Event handling and GUI programming

Comparison of AWT and SWING, Applet class, Applet API hierarchy , Life cycle of Applet, Delegation Event Model, Event handling mechanisms, Swing components, Swing Component Hierarchy-Basic and Advanced Components, JApplet, Layout managers, Adapter class, Innerclass.

What is AWT?

AWT also known as **Abstract window toolkit**. It is a platform dependent API used for **developing GUI** (graphical user interface) or applications that are window based. It is generated by system's host operating system and contains large number of methods and class which are used for creating and managing UI of an application.

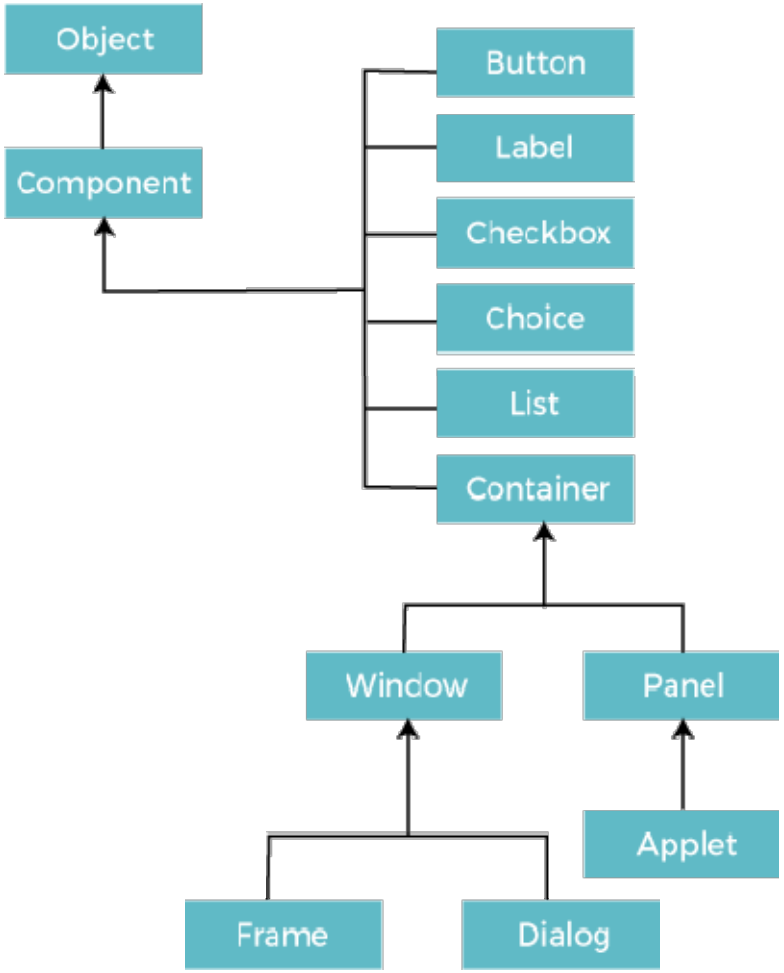
What is Swing?

In Java, a swing is a light-weighted, GUI (graphical user interface) that is used for creating different applications. It has platform-independent components and enables users to create buttons as well as scroll bars. It also includes a package for creating applications for desktops. The components of swing are written in Java and are a part of the Java foundation class.

- The main difference between AWT and Swing is that AWT is purely used for GUI whereas Swing is used for both, GUI as well as for making web application.

Java AWT Hierarchy

Components: AWT provides various components such as buttons, labels, text fields, checkboxes, etc used for creating GUI elements for Java Applications.
Containers: AWT provides containers like panels, frames, and dialogues to organize and group components in the Application.
Layout Managers: Layout Managers are responsible for arranging data in the containers some of the layout managers are BorderLayout, FlowLayout, etc.
Event Handling: AWT allows the user to handle the events like mouse clicks, key presses, etc. using event listeners and adapters.
Graphics and Drawing: It is the feature of AWT that helps to draw shapes, insert images and write text in the components of a Java Application.



AWT is platform dependent

- **JVM**
- **Abstract APIs**
- **Platform-Independent Libraries**

- ***Points about Java AWT components***

i. Components of AWT are heavy and platform dependent

ii. AWT has less control as the result can differ because of components are platform dependent.

Types of containers in AWT

- **Window:** Window is a top-level container that represents a graphical window or dialog box. The Window class extends the Container class, which means it can contain other components, such as buttons, labels, and text fields.
- **Panel:** Panel is a container class in Java. It is a lightweight container that can be used for grouping other components together within a window or a frame.
- **Frame**: The Frame is the container that contains the title bar and border and can have menu bars.
- **Dialog:** A dialog box is a temporary window an application creates to retrieve user input.

