

CSE201: Monsoon 2020
Advanced Programming

Lecture 16: Event Driven Programming using JavaFX

Vivek Kumar

Computer Science and Engineering

IIT Delhi

vivekk@iiitd.ac.in

Today's Lecture

- Introduction to JavaFX
- Event driven programming
- *Note that JavaFX is vast and we are only covering very basic concepts in this lecture. For your project you might require some advanced features in JavaFX*

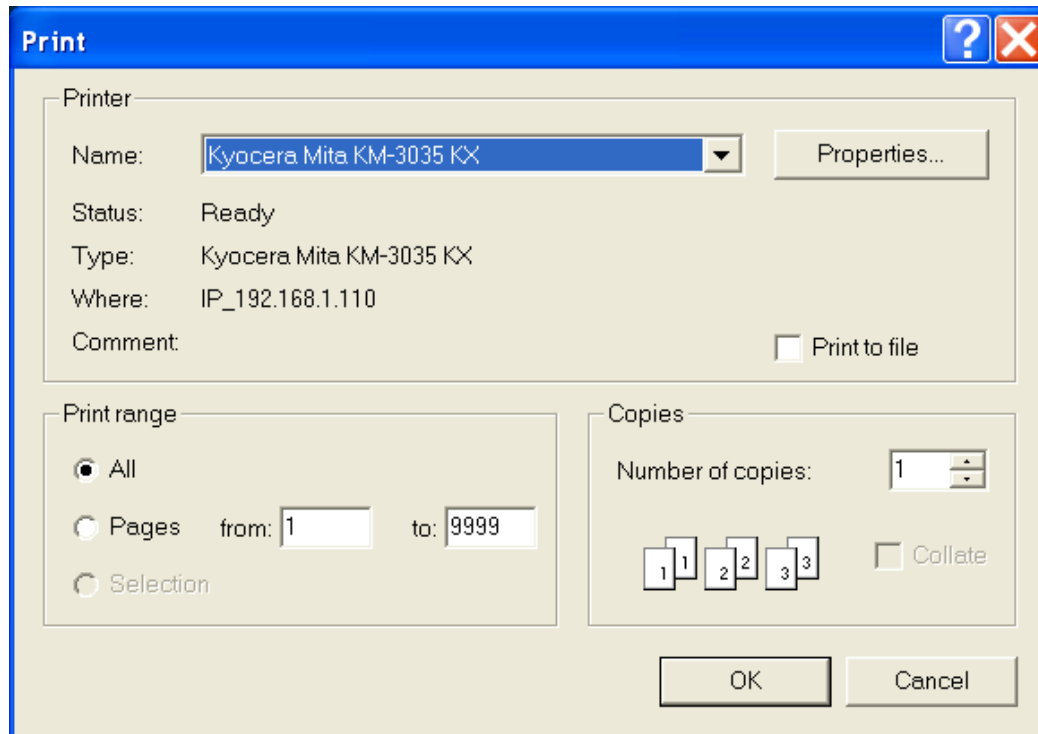
JavaFX slides Acknowledgements: CSE114, Stony Brook University
(<http://www3.cs.stonybrook.edu/~pfodor/courses/cse114.html>)

+ Oracle online documentation

GUI

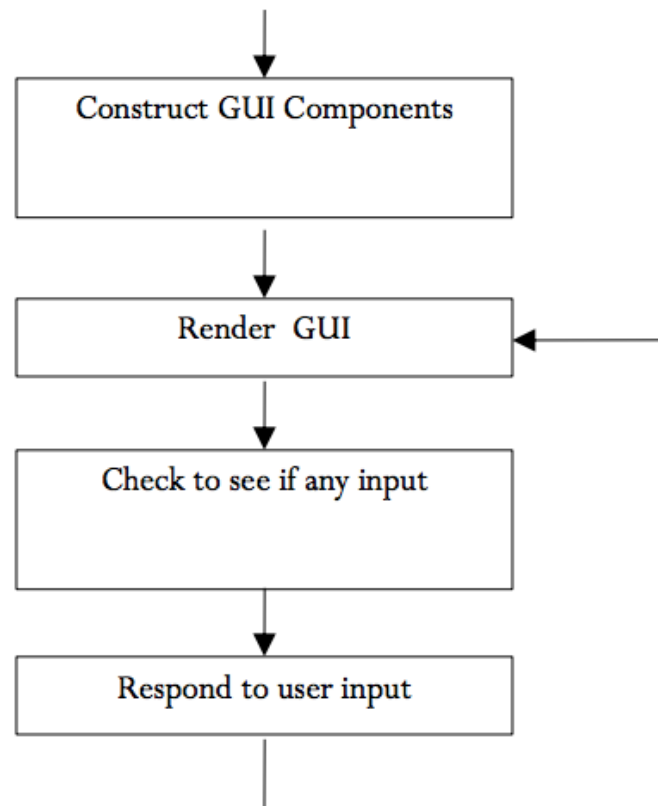
- Graphical User Interface
 - Provides user-friendly human interaction
- History of GUI programming
 - Abstract Window Toolkit
 - Swings
 - JavaFX script
 - JavaFX library

