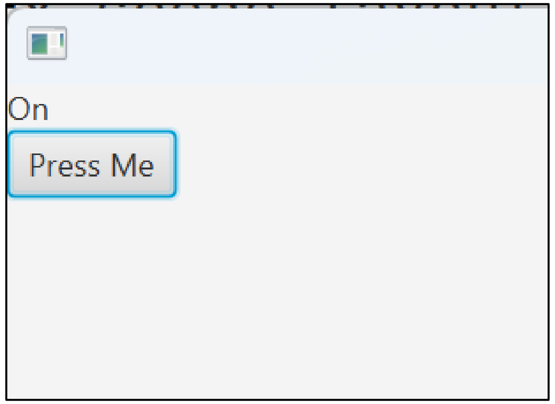
```
Hello Worldfoo  How are you?
          bar
          taco
```

```
@Override
public void start(Stage primaryStage) {
   primaryStage.setTitle("Simple Window");
   primaryStage.setX(0);
   primaryStage.setY(0);

   Pane root;
   root = new HBox();
   VBox innerPane = new VBox();

   Label l1 = new Label("Hello World");
   Label l2 = new Label("How are you?");
   innerPane.getChildren().addAll(new Label("foo"),
         new Label("bar"), new Label("taco"));

   root.getChildren().addAll(l1, innerPane, l2);
   Scene scene = new Scene(root, 600, 400);
   primaryStage.setScene(scene);
   primaryStage.show();
}
```

Button

- Let’s add another component, a Button, which is a Control object that simulates pressing a button
- Right now, if we push the button, nothing happens
- To tell the Button what to do when it is pressed, we must call the setOnAction() method and pass in the action to perform

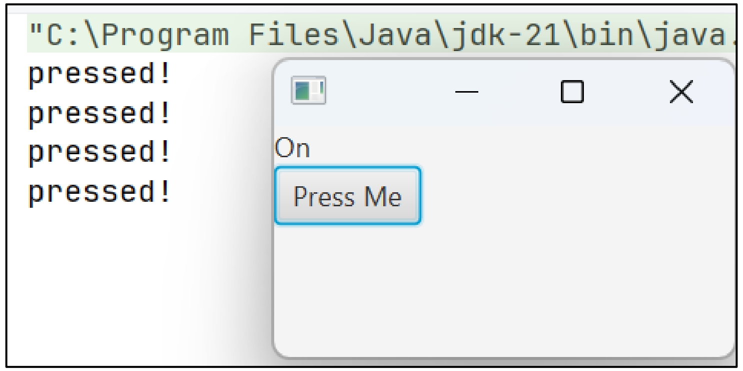
```
public class ControlObjects extends Application {
   public static void main(String[] args) {
      launch(args);
   }
   @Override
   public void start(Stage stage) throws Exception {
      Pane root = new VBox();

      Label label1 = new Label("On");
      Button button1 = new Button("Press Me");

      root.getChildren().addAll(label1, button1);
      Scene scene = new Scene(root, 600, 400);
      stage.setScene(scene);
      stage.show();
   }
}
```



setOnAction()

- setOnAction() takes in an EventHandler<ActionEvent> object (more on that later) that tells the program what method to run when the button is pressed
  - Essentially, we are assigning the action we want the button to do when it is pressed
- One way to create this EventHandler<ActionEvent> object is through a method reference

```
public void start(Stage stage) throws IOException {
   Pane root = new VBox();
   Label label1 = new Label("Press the Button.");

   Button button1 = new Button("On");
   button1.setOnAction(this::respond);

   root.getChildren().addAll(label1, button1);

   //200 and 100 are the width and height
   Scene scene = new Scene(root,200,100);
   stage.setScene(scene);
   stage.show();
}
private void respond(ActionEvent event){
   System.out.println("Button pressed!");
}
```



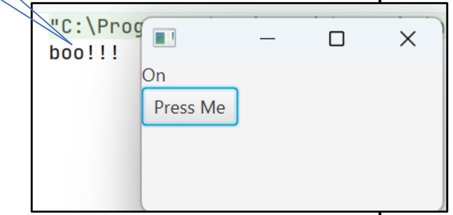
```
"C:\Program Files\Java\jdk-21\bin\java.
pressed!
pressed!
pressed!
pressed!
```

Method reference

- A method reference allows you to assign a method to be run when the button is pressed
- The only catch is that the method must have a void return type and take in an Event argument
- Using a method reference is like using a method from a class
- If it is an instance method, then you can make an instance of the class and then do varName::methodName
- If it is a static method, you can do ClassName::staticMethodName

```
public class SomeClass {
   public void someMethod(ActionEvent event){
      System.out.println("boo!!!");
   }
   public static void someStaticMethod(ActionEvent
event){
      System.out.println("yeah!!!");
   }
}
```

```
public class ControlObjects extends Application {
   public static void main(String[] args) {
      launch(args);
   }
   @Override
   public void start(Stage stage) throws Exception {
      Pane root = new VBox();

      Label l1 = new Label("On");
      Button b1 = new Button("Press Me");

      //Method reference from a method in this class
      EventHandler<ActionEvent> h1 = this::respond;
      //Method reference from an instance method in another class
      SomeClass s1 = new SomeClass();
      EventHandler<ActionEvent> h2 = s1::someMethod;
      //Method reference from a static method in another class
      EventHandler h3 = SomeClass::someStaticMethod;
      b1.setOnAction(h2);

      root.getChildren().addAll(l1, b1);
      Scene scene = new Scene(root, 200, 100);
      stage.setScene(scene);
      stage.show();
   }
   private void respond(ActionEvent event){
      System.out.println("pressed!");
   }
}
```



```
"C:\Prog
boo!!!
```