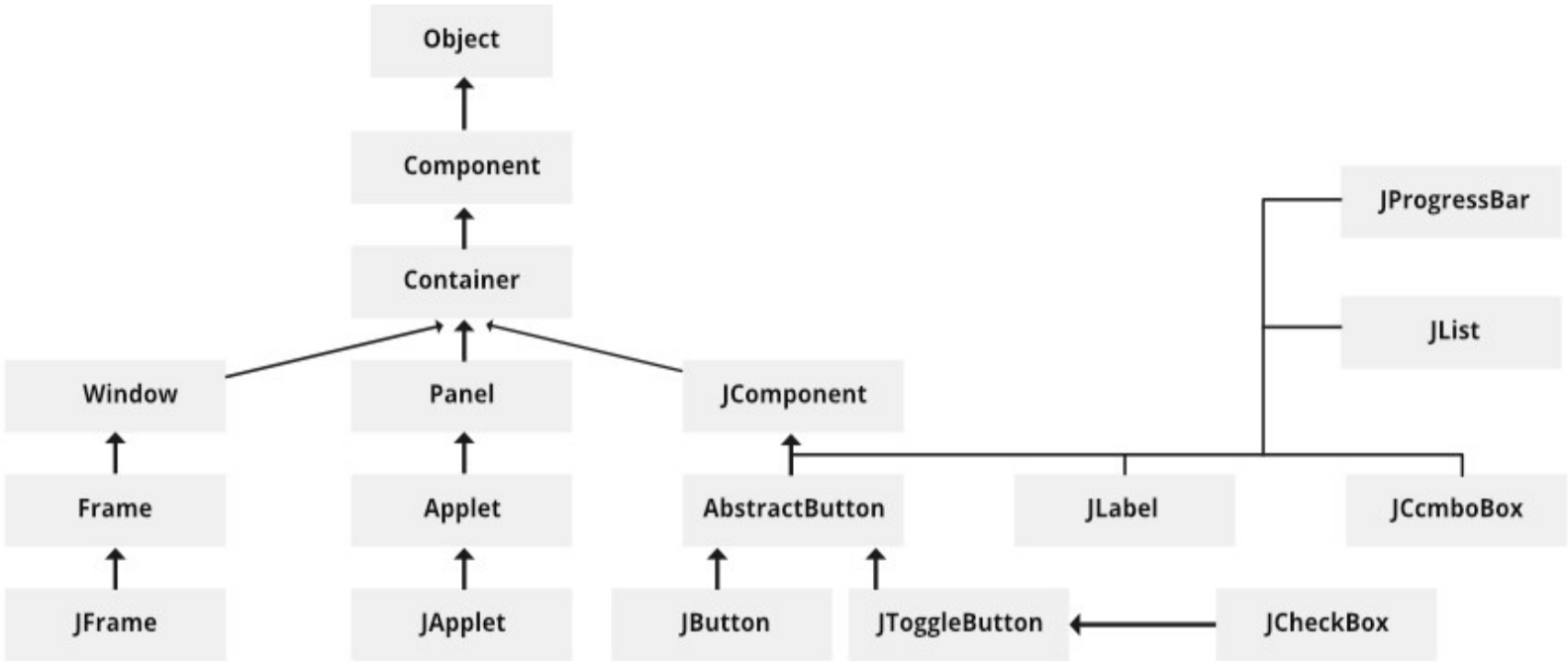
Java Swing

- **Platform Independence:** One of the primary advantages of Swing is its platform independence. Applications developed using Swing can run on any platform that supports Java, without requiring modifications.
- **Rich Set of Components:** Swing offers a wide range of components that can be used to create complex GUIs. Developers can choose from basic components like buttons and labels to advanced components such as tables, trees, and scroll panes.
- **Customizability:** Swing components are highly customizable, allowing developers to control various aspects of their appearance and behavior. Properties such as size, color, font, and layout can be easily adjusted to meet specific design requirements.
- **Event Handling:** Swing provides a robust event handling mechanism that allows developers to respond to user interactions, such as button clicks and mouse movements. This enables the creation of interactive and responsive applications.
- **Layout Managers:** Swing includes a set of layout managers that facilitate the arrangement of components within a container. Layout managers automatically adjust the position and size of components based on the container's size, ensuring consistent behavior across different screen resolutions and devices.

Features of Java Swing

- **Platform Independence and Consistency**
- **Extensive Component Library**
- **Flexible Layout Management**
- **Event-Driven Architecture**
- **Graphics and Rendering**

Swing Classes Hierarchy



AWT	Swing
In Java, awt is an API that is used for developing GUI (Graphical user interface) applications.	Swing on another hand, in Java, is used for creating various applications including the web.
Java AWT components are heavily weighted.	Java swing components are light-weighted.
Java AWT has fewer functionalities as compared to that of swing.	Swing has greater functionalities as compared to awt in Java.
The code execution time in Java AWT is more.	The execution time of code in Java swing is less compared to that of awt.
In Java AWT, the components are platform-dependent.	The swing components are platform-independent.
MVC (Model-View-Controller) is not supported by Java awt.	MVC (Model-View-Controller) is supported by Java swing.
In Java, awt provides less powerful components.	Swing provide more powerful components.
The awt components are provided by the package java.awt	The swing components are provided by javax.swing package
There is abundant features in awt that are developed by developers and act as a thin layer between development and operating system.	Swing has a higher level of in-built components for developers which facilitates them to write less code.
The awt in Java has slower performance as compared to swing.	Swing is faster than that of awt.
AWT Components are dependent on the operating system.	Swing components are completely scripted in Java. Its components are not dependent on operating system.
Java AWT stands for Abstract Window Toolkit.	Java Swing is mainly referred to as Java Foundation Classes (JFC).
Java AWT includes 21 peers. Each control has its own peer, and there is an additional peer specifically for the dialogue. The operating system supplies these peers as widgets.	Java Swing uses a single peer from the OS as a drawing surface to render its widgets like labels, buttons, and entry fields, all of which are created by the Java Swing Package.

