Object Oriented Programming using Java

Prepared By:
Suyel, PhD
Assistant Professor
Dept. of CSE, NIT Patna

Outline

1. Event Handling
2. Source and Listener
3. Event Class
4. Sources of Event
5. Event Listener Interface
6. Registration Model
7. Using the Delegation Event Model

Event Handling

- **What is an Event?**
  - Change in the state of a object is known as event. It describes the change in state of event.
  - Events are generated, when a user interacts with the GUI components.
  - Example: Clicking on a mouse or button that causes an event
- **Type of Events**
  - **Foreground Event:** These events need direct interaction of user. These are generated, when a user directly interact with GUI.
  - **Background Event:** These events need interaction of end user. For example, operating system interrupt, hardware or software failure

Event Handling (Cont…)

Event Handling

- Event handling is the technique that manages the event and decides what should happen, if an event occurs.
- This technique has the code that is known as event handler.
- Java uses Delegation Event Model to handle events consisting of two participants:
  - Source
  - Listener

Source and Listener

Source:

- The source is an object on which event occurs.
- This occurs when the internal state of that object changes.
- It is sole responsible to provide information of the occurred event to the handler.
- 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. Here is the general form:
  - 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, the method that registers a keyboard event listener is called addKeyListener( ).

Source and Listener (Cont…)

Source (Cont…):

- 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 of such a method is this:

```
public void add<Type>Listener(<Type>Listener el)
    throws java.util.TooManyListenersException
```

- When the above condition 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 of such a method is:

```
public void remove<Type>Listener(<Type>Listener el)
```

Source and Listener (Cont…)

Listener

- A listener is an object that is notified, when an event occurs.
- It must have been registered with one or more sources to receive notifications about specific types of events.
- Listener waits until it gets an event.
- Listener must implement methods to receive and process these notifications.
- java.awt.event package provides many Event Classes and Listener Interfaces for event handling. **Some of them** are shown in the next slide.

Source and Listener (Cont…)

Event Class	Listener Interface
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Event Class

- The most widely used events are those defined by the AWT and those defined by Swing.
- At the root of the Java event class hierarchy is EventObject, which is in java.util. It is the superclass for all events.
- It has one constructor is shown here:
  EventObject(Object *src*) //*src* is the object that generates this event
- EventObject has two methods: getSource( ) and toString( ).
  - **getSource( ):** getSource() method returns the source of the event.
    Object getSource( )
  - **toString( ):** This method returns a string representation of the object.
    Object toString( )

Event Class (Cont…)

- AWTEvent is the superclass of all AWT events that are handled by the delegation event model.
- The AWTEvent class is defined within the java.awt package, and it is a subclass of EventObject.
- Its getID( ) method can be used to determine the type of the event. The signature of this method is given below:

```
int getID( )
```

- **Some event classes** are shown in the next slide with description.

Event Class (Cont…)

Event Class	Description
ActionEvent	Generated, when a button is pressed, a list item is double-clicked or a menu item is selected.
MouseEvent	Generated, when the mouse is dragged, moved, clicked, pressed or released. It also generated, when the mouse enters or exits a component.
MouseWheelEvent	Generated, when the mouse wheel is moved.
KeyEvent	Generated, when input is received from the keyboard.
ItemEvent	Generated, when a check box or list item is clicked. It also occurs, when a choice selection is made or a checkable menu item is selected or deselected.
TextEvent	Generated, when the value of a text area or text field is changed.
AdjustmentEvent	Generated, when a scroll bar is manipulated.
InputEvent	Abstract superclass for all component input event classes.

Event Class (Cont…)

ActionEvent Class:

- An ActionEvent is generated, when a button is pressed, a list item is double-clicked or a menu item is selected.
- It defines four integer constants that can be used to identify any modifiers associated with an action event: ALT_MASK, CTRL_MASK, META_MASK and SHIFT_MASK.
- In addition, there is an integer constant, ACTION_ PERFORMED that can be used to identify action events. There are three constructors:

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*)

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type* and its command string is *cmd*. The argument modifiers indicates the modifier keys. The *when* parameter specifies, when the event occurred.

Event Class (Cont…)

ActionEvent Class (Cont…):

- **getActionCommand( ):** This method returns the command name for invoking ActionEvent object. Its form is given below:

  String getActionCommand( )

  For example, when a button is pressed, an action event is generated that has a command name same as the label on that button.

