Java Programming

UNIT - V

by

Mr. K. Srikar Goud
Asst. Professor
Department of Information Technology

Unit V

GUI Programming with Swing , Event Handling, Applets

Swing

- Provides more powerful and flexible GUI components than AWT

- Origin of Swing
  - Limitations of AWT(Abstract Window Toolkit)
    - Heavyweight Components
    - Threatens the Java philosophy of “Write Once, Run Anywhere”

Limitations of AWT

- AWT supports limited number of GUI components.
- AWT components are heavy weight components.
- AWT components are developed by using platform specific code.
- AWT components behaves differently in different operating systems.

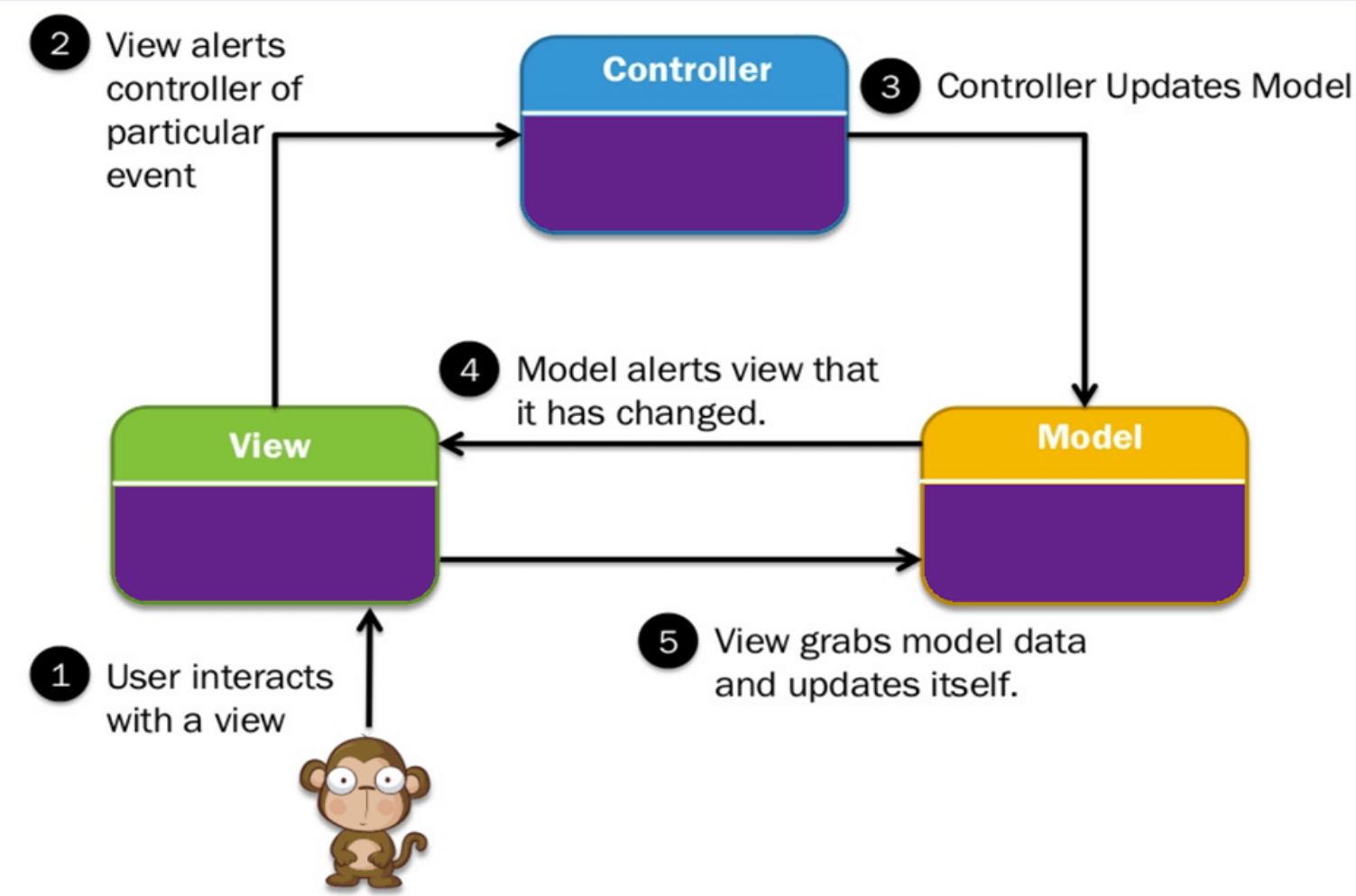
Features of Swing

- Swing Components Are Lightweight
- Swing Supports a Pluggable Look and Feel

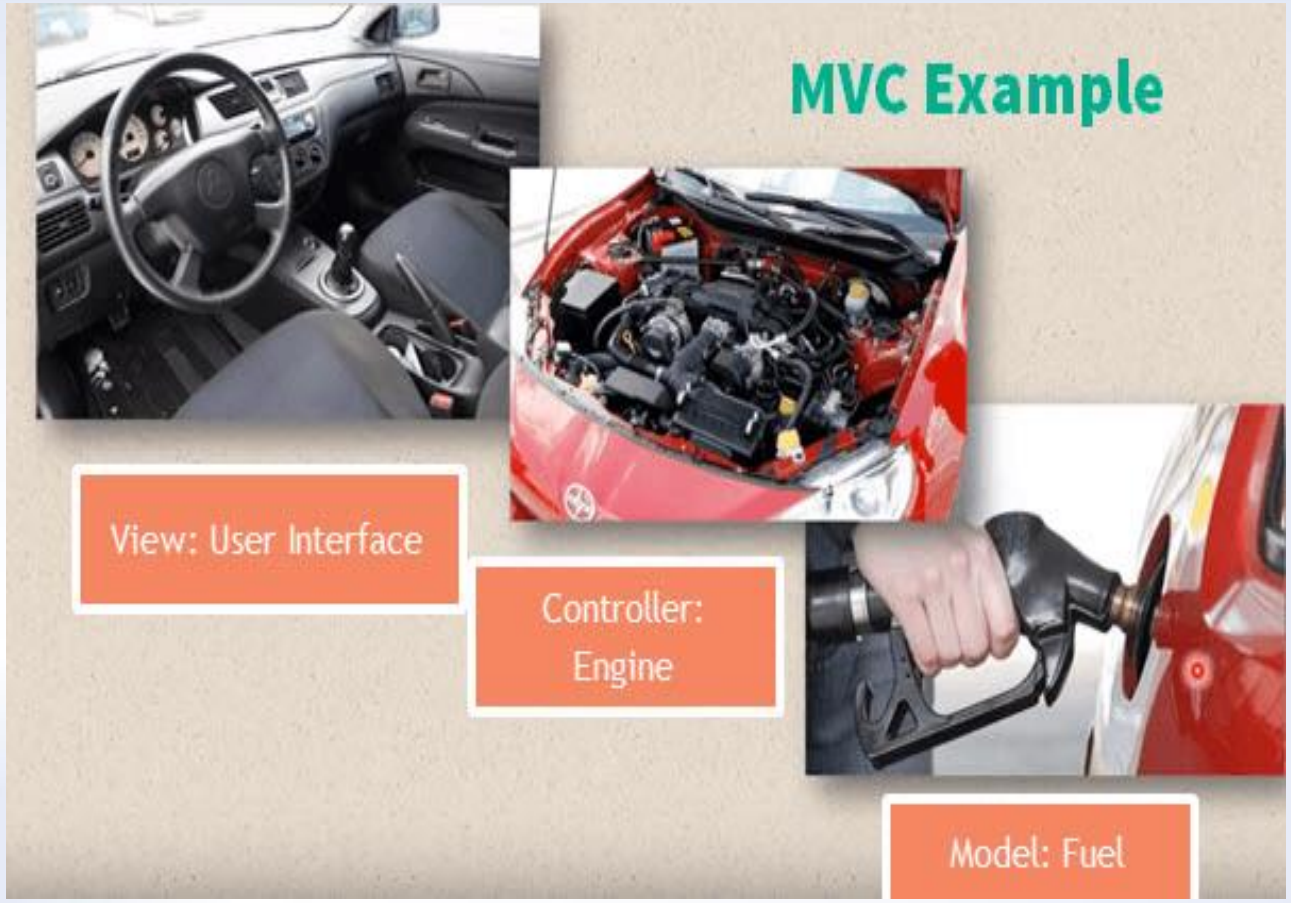
Model View Controller(MVC)

- The state information associated with the component (Model)
- The way that the component looks when rendered on the screen(View)
- The way that the component reacts to the user(Controller)

Model View Controller(MVC)



Model View Controller(MVC)



Model View Controller(MVC)

Swing’s Model View Controller(MVC)

- Swing uses a modified version of MVC that combines the view and

  the controller into a single logical entity called the ***UI delegate.***

- *Swing’s* approach is called either the ***Model-Delegate architecture*** *or the* ***Separable Model architecture***

Components and Containers

- A *component is an* independent visual control.
- A container is a special type of component that is designed to hold

  other components.

- Swing components are derived from the **JComponent class.**
- JComponent inherits the AWT classes Container and Component.
- **javax.swing package**
- JButton, JCheckBox, JRadioButton,JColorChooser, JMenu, JDialog,etc

Containers

- Top level Containers(Heavy weight)
  - JFrame, JApplet,JWindow, and JDialog.
  - do not inherit Jcomponent

- Lightweight containers.
  - Lightweight containers *do inherit* ***JComponent.***
  - ***JPanel***

Top level Container panes

- JRootPane

- The panes that comprise the root pane are called the
  - *Glass pane* - a transparent instance of **JPanel**
  - *Layered pane* - instance of **JLayeredPane**
  - *Content pane* - an opaque instance of **Jpanel**

Layout Managers

- Layout Managers help in arranging components in a particular manner in a frame or container.