GUI Examples



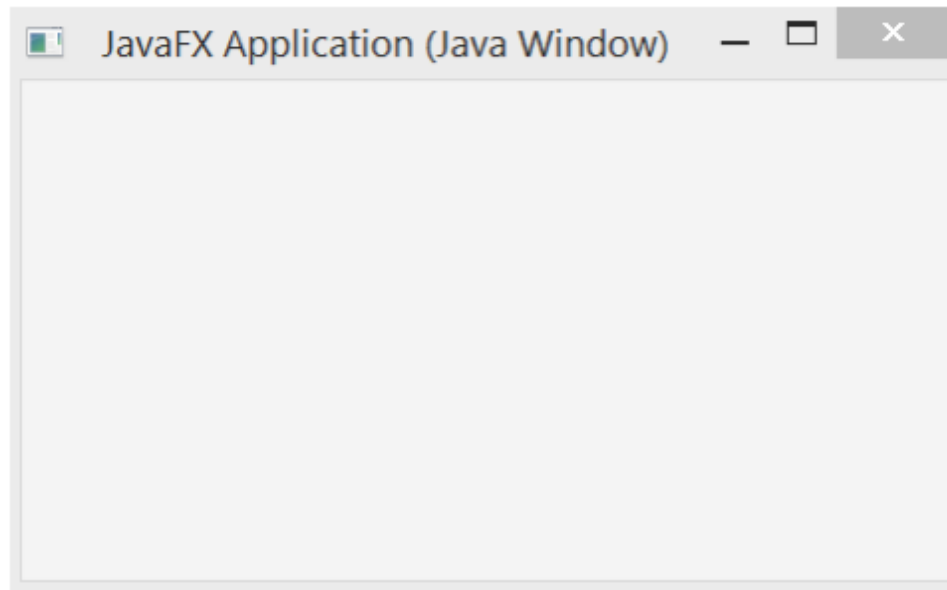
How do GUI Works?

- They loop and respond to events



How does GUI Framework Help?

- Provides ready made visible, interactive, customizable components
 - you wouldn't want to have to code your own window

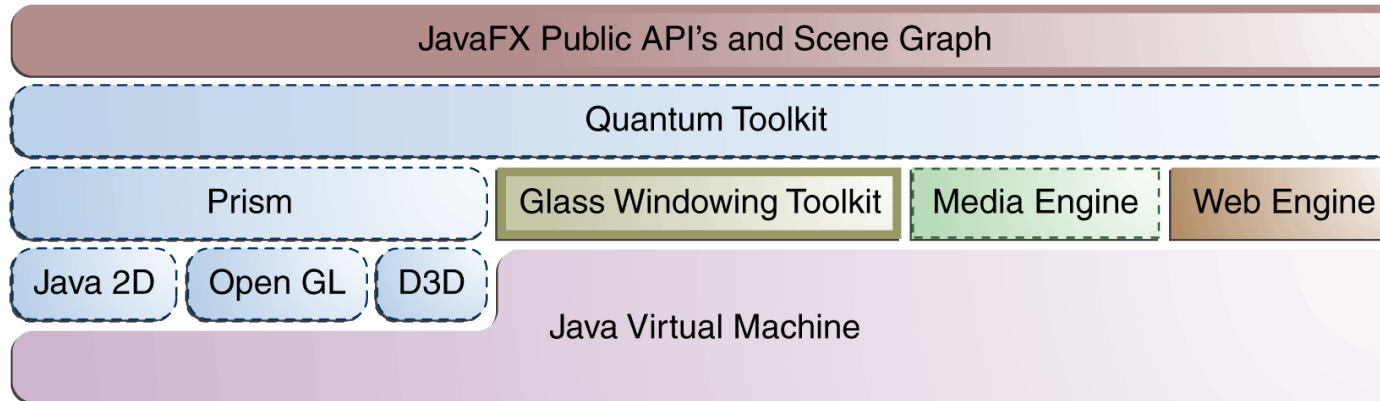


JavaFX: Simplifies Application Development



- JavaFX library simplifies the building of complex graphically rich client applications
- It provides simple APIs to add graphics, media, web content, UI controls etc., in the applications

