

# Java™ Education & Technology Services

## Introduction to GUI Using Java Programming

# Course Content



Java™ Education  
and Technology Services



Invest In Yourself,  
**Develop** Your Career



# Course outline

- [Lesson 1: Event Handling in Java](#)
- [Lesson 2: Introduction to GUI Programming](#)
- [Lesson 3: Building UI Using Java FX](#)
- [Lesson 4: Introducing to JavaFX Event Handling, and Layout Managers](#)
- [References.](#)

# Lesson 1

## Event Handling

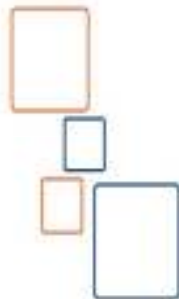
### in Java



Java™ Education  
and Technology Services



Invest In Yourself,  
Develop Your Career



## 1.1 What is an Event?

- Change in the state of an object is known as event i.e. event describes the change in state of source.
- Events are generated as result of user interaction with the graphical user interface components.
- For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page are the activities that causes an event to happen.

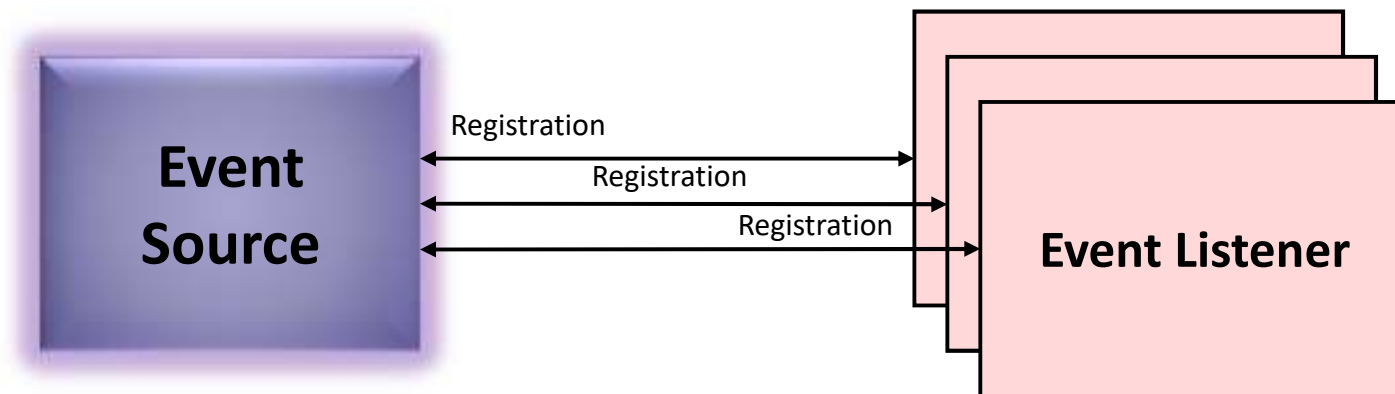
## 1.2 Types of Events

- The events can be classified into two main categories:
  - **Foreground Events:** Those events which require the direct interaction of user. They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page etc.
  - **Background Events:** Those events does not require the user action. Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

## 1.3 What is Event Handling?

- Event Handling is the mechanism that controls the event and decides what should happen if an event occurs.
- This mechanism have the code which is known as *event handler* that is executed when an event occurs.
- Java uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.
- The Delegation Event Model has the following key participants namely:
  - **Source** - The source is an object on which event occurs, i.e. it is the object that fired the event
  - **Listener** - It is also known as event handler, i.e. it is the object that has the code to execute when notified that the event has been fired.

## 1.3 What is Event Handling?



- An Event Source may have one or more Event Listeners.
- The advantage of this model is:
  - The Event Object is only forwarded to the listeners that have registered with the source.





## 1.4 Components of Event Handling

- Event handling has three main components:
  - **Events** : An event is a change in state of an object.
  - **Events Source** : Event source is an object that generates an event.
  - **Listeners** : A listener is an object that listens to the event. A listener gets notified when an event occurs.

## 1.5 How Events Are Handled?

- A source generates an Event and send it to one or more listeners registered with the source.
- Once event is received by the listener, it processes the event and then return.
- Events are supported by a number of Java packages, like `java.awt.event` for AWT and most of SWING components, `javax.swing.event`, for extra events which added with swing components, and `javafx.event` for Java FX components
- The following piece of code is a simplified example of the process:

```
Button b = new Button("Ok"); //Constructing a Component (Source)
MyListener myL = new MyListener(); //Constructing a Listener
b.addActionListener(myL); //Registering Listener with Source
```

## 1.5 How Events Are Handled?

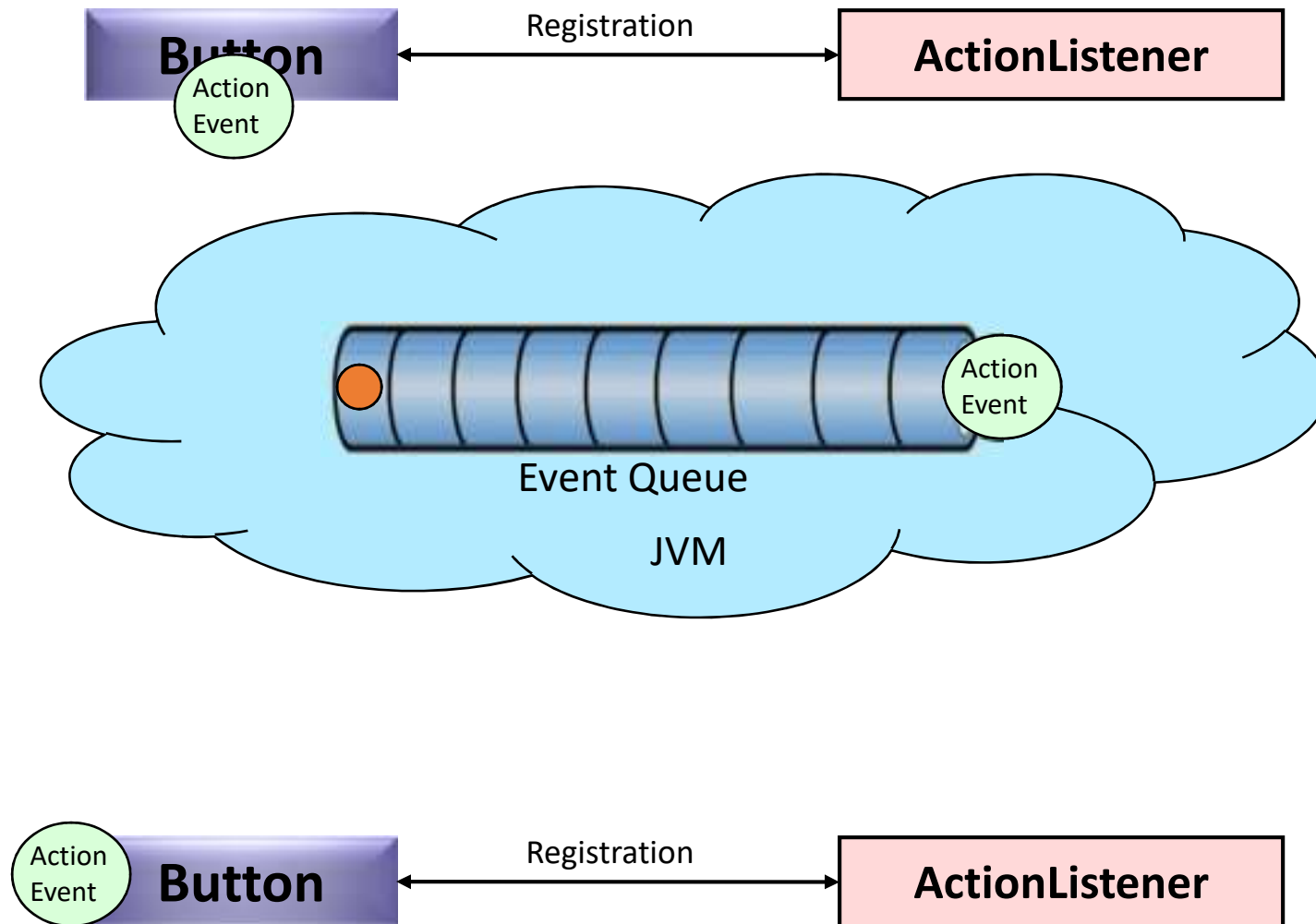
1. Write the listener code:

```
class MyListener implements ActionListener {  
    public void actionPerformed(ActionEvent e) {  
        // handle the event here  
        // (i.e. what you want to do  
        // when the Ok button is clicked)  
    }  
}
```

2. Create the source, and register the listener with it:

```
public class MyApp extends Frame{  
    public MyApp() {  
        Button b = new Button("Ok");  
        MyListener myL = new MyListener();  
        b.addActionListener(myL);  
    }  
}
```

## 1.5 How Events Are Handled?



## 1.5 How Events Are Handled ?

- When examining the previous button example:
  - The button (source) is created.
  - The listener is registered with the button,
  - The user clicks on the Ok button.
    - An **ActionEvent** object is created and placed on the *event queue*.
- The **Event Dispatching Thread** processes the events in the queue.
  - The Event Dispatching Thread checks with the button to see if any listeners have registered themselves.
- The Event Dispatcher then invokes the **actionPerformed()** method, and passes the **ActionEvent** object itself to the method.

## 1.6 Event Handling Example

```
import java.awt.*;      import java.awt.event.*;      import javax.swing.*;

public class TestEventApp extends JFrame{
    JLabel textLabel=new JLabel("Do event using mouse to see handling");
    JLabel handleLabel = new JLabel("-----");
    public TestEventApp(String title){
        super(title);
        this.setLayout(new FlowLayout());
        this.add(textLabel);          this.add(handleLabel);
        textLabel.setOpaque(true);   handleLabel.setOpaque(true);
        textLabel.setFont(new Font("MonoSpaced", Font.BOLD, 20));
        handleLabel.setFont(new Font("MonoSpaced", Font.BOLD, 20));
        textLabel.setBackground(Color.pink);
        handleLabel.setBackground(Color.LIGHT_GRAY);
        this.setDefaultCloseOperation(this.EXIT_ON_CLOSE);
        this.addMouseListener(new MyMouseListener());
        this.addMouseMotionListener(new MyMouseMotionListener());
    }
}
```

## 1.6 Event Handling Example

```
public static void main(String [] args) {  
    TestEventApp app = new TestEventApp("Event Handling Example");  
        app.setSize(600,600);  
        app.setVisible(true);  
}  
  
class MyMouseListener implements MouseListener  
{  
    public void mousePressed(MouseEvent e)  
    {  
        handleLabel.setForeground(Color.blue);  
        handleLabel.setText("Mouse is Pressed ");  
    }  
    public void mouseReleased(MouseEvent e)  
    {  
        handleLabel.setForeground(Color.red);  
        handleLabel.setText("Mouse is Released ");  
    }  
}
```