- The following classes represent the layout managers in Java:
  - FlowLayout
  - BorderLayout
  - CardLayout
  - GridLayout
  - GridBagLayout
  - BoxLayout

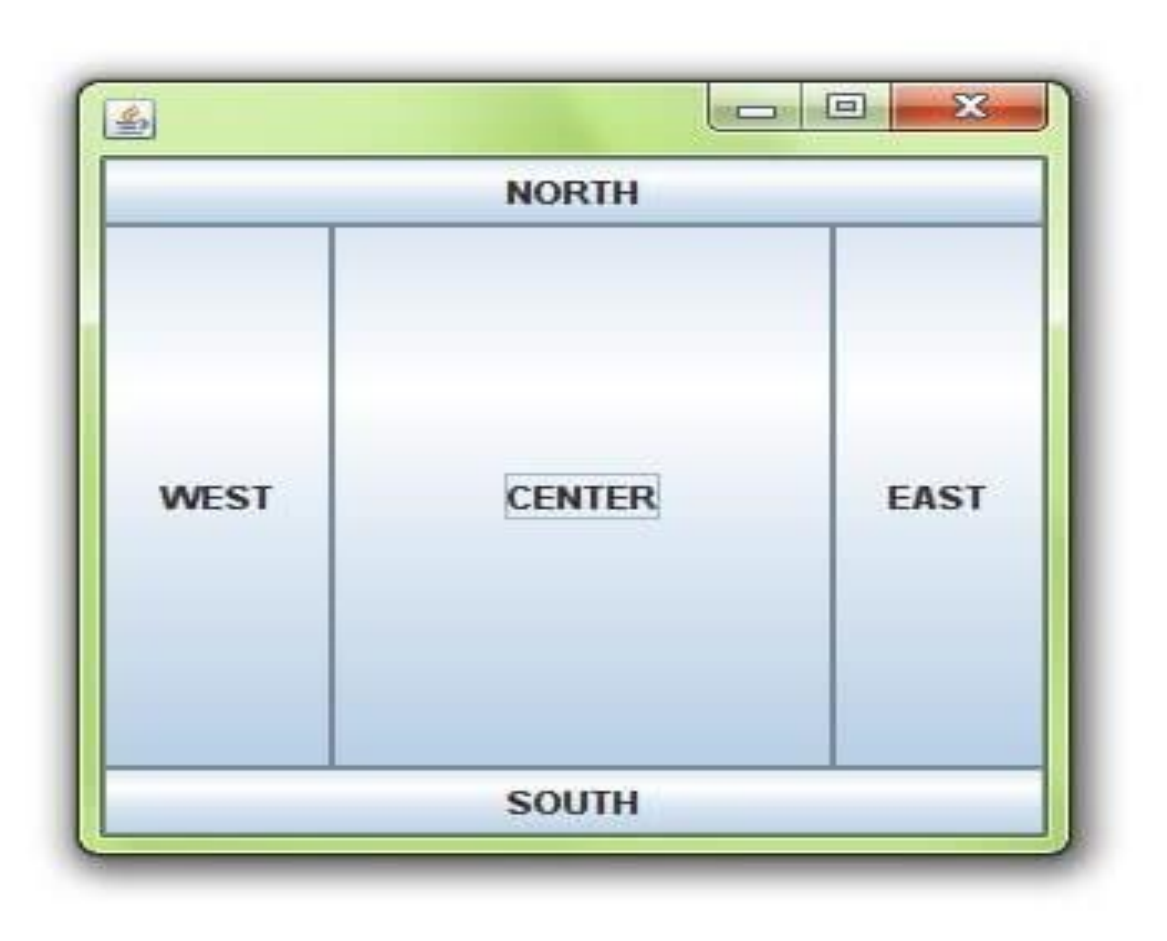
FlowLayout

- FlowLayout obj = new FlowLayout();
- FlowLayout obj = new FlowLayout(int alignment);
- FlowLayout obj = new FlowLayout(int alignment, int hgap, int vgap);
- Alignment can be specified as follows:
  - FlowLayout.LEFT
  - FlowLayout.RIGHT
  - FlowLayout.CENTER

BorderLayout

- BorderLayout obj = new BorderLayout();
- BorderLayout obj = new BorderLayout(int hgap, int vgap);
- While adding the components to the container the direction should be specified, as:
  - c.add(“North”,component)
  - c.add(component,BorderLayout.NORTH)
- Similarly other constants EAST, WEST, SOUTH, CENTER can be used

BorderLayout



CardLayout

- Treats each component as a card.
- Only one card is visible at a time, and the container acts as a stack of cards.
- CardLayout obj = new CardLayout();
- CardLayout· obj = new CardLayout(int hgap, int vgap);

- To add components to the container,
  - c.add(“cardname”,component)

CardLayout

- To retrieve the cards one by one, the following methods can be used:
  - void first (container) : to retrieve the first card.
  - void last (container) : to retrieve the last card.
  - void next (container) : to go to the next card.
  - void previous (container) : to go back to previous card.
  - void show (container, “ cardname") : to see a particular card with the name specified.

GridLayout

- To divide the container into a two-dimensional grid form that contains several rows and columns.