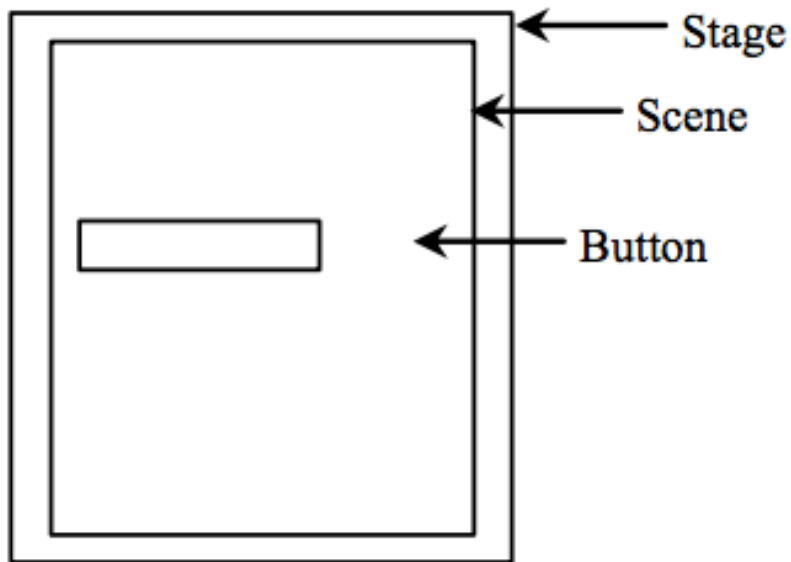
JavaFX Runtime High Level Architecture



● JavaFX Glossary

- **Glass Windowing Toolkit:** Provides native operating services, such as managing the windows, timers, etc.
- **Prism:** Graphics pipeline that can run on hardware and software renderers
- **Quantum Toolkit:** Ties Prism and Glass together and makes them available to the JavaFX APIs

Basic Structure of JavaFX



- Class `javafx.stage.Stage` is the top level JavaFX container
- Class `javafx.scene.Scene` class is the container for all content in a scene graph
- Abstract class `javafx.application.Application` is the entry point for JavaFX applications
 - Executes the user application and processes input events
 - User just need to Override the start method!
- Components can be created/added programmatically

```
Parent p;  
Node n;  
p.getChildren().add(n)
```

JavaFX: Hello World

```
public class HelloWorld extends Application {  
    public static void main(String[] args) {  
        launch(args);  
    }  
  
    //Override the start method in the Application class  
    @Override  
    public void start(Stage primaryStage) {  
        // Set the stage title  
        primaryStage.setTitle("MyJavaFX");  
        // Create a button and place it in the scene  
        Button btn = new Button("Hello World");  
        Scene scene = new Scene(btn, 200, 250);  
        // Place the scene in the stage  
        primaryStage.setScene(scene);  
        // Display the stage  
        primaryStage.show();  
    }  
}
```

- The main class for a JavaFX application extends the `javafx.application.Application` abstract class
 - The `start()` method is the main entry point for all JavaFX applications



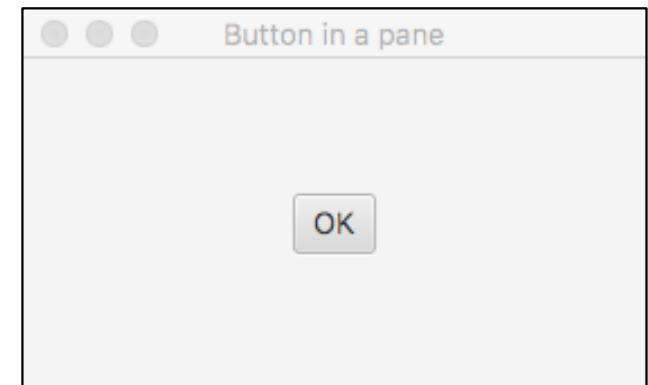
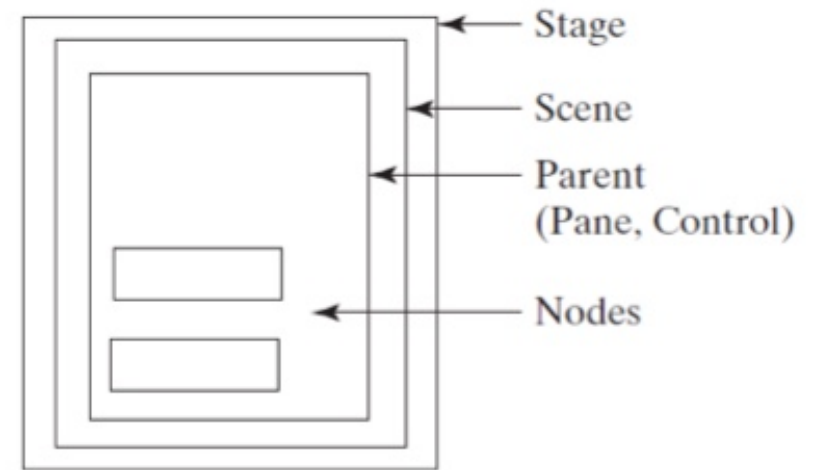
Application's Life Cycle

```
public class HelloWorld extends Application {  
    public static void main(String[] args) {  
        launch(args);  
    }  
  
    //Override the start method in the Application class  
    @Override  
    public void start(Stage primaryStage) {  
        // Set the stage title  
        primaryStage.setTitle("MyJavaFX");  
        // Create a button and place it in the scene  
        Button btn = new Button("Hello World");  
        Scene scene = new Scene(btn, 200, 250);  
        // Place the scene in the stage  
        primaryStage.setScene(scene);  
        // Display the stage  
        primaryStage.show();  
    }  
}
```