Applet class

- Java Applet is a special type of small Java program embedded in the webpage to generate dynamic content. The specialty of the Java applet is it runs inside the browser and works on the Client side (User interface side).
- *Applets, which are small programs that can be run inside a web page, are no longer widely used on the web. Therefore, this package has been deprecated in Java 9 and later versions and is no longer recommended for use. It's expected that this package will be removed in future versions of the language.*
- **In Java, there are two types of Applet**
- Java Applets based on the AWT(Abstract Window Toolkit) packages by extending its Applet class
- Java Applets is based on the Swing package by extending its JApplet Class in it.

- There are two ways to execute a Java Applet:
- By using an HTML file
- By using the appletviewer tool

- **Advantages of Applet**
- It runs inside the browser and works on the Client-side, so it takes less time to respond.
- It is more Secured
- It can be Executed By multi-platforms with any Browsers, i.e., Windows, Mac Os, Linux Os.
- **Disadvantages of Applet**
- A plugin is required at the client browser(User Side) to execute an Applet.

Hierarchy of Applet

The inheritance hierarchy for a Swing applet

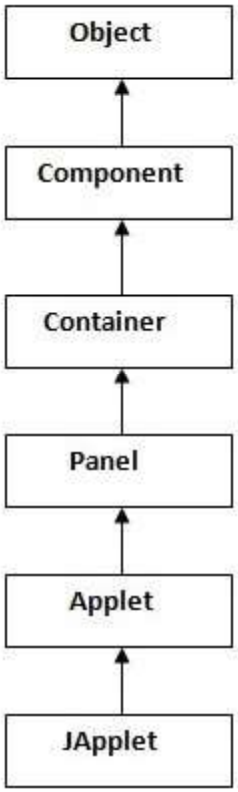
java.awt.Component

java.awt.Container

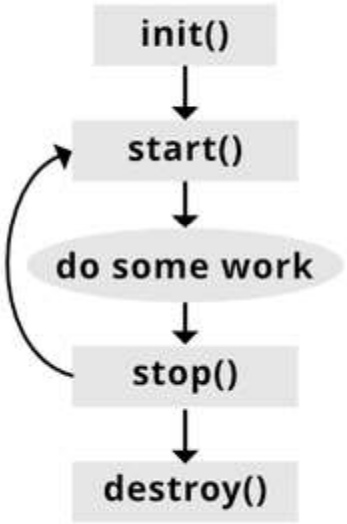
java.awt.Panel

java.applet.Applet

javax.swing.JApplet



Life Cycle Of java Applet Class



- init() and destroy() are only called once each
- start() and stop() are called whenever the browser enters and leaves the page
- do some work is code called by the listeners that may exist in the applet

- **The init() Method**
  - Invoked when the applet is first loaded and again if the applet is reloaded.
  - A subclass of Applet should override this method if the subclass has an initialization to perform. The functions usually implemented in this method include creating new threads, loading images, setting up user-interface components, and getting string parameter values from the <applet> tag in the HTML page.
- **The start() Method**
  - Invoked after the init() method is executed; also called whenever the applet becomes active again after a period of inactivity (for example, when the user returns to the page containing the applet after surfing other Web pages).
  - A subclass of Applet overrides this method if it has any operation that needs to be performed whenever the Web page containing the applet is visited. An applet with animation, for example, might use the start method to resume animation.
- **The stop() Method**
  - The opposite of the start() method, which is called when the user moves back to the page containing the applet; the stop() method is invoked when the user moves off the page.
  - A subclass of Applet overrides this method if it has any operation that needs to be performed each time the Web page containing the applet is no longer visible. When the user leaves the page, any threads the applet has started but not completed will continue to run. You should override the stop method to suspend the running threads so that the applet does not take up system resources when it is inactive.
- **The destroy() Method**
  - Invoked when the browser exits normally to inform the applet that it is no longer needed and that it should release any resources it has allocated.
  - A subclass of Applet overrides this method if it has any operation that needs to be performed before it is destroyed. Usually, you won't need to override this method unless you wish to release specific resources, such as threads that the applet created.