- GridLayout obj = new GridLayout();
- GridLayout obj = new GridLayout(int rows, int cols);
- GridLayout obj = new GridLayout(int rows, int cols, int hgap, int vgap);

GridBagLayout

- The components are arranged in rows and columns.
- This layout is more flexible as compared to other layouts since in this layout, the components can span more than one row or column and the size of the components can be adjusted to fit the display area

- GridBagLayout obj = new GridBagLayout();

- GridBagConstraints cons = new GridBagConstraints();

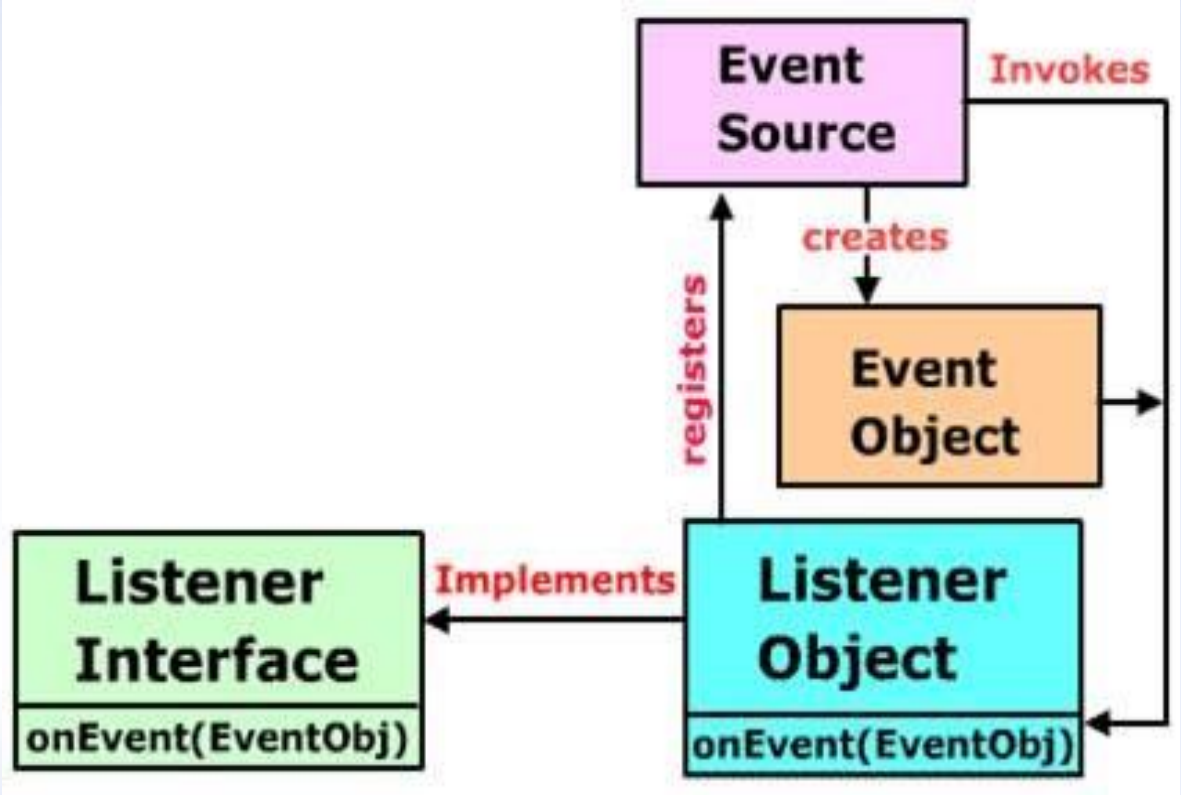
- GridBagConstraints cons new GridBagConstraints (int gridx, int gridy, int gridwidth, int gridheight, double weightx, double weighty, int anchor, int fill, Insets insets, int ipadx, int ipady);