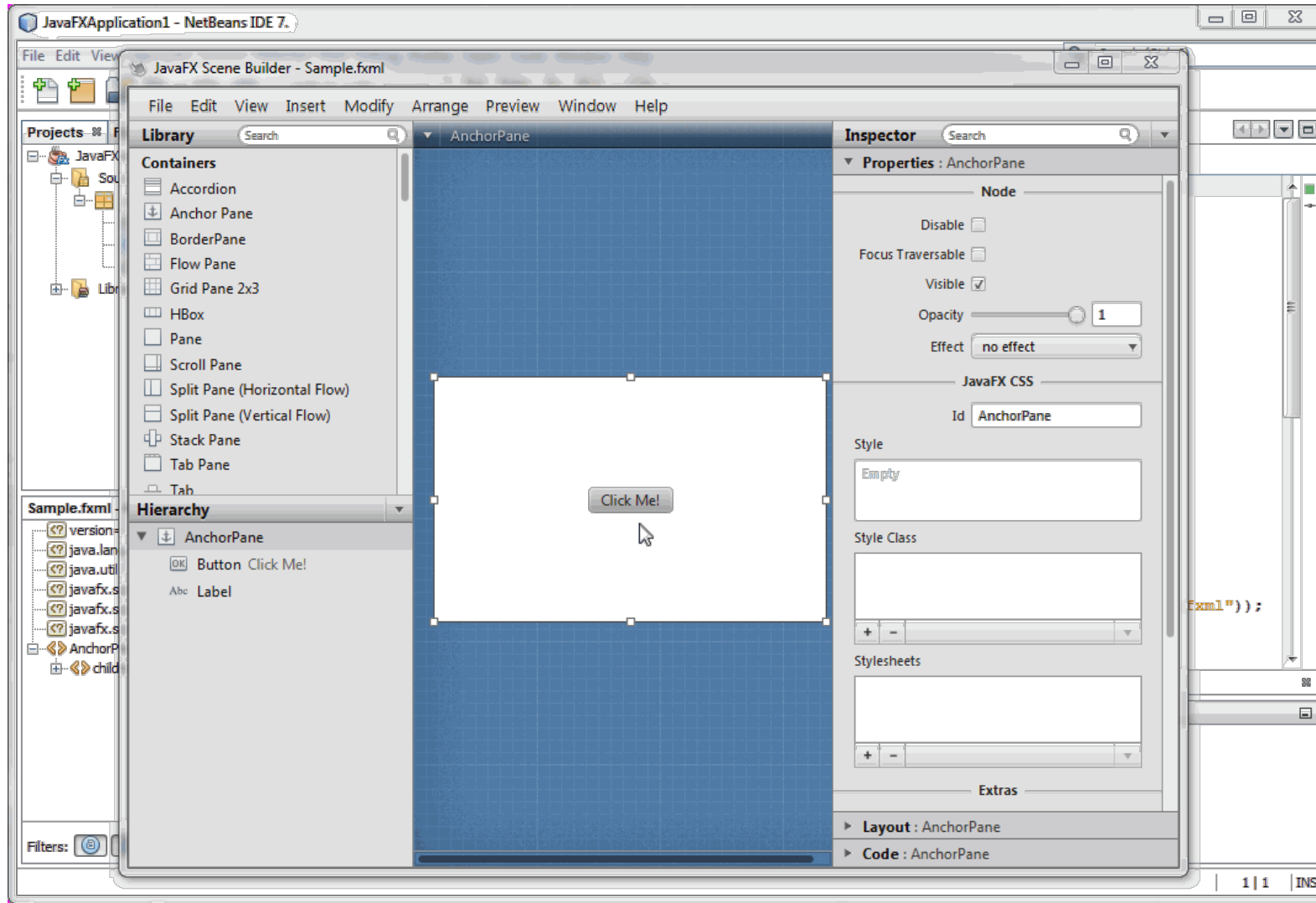
1. Constructs an instance of the specified Application class
2. Calls the concrete method `init()`
3. Calls `start(javafx.stage.Stage)` method (must be Overridden)
4. Waits for the application to finish
5. Calls the concrete method `stop()`