- **getModifiers( ):** It returns a value that indicates which modifier keys (ALT, CTRL, META and/or SHIFT) were pressed, when the event was generated. Its form is given below:

  int getModifiers( )

- **getWhen( ):** This method returns the time, when the event took place. This is called the event's *timestamp*. Its form is given below:
  long getWhen( )

Event Class (Cont…)

AdjustmentEvent Class:

- An AdjustmentEvent is generated by a scroll bar.
- This class defines integer constants that can be used to identify them.

Integer Constant	Description
BLOCK_DECREMENT	The user clicked inside the scroll bar to decrease its value.
BLOCK_INCREMENT	The user clicked inside the scroll bar to increase its value.
TRACK	The slider was dragged.
UNIT_DECREMENT	The button at the end of the scroll bar was clicked to decrease its value.
UNIT_INCREMENT	The button at the end of the scroll bar was clicked to increase its value.

- There is an integer constant, ADJUSTMENT_VALUE_CHANGED that indicates that a change has occurred.

Event Class (Cont…)

AdjustmentEvent Class (Cont…):

- There is one AdjustmentEvent constructor:

  AdjustmentEvent(Adjustable *src*, int *id*, int *type*, int *data*)

  Here, *id* specifies the event. The type of the adjustment is specified by *type*, and its associated data is *data*.

- **getAdjustable( ):** It returns the object that generated the event.

  Adjustable getAdjustable( )

- **getAdjustmentType( ):** This method is used to obtain the type of the adjustment event and it returns one of the constants.

  getAdjustmentType( )

- **getValue( ):** The amount of the adjustment is obtained by this method.

  int getValue( )

Event Class (Cont…)

ComponentEvent Class:

- A ComponentEvent is generated, when the size, position, or visibility of a component is changed.
- There are 4 types of component events. ComponentEvent class defines integer constants that can be used to identify them.

Integer Constant	Description
COMPONENT_HIDDEN	The component was hidden.
COMPONENT_MOVED	The component was moved.
COMPONENT_RESIZED	The component was resized.
COMPONENT_SHOWN	The component became visible.

- There is one constructor:

ComponentEvent(Component *src*, int *type*) //Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*.

Event Class (Cont…)

ComponentEvent Class (Cont…):

- **ComponentEvent is the superclass either directly or indirectly of many event, namely ContainerEvent, FocusEvent, KeyEvent, MouseEvent and WindowEvent.**

- **getComponent( ):** This method returns the component that generated the event. It is shown here:

  Component getComponent( )

Event Class (Cont…)

ContainerEvent Class:

- It is caused, when a component is added to or removed from a container.
- There are two types of container events and integer constants are used to identify them: COMPONENT_ADDED and COMPONENT_REMOVED. It has one constructor:

  ContainerEvent(Component *src*, int *type*, Component *comp*) // Here, the component that has been added to or removed from the container is *comp*.

- **getContainer( ):** This method is used to obtain a reference to the container that generated this event.

  Container getContainer( )

- **getChild( ):** It returns a reference to the component that was added to or removed from the container.

  Component getChild( )

Event Class (Cont…)

FocusEvent Class:

- A event is generated, when a component gains or loses input focus.
- These events are identified by the integer constants: FOCUS_GAINED and FOCUS_LOST. This class is a subclass of ComponentEvent.
- There are 3 constructors:

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

  Here, *src* is a reference to the component that generated this event. The argument *temporaryFlag* is set to true, if the focus event is temporary. The other component involved in the focus change, called the *opposite component*, is passed in *other*.

Event Class (Cont…)

FocusEvent Class (Cont…):

- **getOppositeComponent( ):** This method is used to determine the other component by calling:

  Component getOppositeComponent( )

- **isTemporary( ):** This method indicates, if this focus change is temporary. Its form is shown here:

  boolean isTemporary( )

  The method returns true, if the change is temporary. Otherwise, it returns false.

Event Class (Cont…)

InputEvent Class:

- The abstract class InputEvent is a subclass of ComponentEvent.
- It is the superclass for the component input events. Its subclasses are KeyEvent and MouseEvent.
- InputEvent defines eight integer constants that represent any modifiers, such as the control key being pressed. These modifiers are:

1. ALT_DOWN_MASK
2. BUTTON2_DOWN_MASK
3. META_DOWN_MASK
4. ALT_GRAPH_DOWN_MASK
5. BUTTON3_DOWN_MASK
6. SHIFT_DOWN_MASK
7. BUTTON1_DOWN_MASK
8. CTRL_DOWN_MASK

Event Class (Cont…)

InputEvent Class (Cont…):

- If a modifier was pressed at the time, when an event is generated, we use the isAltDown( ), isAltGraphDown( ), isControlDown( ), isMetaDown( ), and isShiftDown( ) methods to test. The forms are:

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

- **getModifiers( ):** We can obtain a value that contains all of the original modifier flags by calling this method. It is shown below:

```
int getModifiers( )
```

Event Class (Cont…)

ItemEvent Class:

- An ItemEvent is generated, when a check box or a list item is clicked or when a checkable menu item is selected or deselected.
- There are two types of item events and one additional integer constants, which are identified by the following integer constants:

DESELECTED	//The user deselected an item
SELECTED	//The user selected an item
ITEM_STATE_CHANGED	//Signifies the change of state

- It has one constructor:

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

Here, *src* is a reference to the component that generated this event. This might be a list or choice element. Type is the *type* of event. The specific item that generated the item event is passed in *entry*. The current state of that item is in *state*.