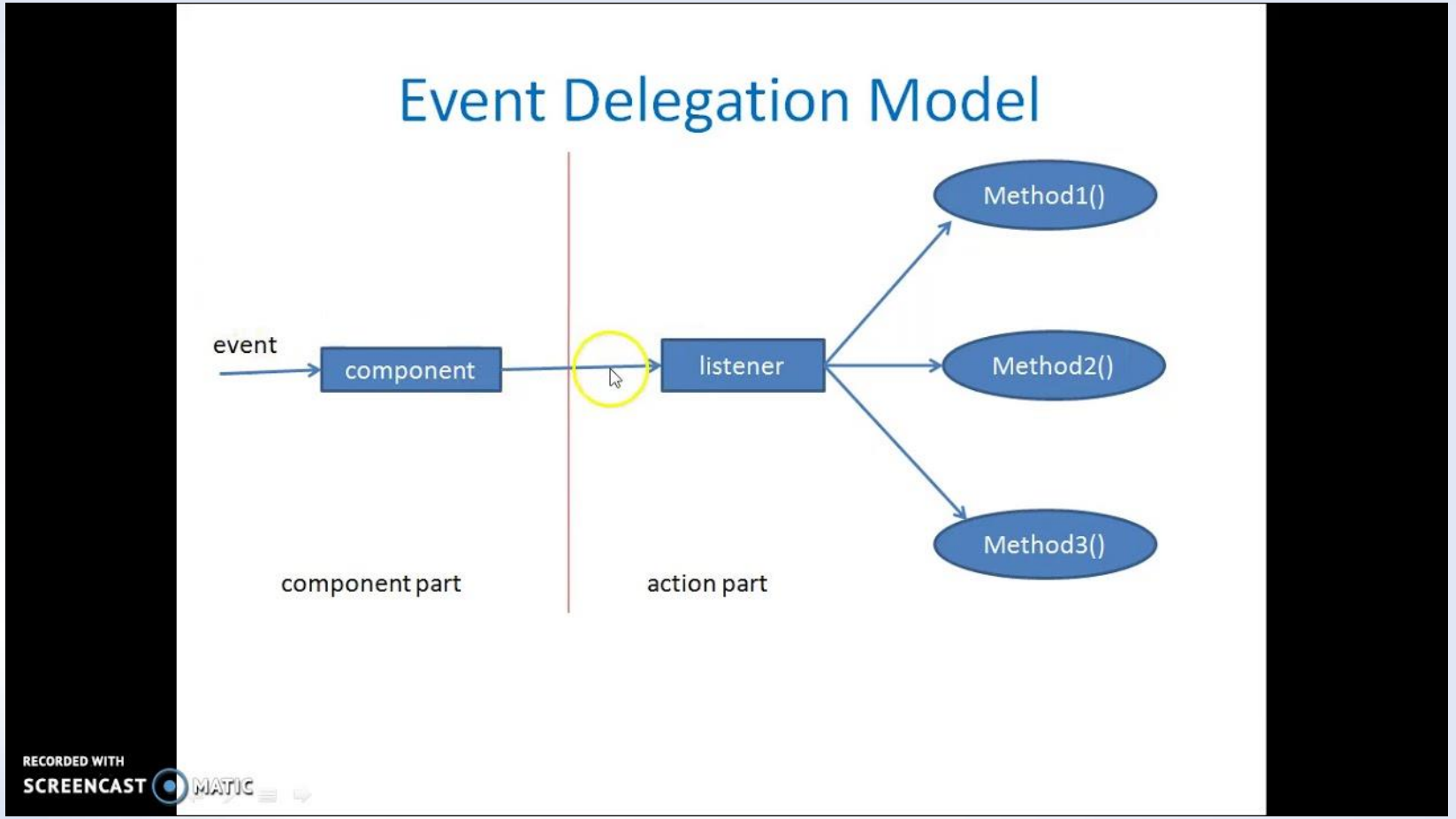
Delegation event model

- A *source* generates an event and sends it to one or more *listeners*.
- In this scheme, the listener simply waits until it receives an event.
- Once received, the listener processes the event and then returns.

- Listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them.

.


Event Source
Invokes
creates
registers
Event Object
Listener Interface
Implements
Listener Object
onEvent(EventObj)
onEvent(EventObj)


Event Delegation Model
event
component
listener
Method1()
Method2()
Method3()
component part
action part
RECORDED WITH
SCREENCAST MATIC

Events

- An *event* is an object that describes a state change in a source.
- It can be generated as a consequence of a person interacting with the elements in a graphical user interface.
- Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.

- Events may also occur that are not directly caused by interactions with a user interface.
- For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed.

Event sources

- A *source* is an object that generates an event.
- Sources may generate more than one type of event.
- A source must register listeners in order for the listeners to receive notifications about a specific type of event.
- Each type of event has its own registration method.
- General form is:

  public void add*Type*Listener(*Type*Listener *el*)

  Here, *Type* is the name of the event and *el* is a reference to the event listener.
- For example,

  1. The method that registers a keyboard event listener is called **addKeyListener()**.

  2. The method that registers a mouse motion listener is called **addMouseMotionListener( )**.

- When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as *multicasting* the event.
- Some sources may allow only one listener to register. The general form is:

  public void add*Type*Listener(*Type*Listener *el*) throws java.util.TooManyListenersException

  Here Type is the name of the event and *el* is a reference to the event listener.
- When such an event occurs, the registered listener is notified. This is known as *unicasting* the event.

- A source must also provide a method that allows a listener to unregister an interest in a specific type of event.
- The general form is:

  public void remove*Type*Listener(*Type*Listener *el*)

  Here, *Type* is the name of the event and *el* is a reference to the event listener.
- For example, to remove a keyboard listener, you would call **removeKeyListener( )**.

Event Listeners

- A *listener* is an object that is notified when an event occurs.
- Event has two major requirements.
  1. It must have been registered with one or more sources to receive notifications about specific types of events.
  2. It must implement methods to receive and process these notifications.
- The methods that receive and process events are defined in a set of interfaces found in **java.awt.event**.
- For example, the **MouseMotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved.

Event classes

- The Event classes that represent events are at the core of Java's event handling mechanism.
- Super class of the Java event class hierarchy is **EventObject**, which is in **java.util**. for all events.
- Constructor is :

EventObject(Object *src*)

Here, *src* is the object that generates this event.

- **EventObject** contains two methods: **getSource( )** and **toString( )**.
- 1. The **getSource( )** method returns the source of the event. General form is : Object getSource( )
- **2.** The **toString( )** returns the string equivalent of the event.

- EventObject is a superclass of all events.
- AWTEvent is a superclass of all AWT events that are handled by the delegation event model.
- The package **java.awt.event** defines several types of events that are generated by various user interface elements.

Event Classes in java.awt.event

- ActionEvent: Generated when a button is pressed, a list item is double clicked, or a menu item is selected.
- AdjustmentEvent: Generated when a scroll bar is manipulated.
- ComponentEvent: Generated when a component is hidden, moved, resized, or becomes visible.
- ContainerEvent: Generated when a component is added to or removed from a container.
- FocusEvent: Generated when a component gains or loses keyboard focus.