JavaFX: Button in a Pane

```
public class ButtonInPane extends Application {  
    public static void main(String[] args) {  
        launch(args);  
    }  
  
    @Override  
    public void start(Stage primaryStage) {  
        // Set the stage title  
        primaryStage.setTitle("Button in a Pane");  
        // Create a button and place it in the scene  
        Button btn = new Button("OK");  
        // Create a pane and place a button in the pane  
        StackPane pane = new StackPane();  
        pane.getChildren().add(btn);  
        // Create scene with a pane inside it  
        Scene scene = new Scene(pane, 200, 50);  
        // Place the scene in the stage  
        primaryStage.setScene(scene);  
        // Display the stage  
        primaryStage.show();  
    }  
}
```



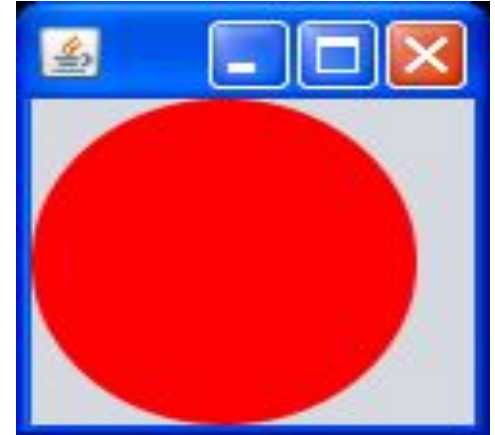
Alternatively: Scene Builder + FXML



- Scene Builder provides a graphical interface for designing and constructing user interfaces
- Scene Builder allows for components to be created, placed, and for many of their properties to be modified
- Saves your layout in an FXML file, which could be read in the Java file to create the GUI

Let's Compare: JavaFX 2.0

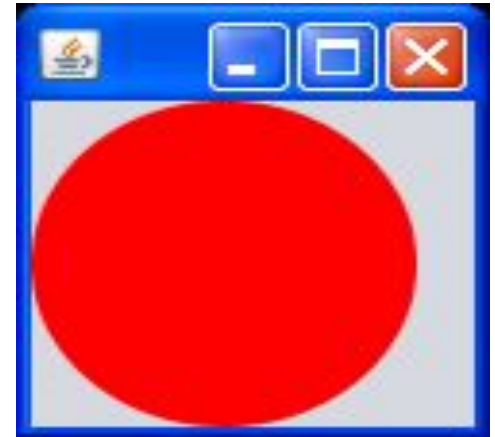
```
public class JavaFXTest extends Application {  
    @Override public void start(Stage stage) {  
        stage.setTitle("FXML Example");  
        Group root = new Group();  
        Circle c1 = new Circle(50.0f, 50.0f, 50.0f, Color.RED);  
  
        root.getChildren().add(c1);  
        stage.setScene(new Scene(root, 100, 100));  
        stage.setVisible(true);  
        stage.show();  
    }  
  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```



Let's Compare: FXML

```
<BorderPane>
  <center>
    <Circle radius="50" centerX="50" centerY="50"/>
  </center>
</BorderPane>
```

```
public class JavaFXTest extends Application {
    @Override public void start(Stage stage) {
        stage.setTitle("FXML Example");
        Parent root = FXMLLoader.load(getClass().getResource("example.fxml"),
            ResourceBundle.getBundle("r.fxml_example"));
        stage.setScene(new Scene(root));
        stage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```