## 1.6 Event Handling Example

```
public void mouseClicked(MouseEvent e)
{
    handleLabel.setForeground(Color.yellow);
    handleLabel.setText("Mouse is Clicked ");
}

public void mouseEntered(MouseEvent e)
{
    handleLabel.setForeground(Color.GRAY);
    handleLabel.setText("Mouse is Entered ");
}

public void mouseExited(MouseEvent e)
{
    handleLabel.setForeground(Color.magenta);
    handleLabel.setText("Mouse is Exited ");
}

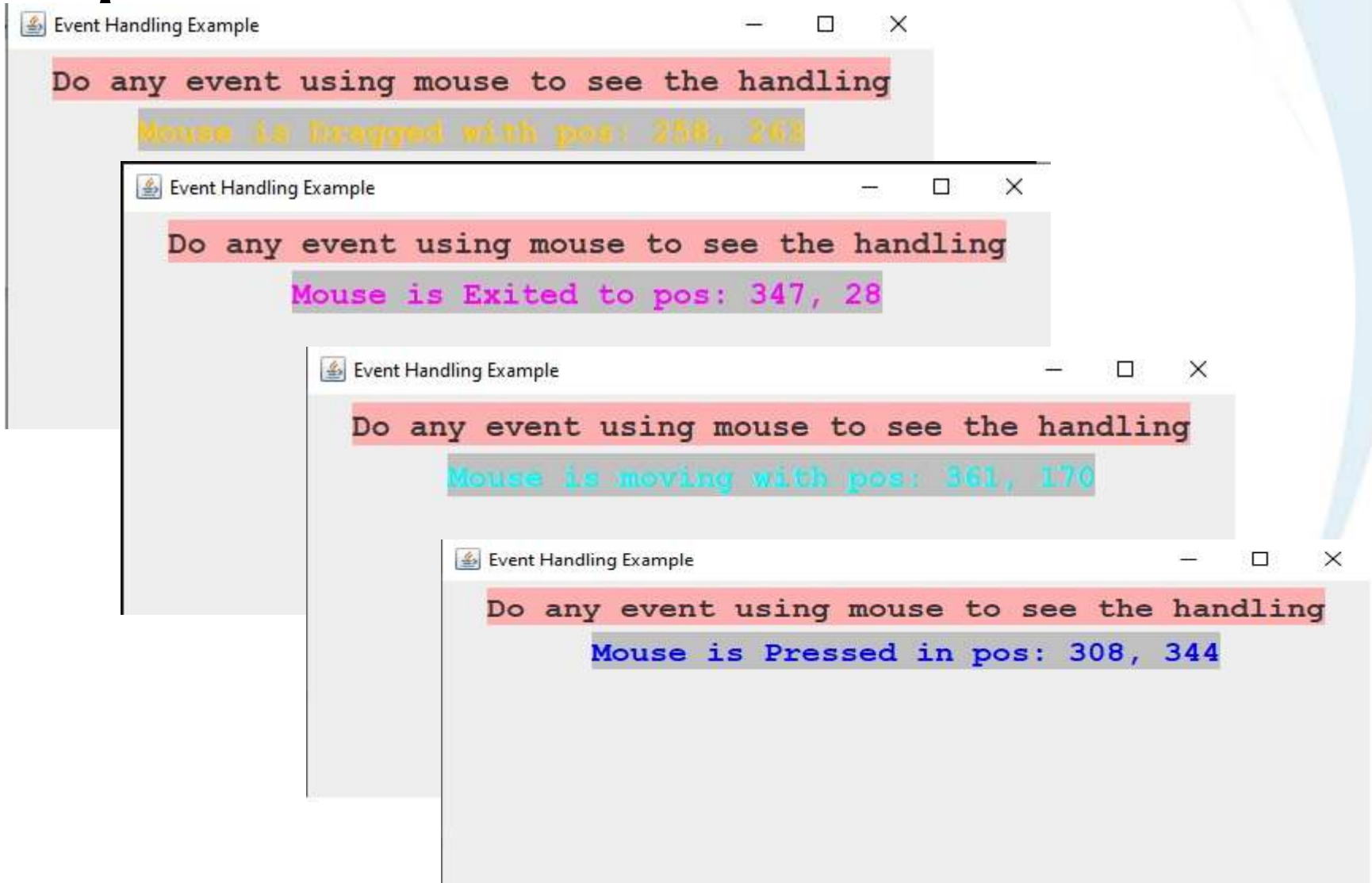
} // end of the first inner class
```



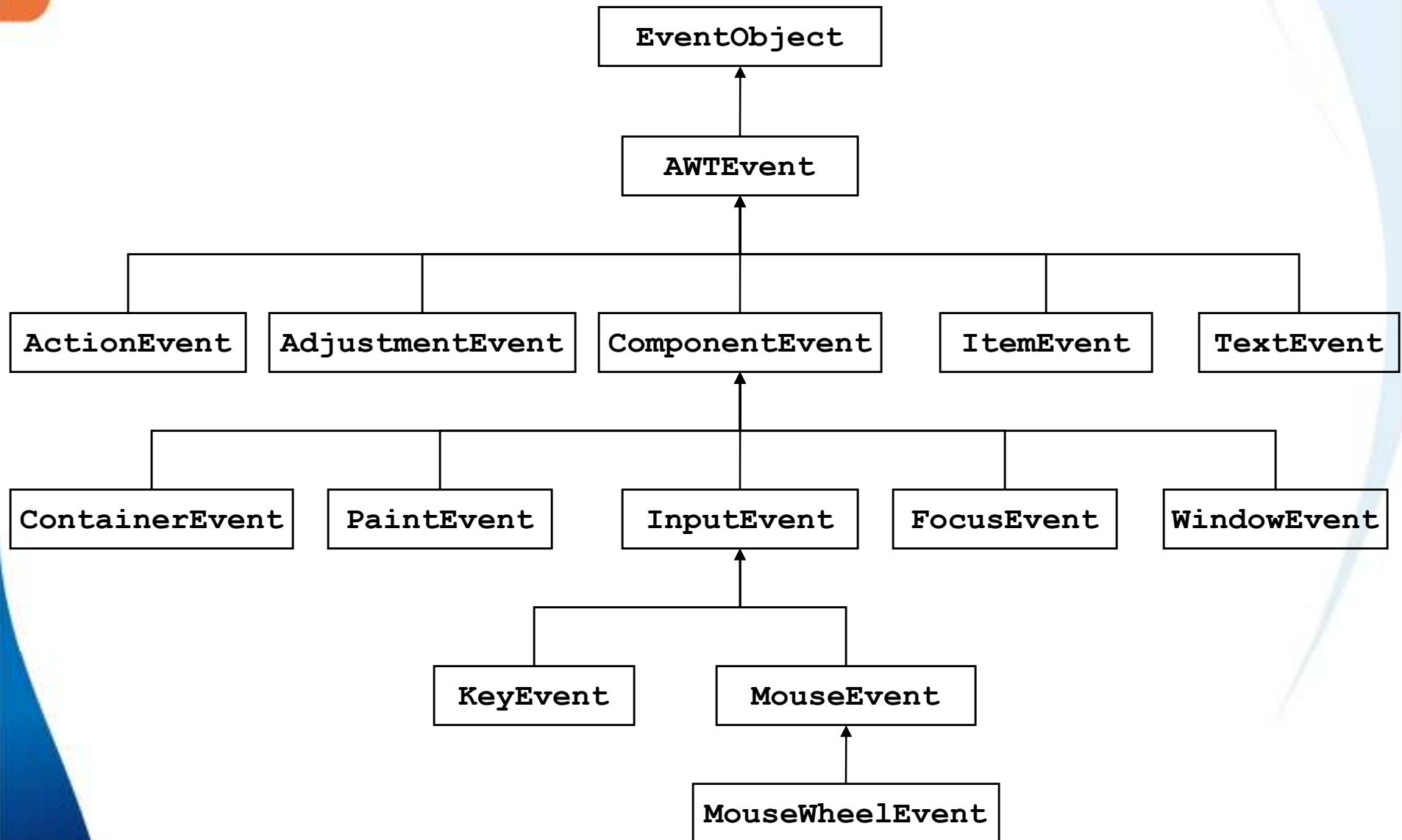
## 1.6 Event Handling Example

```
class MyMouseMotionListener implements MouseMotionListener
{
    public void mouseMoved(MouseEvent e)
    {
        handleLabel.setForeground(new Color(0.0f,1.0f,1.0f));
        handleLabel.setText("Mouse is moving ");
    }
    public void mouseDragged(MouseEvent e)
    {
        handleLabel.setForeground(Color.ORANGE);
        handleLabel.setText("Mouse is Dragged ");
    }
} // end of second inner class
}
```

# Output



# 1.7 Events Classes Hierarchy



## 1.7.1 Important Event Classes and Interfaces

Event Classes	Description	Listener Interface
<b>ActionEvent</b>	generated when a Button is pressed, MenuItem is selected, list-item is double clicked, key stroke the ENTER key inside TextFiled	<b>ActionListener</b>
<b>MouseEvent</b>	generated when mouse is dragged, moved, clicked, pressed or released and also when it enters or exit a component	<b>MouseListener</b>
<b>KeyEvent</b>	generated when input is received from keyboard	<b>KeyListener</b>
<b>ItemEvent</b>	generated when check-box or list item is clicked (select or deslect)	<b>ItemListener</b>
<b>TextEvent</b>	generated when value of TextArea or TextField is changed	<b>TextListener</b>
<b>MouseWheelEvent</b>	generated when mouse wheel is moved	<b>MouseWheelListener</b>

## 1.7.1 Important Event Classes and Interfaces

Event Classes	Description	Listener Interface
<b>WindowEvent</b>	generated when window is activated, deactivated, de-iconified, iconified, opened, opening or closed	<b>WindowListener</b>
<b>ComponentEvent</b>	generated when component is hidden, moved, resized or shown	<b>ComponentListener</b>
<b>ContainerEvent</b>	generated when component is added or removed from container	<b>ContainerListener</b>
<b>AdjustmentEvent</b>	generated when scroll bar is manipulated	<b>AdjustmentListener</b>
<b>FocusEvent</b>	generated when component gains or loses keyboard focus	<b>FocusListener</b>

## 1.7.2 Listeners' Methods

Event	Listener Interface(s)	Method(s)
<b>ActionEvent</b>	<b>ActionListener</b>	actionPerformed (ActionEvent e)
<b>AdjustmentEvent</b>	<b>AdjustmentListener</b>	adjustmentValueChanged (AdjustmentEvent e)
<b>ComponentEvent</b>	<b>ComponentListener</b>	componentHidden (ComponentEvent e) componentShown (ComponentEvent e) componentMoved (ComponentEvent e) componentResized (ComponentEvent e)
<b>ItemEvent</b>	<b>ItemListener</b>	itemStateChanged (ItemEvent e)
<b>TextEvent</b>	<b>TextListener</b>	textValueChanged (TextEvent e)
<b>ContainerEvent</b>	<b>ContainerListener</b>	componentAdded (ContainerEvent e) componentRemoved (ContainerEvent e)

## 1.7.2 Listeners methods

Event	Listener Interface(s)	Method(s)
<b>FocusEvent</b>	<b>FocusListener</b>	<b>focusGained (FocusEvent e)</b> <b>focusLost (FocusEvent e)</b>
<b>WindowEvent</b>	<b>WindowListener</b>	<b>windowClosed (WindowEvent e)</b> <b>windowClosing (WindowEvent e)</b> <b>windowOpened (WindowEvent e)</b> <b>windowActivated (WindowEvent e)</b> <b>windowDeactivated (WindowEvent e)</b> <b>windowIconified (WindowEvent e)</b> <b>windowDeiconfied (WindowEvent e)</b>

## 1.7.2 Listeners methods

Event	Listener Interface(s)	Method(s)
<b>KeyEvent</b>	<b>KeyListener</b>	<b>keyPressed (KeyEvent e)</b> <b>keyReleased (KeyEvent e)</b> <b>keyTyped (KeyEvent e)</b>
<b>MouseEvent</b>	<b>MouseListener</b>	<b>mousePressed (MouseEvent e)</b> <b>mouseReleased (MouseEvent e)</b> <b>mouseClicked (MouseEvent e)</b> <b>mouseEntered (MouseEvent e)</b> <b>mouseExited (MouseEvent e)</b>
	<b>MouseMotionListener</b>	<b>mouseMoved (MouseEvent e)</b> <b>mouseDragged (MouseEvent e)</b>
<b>MouseWheelEvent</b>	<b>MouseWheelListener</b>	<b>mouseWheelMoved (MouseWheelEvent e)</b>



## 1.8 Adapters

- When working with listener interfaces that have more than one method, it is a tedious task to override all the methods of the interface when we only need to implement some of them.
- Therefore, for each listener interface with multiple methods, there is a special Adapter class that has implemented the interface and overridden all the methods with empty bodies.

## 1.8 Adapters

Adapter Class	Listener Interface
<b>WindowAdapter</b>	<b>WindowListener</b>
<b>ComponentAdapter</b>	<b>ComponentListener</b>
<b>ContainerAdapter</b>	<b>ContainerListener</b>
<b>KeyAdapter</b>	<b>KeyListener</b>
<b>MouseAdapter</b>	<b>MouseListener</b>
<b>MouseMotionAdapter</b>	<b>MouseMotionListener</b>
<b>FocusAdapter</b>	<b>FocusListener</b>

## 1.9 Another Event Handling Example

```
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
public class SampleUI extends JFrame {  
    public SampleUI() {  
        this.setLayout(new FlowLayout());  
        JTextArea ta = new JTextArea(20, 50);  
        JScrollPane scroll = new JScrollPane(ta);  
        JTextField tf = new JTextField(40);  
        JButton okButton = new JButton("Send");  
  
        okButton.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent ae) {  
                ta.append(tf.getText() + "\n");  
                tf.setText("");  
            }  
        });  
    }  
}
```

## 1.9 Another Event Handling Example

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class SampleUI extends JFrame {
    public SampleUI() {
        this.setLayout(new FlowLayout());
        JTextArea ta = new JTextArea(20, 50);
        JScrollPane scroll = new JScrollPane(ta);
        JTextField tf = new JTextField(40);
        JButton okButton = new JButton("Send");

        okButton.addActionListener((ae) -> {
            ta.append(tf.getText() + "\n");
            tf.setText("");
        });
    }
}
```

## 1.9 Another Event Handling Example