- InputEvent: Abstract super class for all component input event classes.
- ItemEvent: Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
- KeyEvent: Generated when input is received from the keyboard.
- MouseEvent: Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
- TextEvent: Generated when the value of a text area or text field is changed.
- WindowEvent: Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

AdjustmentEvent Class

- 5 types of Adjustment Events:
  - BLOCK_DECREMENT
  - BLOCK_INCREMENT
  - TRACK
  - UNIT_DECREMENT
  - UNIT_INCREMENT

- int getAdjustmentType( )
- int getValue( )

ComponentEvent Class

- ComponentEvent(Component *src, int type)*
- Component getComponent( )
- Types of Component Events:
  - COMPONENT_HIDDEN
  - COMPONENT_MOVED
  - COMPONENT_RESIZED
  - COMPONENT_SHOWN

ContainerEvent Class

- ContainerEvent(Component *src, int type, Component comp)*

- Container getContainer( )
- Component getChild( )

- Types of Container Events:
  - COMPONENT_ADDED
  - COMPONENT_REMOVED

FocusEvent Class

- FocusEvent(Component *src, int type)*
- FocusEvent(Component *src, int type, boolean temporaryFlag)*
- FocusEvent(Component *src, int type, boolean temporaryFlag, Component other)*

- Component getOppositeComponent( )
- boolean isTemporary( )

- Types of Focus Events:
  - FOCUS_GAINED
  - FOCUS_LOST

ItemEvent Class

- ItemEvent(ItemSelectable *src, int type, Object entry, int state)*

- Object getItem( )
- int getStateChange( )

- Types of Events:
  - DESELECTED
  - SELECTED

ActionEvent Class

- ActionEvent(Object *src, int type, String cmd)*
- ActionEvent(Object *src, int type, String cmd, int modifiers)*
- ActionEvent(Object *src, int type, String cmd, long when, int modifiers)*

- String getActionCommand( )
- int getModifiers( )
- long getWhen( )

- ALT_MASK, CTRL_MASK,META_MASK, and SHIFT_MASK
- Types of Events:
  - ACTION_PERFORMED

InputEvent Class

- boolean isAltDown( )
- boolean isAltGraphDown( )
- boolean isControlDown( )
- boolean isMetaDown( )
- boolean isShiftDown( )

- int getModifiers( )
- int getModifiersEx( )

KeyEvent Class

- KeyEvent(Component *src, int type, long when, int modifiers, int code, char ch)*

- Types of Events:
  - KEY_PRESSED
  - KEY_RELEASED
  - KEY_TYPED

MouseEvent Class

- MouseEvent(Component *src, int type, long when, int modifiers,* int *x, int y, int clicks, boolean triggersPopup)*

- int getX( )
- int getY( )
- int getClickCount( )
- boolean isPopupTrigger( )

- Types of Events:
  - MOUSE_CLICKED ,MOUSE_DRAGGED ,MOUSE_ENTERED ,MOUSE_EXITED,MOUSE_MOVED, MOUSE_PRESSED, MOUSE_RELEASED ,MOUSE_WHEEL

MouseWheelEvent Class

- MouseWheelEvent(Component *src, int type, long when, int modifiers,*

  int *x, int y, int clicks, boolean triggersPopup,* int *scrollHow, int amount, int count)*

- int getScrollType( )
- int getScrollAmount( )

- Types of Events:
  - WHEEL_BLOCK_SCROLL
  - WHEEL_UNIT_SCROLL

WindowEvent Class

- WindowEvent(Window *src, int type)*
- WindowEvent(Window *src, int type, int fromState, int toState)*

- Window getWindow( )
- int getOldState( )
- int getNewState( )

- Types of Events:
  - WINDOW_CLOSED,WINDOW_CLOSING, WINDOW_GAINED_FOCUS, WINDOW_LOST_FOCUS, WINDOW_OPENED etc.

Event Listener Interfaces

- ActionListener
- AdjustmentListener
- ComponentListener
- ContainerListener
- FocusListener
- ItemListener
- KeyListener
- MouseListener
- MouseMotionListener
- MouseWheelListener
- TextListener
- WindowFocusListener
- WindowListener

Applets

- *Applets* are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a Web document.
- After an applet arrives on the client, it has limited access to resources, so that it can produce an arbitrary multimedia user interface and run complex computations without introducing the risk of viruses or breaching data integrity.

Applets

- An applet cannot run any executable program in the client system.
- An applet cannot communicate with any server other than the server from which it is downloaded.
- An applet cannot read from a fIle or write into a fIle that belongs to the client computer.
- An applet cannot find sensitive data like user's login name;email-address, etc.

Applet

- Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer.

- An appletviewer executes your applet in a window. This is generally the fastest and easiest way to test an applet.