JavaFX UI Controls



Event Programming

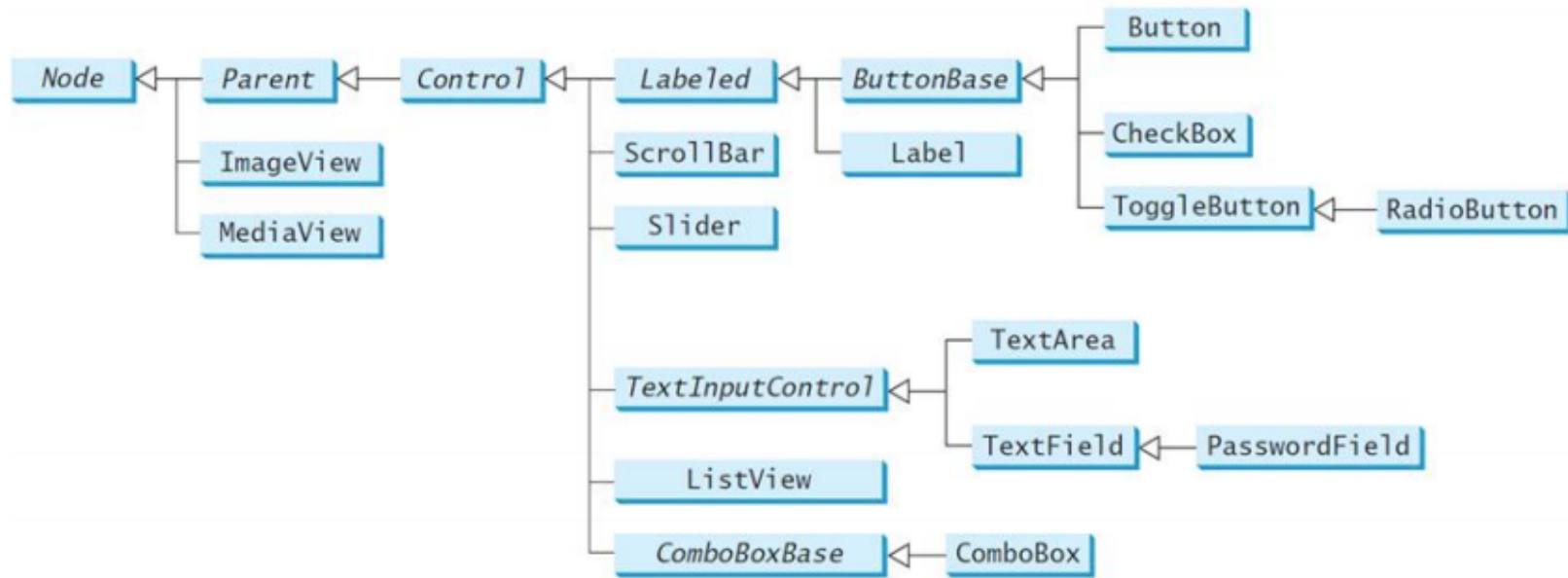
- Procedural programming is executed in procedural/statement order
- In event-driven programming, code is executed upon activation of events
- Operating Systems constantly monitor events
 - Ex: keystrokes, mouse clicks, etc...
- The OS:
 - sorts out these events
 - reports them to the appropriate programs

How to do Event Programming?

- For each control (button, combo box, etc.)
 - define an event handler
 - construct an instance of event handler
 - tell the control who its event handler is
- Event Handler?
 - code with response to event
 - a.k.a. event listener

Java's Event Handling

- An event source is a GUI control
 - JavaFX: Button, ChoiceBox, etc.
- Different types of sources:
 - can detect different types of events
 - can register different types of listeners (handlers)



Event Creation

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) { // entry point
        primaryStage.setTitle("Hello World!");
        Button btn = new Button("Say Hello World");

        btn.setOnAction(new HelloEvent());
        StackPane pane = new StackPane();
        pane.getChildren().add(btn);
        Scene scene = new Scene(pane, 200, 50);
        // Place the scene in the stage
        primaryStage.setScene(scene);
        // Display the stage
        primaryStage.show();
    }
}

.....
```

- When the user interacts with a control (source):
 - an event object is constructed
 - Contain information about the event
 - Like what?
 - location of mouse click
 - event source that was interacted with, etc.
 - the event object is sent to all registered listener objects
 - the listener object (handler) responds as you defined it to

Event Listeners

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) { // entry point
        primaryStage.setTitle("Hello World!");
        Button btn = new Button("Say Hello World");

        btn.setOnAction(new HelloEvent());
        StackPane pane = new StackPane();
        pane.getChildren().add(btn);
        Scene scene = new Scene(pane, 200, 50);
        // Place the scene in the stage
        primaryStage.setScene(scene);
        // Display the stage
        primaryStage.show();
    }
}

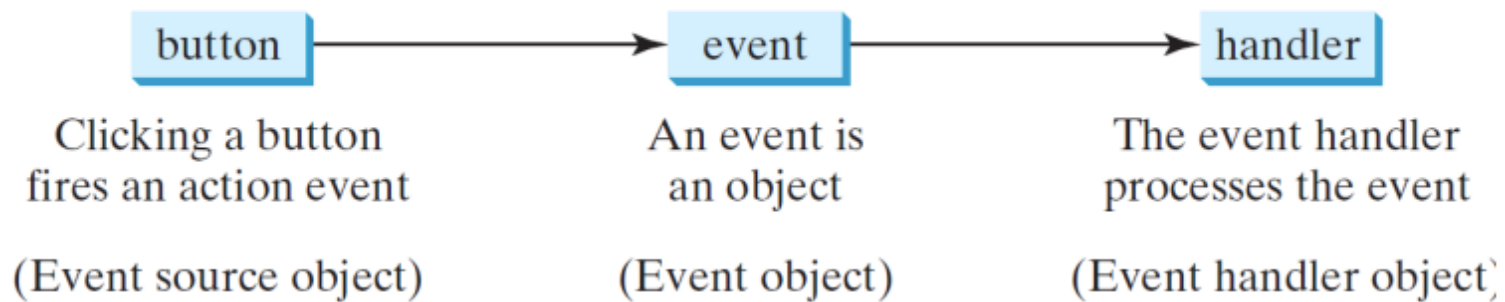
class HelloEvent implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Hello World!");
    }
}
```

© Vivek Kumar

- Event listeners (event handler)
 - Defined by you, the application programmer
 - you customize the response
 - How?
 - Inheritance & Polymorphism
 - You define your own listener class
 - implement the appropriate interface
 - define responses in all necessary methods

Summary: How to Handle GUI Events

- Source object: button
 - An event is generated by external user actions such as mouse movements, mouse clicks, or keystrokes
- An event can be defined as a type of signal to the program that something has happened
- Listener object contains a method for processing the event.