Event Handling

- Event handling is fundamental to Java programming because it is used to create event driven programs eg
  - Applets
  - GUI based windows application
  - Web Application
- Event handling mechanism have been changed significantly between the original version of Java (1.0) and all subsequent versions of Java, beginning with version 1.1.
- The modern approach to handling events is based on the delegation event model,

Basics

- **Events** : **The Events are the objects that define state change in a source**. An event can be generated as a **reaction of a user while interacting with GUI elements**. Some of the event generation activities are moving the mouse pointer, clicking on a button, pressing the keyboard key, selecting an item from the list, and so on. The Events may also occur that may be not related to user interaction, such as a timer expires, counter exceeded, system failures, or a task is completed, etc. We can define events for any of the applied actions.
- **Event Sources** : A **source is an object that causes and generates an event**. It generates an event when the **internal state of the object is changed**. The sources are allowed to generate several different types of events. An example:

  public void addTypeListener (TypeListener e1)

  public **void** addTypeListener(TypeListener e2) **throws** java.util.TooManyListenersException
- Event Listeners**: An event listener is an object that is invoked when an event triggers**. The listeners require two things;

  1. it must be registered with a source; however, it can be registered with several resources to receive notification about the events

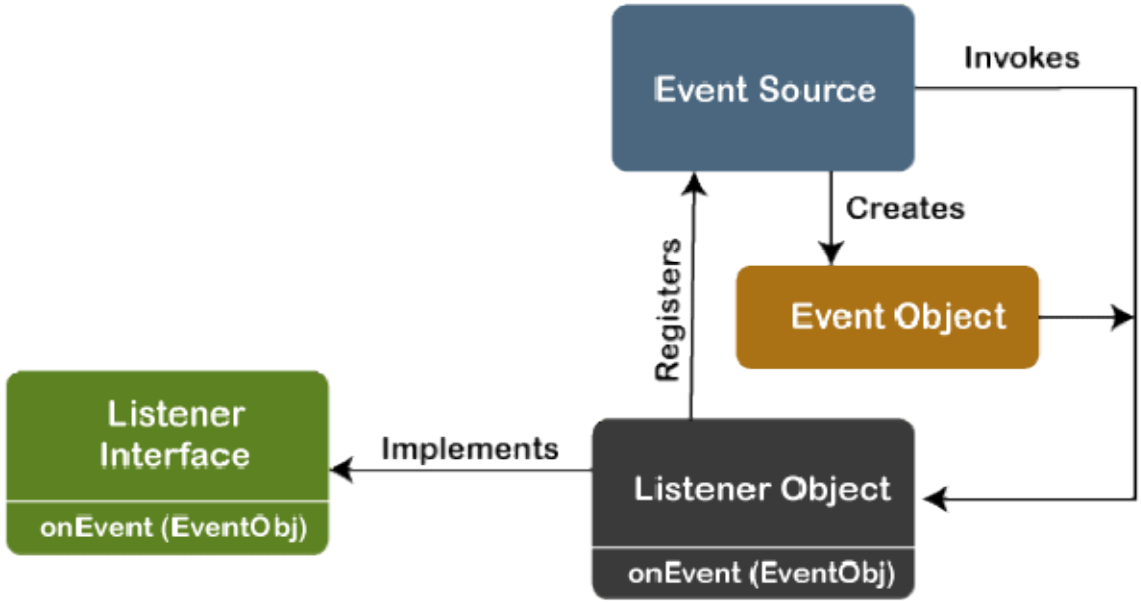
  2. it must implement the methods to receive and process the received notifications. The methods that deal with the events are defined in a set of interfaces. These interfaces can be found in the java.awt.event package.

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. Let's have a brief introduction to this model.
  The Delegation Event Model has the following key participants namely:
  Source - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provides classes for source object.
  Listener - It is also known as event handler. Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.

- Advantages of event Handling
  The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event. The user interface element is able to delegate the processing of an event to the separate piece of code. In this model ,Listener needs to be registered with the source object so that the listener can receive the event notification. This is an efficient way of handling the event because the event notifications are sent only to those listener that want to receive them.

Event Processing in Java



Events and Event Classes

- The root class is called java.util.EventObject. The only common feature shared by all events is a source object. So we find the following two methods in the EventObject class :
  public Object getSource();
  Returns the source of the event.
  String toString( );
  returns the string equivalent of the event.
- The class AWTEvent, defined within the java.awt package, is a subclass of EventObject.
- It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model.
- Its getID( ) method can be used to determine the type of the event.
  int getID( );
- The package java.awt.event defines many types of events that are generated by various user interface elements.

Event handling mechanisms

- Events and Event Clases
  The root class is called java.util.EventObject. The only common feature shared by all events is a source object.
- So we find the following two methods in the EventObject class :
  - public Object getSource(); : Returns the source of the event.
  - String toString( ); : returns the string equivalent of the event.

Delegation Event Model

- The Delegation Event Model is a programming pattern used in **Java for handling events in graphical user interfaces (GUIs).** When a GUI component generates an event (such as a button press), it is delegated to **event listeners of that component**. These listeners then handle the event by executing the appropriate code.
- The Delegation Event Model is important for several reasons:
  - It allows for a **clean separation of concerns between the GUI components and the code that handles their events**.
  - It simplifies adding or removing event listeners, **as they can be added or removed independently of the components they listen to**.
  - It provides a **flexible and extensible architecture** for handling events, allowing various event types.

- The hierarchy of event listeners in the Delegation Event Model consists of three levels:
- **Top-level container listeners:** These listeners register with the **top-level container of a GUI component** (such as a JFrame). **They handle events that the element or its children do not govern**.
- **Component-level listeners:** They register with a specific **GUI component (such as a JButton).** They handle events generated by that component.
- **Component-level listeners:** These **register with the system itself (such as the AWTEventMulticaster).** They handle events that any other listener cannot regulate.