Event-delegation model

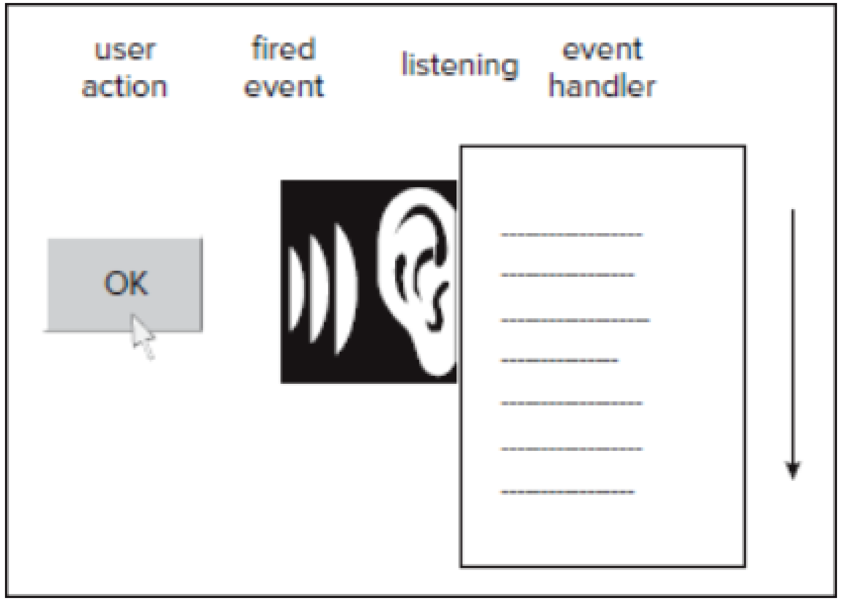
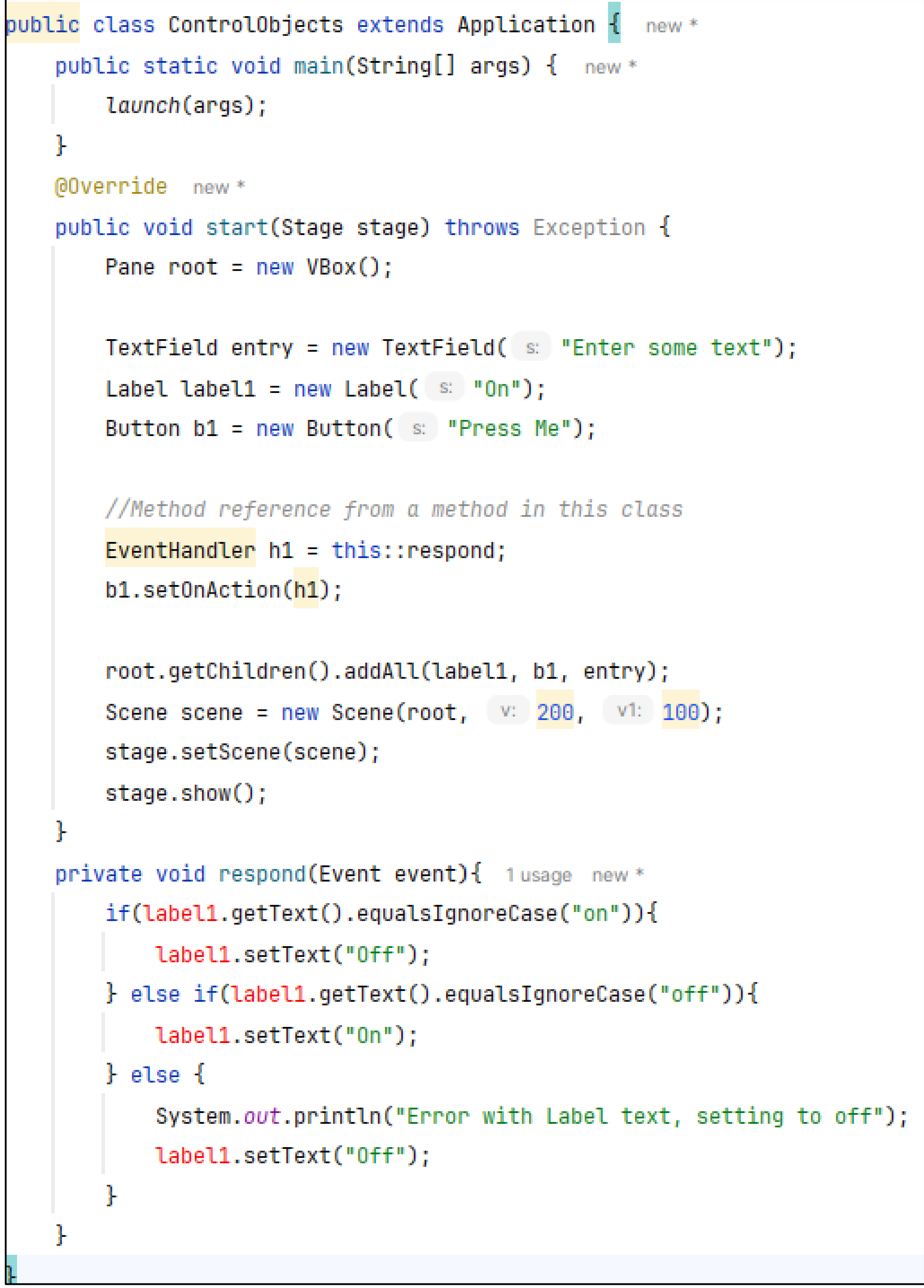


Figure 17.5 What happens when a button is pressed

- This system of an object listening for an event and then executing some defined operation in response is called the event-delegation model
- When you interact with a source object like a Button it fires/creates an event
    - If an EventHandler<ActionEvent> is registered to the Button and ***listening*** for an Event, it will catch the event and execute some code in response
    - If no EventHandler<ActionEvent> is registered for the Button, the event falls on deaf ears
- In this way, the source object (i.e., Button) delegates the act of responding to the event to something else, i.e., the EventHandler<ActionEvent>

Button Press

- For this example, we’ll use the respond method within the current class for the method reference
- Let’s say we wanted to modify the Label from “On” to “Off” or “Off” to “On” every time we pushed it
- We can get and set the text of the Label with the methods getText() and setText(), but we need a reference to the Label first
- But we can not access l1 from respond() because it is declared in start() and is therefore out of scope in respond()

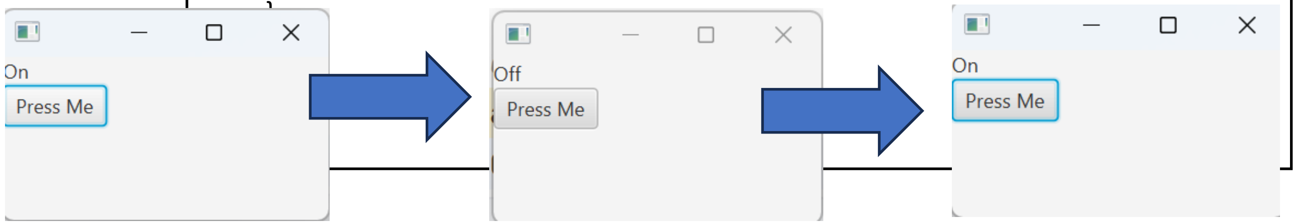


```java
public class ControlObjects extends Application {
    public static void main(String[] args) {
        launch(args);
    }
    @Override
    public void start(Stage stage) throws Exception {
        Pane root = new VBox();

        TextField entry = new TextField("Enter some text");
        Label label1 = new Label("On");
        Button b1 = new Button("Press Me");

        //Method reference from a method in this class
        EventHandler h1 = this::respond;
        b1.setOnAction(h1);

        root.getChildren().addAll(label1, b1, entry);
        Scene scene = new Scene(root, 200, 100);
        stage.setScene(scene);
        stage.show();
    }
    private void respond(Event event){
        if(label1.getText().equalsIgnoreCase("on")){
            label1.setText("Off");
        } else if(label1.getText().equalsIgnoreCase("off")){
            label1.setText("On");
        } else {
            System.out.println("Error with Label text, setting to off");
            label1.setText("Off");
        }
    }
}
```

Button Press

- If we declared l1 outside of start(), as an instance variable, then it will be in the scope of both start() and respond()
- It can then be access and modified in both methods
- Now clicking the button toggles the text on the Label

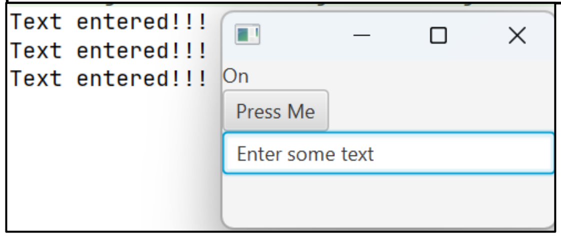
```java
public class ControlObjects extends Application {
  public static void main(String[] args) {
    launch(args);
  }
  private Label label1;
  @Override
  public void start(Stage stage) throws Exception {
    Pane root = new VBox();

    TextField entry = new TextField("Enter some text");
    label1 = new Label("On");
    Button button1 = new Button("Press Me");

    //Method reference from a method in this class
    EventHandler<ActionEvent> h1 = this::respond;
    button1.setOnAction(h1);

    root.getChildren().addAll(label1, b1, entry);
    Scene scene = new Scene(root, 200, 100);
    stage.setScene(scene);
    stage.show();
  }
  private void respond(ActionEvent event){
    if(label1.getText().equalsIgnoreCase("on")){
      label1.setText("Off");
    } else if(label1.getText().equalsIgnoreCase("off")){
      label1.setText("On");
    } else {
      System.out.println("Error with Label text, setting to off");
      label1.setText("Off");
    }
```



TextField

- Let’s look at another Control component we could add to our program
- A TextField object allows you to enter a single line of text
  - If you wanted to enter multiple lines of text you would use a TextArea
- When you hit Enter with your cursor on the TextField it will trigger an Event
- We can use setOnAction() to register an EventHandler that will run when we trigger the TextField