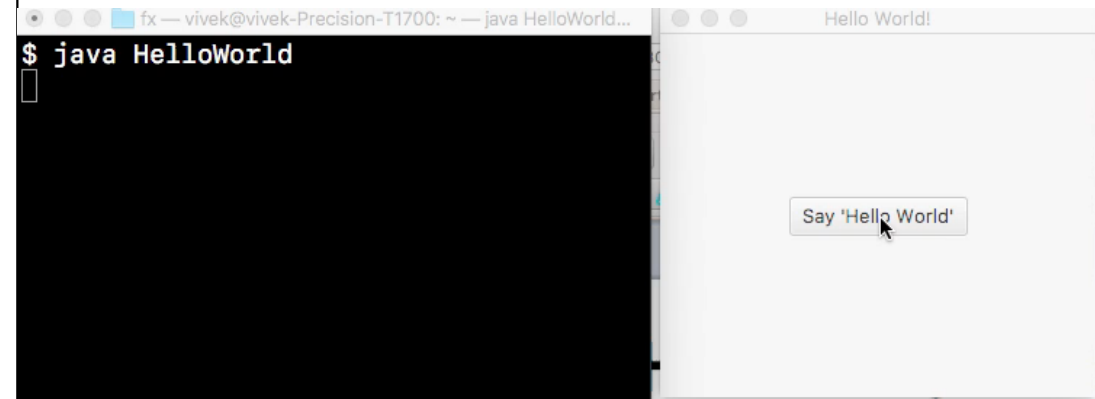
Working of Our Hello World GUI

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) { // entry point
        primaryStage.setTitle("Hello World!");
        Button btn = new Button("Say Hello World");

        btn.setOnAction(new HelloEvent());
        StackPane pane = new StackPane();
        pane.getChildren().add(btn);
        Scene scene = new Scene(pane, 200, 50);
        // Place the scene in the stage
        primaryStage.setScene(scene);
        // Display the stage
        primaryStage.show();
    }
}

class HelloEvent implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Hello World!");
    }
}
```



Productivity in Event Programming

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) { // entry point
        primaryStage.setTitle("Hello World!");
        Button btn = new Button("Say Hello World");

        btn.setOnAction(new HelloEvent());
        StackPane pane = new StackPane();
        pane.getChildren().add(btn);
        Scene scene = new Scene(pane, 200, 50);
        // Place the scene in the stage
        primaryStage.setScene(scene);
        // Display the stage
        primaryStage.show();
    }
}

class HelloEvent implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Hello World!");
    }
}
```

- Can we write this code in much better way?

Productivity in Event Programming (1/3)

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) { // entry point
        primaryStage.setTitle("Hello World!");
        Button btn = new Button("Say Hello World");

        btn.setOnAction(new HelloEvent());
        StackPane pane = new StackPane();
        pane.getChildren().add(btn);
        Scene scene = new Scene(pane, 200, 50);
        // Place the scene in the stage
        primaryStage.setScene(scene);
        // Display the stage
        primaryStage.show();
    }
    class HelloEvent implements EventHandler<ActionEvent> {
        @Override
        public void handle(ActionEvent event) {
            System.out.println("Hello World!");
        }
    }
}
```

- Using **inner** classes for creating listener objects
- Description
 - Class defined in scope of another class
- Property
 - Can directly access **all** variables & methods of enclosing class (including private fields & methods)
- Why inner class?
 - Logical grouping of functionality
 - Increases encapsulation
 - Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world
 - More readability and maintainable code

Productivity in Event Programming (2/3)

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) { // entry point
        primaryStage.setTitle("Hello World!");
        Button btn = new Button("Say Hello World");

        btn.setOnAction(new EventHandler<ActionEvent>() {

            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });

        StackPane pane = new StackPane();
        pane.getChildren().add(btn);
        Scene scene = new Scene(pane, 200, 50);
        // Place the scene in the stage
        primaryStage.setScene(scene);
        // Display the stage
        primaryStage.show();
    }
}
```

- Using **anonymous** inner classes for creating listener objects
 - It combines declaring an inner class and creating an instance of the class in one step
 - An anonymous inner class must always extend a superclass or implement an interface, but it cannot have an explicit extends or implements clause
 - An anonymous inner class must implement all the abstract methods in the superclass or in the interface
 - An anonymous inner class always uses the no-arg constructor from its superclass to create an instance