- Deploying an applet is to specify the applet directly in an HTML file.

```
//First.java
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{
public void paint(Graphics g){
g.drawString("welcome",150,150);
}
 }
```

```
<html>
<body>
<applet code="First.class" width="300" height="300">
</applet>
</body>
</html>
```

```
//First.java
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{
   public void paint(Graphics g){
   g.drawString("welcome to applet",150,150);
}
}
/*
<applet code="First" width="300" height="300">
</applet>
*/
```

Differences between applets and applications

- Java can be used to create two types of programs: applications and applets.
- An *application* is a program that runs on your computer, under the operating system of that Computer(i.e an application created by Java is more or less like one created using C or C++).
- When used to create applications, Java is not much different from any other computer language.
- An *applet* is an application designed to be transmitted over the Internet and executed by a Java-compatible Web browser.
- An applet is actually a tiny Java program, dynamically downloaded across the network, just like an image, sound file, or video clip.

- The important difference is that an applet is an *intelligent program*, not just an animation or media file(i.e an applet is a program that can react to user input and dynamically change—not just run the same animation or sound over and over.)
- Applications require main method to execute.
- Applets do not require main method.
- Java's console input is quite limited
- Applets are graphical and window-based.

Life cycle of an applet

- Applets life cycle includes the following methods
    1. **init( )**
    2. **start( )**
    3. **paint( )**
    4. **stop( )**
    5. **destroy( )**
- When an applet begins, the AWT calls the following methods, in this sequence:

  init( )

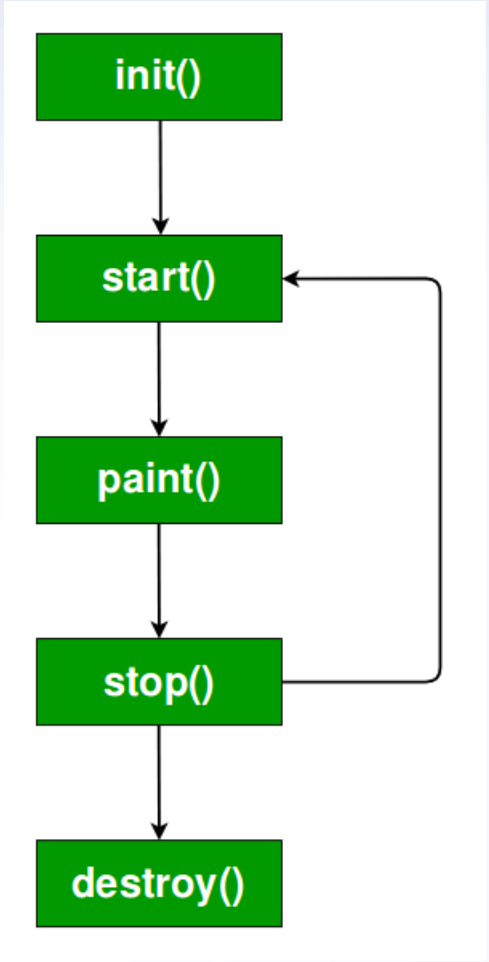
  start( )

  paint( )
- When an applet is terminated, the following sequence of method calls takes place:

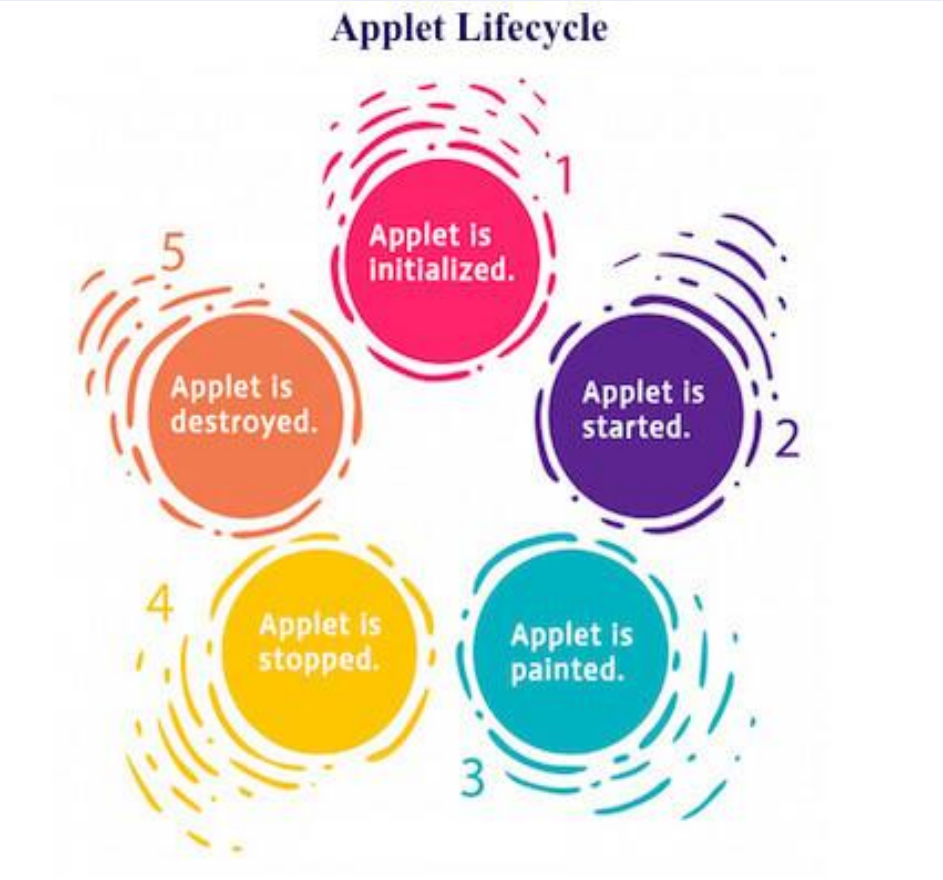
  stop( )

  destroy( )