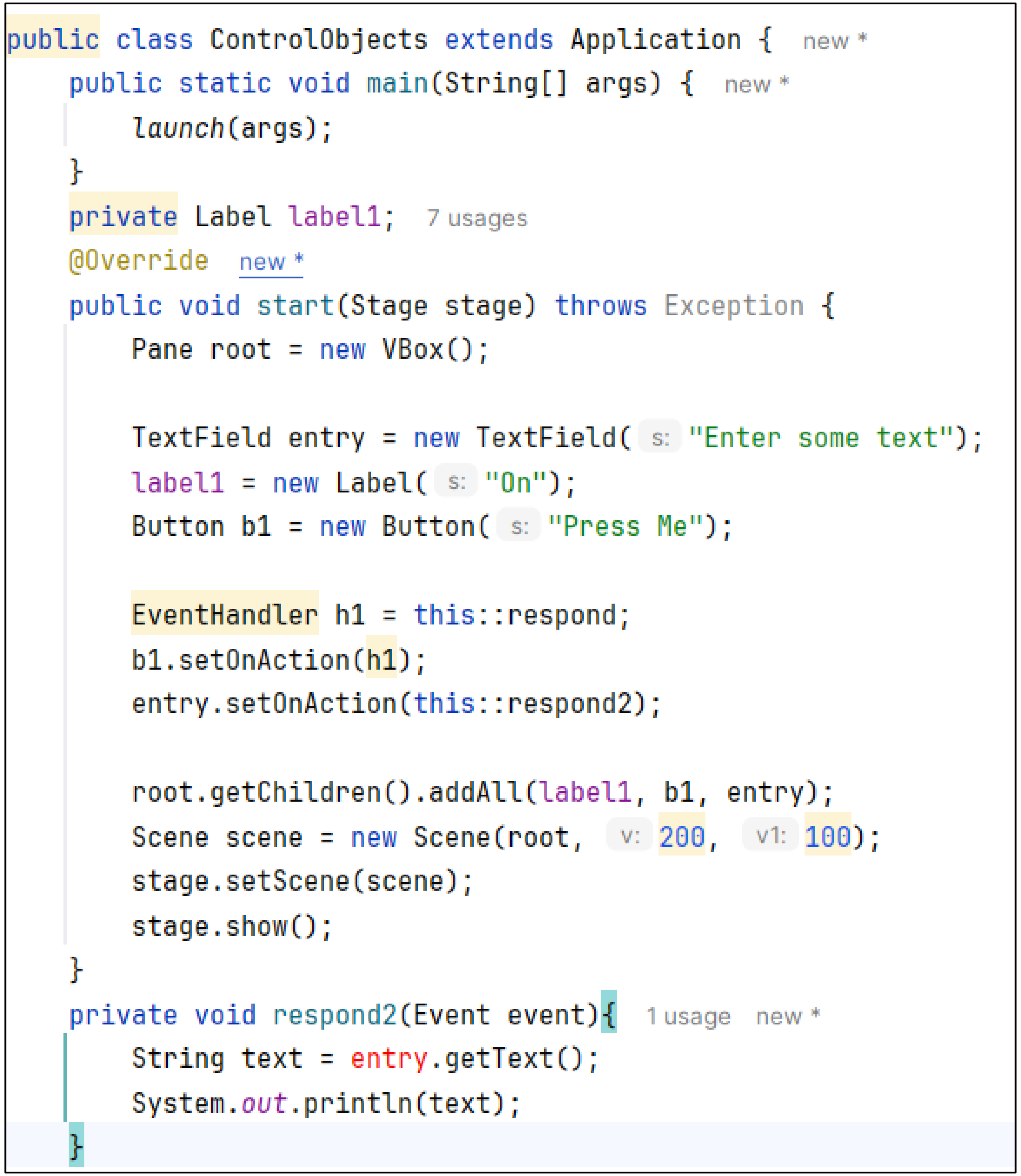
```
public class ControlObjects extends Application {
  public static void main(String[] args) {
    launch(args);
  }
  private Label label1;
  @Override
  public void start(Stage stage) throws Exception {
    Pane root = new VBox();

    TextField entry = new TextField("Enter some text");
    label1 = new Label("On");
    Button b1 = new Button("Press Me");



    EventHandler<ActionEvent> h1 = this::respond;
    b1.setOnAction(h1);
    entry.setOnAction(this::respond2);

    root.getChildren().addAll(label1, b1, entry);
    Scene scene = new Scene(root, 200, 100);
    stage.setScene(scene);
    stage.show();
  }
  private void respond2(ActionEvent event){
    System.out.println("Text entered!!!");
  }
```



```
Text entered!!!
Text entered!!!
Text entered!!!
```

TextField

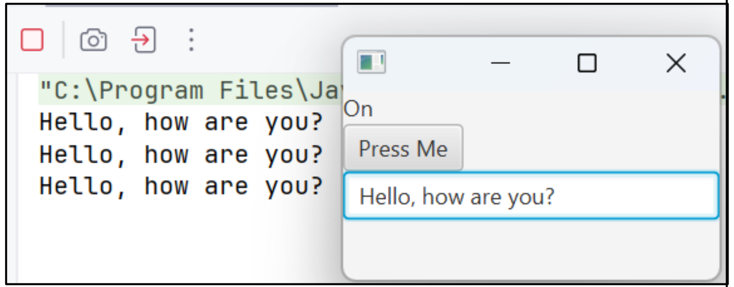
- For a TextField, we would like to call getText() to get what text is written in the TextField, and then do something with that text
- To call getText() on the TextField entry in respond2(), we need a reference to it just like with Label label1
- We could make the TextField entry an instance variable like we did label1, but here is another way to get a reference to it



```
public class ControlObjects extends Application {  new *
    public static void main(String[] args) {  new *
        launch(args);
    }
    private Label label1;  7 usages
    @Override  new *
    public void start(Stage stage) throws Exception {
        Pane root = new VBox();

        TextField entry = new TextField( s: "Enter some text");
        label1 = new Label( s: "On");
        Button b1 = new Button( s: "Press Me");

        EventHandler h1 = this::respond;
        b1.setOnAction(h1);
        entry.setOnAction(this::respond2);

        root.getChildren().addAll(label1, b1, entry);
        Scene scene = new Scene(root,  v: 200,  v1: 100);
        stage.setScene(scene);
        stage.show();
    }
    private void respond2(Event event){  1 usage  new *
        String text = entry.getText();
        System.out.println(text);
    }
```

getSource()

- The Event object being passed into respond2, has information about what caused the Event
  - One of those pieces of information is the source of the Event
- event.getSource() will return a reference to the object that triggered the Event
  - In this case, that is the TextField entry

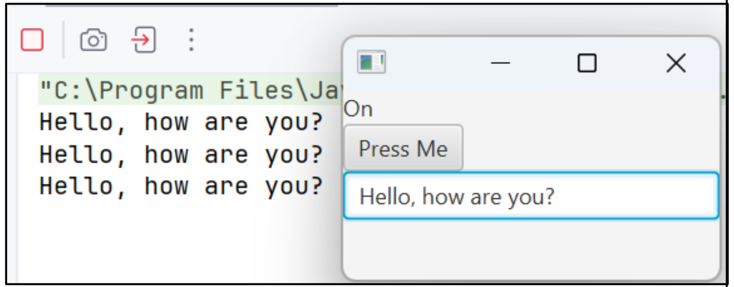


```
"C:\Program Files\Ja
Hello, how are you?
Hello, how are you?
Hello, how are you?
```

```java
public class ControlObjects extends Application {
    public static void main(String[] args) {
        launch(args);
    }
    private Label label1;
    @Override
    public void start(Stage stage) throws Exception {
        Pane root = new VBox();

        TextField entry = new TextField("Enter some text");
        label1 = new Label("On");
        Button b1 = new Button("Press Me");

        EventHandler<ActionEvent> h1 = this::respond;
        b1.setOnAction(h1);
        entry.setOnAction(this::respond2);

        root.getChildren().addAll(label1, b1, entry);
        Scene scene = new Scene(root, 200, 100);
        stage.setScene(scene);
        stage.show();
    }
    private void respond2(ActionEvent event){
        TextField text = (TextField) event.getSource();
        System.out.println(text.getText());
    }
    private void respond(ActionEvent event){
        if(label1.getText().equalsIgnoreCase("on")){
            label1.setText("Off");
        } else if(label1.getText().equalsIgnoreCase("off")){
            label1.setText("On");
        } else {
            System.out.println("Error with Label text, setting to off");
            label1.setText("Off");
        }
    }
}
```

getSource()