Implementing the Delegation Event Model in Java

- **Create an event listener interface** that defines the methods to call when an event is generated.
- **Implement the event listener interface** in one or more classes to handle the events.
- **Register the event listener** with the appropriate GUI component (s) using the addComponentListener or addMouseListener method.

- The events are categories into the following two categories:
- **The Foreground Events:** The foreground events are those events that require direct interaction of the user. These types of events are generated as a result of user interaction with the GUI component. For example, clicking on a button, mouse movement, pressing a keyboard key, selecting an option from the list, etc.
- **The Background Events :** The Background events are those events that result from the interaction of the end-user. For example, an Operating system interrupts system failure (Hardware or Software).

Inner class

- Listener classes are often designed to facilitate the creation of listener objects for some GUI object (e.g., a button, window, checkbox, etc...). In this context, separating the code for the Listener class from the code for the class of the related GUI object seems to, in a certain sense, work against the notion of the benefits of code encapsulation.
- Fortunately, in Java, within the scope of one class, we are allowed to define one or more *inner classes* (or *nested classes*). (*Technically, the compiler turns an inner class into a regular class file named: OuterClassName$InnerClassName.class*.
- Such inner classes can be defined inside the framing class, but outside its methods -- or it may even be defined inside a method, and they can reference data and methods defined in the outer class in which it nests.

```
public class OuterClass {
   private int data;

   public void myMethod() {
       //Do something
   }
   class InnerClass {
       public void myInnerMethod() {
           data++;                  //we can reference instance
                                    //variables of the outer class,
           myMethod();              //and we can call methods of
                                    //the outer class
       }
    }
}
```

Anonymous Inner Classes

- We can also create *Anonymous* inner classes (i.e., inner classes that have no name) to shorten the process of declaring an inner class and creating an instance of that class into one step.
- There are some restrictions on creating anonymous inner classes, however. In particular, an anonymous inner class:
- Must always extend a superclass or implement an interface
- Must implement all the abstract methods in the superclass or in the interface
- Always uses the no-arg constructor from its superclass to create an instance
  - Note, if the anonymous inner class implements an interface, the constructor used is Object()
- Is compiled into a class named *OuterClassName*$$*n.class*
  - So for example, if the outer class Test has two anonymous inner classes, these two classes are compiled into Test$$1.class and Test$$2.class

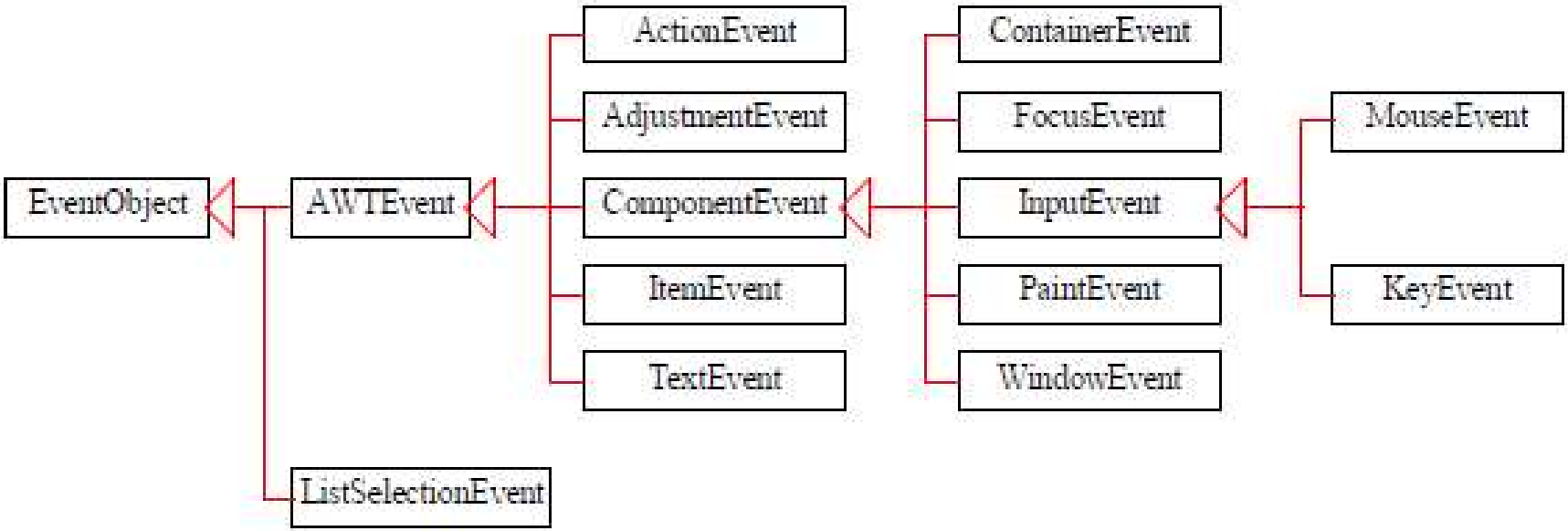
Inner Class vs. Nested Class

Feature	Inner Class	Static Nested Class
Association	Associated with an instance of the outer class	Not associated with an instance of the outer class
Access to Outer Class Members	Can access all members, including private	Can only access static members
Usage	Useful for event handling and encapsulation	Useful for utility classes related to the outer class without accessing instance-specific data