Event Class (Cont…)

ItemEvent Class (Cont…):

- **getItem( ):** This method can be used to obtain a reference to the item that generated an event. Its signature is shown below:

  Object getItem( )

- **getItemSelectable( ):** This method can be used to obtain a reference to the ItemSelectable object that generated an event.

  ItemSelectable getItemSelectable( )

  Lists and choices are examples of user interface elements that implement the ItemSelectable interface.

- **getStateChange( ):** This method returns the state change (SELECTED or DESELECTED) for the event. It is shown below:

  int getStateChange( )

Event Class (Cont…)

KeyEvent Class:

- A KeyEvent is generated, when keyboard input occurs. KeyEvent is a subclass of InputEvent.
- There are 3 types of key events, which are identified by these integer constants: KEY_PRESSED, KEY_RELEASED and KEY_TYPED. The last event occurs only when a character is generated.
- There are many other integer constants. For example, VK_0 through VK_9 and VK_A through VK_Z define the ASCII equivalents of the numbers and letters. Here are some others:

VK_ALT	VK_DOWN	VK_LEFT	VK_RIGHT
VK_CANCEL	VK_ENTER	VK_PAGE_DOWN	VK_SHIFT
VK_CONTROL	VK_ESCAPE	VK_PAGE_UP	VK_UP

Here, VK constants specify *virtual key codes* and are independent of any modifiers, such as control, shift, or alt.

Event Class (Cont…)

KeyEvent Class (Cont…):

- There is one constructor:

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

  Here, *src* is a reference to the component that generated the event. The *modifiers* argument indicates, which modifiers were pressed when this key event occurred. The virtual key code is passed in *code*. The character equivalent (if one exists) is passed in *ch*.

  If no valid character exists, then *ch* contains CHAR_UNDEFINED. For KEY_TYPED events, *code* contains VK_UNDEFINED.

- getKeyChar( ) and getKeyCode( ): These return the character that was entered and the key code, respectively. Their general forms are:

  char getKeyChar( )
  int getKeyCode( )

Event Class (Cont…)

MouseEvent Class:

- There are eight types of mouse events. The MouseEvent class defines the following integer constants:

MOUSE_CLICKED	The user clicked the mouse.
MOUSE_DRAGGED	The user dragged the mouse.
MOUSE_ENTERED	The mouse entered a component.
MOUSE_EXITED	The mouse exited from a component.
MOUSE_MOVED	The mouse moved.
MOUSE_PRESSED	The mouse was pressed.
MOUSE_RELEASED	The mouse was released.
MOUSE_WHEEL	The mouse wheel was moved.

- MouseEvent is a subclass of InputEvent. Here, is one of its constructors:

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

Here, the *modifiers* indicates, which modifiers were pressed when a mouse event occurred. The coordinates of the mouse are passed in *x* and *y*. The click count is passed in *clicks*. The *triggersPopup* flag indicates, if this event causes a pop-up menu.

Event Class (Cont…)

MouseEvent Class (Cont…):

- **getX( ) and getY( ):** These two methods return the X and Y coordinates of the mouse within the component. Their forms are:

  int getX( ) and int getY( ), respectively.

- **getPoint( ):** It returns a Point object that contains the X and Y coordinates in its integer members: *x* and *y*. It is shown below:

  Point getPoint( )

- **translatePoint( ):** It changes the location of the event. Here, the arguments *x* and *y* are added to the coordinates of the event. Its form:

  void translatePoint(int *x*, int *y*)

- **getClickCount( ):** It returns the number of mouse clicks for the event.

  int getClickCount( )

Event Class (Cont…)

MouseEvent Class (Cont…):

- **isPopupTrigger( ):** This method tests, if this event causes a pop-up menu to appear on this platform. Its form is shown below:

```
boolean isPopupTrigger( )
```

- **getButton( ):** It returns a value that represents the button that caused the event. The return value is one of these constants defined by MouseEvent: NOBUTTON, BUTTON1, BUTTON2 and BUTTON3.

```
int getButton( )
```