```
    add(scroll);
    add(tf);
    add(okButton);
    this.setDefaultCloseOperation(this.EXIT_ON_CLOSE);
}
public static void main(String args[])
{
    SampleUI ui=new SampleUI();
    ui.setSize(600, 400);
    ui.setResizable(false);
    ui.setVisible(true);
}
}
```

## 1.9 Another Event Handling Example



# Lab Exercise



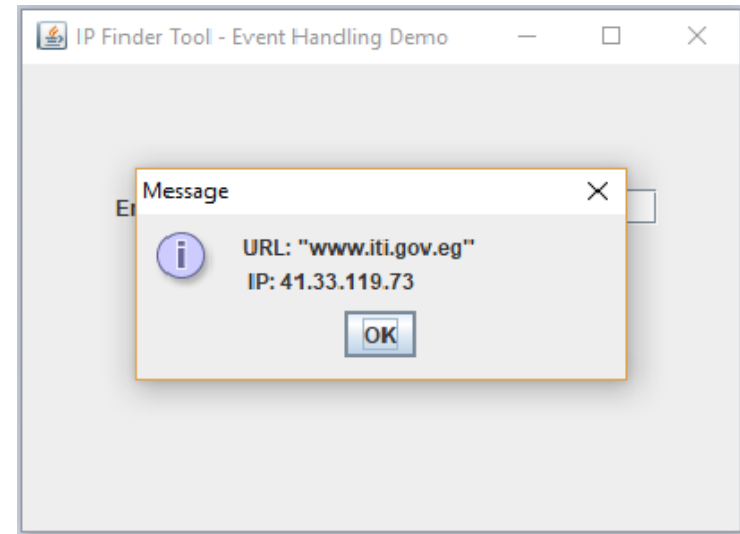
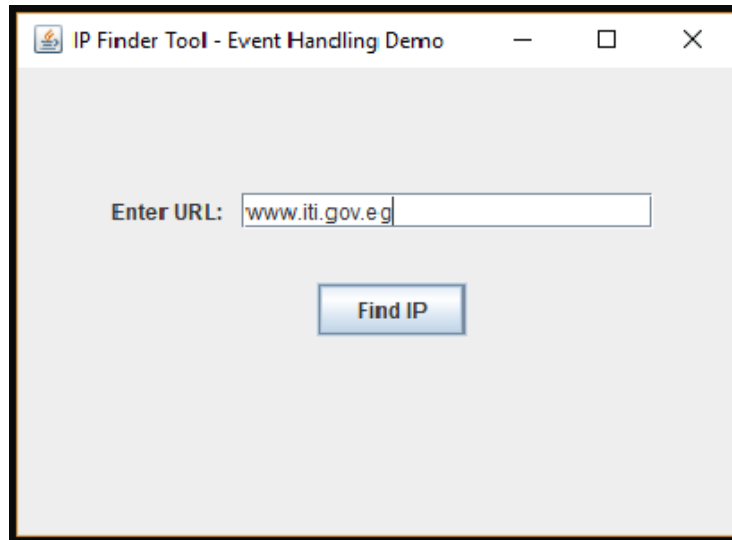
Java™ Education  
and Technology Services



Invest In Yourself,  
**Develop** Your Career

# 1. GUI IP Finder

- Write a Java GUI desktop application to find the IP address of given URL in a text field.

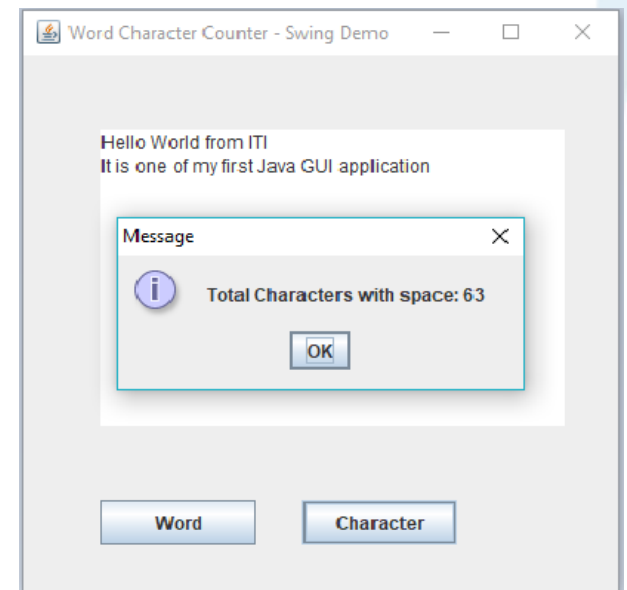
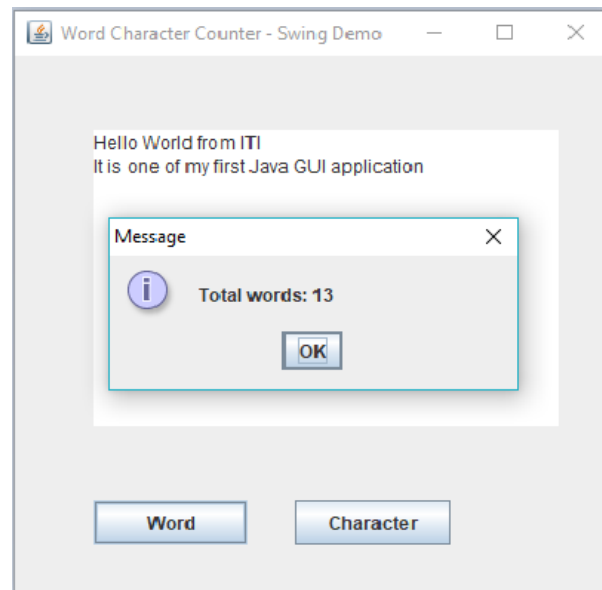
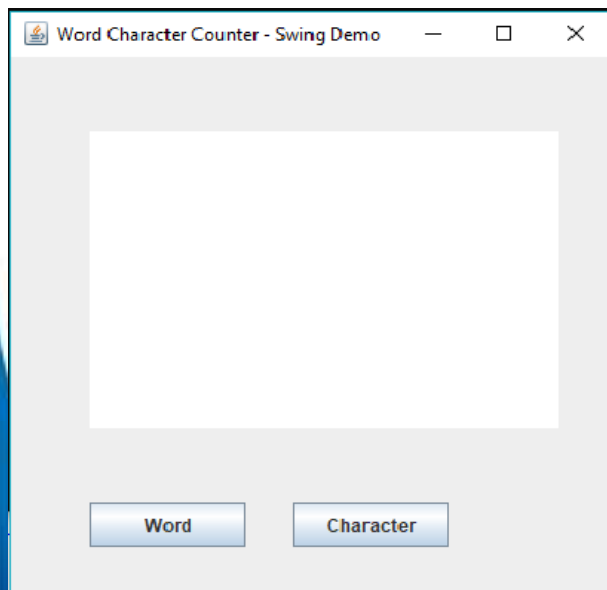


- Note: You may use swing components: JLabel, JTextField, JButton, JOptionPane, and so using the class InetAddress
- You may setLayout(null) of JFrame and then you have to setBounds() of each components with its' positions and size



## 2. Words and Characters Counter

- Write a Java GUI application which has an editable TextArea and two Buttons: one for counting number of words of the string you may write it inside the TextArea, and one for counting the total number of characters –including the spaces.



# Lesson 2

## Introduction to GUI Programming



Java™ Education  
and Technology Services

[Course Outlines](#)

[Course Outlines](#)

Invest In Yourself,  
Develop Your Career

## 2.1 Java UI Components History

- There are current three sets of Java APIs for graphics programming: AWT (Abstract Windowing Toolkit), Swing and JavaFX.
  1. AWT API was introduced in JDK 1.0. Most of the AWT components have become obsolete and should be replaced by newer Swing components.
  2. Swing API, a much more comprehensive set of graphics libraries that enhances the AWT, was introduced as part of Java Foundation Classes (JFC) after the release of JDK 1.1. JFC consists of Swing, Java2D, Accessibility, Internationalization, and Pluggable Look-and-Feel Support APIs. JFC has been integrated into core Java since JDK 1.2.

## 2.1 Java UI Components History

3. The latest JavaFX, which was integrated into JDK 8, is meant to replace Swing.
- Other than AWT/Swing/JavaFX graphics APIs provided in JDK, other organizations/vendors have also provided graphics APIs that work with Java, such as Eclipse's Standard Widget Toolkit (SWT) (used in Eclipse), Google Web Toolkit (GWT) (used in Android), 3D Graphics API such as Java bindings for OpenGL (JOGL) and Java3D.
  - We will start with the AWT before moving into Swing to give you a complete picture of Java Graphics.

## 2.2 Programming GUI with AWT

- AWT Abstract Window Toolkit :
  - is an API to develop GUI or *window-based* application in java.
  - AWT components are *platform-dependent* [ components are displayed according to the view of operating system ].
  - AWT is *heavyweight* [ its components use the resources of system ].

## 2.2.1 AWT Packages

- AWT is huge! It consists of 12 packages of 370 classes (Swing is even bigger, with 18 packages of 737 classes as of JDK 8). Fortunately, only 2 packages - **java.awt** and **java.awt.event** - are commonly-used.
  1. The **java.awt** package contains the core AWT graphics classes:
    - GUI **Component** classes, such as **Button**, **TextField**, and **Label**.
    - GUI **Container** classes, such as **Frame** and **Panel**.
    - Layout managers, such as **FlowLayout**, **BorderLayout** and **GridLayout**.
    - Custom graphics classes, such as **Graphics**, **Color** and **Font**.

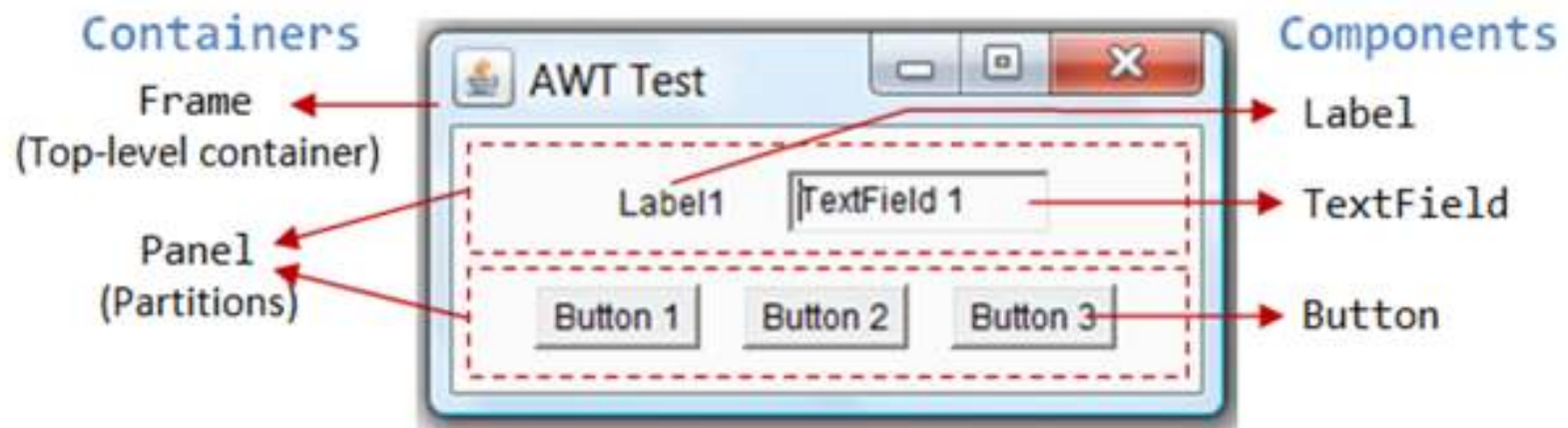
## 2.2.1 AWT Packages

2. The `java.awt.event` package supports event handling:

- **Event** classes, such as **ActionEvent**, **MouseEvent**, **KeyEvent** and **WindowEvent**,
  - Event Listener Interfaces, such as **ActionListener**, **MouseListener**, **MouseMotionListener**, **KeyListener** and **WindowListener**,
  - Event Listener Adapter classes, such as **MouseAdapter**, **KeyAdapter**, and **WindowAdapter**.
- AWT provides a *platform-independent* and *device-independent* interface to develop graphic programs that runs on all platforms, including Windows, Mac OS X, and Unixes.

## 2.2.2 Containers and Components

- There are two types of GUI elements:
  1. Component: Components are elementary GUI entities, such as **Button**, **Label**, and **TextField**.
  2. Container: Containers, such as **Frame** and **Panel**, are used to hold components in a specific layout (such as **FlowLayout** or **GridLayout**). A container can also hold sub-containers.

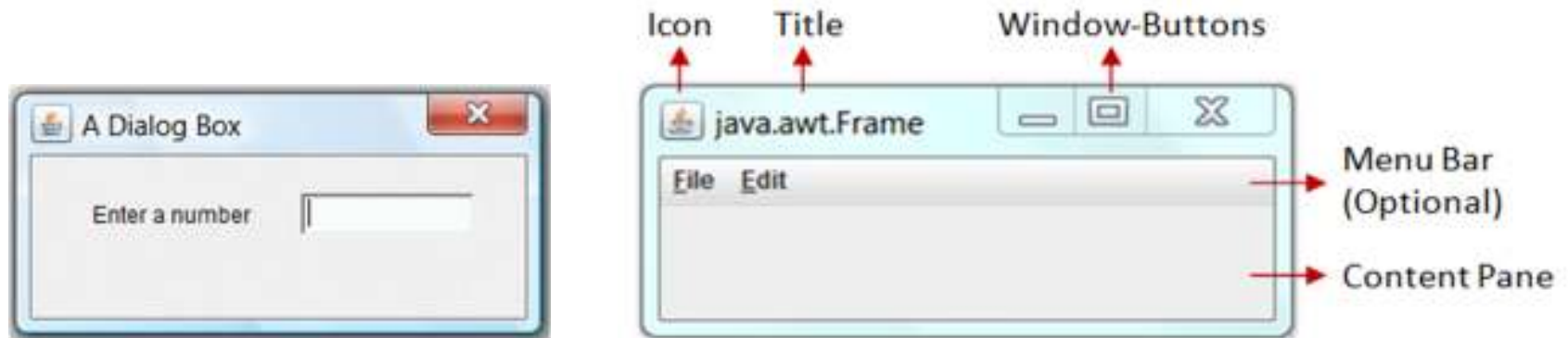


- GUI components are also called controls



## 2.2.2.1 AWT Container Classes

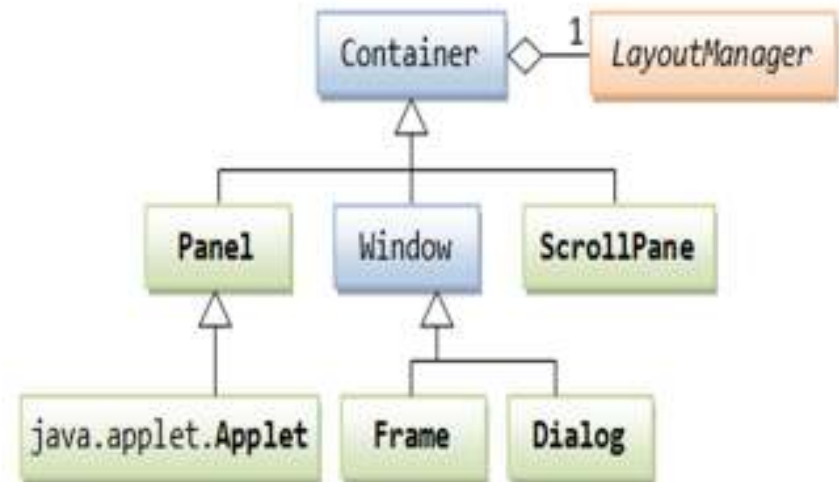
- Top-Level Containers: **Window**, **Frame**, **Dialog** and **Applet**. Each GUI program has a top-level container. The commonly-used top-level containers in AWT are **Frame**, **Dialog** and **Applet**.
  - A **Frame** provides the "main window" for your GUI application.



- An AWT **Dialog** is a "pop-up window" used for interacting with the users.
- An AWT **Applet** ( package `java.applet`) is the top-level container for an applet, which is a Java program running inside a browser.

## 2.2.2.1 AWT Container Classes

- Secondary Containers: **Panel** and **ScrollPane**., are placed inside a top-level container or another secondary container.
  - **Panel**: a rectangular box used to layout a set of related GUI components in pattern such as grid or flow.
  - **ScrollPane**: provides automatic horizontal and/or vertical scrolling for a single child component.
  - others.
- Hierarchy of the AWT Containers



## 2.2.2.2 AWT Component Classes

- AWT provides many ready-made and reusable GUI components in package `java.awt`. The frequently-used are: **Button**, **TextField**, **Label**, **Checkbox**, **CheckboxGroup** (radio buttons), **List**, and **Choice**, as illustrated below.

Enter your name here

TextField

Click Me!

Button

This is Label

Label

Red  
Red  
Green  
Blue

Choice

☒ one ☐ two ☐ three

CheckBox

☒ Alpha ☐ Beta ☐ Charlie

CheckboxGroup

Mercury  
Venus  
Earth  
Mars  
Jupiter  
Saturn  
Uranus  
Neptune

List

## 2.3 Layout Managers

- A container has a *so-called layout manager* to arrange its components.
- The layout managers provide a level of abstraction to map your user interface on all windowing systems, so that the layout can be *platform-independent*.
- AWT provides the following layout managers (in package `java.awt`): **FlowLayout**, **GridLayout**, **BorderLayout**, **GridBagLayout**, **BoxLayout**, **CardLayout**, and others.
- Swing added more layout manager in package `javax.swing`.
- A container has a **`setLayout()`** method to set its layout manager.

## 2.3.1 FlowLayout

- In the `java.awt.FlowLayout`, components are arranged from left-to-right inside the container in the order that they are added (via method `add(Component)`).
- When one row is filled, a new row will be started. The actual appearance depends on the width of the display window.



## 2.3.2 GridLayout

- In `java.awt.GridLayout`, components are arranged in a grid (matrix) of rows and columns inside the **Container**.
- Components are added in a left-to-right, top-to-bottom manner in the order they are added (via method `add(Component)`).





## 2.3.3 BorderLayout

- In `java.awt.BorderLayout`, the container is divided into 5 zones: **EAST**, **WEST**, **SOUTH**, **NORTH**, and **CENTER**.
- Components are added using method `Container.add(aComponent, zone)`,

where zone is either: `BorderLayout.NORTH`,

`BorderLayout.SOUTH`,

`BorderLayout.WEST`,

`BorderLayout.EAST`,

or `BorderLayout.CENTER`.



- You need not place components to all zones. The **NORTH** and **SOUTH** components may be stretched horizontally; the **EAST** and **WEST** components may be stretched vertically; the **CENTER** component may stretch both horizontally and vertically to fill any space left over.

## 2.4 Programming GUI with Swing

- Swing is part of the so-called "*Java Foundation Classes (JFC)*" which was introduced in 1997 after the release of JDK 1.1. JFC was subsequently included as an integral part of JDK since JDK 1.2. JFC consists of:
  - **Swing API:** for advanced graphical programming.
  - **Accessibility API:** provides assistive technology for the disabled.
  - **Java 2D API:** for high quality 2D graphics and images.
  - **Pluggable look and feel supports.**
  - **Drag-and-drop support** between Java and native applications.
- Swing is a rich set of easy-to-use, easy-to-understand JavaBean GUI components that can be dragged and dropped as "GUI builders" in visual programming environment.



## 2.4.1 Swing's Features

- Swing is huge (consists of 18 packages of 737 classes as in JDK 1.8) and has great depth. Compared with AWT, Swing provides a huge and comprehensive collection of reusable GUI components.



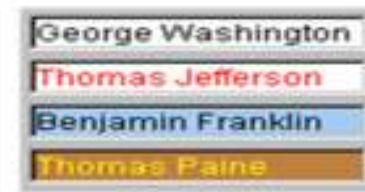
Buttons



Combo Box