Alternatives to Inner Classes

- **Top-Level Classes**: Instead of creating an inner class, define a separate top-level class. This is useful when the class doesn’t need direct access to the outer class's private fields and methods. It also improves code readability and reusability.
- **Static Nested Classes**: If you don’t need an inner class to access non-static members of the outer class, you can use a static nested class. Static nested classes don’t hold a reference to the outer class instance, so they are more memory-efficient.
- **Anonymous Classes with Functional Interfaces**: For single-use implementations, especially for interfaces with one method (functional interfaces), use anonymous classes or lambda expressions. These are lightweight alternatives to inner classes and can be used inline.
- **Factory Pattern**: If you need controlled access to class instances and want to avoid inner classes, consider using the Factory Design Pattern. This approach helps create object instances without exposing the implementation details, keeping code modular and maintainable.

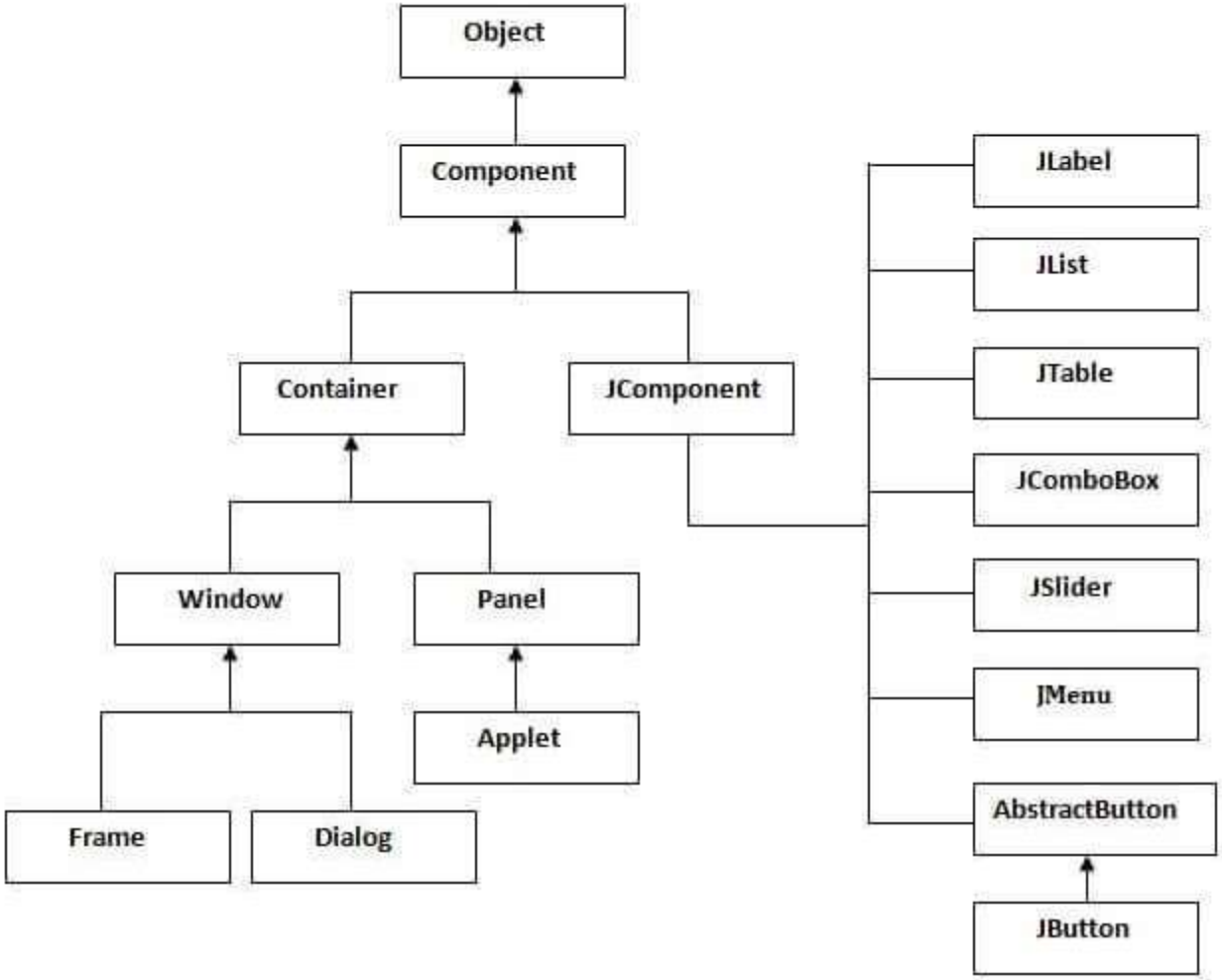
- different types of events for which one can listen, with many being subclasses of others