- We can then cast the return value from getSource() to a TextField object, because we know that was the object we registered the EventHandler<ActionEvent> to
  - If we registered respond2 to both the TextField and the Button, we could do an instanceof on the return value of getSource() to figure out which of the two objects caused the Event
- We can then treat this TextField object like the TextField entry defined in start()



```
"C:\Program Files\Ja
Hello, how are you?
Hello, how are you?
Hello, how are you?
```

Hello, how are you?

```
public class ControlObjects extends Application {
   public static void main(String[] args) {
      launch(args);
   }
   private Label label1;
   @Override
   public void start(Stage stage) throws Exception {
      Pane root = new VBox();

      TextField entry = new TextField("Enter some text");
      label1 = new Label("On");
      Button b1 = new Button("Press Me");

      EventHandler<ActionEvent> h1 = this::respond;
      b1.setOnAction(h1);
      entry.setOnAction(this::respond2);

      root.getChildren().addAll(label1, b1, entry);
      Scene scene = new Scene(root, 200, 100);
      stage.setScene(scene);
      stage.show();
   }
   private void respond2(ActionEvent event){
      TextField text = (TextField) event.getSource();
      System.out.println(text.getText());
   }
   private void respond(ActionEvent event){
      if(label1.getText().equalsIgnoreCase("on")){
         label1.setText("Off");
      } else if(label1.getText().equalsIgnoreCase("off")){
         label1.setText("On");
      } else {
         System.out.println("Error with Label text, setting to off");
         label1.setText("Off");
      }
   }
}
```

Multiple objects

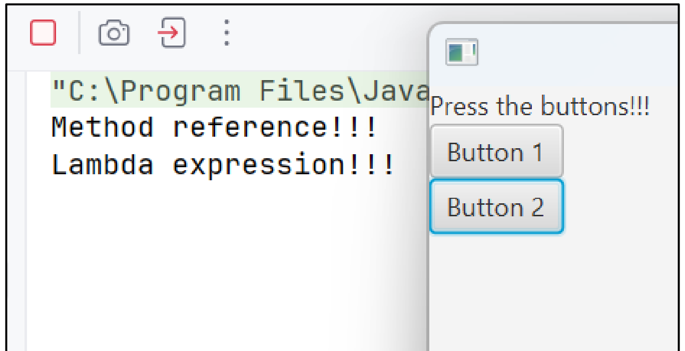
- It is possible to use the same EventHandler<ActionEvent> on multiple objects
- Within the EventHandler<ActionEvent> method, you could do an instanceof on the event source to figure out what object triggered the event

```
public class ControlObjects extends Application {
  public static void main(String[] args) {
    launch(args);
  }
  private Label label1;
  @Override
  public void start(Stage stage) throws Exception {
    Pane root = new VBox();
    TextField entry = new TextField("Enter some text");
    label1 = new Label("On");
    Button b1 = new Button("Press Me");

    EventHandler<ActionEvent> h1 = this::respond3;
    b1.setOnAction(h1);
    entry.setOnAction(h1);

    root.getChildren().addAll(label1, b1, entry);
    Scene scene = new Scene(root, 200, 100);
    stage.setScene(scene);
    stage.show();
  }
  private void respond3(ActionEvent event){
    if(event.getSource() instanceof Button) {
      if (label1.getText().equalsIgnoreCase("on")) {
        label1.setText("Off");
      } else if (label1.getText().equalsIgnoreCase("off")) {
        label1.setText("On");
      } else {
        System.out.println("Error with Label text, setting to off");
        label1.setText("Off");
      }
    } else if(event.getSource() instanceof TextField){
      TextField text = (TextField) event.getSource();
      System.out.println(text.getText());
    }
  }
```

Lambda expression

- We’ve been using a method reference to create an instance of an EventHandler<ActionEvent> that is then passed into the setOnAction() method
    - But there are other ways to create an EventHandler<ActionEvent> object
- A lambda expression creates an object that implements a given function interface
    - EventHandler<ActionEvent> is a functional interface. We will get to functional interfaces in week 4.
- For right now, you can think of a lambda expression as a shorthand for creating a method

```
public class OtherEventHandlers extends Application {
  @Override
  public void start(Stage stage) throws Exception {
    Pane root = new VBox();

    Button b1 = new Button("Button 1");
    b1.setOnAction(this::respond);

    Button b2 = new Button("Button 2");
    b2.setOnAction((ActionEvent event) -> {
      System.out.println("Lambda expression!!!");
    });

    root.getChildren().addAll(new Label("Press the buttons!!!"),
        b1, b2);
    Scene scene= new Scene(root, 400,200);
    stage.setScene(scene);
    stage.show();
  }
  private void respond(ActionEvent event){
    System.out.println("Method reference!!!");
  }
}
```

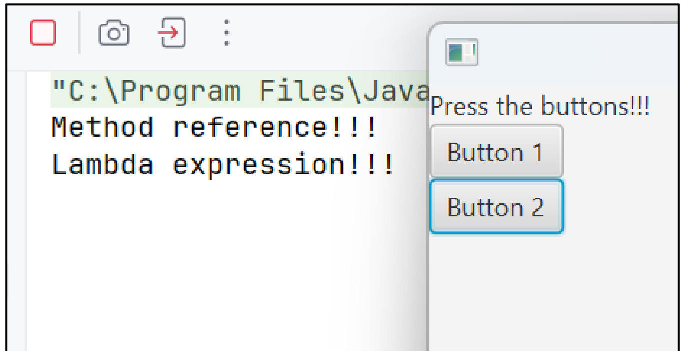


```
"C:\Program Files\Java
Method reference!!!
Lambda expression!!!
```

Lambda expression

- The syntax for a lambda expression is (<args>) -> { <method_body> };
- The first part <args>, are the arguments that will be passed into the method
- The second part <method_body>, is the code that will be executed when the Button or TextField is triggered

```java
public class OtherEventHandlers extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        Pane root = new VBox();

        Button b1 = new Button("Button 1");
        b1.setOnAction(this::respond);

        Button b2 = new Button("Button 2");
        b2.setOnAction((ActionEvent event) -> {
            System.out.println("Lambda expression!!!");
        });

        root.getChildren().addAll(new Label("Press the buttons!!!"),
                b1, b2);
        Scene scene= new Scene(root, 400,200);
        stage.setScene(scene);
        stage.show();
    }
    private void respond(ActionEvent event){
        System.out.println("Method reference!!!");
    }
}
```

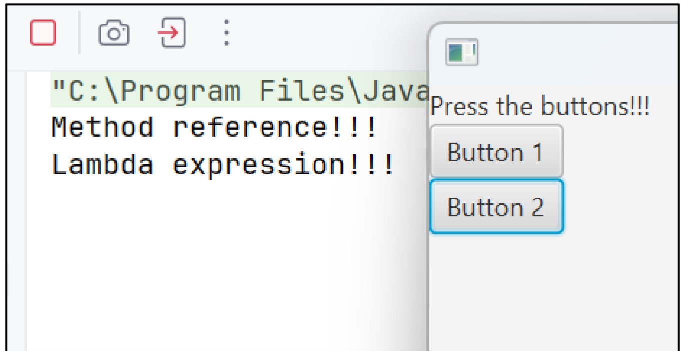


```
"C:\Program Files\Java
Method reference!!!
Lambda expression!!!
```