List



TextField



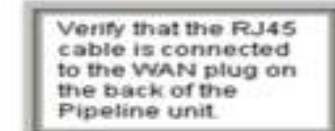
Slider



Menu



Label



Text Area



Tool Tip



Progress Bar



File Chooser



Color Chooser

First Name	Last Name
Mark	Andrews
Tom	Ball
Alan	Chung
Jeff	Dinkins

Table



Tree



Split Pane



Tabbed Pane

## 2.4.1 Swing's Features

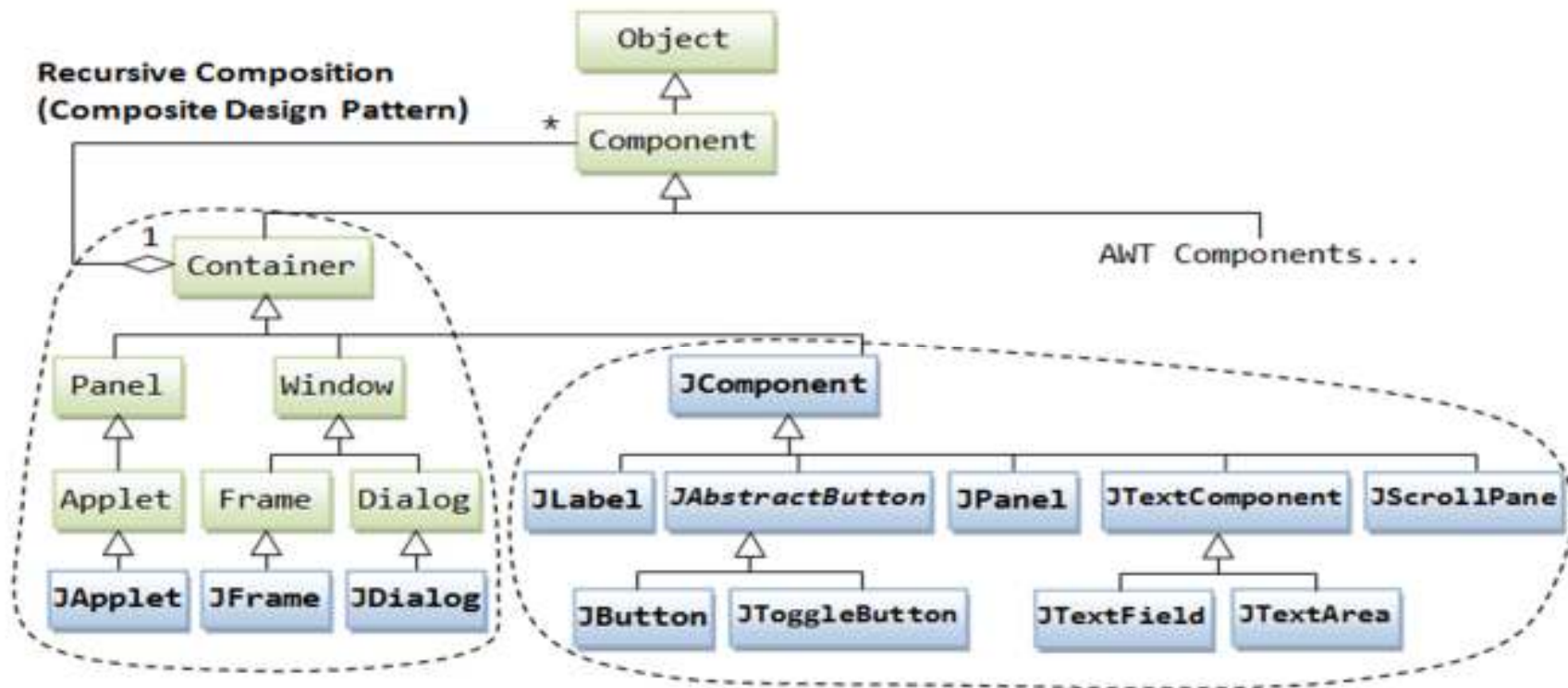
- The main features of Swing are
  1. Swing is written in pure Java (except a few classes) and therefore is 100% portable.
  2. Swing components are lightweight. The AWT components are heavyweight.
  3. Swing components support pluggable look-and-feel.
  4. Swing supports mouse-less operation, i.e., it can operate entirely using keyboard.
  5. Swing components support "tool-tips".
  6. Swing components are *JavaBeans* – a Component-based Model used in Visual Programming. You can drag-and-drop a Swing component into a "design form" using a "GUI builder" and double-click to attach an event handler.

## 2.4.1 Swing's Features

7. Swing application uses AWT event-handling classes (in package **java.awt.event**). Swing added some new classes in package **javax.swing.event**, but they are not frequently used.
8. Swing application uses AWT's layout manager (such as **FlowLayout** and **BorderLayout** in package **java.awt**). It added new layout managers, such as Springs, Struts, and **BoxLayout** (in package **javax.swing**).
9. Swing implements double-buffering and automatic repaint batching for smoother screen repaint.
10. Swing introduces **JLayeredPane** and **JInternalFrame** for creating Multiple Document Interface (MDI) applications.
11. Swing supports floating toolbars (in **JToolBar**), splitter control, "undo".

## 2.4.2 Swing's Components

- Swing component classes (in package javax.swing) begin with a prefix "J", e.g., **JButton**, **JTextField**, **JLabel**, **JPanel**, **JFrame**, or **JApplet**.



## 2.4.2.1 Swing's Top-Level and Secondary Containers

- Just like AWT application, a Swing application requires a top-level container. There are three top-level containers in Swing:
  1. **JFrame**: used for the application's main window (with an icon, a title, minimize/maximize/close buttons, an optional menu-bar, and a content-pane).
  2. **JDialog**: used for secondary pop-up window (with a title, a close button, and a content-pane).
  3. **JApplet**: used for the applet's display-area (content-pane) inside a browser's window.
- Similarly to AWT, there are secondary containers (such as **JPanel**) which can be used to group and layout relevant components.



## 2.4.2.1 Swing's Top-Level and Secondary Containers

- The *Content-Pane* of Swing's Top-Level Container:
  - However, unlike AWT, the **JComponents** shall not be added onto the top-level container (e.g., **JFrame**, **JApplet**) directly because they are lightweight components. The **JComponents** must be added onto the so-called *content-pane* of the top-level container. *Content-pane* is in fact a **java.awt.Container** that can be used to group and layout components.
  - If a component is added directly into a **JFrame**, it is added into the *content-pane* of **JFrame** instead



## 2.4.3 Event-Handling in Swing

- Swing uses the AWT event-handling classes (in package **`java.awt.event`**). Swing introduces a few new event-handling classes (in package **`javax.swing.event`**) but they are not frequently used.

## 2.5 JavaFX: Modern Java GUI packages

- JavaFX is a set of graphics and media packages that enables developers to design, create, test, debug, and deploy rich client applications that operate consistently across diverse platforms.



## 2.5.1 JavaFX History

- F3 (Form Follows Function) by Chris Oliver.
- F3 is a declarative scripting language with good support of IDE, and compile time error reporting unlike java script.
- Chris Oliver became a Sun employee through Sun acquisition of See Beyond Technology Corporation in September 2005.
- At JavaOne 2007 , Its name was changed to JavaFX [Open Source].
- The first version of JavaFX Script was an interpreted language, and was considered a prototype of the compiled JavaFX Script language.

## 2.5.1 JavaFX History

- At JavaOne 2009, the JavaFX SDK 1.2 was released.
- At JavaOne 2010, the JavaFX SDK 2.0 was announced.
- JavaFX 2.0 road-map :
  - Deprecating JavaFX script.
  - Porting all JavaFX scripting features into JavaFX 2.0 APIs.
  - Providing web component for embedding HTML and JavaScript into JavaFX code.
  - Enable JavaFX interoperability with swing.
- At JavaOne 2011, the JavaFX SDK 2.0 was released

## 2.5.2 What is JavaFX?

- JavaFX is a next generation graphical user interface toolkit.
- It is intended to replace java Swing as the standard GUI library for JavaSE.
- It is a set of graphics and media packages that enables developers to *design, create, test, debug*, and *deploy* rich client applications that operate consistently across diverse platforms.
- JavaFX has included a feature of customized style using Cascading Style Sheets (CSS) style.

## 2.5.3 JavaFX Features

- It is easy to learn because it is very similar to Java swing.
- **Java APIs.** JavaFX is a Java library that consists of classes and interfaces that are written in Java code.
- **FXML and Scene Builder.** FXML is an XML-based declarative markup language for constructing a JavaFX application user interface.
- **WebView.** A web component that uses WebKitHTML technology to make it possible to embed web pages within a JavaFX application.
- **Swing interoperability.** Existing Swing applications can be updated with JavaFX features, such as rich graphics media playback and embedded Web content.

## 2.5.3 JavaFX Features

- **Built-in UI controls and CSS.** JavaFX provides all the major UI controls that are required to develop a full-featured application. Components can be skinned with standard Web technologies such as CSS.
- **Animations, 2D and 3D graphics .**
- **Canvas API.** The Canvas API enables drawing directly within an area of the JavaFX scene that consists of one graphical element (node).
- **Rich Text Support.** JavaFX 8 brings enhanced text support to JavaFX, including bi-directional text and complex text scripts, such as Thai and Hindi in controls, and multi-line, multi-style text in text nodes.

## 2.5.3 JavaFX Features

- **Multitouch Support.** JavaFX provides support for multitouch operations, based on the capabilities of the underlying platform.
- **Hardware-accelerated graphics pipeline.** JavaFX graphics are based on the graphics rendering pipeline (Prism).
- **High-performance media engine.** The media pipeline supports the playback of web multimedia content. It provides a stable, low-latency media framework that is based on the GStreamer multimedia framework.
- Many extra features like date-picker, accordion pane, tabbed pane and pie-chart.

## 2.6 Java UI Components

### ❑ **AWT**

- ❑ Platform Specific.



### ❑ **Swing**

- ❑ Only for Desktop Applications

### ❑ **JavaFX**

- ❑ Desktop, websites, Handheld Devices friendly.

## 2.6.1 Java Swing Vs JavaFX

- In Swing,

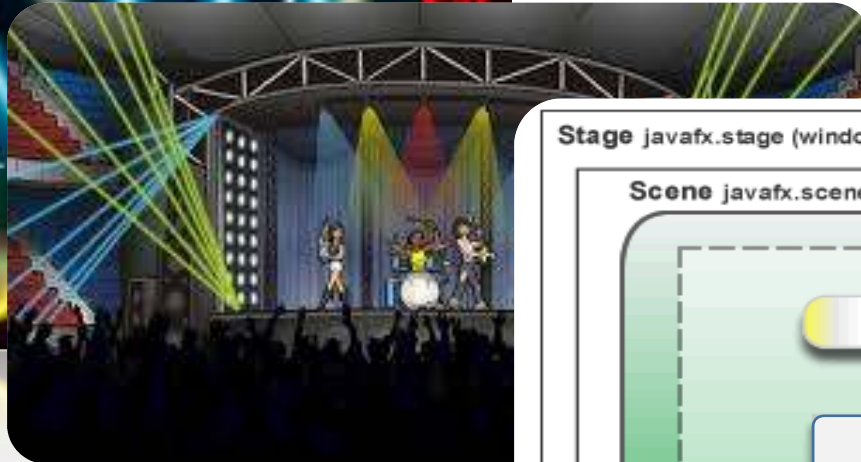


- The **JFrame** class holds all user interface components.
- A **JFrame** is an empty window to which you can add a **JPanel**, which serves as an intermediate (or secondary) container for user-interface elements.
- A Swing application is actually a class that extends the **JFrame** class. To display user-interface components, you add components to a **JPanel** and then add the panel to the frame.



## 2.6.1 Java Swing Vs JavaFX

- In JavaFX, All the world is Stage



Stage javafx.stage (window)

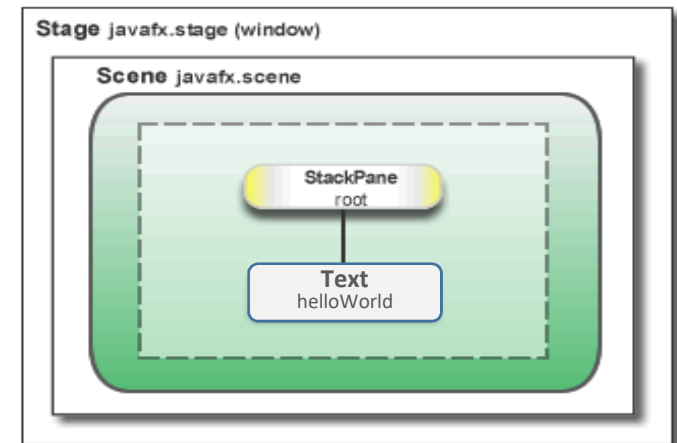
Scene javafx.scene

StackPane  
root

Text  
helloWorld

## 2.6.1 Java Swing Vs JavaFX

- In JavaFX, All the world is Stage



- A **stage** is the highest level container.
- The individual controls and other components that make up the user interface are contained in a **scene**.
- An application can have more than one scene, but only one of the scenes can be displayed on the stage at any given time.
- A scene contains a **scene graph**, is a collection of all the elements [**nodes**] that make up a user interface.

## 2.6.1 Java Swing Vs JavaFX

- **JavaFX** formatting can be controlled with CSS.
- **JavaFX** has several interesting controls that **Swing** doesn't have, such as the collapsible **TitledPane** control and the **Accordion** control.
- The `javafx.scene.effect` package contains a number of classes that can easily apply special effects to any node in the scene graph.
- **Swing** does not provide any direct support for animation.
- **JavaFX** has built-in support for sophisticated animations that can be applied to any node in the scene graph.
- **JavaFX** supports multi-touch operations.

## 2.7 Hello World Program in JavaFX

- There is three way to make a hello world program
  - **Simple code using classes**
  - Using FXML (XML)
  - Using Scene builder

## 2.7 Hello World Program in JavaFX