User Action	Event Type Generated	Source Object
Click a button	ActionEvent	JButton
Mouse button pressed, released, etc...	MouseEvent	Component
Key pressed, released, etc...	KeyEvent	Component
Window opened, closed, etc...	WindowEvent	Window
Press return on a text field	ItemEvent, Action Event	JTextField
Click a check box	ItemEvent, ActionEvent	JCheckBox
Click a radio button	ItemEvent, ActionEvent	JRadioButton
Selecting a new item from a combo box	ItemEvent, ActionEvent	JComboBox

Event Class	Listener Interface	Listener Methods (Handlers)
ActionEvent	ActionListener	actionPerformed(ActionEvent)
ItemEvent	ItemListener	itemStateChanged(ItemEvent)
WindowEvent	WindowListener	windowClosing(WindowEvent) windowOpened(WindowEvent) windowIconified(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent)
ContainerEvent	ContainerListener	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
MouseEvent	MouseListener	mousePressed(MouseEvent) mouseReleased(MouseEvent) mouseClicked(MouseEvent) mouseExited(MouseEvent) mouseEntered(MouseEvent)
KeyEvent	KeyListener	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTypeed(KeyEvent)

Hierarchy of Java Swing classes



Components of Swing Class the task's percentage

Class	Description
Component	A Component is the Abstract base class for about the non-menu user-interface controls of Java SWING. Components are representing an object with a graphical representation.
Container	A Container is a component that can container Java SWING Components
JComponent	A JComponent is a base class for all swing UI Components In order to use a swing component that inherits from JComponent, the component must be in a containment hierarchy whose root is a top-level Java Swing container.
JLabel	A JLabel is an object component for placing text in a container.
JButton	This class creates a labeled button.
JColorChooser	A JColorChooser provides a pane of controls designed to allow the user to manipulate and select a color.
JCheckBox	A JCheckBox is a graphical (GUI) component that can be in either an on-(true) or off-(false) state.
JRadioButton	The JRadioButton class is a graphical (GUI) component that can be in either an on-(true) or off-(false) state. in the group
JList	A JList component represents the user with the scrolling list of text items.
JComboBox	A JComboBox component is Presents the User with a show up Menu of choices.
JTextField	A JTextField object is a text component that will allow for the editing of a single line of text.
JPasswordField	A JPasswordField object it is a text component specialized for password entry.
JTextArea	A JTextArea object is a text component that allows for the editing of multiple lines of text.
ImageIcon	A ImageIcon control is an implementation of the Icon interface that paints Icons from Images
JScrollbar	A JScrollbar control represents a scroll bar component in order to enable users to Select from range values.
JOptionPane	JOptionPane provides set of standard dialog boxes that prompt users for a value or Something.
JFileChooser	A JFileChooser it Controls represents a dialog window from which the user can select a file.
JProgressBar	As the task progresses towards completion, the progress bar displays the tasks percentage on its completion.
JSlider	A JSlider this class is letting the user graphically (GUI) select by using a value by sliding a knob within a bounded interval.
	A JSpinner this class is a single line input where the field that lets the user select by using a number or an object

Java Swing Components and Containers

- Java Swing Framework contains a large set of these components which provide rich functionalities and allow high level of customization. All these components are lightweight components.
- Swing components are platform-independent, customizable, and offer extensive event handling.
- This class provides some common functionality like pluggable look and feel, support for accessibility, drag and drop, layout, etc.
- A container holds a group of components. It provides a space where a component can be managed and displayed. Containers are of two types:
- **Top level Containers**
  - It inherits Component and Container of AWT.
  - It cannot be contained within other containers.
  - Heavyweight.
  - Example: JFrame, JDialog, JApplet
- **Lightweight Containers**
  - It inherits JComponent class.
  - It is a general purpose container.
  - It can be used to organize related components together.
  - Example: JPanel

JApplet

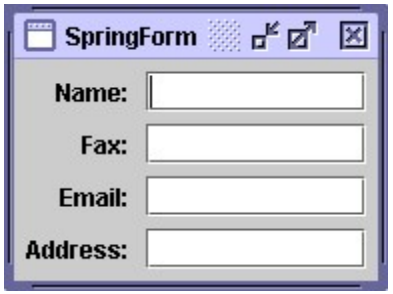
- A JApplet is a **graphical user interface application based on the Java programming language, specifically developed for web-oriented tasks.**
- Created as part of the **Java Foundation Classes (JFC)** included in package javax.swing, JApplets provide **a platform for designing interactive, rich media applications that can be embedded in a web page,** enabling the execution of Java bytecode on a client's browser.
- This significantly enhances the **user experience by delivering superior, dynamic content compared to static HTML** content traditionally provided by websites.
- The primary purpose of **JApplet is to allow developers to build and embed sophisticated, platform-independent applications directly onto web pages**.
- This can vastly expand the functionality of the page, ranging from simple animations and games to data visualization tools, interactive forms, and complex business applications.
- A JApplet expects to be embedded in a page and used in a viewing environment that provides it with resources. In all other respects, however, applets are just ordinary Panel objects.