Lambda expression vs method reference

- The advantage of a lambda expression is that it is simpler and faster to write
- The main downside is that it’s less reusable and becomes awkward for larger, more complex sets of operations

```
public class OtherEventHandlers extends Application {
  @Override
  public void start(Stage stage) throws Exception {
    Pane root = new VBox();

    Button b1 = new Button("Button 1");
    b1.setOnAction(this::respond);

    Button b2 = new Button("Button 2");
    b2.setOnAction((ActionEvent event) -> {
      System.out.println("Lambda expression!!!");
    });

    root.getChildren().addAll(new Label("Press the buttons!!!"),
        b1, b2);
    Scene scene= new Scene(root, 400,200);
    stage.setScene(scene);
    stage.show();
  }
  private void respond(ActionEvent event){
    System.out.println("Method reference!!!");
  }
}
```



```
"C:\Program Files\Java
Method reference!!!
Lambda expression!!!
```

EventHandler<ActionEvent>

- There are three other ways of making an instance of an EventHandler<ActionEvent> that have their own advantages
- If we look up the documentation for EventHandler <ActionEvent>, we see that it is an interface
  - Specifically, it is a functional interface, which means it only has a single method
- Because it is an interface, there's nothing stopping us from making our own class that implements it

```
public class MyEventHandler implements
EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Do something.");
    }
}
```

```
public class OtherEventHandlers extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        Pane root = new VBox();

        Button b1 = new Button("Button 1");
        b1.setOnAction(this::respond);

        Button b2 = new Button("Button 2");
        b2.setOnAction((ActionEvent event) -> {
            System.out.println("Lambda expression!!!");
        });

        Button b3 = new Button("Button 3");
        EventHandler<ActionEvent> h1 = new MyEventHandler();
        b3.setOnAction(h1);

        root.getChildren().addAll(new Label("Press the buttons!!!"),
                b1, b2, b3);
        Scene scene = new Scene(root, 400,200);
        stage.setScene(scene);
        stage.show();
    }
    private void respond(ActionEvent event){
        System.out.println("Method reference!!!");
    }
}
```

EventHandler <ActionEvent>

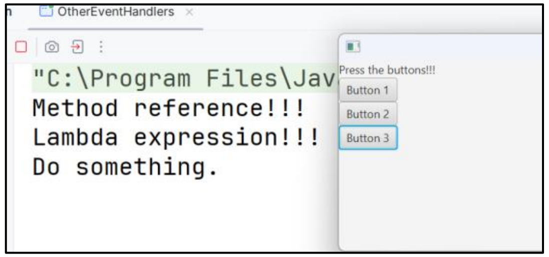
- Because MyEventHandler implements EventHandler<ActionEvent>, we can make an instance of it and assign it to an EventHandler<ActionEvent> reference
- We can then pass that reference into setOnAction()
- But what happens when we press the button?

```
public class MyEventHandler implements
EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Do something.");
    }
}
```

```
public class OtherEventHandlers extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        Pane root = new VBox();

        Button b1 = new Button("Button 1");
        b1.setOnAction(this::respond);

        Button b2 = new Button("Button 2");
        b2.setOnAction((ActionEvent event) -> {
            System.out.println("Lambda expression!!!");
        });

        Button b3 = new Button("Button 3");
        EventHandler<ActionEvent> h1 = new MyEventHandler();
        b3.setOnAction(h1);

        root.getChildren().addAll(new Label("Press the buttons!!!"),
                b1, b2, b3);
        Scene scene = new Scene(root, 400,200);
        stage.setScene(scene);
        stage.show();
    }
    private void respond(ActionEvent event){
        System.out.println("Method reference!!!");
    }
}
```

EventHandler <ActionEvent>

- When we press button3, it calls the handle() method for MyEventHandler
  - What is going on here?
- When we pass an instance of MyEventHandler into setOnAction() for the Button b3, b3 saved a reference to the event handler
- When b3 is pressed, it calls the handle() method of the event handler that was saved



```
"C:\Program Files\Jav
Method reference!!!
Lambda expression!!!
Do something.
```

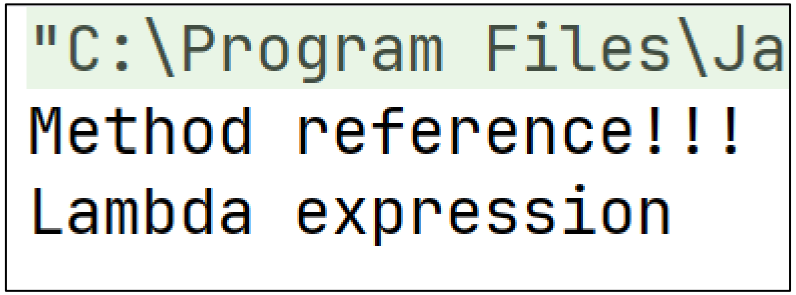
```java
public class MyEventHandler implements
EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Do something.");
    }
}
```

```java
public class OtherEventHandlers extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        Pane root = new VBox();

        Button b1 = new Button("Button 1");
        b1.setOnAction(this::respond);

        Button b2 = new Button("Button 2");
        b2.setOnAction((ActionEvent event) -> {
            System.out.println("Lambda expression!!!");
        });

        Button b3 = new Button("Button 3");
        EventHandler<ActionEvent> h1 = new MyEventHandler();
        b3.setOnAction(h1);

        root.getChildren().addAll(new Label("Press the buttons!!!"),
                b1, b2, b3);
        Scene scene = new Scene(root, 400,200);
        stage.setScene(scene);
        stage.show();
    }
    private void respond(ActionEvent event){
        System.out.println("Method reference!!!");
    }
}
```

EventHandler<ActionEvent>

- So what was going on with the method reference and lambda expression?
- Method references or a lambda expressions are a shorthand for making an object that implements some interface
  - In this case the interface was an EventHandler<ActionEvent>
- All that mattered was that the signature of the method reference and lambda expression matched the handle() method of EventHandler<ActionEvent>
  - I.E., returns void and takes in an ActionEvent as an argument

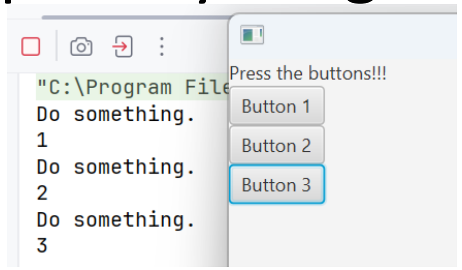
```java
public class OtherEventHandlers extends Application {
  @Override
  public void start(Stage stage) throws Exception {

    EventHandler<ActionEvent> h1 = this::respond;
    h1.handle(null);

    EventHandler<ActionEvent> h2 = (ActionEvent event) -> {
      System.out.println("Lambda expression");
    };
    h2.handle(null);

  }
  private void respond(ActionEvent event){
    System.out.println("Method reference!!!");
  }
}
```



```
"C:\Program Files\Ja
Method reference!!!
Lambda expression
```

MyEventHandler

- The advantage of defining a whole class that implements EventHandler<ActionEvent>, instead of using a method reference or lambda expression, is that we have access to instance variables
- Let’s say we wanted to count how many times the button is pressed
- We could add a numPressed attribute to MyEventHandler that ticks up everything handle() is called