Life cycle of an applet



Life cycle of an applet



Life cycle of an applet

- **public void init():** is used to initialized the Applet. It is invoked only once.
- **public void start():** is invoked after the init() method or browser is maximized. It is used to start the Applet.
- **public void paint(Graphics g):** is used to paint the Applet. It provides Graphics class object that can be used for drawing oval, rectangle, arc etc. The paint( ) method is called each time applet's output must be redrawn. paint( ) is also called when the applet begins execution
- **public void stop():** is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.
- **public void destroy():** is used to destroy the Applet. It is invoked only once.

Adapter classes

- Java provides a special feature, called an *adapter class*, that can simplify the creation of event handlers.
- An adapter class provides an empty implementation of all methods in an event listener interface.
- Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface.
- You can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested.

- Adapter classes in **java.awt.event are**.

Adapter Class	**Listener Interface**
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

Inner classes

- Inner classes, which allow one class to be defined within another.
- An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.
- An inner class is fully within the scope of its enclosing class.
- An inner class has access to all of the members of its enclosing class, but the reverse is not true.
- Members of the inner class are known only within the scope of the inner class and may not be used by the outer class

Painting in Swing

- The AWT class Component defines a method called paint( ) that is used to draw output directly to the surface of a component
- Swing uses a bit more sophisticated approach to painting that involves three distinct methods: paintComponent( ), paintBorder( ), and paintChildren( ).

- protected void paintComponent(Graphics *g)*

Compute the Paintable Area

- To obtain the border width
  - Insets getInsets( )
- The inset values can be obtained by using these fields:
  - int top;
  - int bottom;
  - int left;
  - int right;
- The width and height of the component can be obtained by calling getWidth( ) and getHeight( ) on the component

JLabel and ImageIcon

- JLabel can be used to display text and/or an icon. It is a passive component in that it does not respond to user input.
- JLabel(Icon *icon)*
- JLabel(String *str)*
- JLabel(String *str, Icon icon, int align)*
- Icon getIcon( )
- String getText( )
- void setIcon(Icon *icon)*
- void setText(String *str)*

JTextField

- JTextField allows you to edit one line of text.

- JTextField(int *cols)*
- JTextField(String *str, int cols)*
- JTextField(String *str)*

- To obtain the text currently in the text field, call **getText( ).**

Swing Buttons

- JButton, JToggleButton, JCheckBox, and JRadioButton

- All are subclasses of the AbstractButton class, which extends Jcomponent

- **JButton:**
- JButton(Icon *icon)*
- JButton(String *str)*
- JButton(String *str, Icon icon)*

JToggleButton

- A toggle button looks just like a push button, but it acts differently because it has two states: pushed and released.
- JToggleButton is a superclass for two other Swing components: JCheckBox and JRadioButton
- JToggleButton(String *str)*
- JToggleButton also generates an item event.
- To handle item events, we implement the ItemListener interface
- Object getItem( )
- boolean isSelected( )

JRadioButton

- Radio buttons are a group of mutually exclusive buttons, in which only one button can be selected at any one time.
- JRadioButton(String *str)*
- Radio buttons must be configured into a group. A button group is created by the **ButtonGroup class.**

  void add(AbstractButton *ab)*

- getActionCommand( )
- getSource( ) on the ActionEvent object
- isSelected( )

JTabbedPane

- It manages a set of components by linking them with tabs.

- void addTab(String *name, Component comp)*

1. Create an instance of **JTabbedPane.**
2. Add each tab by calling **addTab( ).**
3. Add the tabbed pane to the content pane.

JScrollPane

- JScrollPane is a lightweight container that automatically handles the scrolling of another component.
- The viewable area of a scroll pane is called the *viewport.*
- JScrollPane will dynamically add or remove a scroll bar as needed**.**
- JScrollPane(Component *comp)*

1.Create the component to be scrolled.

2. Create an instance of JScrollPane, passing to it the object to scroll.

3. Add the scroll pane to the content pane.

JList

- It supports the selection of one or more items from a list.
- JList(E[ ] *items)*
- JList generates a ListSelectionEvent when the user makes or changes a selection.
- ListSelectionListener specifies only one method, called valueChanged( )

void valueChanged(ListSelectionEvent *le)*

void setSelectionMode(int *mode)*

- SINGLE_SELECTION
- SINGLE_INTERVAL_SELECTION
- MULTIPLE_INTERVAL_SELECTION

int getSelectedIndex( )

E getSelectedValue( )

JComboBox

- A combination of a text field and a drop-down list

- JComboBox(E[ ] *items)*
- void addItem(E *obj)*
- Object getSelectedItem( )

- JComboBox generates an action event and Item event

JMenu

- A menu represents a group of items or options for the user to select from.

- JMenuBar mb=new JMenuBar();

- JMenu file=new JMenu();
- mb.add(file);

- JMenuItem op=new JMenuItem(“Open”);
- file.add(op);