```
public class HelloWorld extends Application{

    @Override
    public void start(Stage primaryStage) throws Exception {

        Text helloWorld = new Text("Hello World FX!!!");

        StackPane rootPane = new StackPane(helloWorld);

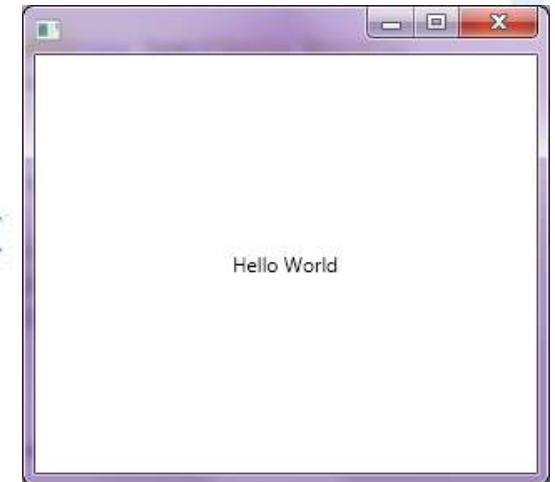
        Scene scene = new Scene(rootPane, 400, 300);

        primaryStage.setScene(scene);
        primaryStage.show();

    }

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

}
```



## 2.7 Hello World Program in JavaFX

- In the above example, there are important things to know about the basic structure of a JavaFX application:
  - The main class for a JavaFX application extends the `javafx.application.Application` class. The `start()` method is the main entry point for all JavaFX applications.
  - A JavaFX application defines the user interface container by means of a stage and a scene.
  - The JavaFX **Stage** class is the top-level JavaFX container.
  - The JavaFX **Scene** class is the container for all content.

## 2.7 Hello World Program in JavaFX

- In JavaFX, the content of the scene is represented as a hierarchical scene graph of nodes. In above example, the **rootPane** node is a **StackPane** object, which is a resizable layout node. This means that the **rootPane** node's size tracks the scene's size and changes when the stage is resized by a user.
- The **rootPane** node contains one child node, a **Text** component.

## 2.7 Hello World Program in JavaFX

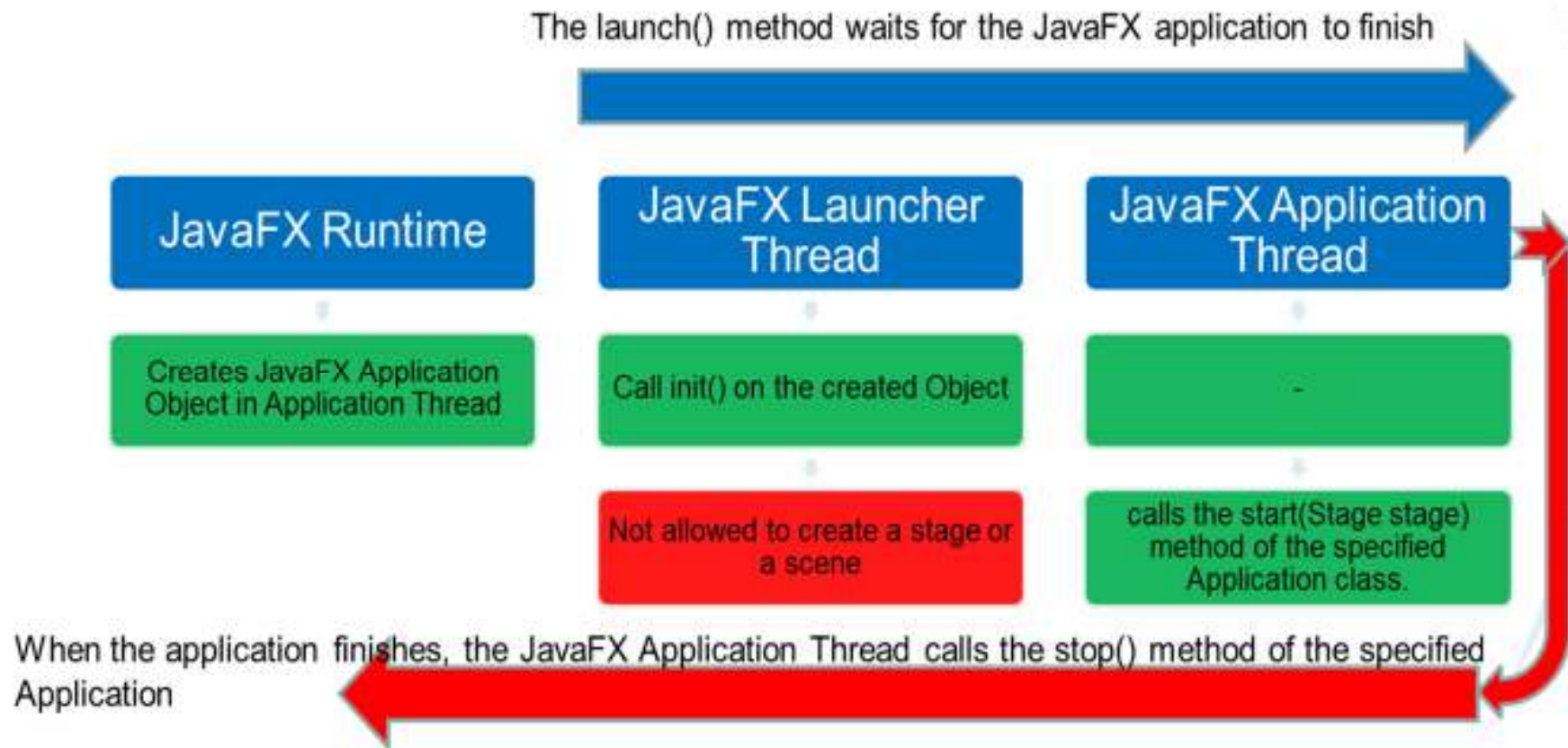
- Application class is the entry point for JavaFX applications.
- The Application class is an abstract class with a single abstract method start.
- The application class has some important methods :
  - `public void init() throws Exception`
  - `public abstract void start(Stage primaryStage)  
throws Exception`
  - `public void stop() throws Exception`
  - `public static void launch(Class<? extends  
Application> appClass, String... args)`
  - `public static void launch(String... args)`



## 2.7 Hello World Program in JavaFX

- **launch ()** method:
  - This method is typically called from the main method().
  - It must not be called more than once or an exception will be thrown.
  - The **launch** method does not return until the application has exited , either via a call to **Platform.exit** or all of the application windows have been closed and the **Platform** attribute **implicitExit** is set to true.

## 2.8 The Life Cycle of a JavaFX Application



## 2.8 The Life Cycle of a JavaFX Application

- JavaFX runtime is responsible for creating several threads.
- At different phases in the application, threads are used to perform different tasks.
- The JavaFX runtime creates, among other threads, two threads:
  - JavaFX-Launcher Thread
  - JavaFX Application Thread
- The `launch()` method of the `Application` class create these threads.

## 2.8 The Life Cycle of a JavaFX Application

- During the lifetime of a JavaFX application, the JavaFX runtime calls the following methods of the specified JavaFX Application class in order:
  - The default (no-args) constructor [ in Application Thread]
  - The init() method [ in Launcher Thread]
  - The start() method [ in Application Thread]
  - The stop() method [ in Application Thread]

## 2.8 The Life Cycle of a JavaFX Application

- **Launcher Thread**

- Is the thread that is used to launch the application.
- Constructing and modifying the **Stage** on the launcher thread is **not allowed** as it will throw an exception.
- Also modifying objects that are attached to the scene graph.

- **Application Thread**

- Is the thread that is used to invoke the **start()** and **stop()** methods.
- This thread is used to construct and modify the JavaFX **Stage** and **Scene**, Process input events, running animation timeline, and apply modifications to *live objects* (objects that are attached to the scene).

## 2.8 The Life Cycle of a JavaFX Application

**Example illustrates the life cycle of JavaFX**