- There are three methods of MouseEvent that obtain the coordinates of the mouse relative to the screen rather than the component. They are:

```
Point getLocationOnScreen( )    //Returns a Point object that has both the coordinate
int getXOnScreen( )             //Return the indicated coordinate
int getYOnScreen( )             //Return the indicated coordinate
```

Event Class (Cont…)

MouseWheelEvent Class:

- MouseWheelEvent class encapsulates a mouse wheel event. It is a subclass of MouseEvent.
- If a mouse has a wheel, it is located between the left and right buttons. Mouse wheels are used for scrolling.
- It defines these two integer constants:

WHEEL_BLOCK_SCROLL	A page-up or page-down scroll event occurred.
WHEEL_UNIT_SCROLL	A line-up or line-down scroll event occurred.

- The constructor is shown here:

MouseWheelEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *x*, int *y*, int *clicks*, boolean *triggersPopup*, int *scrollHow*, int *amount*, int *count*)

The scrollHow value must be either WHEEL_UNIT_SCROLL or WHEEL_BLOCK_SCROLL. The number of units to scroll is passed in amount. The count parameter indicates the number of rotational units that the wheel moved.

Event Class (Cont…)

MouseWheelEvent Class (Cont…):

- **getWheelRotation( ):** This method is used to obtain the number of rotational units. Its form is shown below:

  int getWheelRotation( )

  If the value is positive, the wheel moved counterclockwise. If the value is negative, the wheel moved clockwise.

- **getScrollType( ):** To obtain the type of scroll, we call this method.

  int getScrollType( )

  It returns either WHEEL_UNIT_SCROLL or WHEEL_BLOCK_SCROLL.

- **getScrollAmount( ):** If the scroll type is WHEEL_UNIT_SCROLL, we can obtain the number of units of scroll by calling this method.

  int getScrollAmount( )

Event Class (Cont…)

- **TextEvent Class:**
  - Text events are generated by text fields and text areas, when characters are entered by a user or program.
  - TextEvent defines the integer constant TEXT_VALUE_CHANGED.
  - The one constructor of this class is shown below:

    TextEvent(Object *src*, int *type*)
  - The TextEvent object does not include the characters currently in the text component that generated the event. Here, the program must use other methods associated with the text component to retrieve the information.
  - For the above mentioned reason, no methods are discussed here for the TextEvent.

Event Class (Cont…)

WindowEvent Class:

- WindowEvent is a subclass of ComponentEvent. It defines integer constants that can be used to identify them.

Integer Constant	Description
WINDOW_ACTIVATED	The window was activated.
WINDOW_CLOSED	The window has been closed.
WINDOW_CLOSING	The user requested that the window be closed.
WINDOW_DEACTIVATED	The window was deactivated.
WINDOW_ICONIFIED	The window was iconified.
WINDOW_DEICONIFIED	The window was deiconified.
WINDOW_GAINED_FOCUS	The window gained input focus.
WINDOW_LOST_FOCUS	The window lost input focus.
WINDOW_OPENED	The window was opened.
WINDOW_STATE_CHANGED	The state of the window changed.

Event Class (Cont…)

WindowEvent Class (Cont…):

- There are four constructors. The first one is basic constructor and last three offer more detailed control:

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

  Here, *other* specifies the opposite window, when a focus or activation event occurs. The *fromState* specifies the prior state of the window, and *toState* specifies the new state, when a window state change occurs.

- **getWindow( ):** It returns the Window object that generated the event.
  Window getWindow( )

- **getOppositeWindow( ):** It returns the opposite window, when a focus or activation event has occurred. Its form is getOppositeWindow( ).

Sources of Event

Event Source	Description
Button	Generates action events, when a button is pressed.
Check box	Generates item events, when a check box is selected or deselected.
Choice	Generates item events, when a choice is changed.
List	Generates action events, when an item is double-clicked; generates item events, when an item is selected or deselected.
Menu Item	Generates action events, when a menu item is selected; generates item events, when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events, when the scroll bar is manipulated.
Text components	Generates text events, when a user enters a character.
Window	Generates window events, when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Event Listener Interface

- Listeners are created by implementing one or more interfaces.

- When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument.