Types of LayoutManager

- The Layout managers enable us to control the way in which visual components are arranged in the GUI forms by determining the size and position of components within the containers. There are 6 layout managers in Java

Types of LayoutManager

- SpringLayout is a flexible layout manager designed for use by GUI builders. It lets you specify precise relationships between the edges of components under its control. For example, you might define that the left edge of one component is a certain distance (which can be dynamically calculated) from the right edge of a second component.



Types of LayoutManager

- **GridBagLayout**: It is a powerful layout which arranges all the components in a **grid of cells and maintains the aspect ration of the object** whenever the container is resized. In this layout, cells may be different in size. It assigns a consistent horizontal and vertical gap among components. It allows us to specify a default alignment for components within the columns or rows.

Types of LayoutManager

- **GridLayout**: It arranges all the components in a grid of **equally sized cells**, adding them from the **left to right** and **top to bottom**. Only one component can be placed in a cell and each region of the grid will have the same size. When the container is resized, all cells are automatically resized. The order of placing the components in a cell is determined as they were added.

Types of LayoutManager

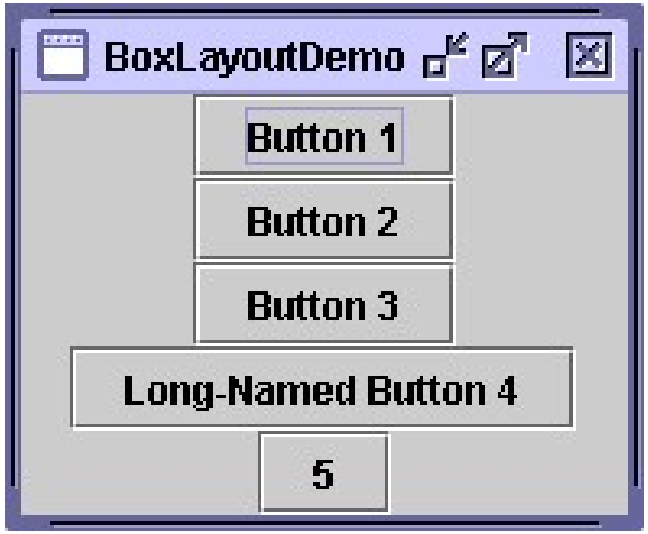
- **FlowLayout**: It arranges the components in a container like the words on a page. It fills the top line from **left to right and top to bottom**. The components are arranged in the order as they are added i.e. first components appears at top left, if the container is not wide enough to display all the components, it is wrapped around the line. Vertical and horizontal gap between components can be controlled. The components can be **left, center or right aligned.**

Types of LayoutManager

- **CardLayout**: It arranges two or more components having the same size. The components are **arranged in a deck**, where all the cards of the same size and the **only top card are visible at any time**. The first component added in the container will be kept at the top of the deck. The default gap at the left, right, top and bottom edges are zero and the card components are displayed either **horizontally or vertically.**

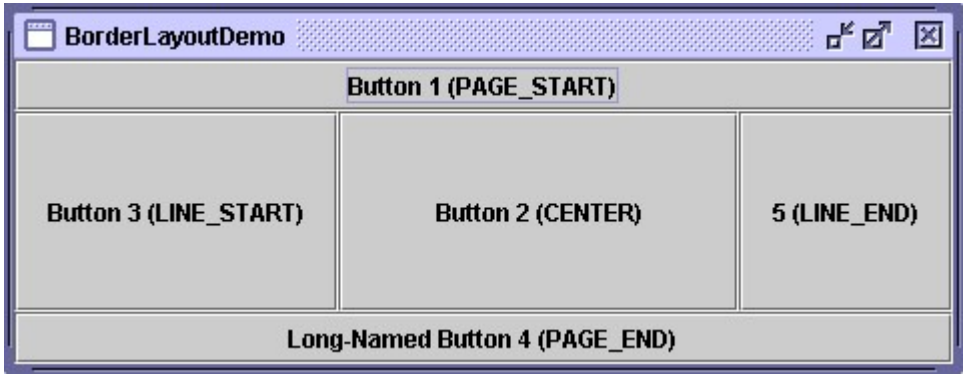
Types of LayoutManager

- **BoxLayout**: It arranges multiple components in either **vertically or horizontally**, but not both. The components are arranged from **left to right or top to bottom**. If the components are aligned **horizontally**, the height of all components will be the same and equal to the largest sized components. If the components are aligned **vertically**, the width of all components will be the same and equal to the largest width components.



Types of LayoutManager

- **BorderLayout**: It arranges all the components along the edges or the middle of the container i.e. **top, bottom, right and left** edges of the area. The components added to the top or bottom gets its preferred height, but its width will be the width of the container and also the components added to the left or right gets its preferred width, but its height will be the remaining height of the container. The components added to the center gets neither its preferred height or width. It covers the remaining area of the container.



Adapter class

- Java adapter classes *provide the default implementation of listener interfaces*. If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces. So it *saves code*.
- Pros of using Adapter classes:
- It assists the unrelated classes to work combinedly.
- It provides ways to use classes in different ways.
- It increases the transparency of classes.
- It provides a way to include related patterns in the class.
- It provides a pluggable kit for developing an application.
- It increases the reusability of the class.
- The adapter classes are found in **java.awt.event, java.awt.dnd** and **javax.swing.event** packages.