```
public class LifeCycleTest extends Application {

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

    public LifeCycleTest() {
        String name = Thread.currentThread().getName();
        System.out.println("Constructor() method: current Thread:" + name);
    }

    @Override
    public void init() throws Exception {
        String name = Thread.currentThread().getName();
        System.out.println("init() method: current Thread:" + name);
        super.init();
    }
}
```

## 2.8 The Life Cycle of a JavaFX Application

```
public void start(Stage primaryStage) {

    String name = Thread.currentThread().getName();
    System.out.println("start() method: current Thread:" + name);

    StackPane root = new StackPane();
    root.getChildren().add(new Text("Hello Life Cycle"));
    Scene scene = new Scene(root, 300, 250);
    primaryStage.setScene(scene);
    primaryStage.show();
}
```

```
@Override
public void stop() throws Exception {
    String name = Thread.currentThread().getName();
    System.out.println("Stop() method: current Thread:" + name);
    super.stop();
}
```

Constructor() method: current Thread:JavaFX Application Thread  
 init() method: current Thread:JavaFX-Launcher  
 start() method: current Thread:JavaFX Application Thread  
 Stop() method: current Thread:JavaFX Application Thread

## 2.8 The Life Cycle of a JavaFX Application

- **Platform class**

- Is the JavaFX application **Platform** support class.
- It can check the current running thread (application thread or not),
- enqueue a task into the application thread,
- and control the default exit behavior.

- `public static boolean isFxApplicationThread ()`  
//Returns true if the calling thread is the JavaFX Application Thread
- `public static void runLater (Runnable runnable)`  
//Run the specified Runnable on the JavaFX Application Thread in the future
- `public static void exit ()` //Causes the JavaFX application to terminate



## 2.9 JavaFX Program Structure

- A JavaFX application consists of three main components.

- **Nodes.**

- **Scene.**

- **Stage.**



- **Node** is the main actor of the application, and the visible component in our application.
- **Scene** is the component that the nodes are displayed on it.
- **Stage** is the base for the scene, and nodes.

## 2.9.1 Node – GUI components

- A *scene graph* is a set of tree data structures where every item is a **Node**.
- Each item is either a "*leaf*" with zero sub-items or a "*branch*" with zero or more sub-items.
- A **Node** may occur at most once anywhere in the scene graph.
- **Node** objects may be constructed and modified on any thread as long they are not yet attached to a **Scene**.
- Modifying **Nodes** that are already attached to a **Scene** (live objects), on the *JavaFX Application Thread* only.

## 2.9.1 Node – GUI components

- One of the greatest advantages of JavaFX is the ability to use **CSS** to style your nodes in the scene graph.
- JavaFX CSS are based on the **W3C CSS** version 2.1 specification.
- The default style sheet for JavaFX applications is *caspiant.css*\*, which is found in the JavaFX runtime JAR file, **jfxrt.jar**. This style sheet defines styles for the root node and the UI controls.
- To change the default style of a node you can use the **setStyle** method.

```
helloWorld.setStyle("-fx-fill: #09f415;"  
                    + "-fx-cursor: hand;");
```

\* See *modena.css* as the modification in release 8

## 2.9.2 Scene: The Collection of All Nodes

- The JavaFX **Scene** class is the container for all content in a scene graph.
- The application must specify the root **Node** for the scene graph by setting the root property.

```
StackPane rootPane = new StackPane(helloWorld);  
Scene scene = new Scene(rootPane, 400, 300);
```

- The scene's size may be initialized by the application during construction. If no size is specified, the scene will automatically compute its initial size based on the preferred size of its content.
- **Scene** objects must be constructed and modified on the **JavaFX Application Thread**.

## 2.9.2.1 Scene and Style

- Any **Node** that will be displayed on the screen must be attached to the **Scene** some how.
- Each **Node** can have an Id property to identify it.

```
helloWorld.setId("text");
```

- The **Node** can be later re-located using the lookup method.
- This is very helpful when using CSS styles, as we will not write a style for node by node we use style classes.

```
Node x= scene.lookup("text");
```

- To create a style class we first create a .css file to indicate a CSS style sheet.

```
.root{  
    -fx-font: 25px "sans-serif";  
    -fx-fill: #eb2020;  
}
```

## 2.9.2.1 Scene and Style

- To apply this style to our scene we must add this sheet to the scene styles.

```
scene.getStylesheets().add(getClass()  
    .getResource("styles/styles.css").toString());
```

- To add a certain class to a node you can use the **getStyleClasses().add()** method as illustrated in the following slide.

## 2.9.2.1 Scene and Style

### MyStyles.css

```
.myStyleClass{
    -fx-fill: #115ee5;
    -fx-font: 25px sans-serif;
}
```

### Application **start()** method

```
@Override
public void start(Stage primaryStage) {

    Text txt = new Text("Hello World");

    StackPane root = new StackPane();
    root.getChildren().add(txt);

    Scene scene = new Scene(root, 300, 250);

    scene.getStylesheets().add(getClass()
        .getResource("../styles/MyStyles.css").toString());
    txt.getStyleClass().add("myStyleClass");

    primaryStage.setTitle("Hello World!");
    primaryStage.setScene(scene);
    primaryStage.show();
}
```



## 2.9.2.1 Scene and Style

- You can create a style class for a certain **Node** in the **Scene** using the hash symbol (#) and the **Node** Id.

MyStyle.css

```
#text{
    -fx-font: 25px "sans-serif";
    -fx-fill: #eb2020;
}
```

Application **start()** method

```
@Override
public void start(Stage primaryStage) {

    Text txt = new Text("Hello World");
    txt.setId("text");
    StackPane root = new StackPane();
    root.getChildren().add(txt);

    Scene scene = new Scene(root, 300, 250);

    scene.getStylesheets().add(getClass()
        .getResource("../styles/MyStyles.css").toString());

    primaryStage.setTitle("Hello World!");
    primaryStage.setScene(scene);
    primaryStage.show();
}
```

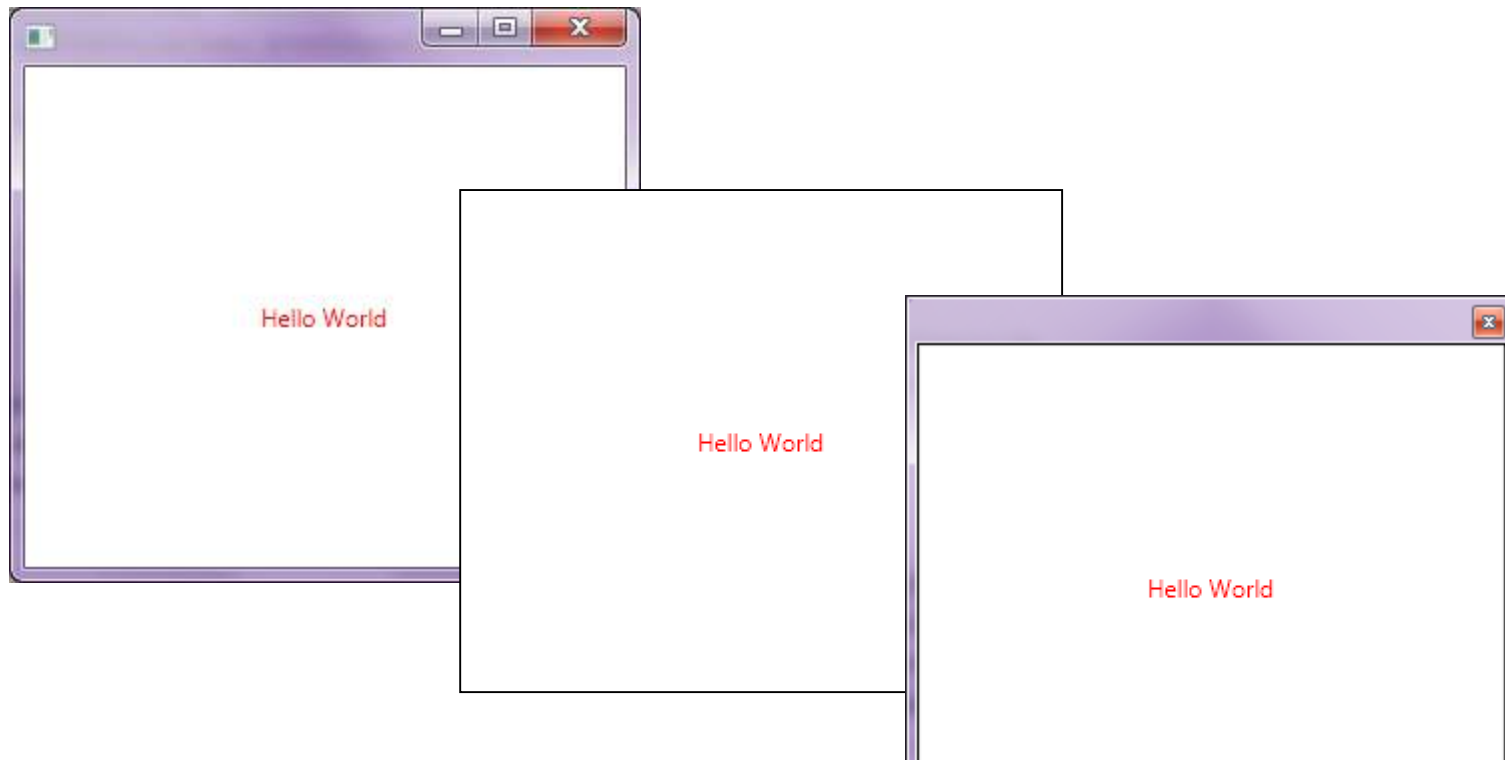


## 2.9.3 Stage: The Main Container

- The JavaFX Stage class is the top level JavaFX container.
- The primary **Stage** is constructed by the platform.
- Stage object must be constructed and modified on the JavaFX Application Thread.
- A stage has one of the following styles:
  - **StageStyle.DECORATED**: a stage with a solid white background and platform decorations.
  - **StageStyle.UNDECORATED**: a stage with a solid white background and no decorations.
  - **StageStyle.TRANSPARENT**: a stage with a transparent background and no decorations.
  - **StageStyle.UTILITY**: a stage with a solid white background and minimal platform decorations.
- The style must be initialized before the stage is made visible.

## 2.9.3 Stage: The Main Container

- A stage has one of the following styles:



```
primaryStage.initStyle(StageStyle.UTILITY);
```

## 2.9.3 Stage: The Main Container

- A stage can optionally have an owner **Window**.