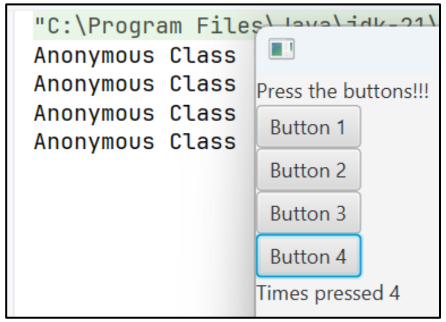
```
"C:\Program File
Do something.
1
Do something.
2
Do something.
3
```

```
public class MyEventHandler implements
EventHandler<ActionEvent> {
  private int numPressed = 0;
  @Override
  public void handle(ActionEvent event) {
    System.out.println("Do something.");
    numPressed += 1;
    System.out.println(numPressed);
  }
}
```

```
public class OtherEventHandlers extends Application {
  @Override
  public void start(Stage stage) throws Exception {
    Pane root = new VBox();

    Button b1 = new Button("Button 1");
    b1.setOnAction(this::respond);

    Button b2 = new Button("Button 2");
    b2.setOnAction((ActionEvent event) -> {
      System.out.println("Lambda expression!!!");
    });
    Button b3 = new Button("Button 3");
    EventHandler<ActionEvent> h1 = new MyEventHandler();
    b3.setOnAction(h1);
    root.getChildren().addAll(new Label("Press the buttons!!!"),
        b1, b2, b3);
    Scene scene = new Scene(root, 400,200);
    stage.setScene(scene);
    stage.show();
  }
  private void respond(ActionEvent event){
    System.out.println("Method reference!!!");
  }
}
```

Anonymous Class

- If we wanted the benefit of a new Class, but didn’t want to write a new class in a new file, we could do an anonymous class or an inner class
- You can think of an anonymous class like a lambda expression in that it can be a quicker way of defining a new class instead of having to make a new file
- A benefit of an anonymous class is that we have access to all the private fields, such as label, of the class we are in



```
"C:\Program Files\Java\jdk-21\
Anonymous Class
Anonymous Class
Anonymous Class
Anonymous Class
```

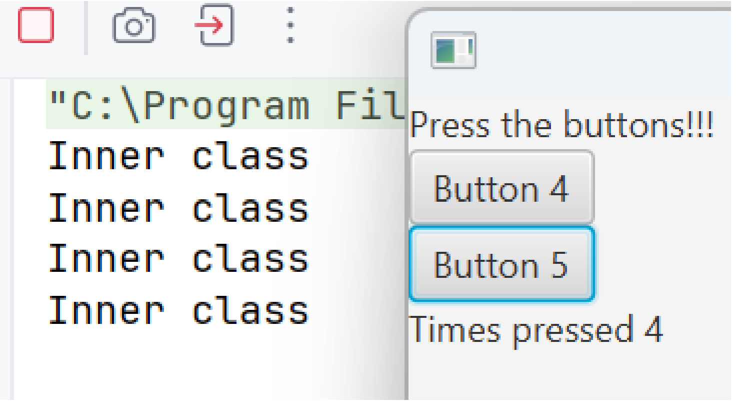
Press the buttons!!!

Times pressed 4

```java
public class OtherEventHandlers extends Application {
   private Label label;
   @Override
   public void start(Stage stage) throws Exception {
      Pane root = new VBox();
      Button b1 = new Button("Button 1");
      b1.setOnAction(this::respond);
      Button b2 = new Button("Button 2");
      b2.setOnAction((ActionEvent event) -> {
         System.out.println("Lambda expression!!!");
      });

      Button b3 = new Button("Button 3");
      EventHandler<ActionEvent> h1 = new MyEventHandler();
      b3.setOnAction(h1);

      label = new Label();
      Button b4 = new Button("Button 4");
      EventHandler<ActionEvent> h2 = new EventHandler<ActionEvent>() {
         private int numPressed = 0;
         @Override
         public void handle(ActionEvent event) {
            System.out.println("Anonymous Class");
            numPressed += 1;
            label.setText("Times pressed " + numPressed);
         }
      };
      b4.setOnAction(h2);

      root.getChildren().addAll(new Label("Press the buttons!!!"),
            b1, b2, b3, b4, label);
      Scene scene = new Scene(root, 400,200);
      stage.setScene(scene);
      stage.show();
   }
   private void respond(ActionEvent event){
      System.out.println("Method reference!!!");
   }
}
```

Inner Class

- An inner class is a class defined within another class
- It has the same benefits as an anonymous class, except like a method reference, it is reusable and might be easier to write for more complex operations



```
"C:\Program Fil
Inner class
Inner class
Inner class
Inner class
```

Press the buttons!!!

Button 4

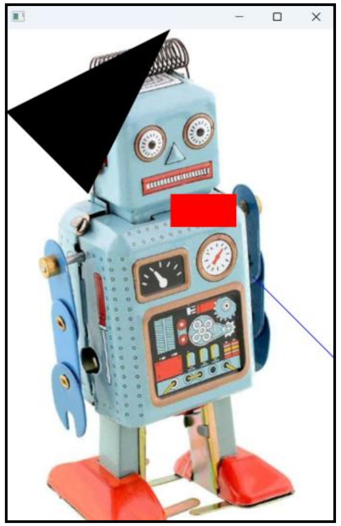
Button 5

Times pressed 4

```java
public class OtherEventHandlers extends Application {
  private class MyEventHandlerInner
      implements EventHandler<ActionEvent> {
    private int numPressed = 0;
    public void handle(ActionEvent event){
      numPressed += 1;
      System.out.println("Inner method");
      label.setText("Times pressed "+numPressed);
    }
  }
  private Label label;
  @Override
  public void start(Stage stage) throws Exception {
    Pane root = new VBox();
    label = new Label();
    Button b4 = new Button("Button 4");
    EventHandler<ActionEvent> h2 = new EventHandler<ActionEvent>() {
      private int numPressed = 0;
      @Override
      public void handle(ActionEvent event) {
        System.out.println("Anonymous Class");
        numPressed += 1;
        label.setText("Times pressed " + numPressed);
      }
    };
    b4.setOnAction(h2);
    Button b5 = new Button("Button 5");
    b5.setOnAction(new MyEventHandlerInner());

    root.getChildren().addAll(new Label("Press the buttons!!!"),
        b4, b5, label);
    Scene scene = new Scene(root, 400,200);
    stage.setScene(scene);
    stage.show();
  }
}
```

Other gui elements

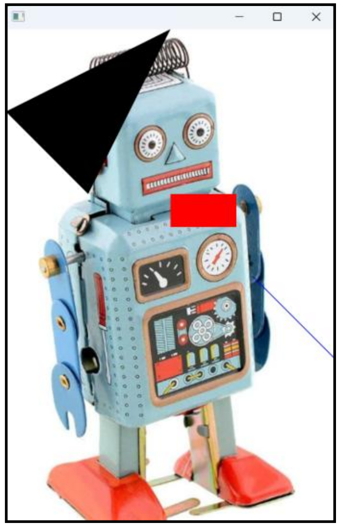
- There are many other types of objects that can be added to our gui
- For example, you can add prebuilt shapes
- You can change many properties of these shapes like their rotation, fill color, or bordercolor
- Note, that you add shapes to a Group object, which is also a subclass of Parent, not Pane object



```
public class MouseActions extends Application {
  @Override
  public void start(Stage stage) throws Exception {

    ImageView imageView = new ImageView();
    imageView.setImage(new Image(new FileInputStream("images/robot.png")));
    imageView.setOnMouseClicked((MouseEvent e) -> {
        double x = e.getX();
        double y = e.getY();
        System.out.println("Moving on image at ("+x+","+y+")");
     });


    Rectangle r1 = new Rectangle(200,200, 80,40);
    r1.setFill(Color.RED);
    r1.setOnMouseMoved(this::respond);

    Polygon p1 = new Polygon();
    Double[] points = {0.0, 0.0, 200.0, 100.0,100.0, 200.0};
    p1.getPoints().addAll(points);
    p1.setRotate(90);

    Line l1 = new Line();
    l1.setStartX(300);
    l1.setStartY(300);
    l1.setEndX(600);
    l1.setEndY(600);
    l1.setStroke(Color.BLUE);

    Group root = new Group();
    root.getChildren().addAll(imageView, r1, p1, l1);
```