Productivity in Event Programming (3/3)

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World");
        Button btn = new Button("Say Hello World");

        btn.setOnAction(e -> {
            System.out.println("Hello World!");
        });

        StackPane pane = new StackPane();
        pane.getChildren().add(btn);
        Scene scene = new Scene(pane, 200, 200);
        // Place the scene in the stage
        primaryStage.setScene(scene);
        // Display the stage
        primaryStage.show();
    }
}
```

BUT, What is a Lambda ???

take a detour...

- **Using Java 8 lambda expressions to simplify event handling**
 - Lambda expressions can be viewed as an anonymous method with a concise syntax
 - The statements in the lambda expression is all for that method
 - If it contains multiple methods, the compiler will not be able to compile the lambda expression
 - So, for the compiler to understand lambda expressions, the interface must contain exactly one method

Let's

Collection Elements (1/4)

```
public class Test {  
    Map<String, Integer> items =  
        new HashMap<String, Integer>();  
  
    public void addElements() { ..... }  
  
    public void print() {  
        for(Map.Entry<String, Integer> entry  
            : items.entrySet()) {  
            System.out.println(entry.getKey()  
                + ", " + entry.getValue());  
        }  
    }  
}
```

- Till now we know only this way to iterate over a collection (e.g., Map)
- Drawback
 - Slightly inconvenient coding

Collection Elements (2/4)

```
public class Test {  
    Map<String, Integer> items =  
        new HashMap<String, Integer>();  
  
    public void addElements() { ..... }  
  
    public void print() {  
        items.forEach( (k, v) -> {  
            System.out.println(k + ", " + v);  
        });  
    }  
}
```

- Java 8 introduces **forEach** statement to ease iterating over the collection elements
- With **lambda expressions** in Java 8 this code becomes very compact now!

Collection Elements (3/4)

```
public class Test {  
    Map<String, Integer> items =  
        new HashMap<String, Integer>();  
  
    public void addElements() { ..... }  
  
    public void print() {  
        items.forEach( (k, v) -> {  
            if("ABC".equals(k)) {  
                System.out.println("Hello ABC!");  
            }  
            System.out.println(k + ", " + v);  
        });  
    }  
}
```

- With lambda expressions in Java 8 this code becomes very compact now!
- You can do some more stuff inside that lambda function!

Collection Elements (4/4)

```
public class Test {  
    Map<String, Integer> items =  
        new HashMap<String, Integer>();  
  
    public void addElements() { ..... }  
  
    public void print() {  
        items.forEach( (String k, Integer v) -> {  
            if("ABC".equals(k)) {  
                System.out.println("Hello ABC!");  
            }  
            System.out.println(k + ", " + v);  
        });  
    }  
}
```

- With lambda expressions in Java 8 this code becomes very compact now!
- You can do some more stuff inside that lambda function!
- You can even declare type of variables in lambda function

Productivity in Event Programming (3/3)

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) { // entry point
        primaryStage.setTitle("Hello World!");
        Button btn = new Button("Say Hello World");

        btn.setOnAction(e -> {
            System.out.println("Hello World!");
        });

        StackPane pane = new StackPane();
        pane.getChildren().add(btn);
        Scene scene = new Scene(pane, 200, 50);
        // Place the scene in the stage
        primaryStage.setScene(scene);
        // Display the stage
        primaryStage.show();
    }
}
```

Now, let's come back to JavaFX...

- Using Java 8 **lambda** expressions to simplify event handling
 - Lambda expressions can be viewed as an anonymous method with a concise syntax
 - The statements in the lambda expressions all on that method
 - If it contains multiple methods, the compiler will not be able to compile the lambda expression
 - So, for the compiler to understand lambda expressions, the interface must contain exactly one method

Productivity in Event Programming (3/3)

```
public class HelloWorld extends Application {  
    public static void main(String[] args) {  
        launch(args);  
    }  
  
    @Override  
    public void start(Stage primaryStage) { // entry point  
        primaryStage.setTitle("Hello World!");  
        Button btn = new Button("Say Hello World");  
  
        btn.setOnAction(e -> {  
            System.out.println("Hello World!");  
        });  
        StackPane pane = new StackPane();  
        pane.getChildren().add(btn);  
        Scene scene = new Scene(pane, 200, 50);  
        // Place the scene in the stage  
        primaryStage.setScene(scene);  
        // Display the stage  
        primaryStage.show();  
    }  
}
```

- Using Java 8 **lambda** expressions to simplify event handling
 - Lambda expressions can be viewed as an anonymous method with a concise syntax
 - The statements in the lambda expression is all for that method
 - If it contains multiple methods, the compiler will not be able to compile the lambda expression
 - So, for the compiler to understand lambda expressions, the interface must contain exactly one method

Next Lecture

- Introduction to process and threads