```
public final void initOwner(Window owner)
```

- When a parent window is closed, all its descendant windows are closed.
- A stage has one of the following modalities:
  - **None**: stage that does not block any other window.
  - **Window Modal**: a stage that blocks input events from being delivered to all windows from its owner (parent) to its root.
  - **Application Modal**: a stage that blocks input events from being delivered to all windows from the same application.

# Lab Exercise



Java™ Education  
and Technology Services



Invest In Yourself  
**Develop** Your Career

# Hello World

- JavaFX Lifecycle application
- Create the Hello World application to match the following style.
  1. Use JavaFX code [ Reflection , LinearGradient ]
  2. Use CSS style sheet [ LinearGradient ]

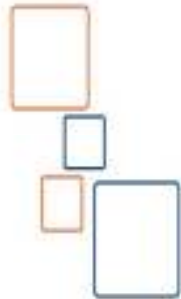


# Lesson 3

## Building UI

### Using JavaFX

**Basic Controls, Event Handling, and Layout**



Java™ Education  
and Technology Services



Invest In Yourself,  
Develop Your Career

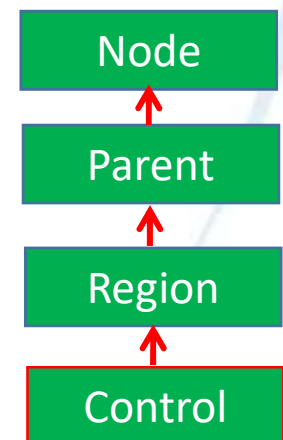


## 3.1 Basic Controls

- Control
- Labeled controls
- Button Classes
- Text Input Control
- Creating Menus

## 3.1.1 Control Class

- Class **Control** is the base class for all Java FX Controls.
- Class **Control** is a sub-class of class **Node**, so it can be treated as node in the scene plus its variables and behaviors as control to support user interactions.
- Controls support explicit skinning to make it easy to leverage the functionality of a control while customizing its appearance (Context menu, skin, **Tooltip**).





## 3.1.2 Labeled controls

- A Labeled Control is one which has as part of its user interface a textual content associated with it.
- It has four sub-classes:
  - **Cell**: used for virtualized controls such as:
    - **ListView**, **TreeView**, and **TableView**.
  - **Label**: is a non-editable text control.
  - **TitledPane**: panel with a title that can be opened and closed.
  - **ButtonBase**: Base class for button-like UI Controls, including **Hyperlinks**, **Buttons**, **ToggleButton**s, **CheckBoxes**, and **RadioButton**s.



## 3.1.2 Labeled controls

- We can customize a **Labeled** control to hold also images and text.

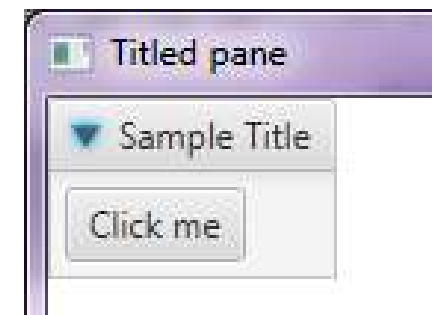
```
Image img = new Image(getClass()
    .getResourceAsStream("images/smile.png"));
ImageView view = new ImageView(img);
Label lable = new Label("Test Lable", view);
lable.setContentDisplay(ContentDisplay.TOP);
```



## 3.1.2 Labeled controls

- The panel in a **TitledPane** can be any **Node** such as UI controls or groups of nodes added to a layout container.
- Note: the inherited properties from class **Labeled** are used to manipulate the header area not the content area.
- It is not recommended to set the **MinHeight**, **PrefHeight**, or **MaxHeight** for this control. Unexpected behavior will occur because the **TitledPane**'s height changes when it is opened or closed.

```
TitledPane pane = new TitledPane("Sample Title", new Button("Click me"));
Scene scene = new Scene(new Group(), 300, 400);
Group root = (Group) scene.getRoot();
root.getChildren().add(pane);
```

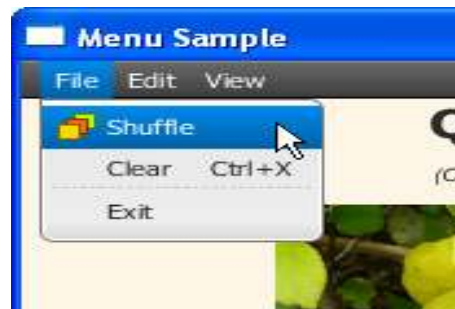


## 3.1.3 ButtonBase Class

- The **ButtonBase** class is an extension of the **Labeled** class. It can display text, an image, or both.

- Sub-classes are:

- **Button.**
- **CheckBox.**
- **HyperLink.**
- **MenuButton.**
- **ToggleButton.**



<http://example.com> — unvisited link

<http://example.com> — link is clicked

<http://example.com> — visited link



## 3.1.3.1 Button Class

- A simple **Button** control.
- The **Button** control can contain text and/or a graphic.

- A **Button** control has three different modes:

➤ **Normal**: A normal push button.

➤ **Default**: A default **Button** is the button that receives a keyboard **VK\_ENTER** press, if no other node in the scene consumes it.

➤ **Cancel**: A Cancel **Button** is the button that receives a keyboard **VK\_ESC** press, if no other node in the scene consumes it.



## 3.1.3.1 Button Class

```
@Override
public void start(Stage primaryStage) throws Exception {
    Button b1 = new Button("Normal");
    Button b2 = new Button("Default");
    Button b3 = new Button("Cancel");

    b2.setDefaultButton(true);
    b3.setCancelButton(true);

    FlowPane root = new FlowPane();
    root.getChildren().addAll(b1,b2,b3);

    Scene scene = new Scene(root, 300, 400);

    primaryStage.setTitle("Button Example");
    primaryStage.setScene(scene);
    primaryStage.show();
}
```

**Button Creation**

**Change Button Type**





## 3.1.3.2 CheckBox

- A tri-state selection **Control** typically skinned as a box with a checkmark or tick mark when checked.
- A **CheckBox** control can be in one of three states:

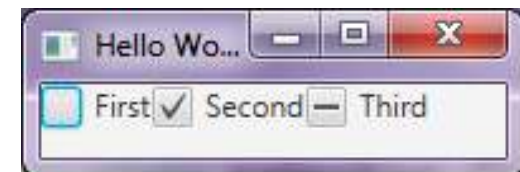
State	Indeterminate	Checked
Checked	false	true
unChecked	false	false
undefined	true	--

- When the checkbox is undefined, it cannot be selected or deselected.

```
cb1.setIndeterminate(false);
cb1.setSelected(false);
```

```
cb1.setIndeterminate(false);
cb2.setSelected(true);
```

```
cb3.setIndeterminate(true);
cb3.setSelected(false);
```

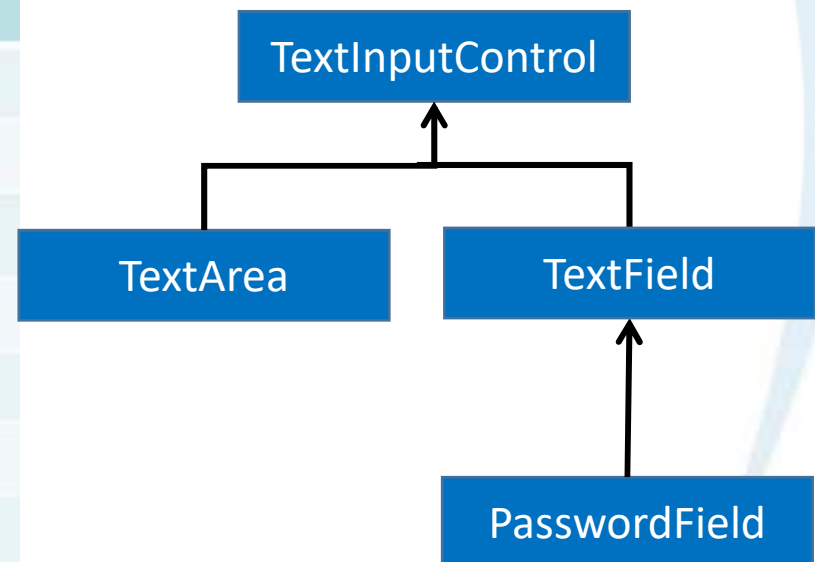


## 3.1.4 TextInputControl Class

- Abstract base class for text input controls

### common methods

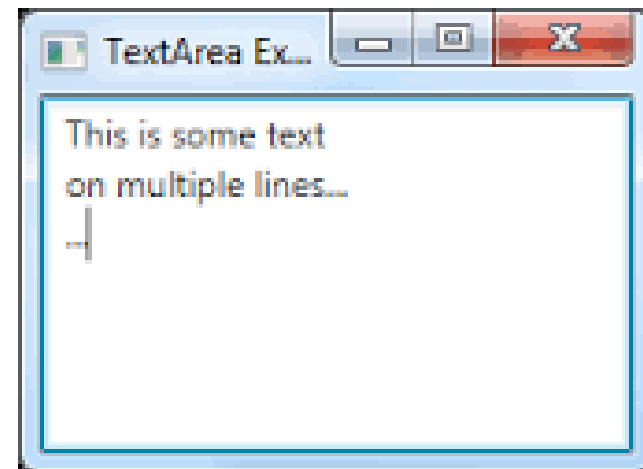
```
void appendText(String text)
void clear()
void deleteText(IndexRange range)
void deleteText(int start,int end)
void deselect()
String getText()
void insertText(int index,String text)
void positionCart(int pos)
void replaceText(int start,int end,String text)
```





## 3.1.4.1 TextArea Class


- Text input component that allows a user to enter multiple lines of plain text.
- You can use the **`setPrefRowCount()`**, and **`setPrefColCount()`** to adjust the preferred size of the **`TextArea`**.



## 3.1.4.2 TextField Class

- Text input component that allows a user to enter a single line of unformatted text.
- As it is one single line **setPrefColCount()** to control the number of columns.
- **TextField** fires **ActionEvent** upon typing the Enter key.

```
TextField lastName = new TextField();  
lastName.setPromptText("Enter your last name.");
```

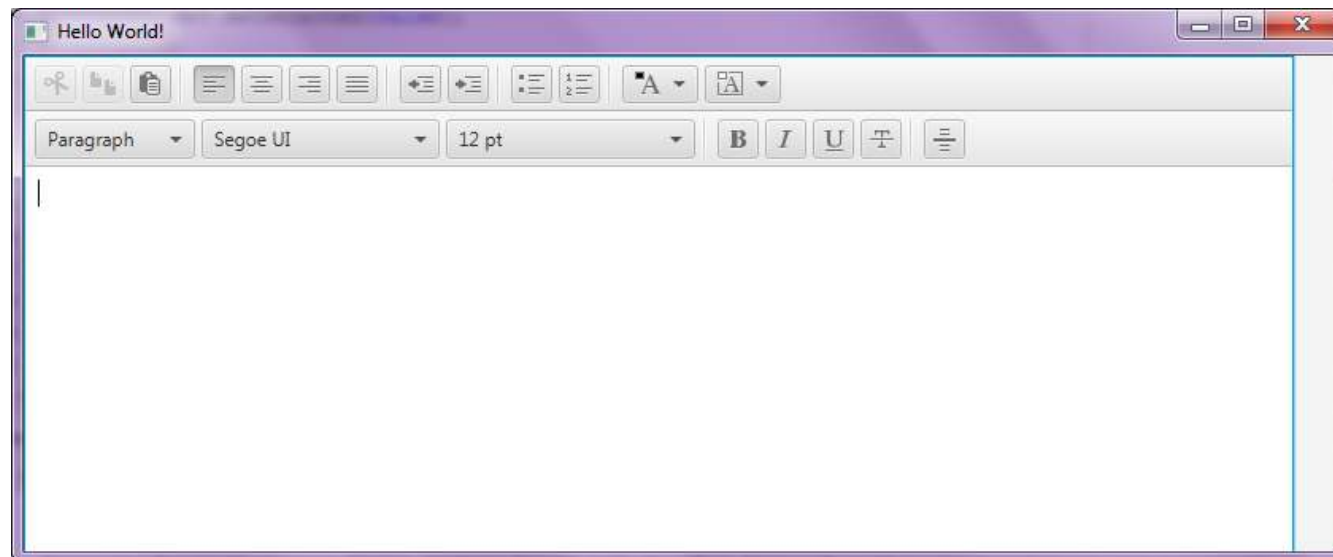


## 3.1.4.3 HTML Editor Class

- It allows you to edit text in your JavaFX applications by using the embedded HTML editor.