Other gui elements

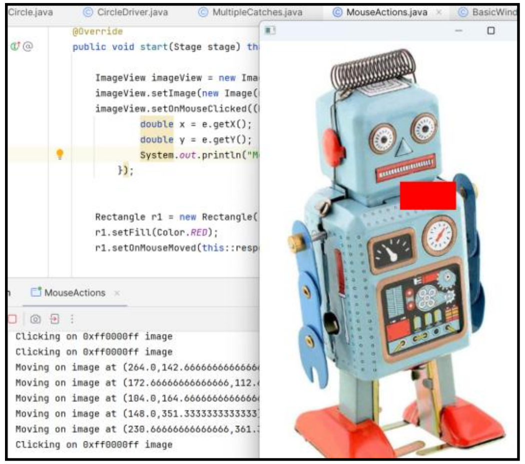
- You can also display images by first creating an Image object with the image's filename
- Then setting the image to an ImageView object
- An ImageView object can be added to a Pane as well as a Group



```
public class MouseActions extends Application {
  @Override
  public void start(Stage stage) throws Exception {

      ImageView imageView = new ImageView();
      imageView.setImage(new Image(new FileInputStream("images/robot.png")));
      imageView.setOnMouseClicked((MouseEvent e) -> {
            double x = e.getX();
            double y = e.getY();
            System.out.println("Moving on image at ("+x+","+y+")");
        });


      Rectangle r1 = new Rectangle(200,200, 80,40);
      r1.setFill(Color.RED);
      r1.setOnMouseMoved(this::respond);

      Polygon p1 = new Polygon();
      Double[] points = {0.0, 0.0, 200.0, 100.0,100.0, 200.0};
      p1.getPoints().addAll(points);
      p1.setRotate(90);

      Line l1 = new Line();
      l1.setStartX(300);
      l1.setStartY(300);
      l1.setEndX(600);
      l1.setEndY(600);
      l1.setStroke(Color.BLUE);

      Group root = new Group();
      root.getChildren().addAll(imageView, r1, p1, l1);
```

Other Events

- There are many other types of Events aside from ActionEvents
- MouseEvents are events that involve the mouse
- Common operations include moving, dragging, and clicking
- One feature of mouse events is the ability to get the x and y position of the event



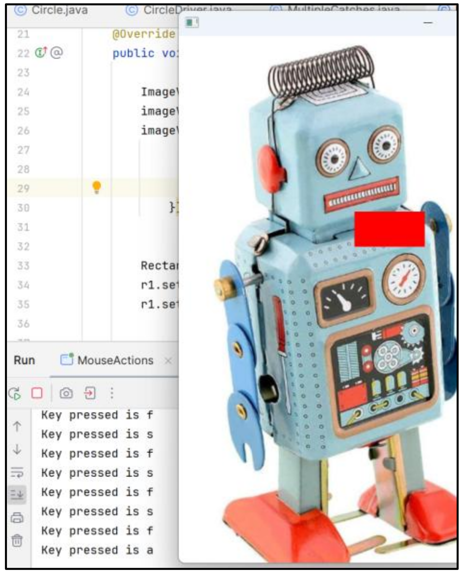
```
    @Override
    public void start(Stage stage) th

        ImageView imageView = new Ima
        imageView.setImage(new Image(
        imageView.setOnMouseClicked((
                double x = e.getX();
                double y = e.getY();
                System.out.println("M
            });


        Rectangle r1 = new Rectangle(
        r1.setFill(Color.RED);
        r1.setOnMouseMoved(this::resp
```

```
Clicking on 0xff0000ff image
Clicking on 0xff0000ff image
Moving on image at (264.0,142.66666666666666
Moving on image at (172.66666666666666,112.
Moving on image at (104.0,164.66666666666666
Moving on image at (148.0,351.3333333333333)
Moving on image at (230.66666666666666,361.
Clicking on 0xff0000ff image
```

```java
public class MouseActions extends Application {
   @Override
   public void start(Stage stage) throws Exception {

      ImageView imageView = new ImageView();
      imageView.setImage(new Image(new FileInputStream("images/robot.png")));
      imageView.setOnMouseClicked((MouseEvent e) -> {
            double x = e.getX();
            double y = e.getY();
            System.out.println("Clicking on image at ("+x+","+y+")");
         });


      Rectangle r1 = new Rectangle(200,200, 80,40);
      r1.setFill(Color.RED);
      r1.setOnMouseMoved(this::respond);


      Group root = new Group();
      root.getChildren().addAll(imageView,r1);

      Scene s1 = new Scene(root,400,600);
      s1.setOnKeyTyped((KeyEvent e) -> System.out.println("Key pressed is "+e.getCharacter()));

      stage.setOnCloseRequest((WindowEvent e) -> System.out.println("Goodbye"));
      stage.setScene(s1);
      stage.show();
   }

   public void respond(MouseEvent e){
      String color = ((Rectangle)e.getSource()).getFill().toString();
      System.out.println("Clicking on "+color+" image");
   }
}
```

Other Events

- There are KeyEvents that involve the keyboard
- Common operations are key up, key down, and key press which is a completely up and down
- One ability of key events is being able to get the character of the key pressed



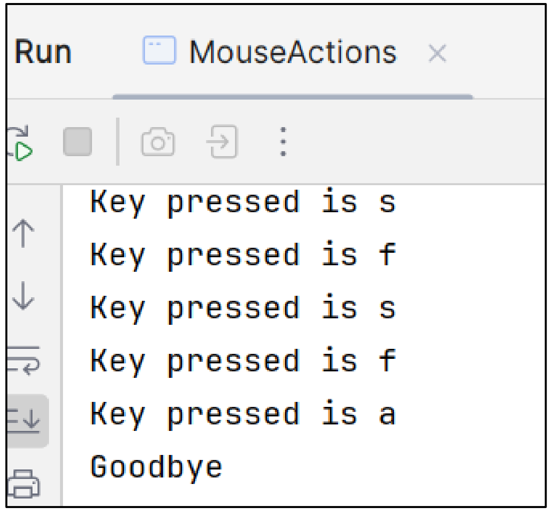
```
    @Override
    public vo
        
        Image
        image
        image
        
        
        
            }
        
        
        Recta
        r1.se
        r1.se
```

```
Key pressed is f
Key pressed is s
Key pressed is f
Key pressed is s
Key pressed is f
Key pressed is s
Key pressed is f
Key pressed is a
```

```
public class MouseActions extends Application {
   @Override
   public void start(Stage stage) throws Exception {

      ImageView imageView = new ImageView();
      imageView.setImage(new Image(new FileInputStream("images/robot.png")));
      imageView.setOnMouseClicked((MouseEvent e) -> {
            double x = e.getX();
            double y = e.getY();
            System.out.println("Clicking on image at ("+x+","+y+")"));
         });


      Rectangle r1 = new Rectangle(200,200, 80,40);
      r1.setFill(Color.RED);
      r1.setOnMouseMoved(this::respond);


      Group root = new Group();
      root.getChildren().addAll(imageView,r1);

      Scene s1 = new Scene(root,400,600);
      s1.setOnKeyTyped((KeyEvent e) -> System.out.println("Key pressed is "+e.getCharacter()));

      stage.setOnCloseRequest((WindowEvent e) -> System.out.println("Goodbye"));
      stage.setScene(s1);
      stage.show();
   }

   public void respond(MouseEvent e){
      String color = ((Rectangle)e.getSource()).getFill().toString();
      System.out.println("Clicking on "+color+" image");
   }
}
```