```
root.getChildren().add(new HTMLEditor());

Scene scene = new Scene(root, 300, 250);
```



## 3.1.5 Creating Menus

- Constructing a **Menu** in JavaFX is no different than Swing, you create **MenuBar**, **Menu**, and **MenuItems**, then we add them to each other.
- The difference between JavaFX and Swing is that JavaFX does not have a pre-made Anchor for the **MenuBar**, so there is no **setMenuBar()** method like Swing.
- **MenuBar** itself is considered a node that can be added to any part of the located **Pane**.

## 3.1.5.1 MenuBar

- A **MenuBar** control traditionally is placed at the very top of the user interface, and embedded within it are **Menus**.
- To add a **Menu** to a **MenuBar** , you add it to the menus **ObservableList**.

### ➤ Constructors

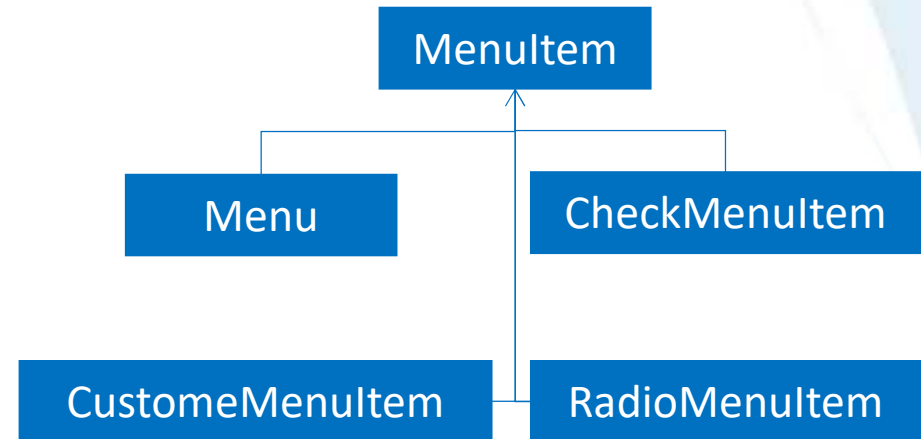
➤ **MenuBar ()**

➤ **MenuBar (Menu . . . )**

### ➤ Methods

➤ **ObservableList<Menu> getMenus ()**

## 3.1.5.2 Menus and MenuItems



- **MenuItem** :
  - To create one actionable option
  - The accelerator property enables accessing the associated action in one keystroke.
- **Menu** : To create a Menu / submenu
- **RadioButtonItem** : To create a mutually exclusive selection
- **CheckMenuItem** : To create an option that can be toggled between selected and unselected states

## 3.1.5.3 Menus and MenuItems

```

public void start(Stage primaryStage) throws Exception {
    MenuBar bar = new MenuBar();
    Menu file = new Menu("File");

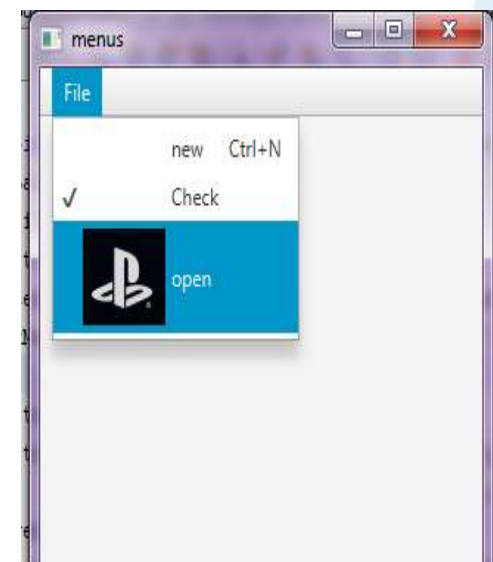
    MenuItem newItem1 = new MenuItem("new");
    newItem1.setAccelerator(KeyCombination.keyCombination("Ctrl+n"));

    CheckMenuItem newItem2 = new CheckMenuItem("Check");

    MenuItem openItem = new MenuItem("open");
    openItem.setGraphic(new ImageView(new Image(getClass()
        .getResourceAsStream("../img/icon.png"))));

    file.getItems().addAll(newItem1, newItem2, openItem);
    bar.getMenus().addAll(file);
    BorderPane pane = new BorderPane();
    pane.setTop(bar);
    Scene scene = new Scene(pane, 300, 400);

```



# Lesson 4

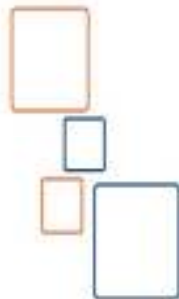
## Introducing to JavaFX Event Handling, and Layout Managers



Java™ Education  
and Technology Services



Invest In Yourself,  
Develop Your Career





## 4.1 JavaFX Event Handling

- Event model in JavaFX is no deferent than Swing, there is an event source and a listener to the event.
- Unlike swing JavaFX consider all event triggers as a *reference property* inside class **Node**. We only need to link the correct event listener to the property we want to respond to.
- JavaFX uses only one generic interface to respond to all events **EventHandler<T extends Event>**. The only method in this interface is **handle(T Event)**.

## 4.1 JavaFX Event Handling

- To handle the event using the event property reference.

```
Button b = new Button("click me !");
b.setOnAction(new EventHandler<ActionEvent>() {

    @Override
    public void handle(ActionEvent event) {
        System.out.print("you clicked me...");
    }

});
```

- Using the **addEventHandler()** method.

```
Button b = new Button("click me !");
b.addEventHandler(ActionEvent.ACTION, new EventHandler<ActionEvent>(){

    @Override
    public void handle(ActionEvent event) {
        //Event Handling code Here
    }

});
```

## 4.2 Layout Mangers

- A JavaFX application can manually lay out the UI by setting the position and size properties for each UI element.
- JavaFX containers (**Panes**) are set of classes used to manage UI components positioning and size over the scene graph.
- Layout pane automatically repositions and resizes the nodes that it contains according to the properties for the nodes and the pane.
- All panes are sub-class of **Node** and they can be added to each other to form more complex layout.

## 4.2.1 BorderLayout

- **BorderPane** lays out children in top, left, right, bottom, and center positions.



- Only one node can be hosted at each position.
- The top and bottom children will be resized to their preferred heights and extend the width of the border pane.
- The left and right children will be resized to their preferred widths and extend the length between the top and bottom nodes.
- And the center node will be resized to fill the available space in the middle.
- **BorderPane** is commonly used as the root of a **Scene**.

## 4.2.1 BorderLayout

- listed below are the commonly used constructors and methods of this pane:

Constructors
<code>BorderPane()</code>
<code>BorderPane(Node center)</code>
<code>BorderPane(Node center, Node top, Node right, Node bottom, Node left)</code>
<code>void setXXX(Node node)</code>
<code>Node getXXX()</code>

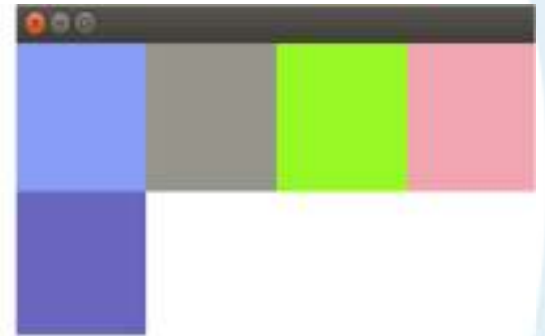
- **Note:** XXX will be replaced with one of the pane positions (center, left, right, top, bottom).

## 4.2.2 FlowPane

- **FlowPane** lays out its children in a flow that wraps at the flowpane's boundary

```
FlowPane pane = new FlowPane();

for (int i = 0; i < 5; i++) {
    pane.getChildren().add(new Rectangle(100, 100,
        new Color(new Random().nextDouble(),
            new Random().nextDouble(),
            new Random().nextDouble(), 1.0)));
}
```



## 4.2.2 FlowPane

- **Nodes** within the **FlowPane** can be aligned Horizontally or Vertically depending on the alignment property value.
- Spacing between nodes can be managed using the `vgap`, and `hgap` properties.
- To add nodes to a **FlowPane** we use the `getChildren()` method to get the node list of this container, then we use `add()`, or `addAll()` method to add nodes.



## 4.2.2 FlowPane

- listed below are the commonly used constructors and methods of this pane:

Constructors
<code>FlowPane()</code>
<code>FlowPane(Node... children)</code>
<code>FlowPane(double hgap, double vgap, Node... children)</code>
<code>FlowPane(Orientation orientation, double hgap, double vgap)</code>
Methods
<code>void setAlignment(Pos value)</code>
<code>void setHgap(double value)</code>
<code>void setVgap(double value)</code>
<code>ObservableList&lt;Node&gt; getChildren() ----&gt; inherited from class Pane</code>
<code>void setOrientation(Orientation value)</code>



## 4.2.3 GridPane

- **GridPane** lays out its children within a flexible grid of rows and columns.
- A child may be placed anywhere within the grid and may span multiple rows/columns.
- A child's placement within the grid is defined by it's layout constraints:

Constrain	Type	Description
columnIndex	integer	column where child's layout area starts.
rowIndex	integer	row where child's layout area starts.
columnSpan	integer	the number of columns the child's layout area spans horizontally.
rowSpan	integer	the number of rows the child's layout area spans vertically.

## 4.2.3 GridPane

- If the row/column indices are not explicitly set, then the child will be placed in the first row/column.
- To add nodes to the **GridPane** we use the **add(node, colIndex, rowIndex)** method.

### Constructor

**GridPane()**

### Methods

void **addColumn**(int columnIndex, Node... children)

void **addRow**(int rowIndex, Node... children)

void **setHgap**(double value)

void **setVgap**(double value)

## 4.2.4 AnchorPane

- **AnchorPane** allows the edges of child nodes to be anchored to an offset from the **AnchorPane**'s edges.
- If the **AnchorPane** has a border and/or padding set, the offsets will be measured from the inside edge of those insets.
- **AnchorPane** has four constrains can be set for each child.

Constrain	type	Description
topAnchor	double	distance from the anchorpane's top insets to the child's top edge.
leftAnchor	double	distance from the anchorpane's left insets to the child's left edge.
bottomAnchor	double	distance from the anchorpane's bottom insets to the child's bottom edge.
rightAnchor	double	distance from the anchorpane's right insets to the child's right edge.

## 4.2.4 AnchorPane

- The following are the commonly used methods of the **AnchorPane**:

### Constructors

`AnchorPane()`

`AnchorPane(Node... children)`

### Methods

**static void setBottomAnchor**(Node child, Double value)

**static void setRightAnchor**(Node child, Double value)

**static void setLeftAnchor**(Node child, Double value)

**static void setTopAnchor**(Node child, Double value)

# Lab Exercise



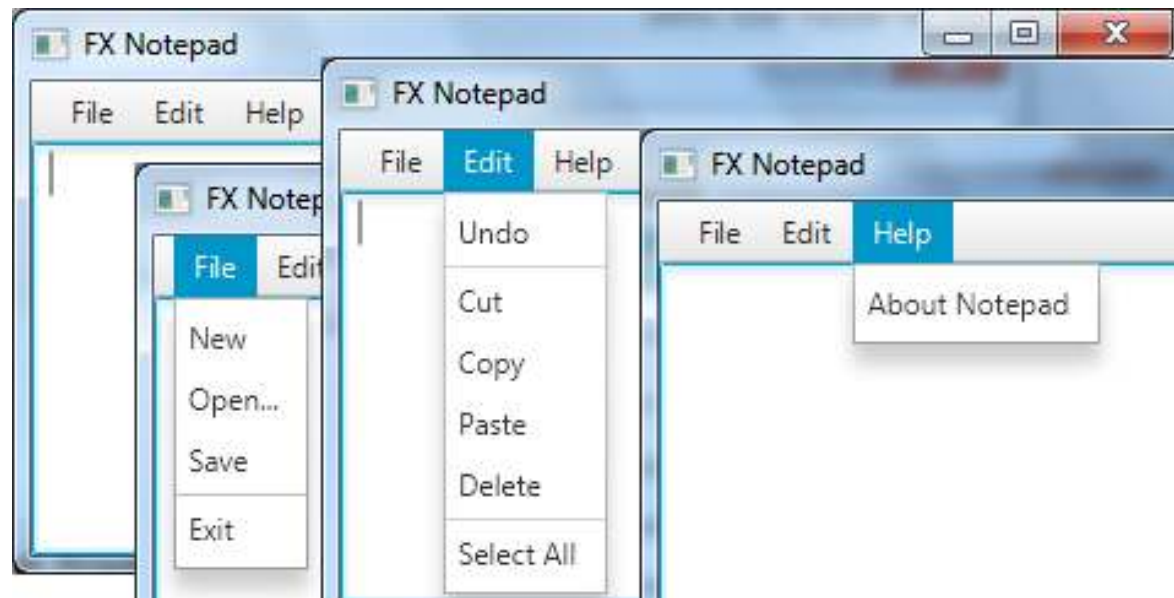
Java™ Education  
and Technology Services



Invest In Yourself,  
**Develop** Your Career

# Create A GUI Desktop Application

- Create a JavaFX NotePad Desktop Application .
  - File menu [new , open, save, Exit]
  - Edit menu [ Cut, copy, Paste, Delete, Select All]
  - Help menu [About]





## References

- Herbert schildet, Java The Complete Reference , Tenth Edition, Oracle Press, McGraw-Hill Education (Publisher), 2018.
- The Java Tutorials:  
<https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html>
- Tutorials Point :  
[https://www.tutorialspoint.com/java/java\\_innerclasses.htm](https://www.tutorialspoint.com/java/java_innerclasses.htm)
- Java T Point: <https://www.javatpoint.com/java-inner-class>
- Java Programming Tutorial: Programming Graphical User Interface (GUI)  
[http://www.ntu.edu.sg/home/ehchua/programming/java/j4a\\_gui.html](http://www.ntu.edu.sg/home/ehchua/programming/java/j4a_gui.html)

# References

- JavaFX: Getting Started with JavaFX:

<https://docs.oracle.com/javase/8/javafx/get-started-tutorial/index.html>

  
**Course Outlines**

*Course Outlines*