Other Events

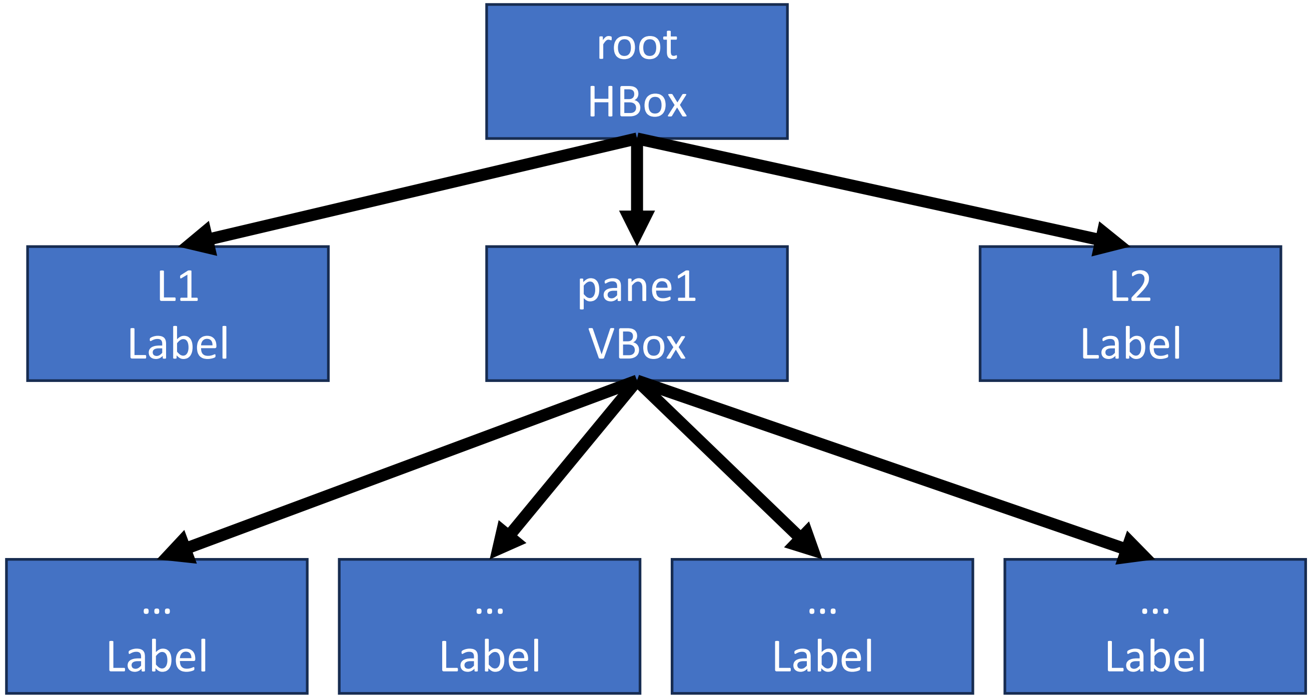
- There are also events that involve the window itself such as moving or closing the window



```
Key pressed is s
Key pressed is f
Key pressed is s
Key pressed is f
Key pressed is a
Goodbye
```

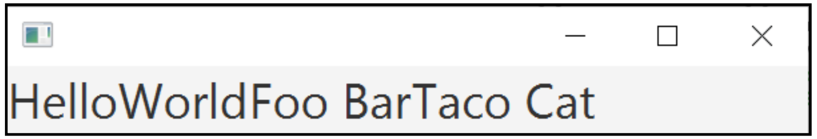
```
public class MouseActions extends Application {
  @Override
  public void start(Stage stage) throws Exception {

    ImageView imageView = new ImageView();
    imageView.setImage(new Image(new FileInputStream("images/robot.png")));
    imageView.setOnMouseClicked((MouseEvent e) -> {
        double x = e.getX();
        double y = e.getY();
        System.out.println("Clicking on image at ("+x+","+y+")");
      });


    Rectangle r1 = new Rectangle(200,200, 80,40);
    r1.setFill(Color.RED);
    r1.setOnMouseMoved(this::respond);


    Group root = new Group();
    root.getChildren().addAll(imageView,r1);

    Scene s1 = new Scene(root,400,600);
    s1.setOnKeyTyped((KeyEvent e) -> System.out.println("Key pressed is "+e.getCharacter()));

    stage.setOnCloseRequest((WindowEvent e) -> System.out.println("Goodbye"));
    stage.setScene(s1);
    stage.show();
  }

  public void respond(MouseEvent e){
    String color = ((Rectangle)e.getSource()).getFill().toString();
    System.out.println("Clicking on "+color+" image");
  }
}
```

Cutouts

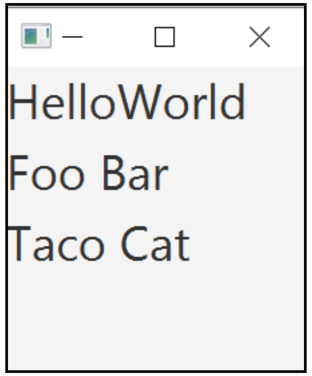

root
HBox
L1
Label
pane1
VBox
L2
Label
…
Label
…
Label
…
Label
…
Label

Layouts

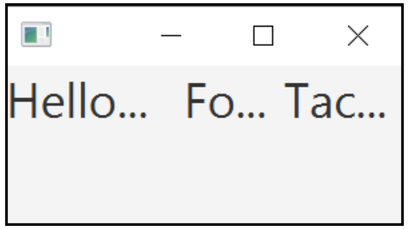
- FlowPane
  - Components flow from left to right
  - If you decrease the width of the window, the components will line wrap to the next line
- Hbox
  - Components are arranged horizontally
- Vbox
  - Components are arranged vertically
- We can also nest layouts within layouts



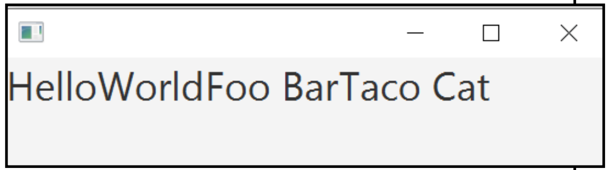
HelloWorldFoo BarTaco Cat



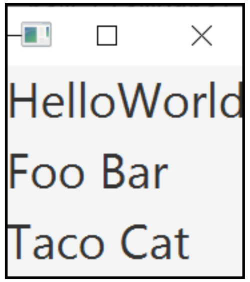
HelloWorld
Foo Bar
Taco Cat



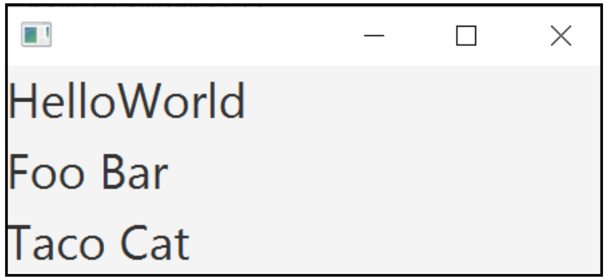
Hello... Fo... Tac...



HelloWorldFoo BarTaco Cat



HelloWorld
Foo Bar
Taco Cat



HelloWorld
Foo Bar
Taco Cat

```
public class LayoutExample extends Application {
  @Override
  public void start(Stage primaryStage) {
    FlowPane layout = new FlowPane();
    //HBox layout = new HBox();
    //VBox layout = new VBox();

    Label l1 = new Label("HelloWorld");
    Label l2 = new Label("Foo Bar");
    Label l3 = new Label("Taco Cat");
    layout.getChildren().addAll(l1,l2,l3);

    l1.setFont(new Font(24));
    l2.setFont(new Font(24));
    l3.setFont(new Font(24));

    Scene s1 = new Scene(layout);
    primaryStage.setScene(s1);
    primaryStage.show();
  }
  public static void main(String[] args) {
    launch(args);
  }
}
```