- **ActionListener Interface:**
  - This interface defines the actionPerformed( ) method that is invoked, when an action event occurs. Its general form is shown here:

    void actionPerformed(ActionEvent *ae*)

Event Listener Interface (Cont…)

- **AdjustmentListener Interface:**
  - This interface defines the adjustmentValueChanged( ) method that is invoked, when an adjustment event occurs. Its general form is:

    void adjustmentValueChanged(AdjustmentEvent *ae*)
- **ComponentListener Interface:**
  - It defines four methods that are invoked, when a component is resized, moved, shown, or hidden. Their general forms are shown below:

    void componentResized(ComponentEvent *ce*)

    void componentMoved(ComponentEvent *ce*)

    void componentShown(ComponentEvent *ce*)

    void componentHidden(ComponentEvent *ce*)

Event Listener Interface (Cont…)

- **ContainerListener Interface:**
  - This interface contains two methods.
  - When a component is added to a container, componentAdded( ) is invoked. When a component is removed from a container, componentRemoved( ) is invoked. Their general forms are:

    void componentAdded(ContainerEvent *ce*)
    void componentRemoved(ContainerEvent *ce*)

- **FocusListener Interface:**
  - This interface also defines two methods.
  - When a component obtains keyboard focus and loses keyboard focus, focusGained( ) and focusLost () are invoked, respectively.

    void focusGained(FocusEvent *fe*)
    void focusLost(FocusEvent *fe*)

Event Listener Interface (Cont…)

- **ItemListener Interface:**
  - It defines the itemStateChanged( ) method that is invoked, when the state of an item changes. Its general form is shown below:

    void itemStateChanged(ItemEvent *ie*)

- **KeyListener Interface:**
  - This interface defines three methods.
  - The keyPressed( ) and keyReleased( ) methods are invoked, when a key is pressed and released, respectively.
  - The keyTyped( ) method is invoked, when a character has been entered. The general forms are given below:

    void keyPressed(KeyEvent *ke*)
    void keyReleased(KeyEvent *ke*)
    void keyTyped(KeyEvent *ke*)

Event Listener Interface (Cont…)

- **MouseListener Interface:**
  - It defines five methods.
  - If the mouse is pressed and released at the same point, mouseClicked() is invoked.
  - When the mouse enters and leaves a component, the mouseEntered( ) and mouseExited( ) are called, respectively.
  - The mousePressed( ) and mouseReleased( ) methods are invoked, when the mouse is pressed and released, respectively.

```
void mouseClicked(MouseEvent me)
void mouseEntered(MouseEvent me)
void mouseExited(MouseEvent me)
void mousePressed(MouseEvent me)
void mouseReleased(MouseEvent me)
```

Event Listener Interface (Cont…)

MouseMotionListener Interface:

- This interface defines two methods.
- The mouseDragged( ) method is called multiple times as the mouse is dragged. The mouseMoved( ) is called, when the mouse is moved.

```
void mouseDragged(MouseEvent me)
void mouseMoved(MouseEvent me)
```

MouseWheelListener Interface:

- This interface defines the mouseWheelMoved( ) method that is invoked, when the mouse wheel is moved.
- Its general form is given here:

```
void mouseWheelMoved(MouseWheelEvent mwe)
```

Event Listener Interface (Cont…)

- **TextListener Interface:**
  - This interface defines the textChanged( ) method that is invoked, when a change occurs in a text area or text field. Its general form is:

    void textChanged(TextEvent *te*)

- **WindowFocusListener Interface:**
  - This interface defines two methods.
  - The windowGainedFocus( ) and windowLostFocus( ) are called, when a window gains and loses input focus, respectively. Their general forms are given below:

    void windowGainedFocus(WindowEvent *we*)
    void windowLostFocus(WindowEvent *we*)

Event Listener Interface (Cont…)

- **WindowListener Interface:**
  - The windowActivated( ) and windowDeactivated( ) methods are invoked, when a window is activated and deactivated, respectively.
  - If a window is iconified and deiconified, the windowIconified( ) and windowDeiconified( ) are called, respectively.
  - When a window is opened and closed, the windowOpened( ) and windowClosed( ) methods are called, respectively. The windowClosing( ) is called, when a window is being closed.

```
void windowActivated(WindowEvent we)
void windowClosed(WindowEvent we)
void windowClosing(WindowEvent we)
void windowDeactivated(WindowEvent we)
void windowDeiconified(WindowEvent we)
void windowIconified(WindowEvent we)
void windowOpened(WindowEvent we)
```

Registration Method

- For registering the component with the Listener, many classes provide the registration methods. For example:

Button: public void addActionListener(ActionListener a){}

MenuItem: public void addActionListener(ActionListener a){}

TextField: public void addActionListener(ActionListener a){}

public void addTextListener(TextListener a){}

TextArea: public void addTextListener(TextListener a){}

Checkbox: public void addItemListener(ItemListener a){}

Choice: public void addItemListener(ItemListener a){}

List: public void addActionListener(ActionListener a){}

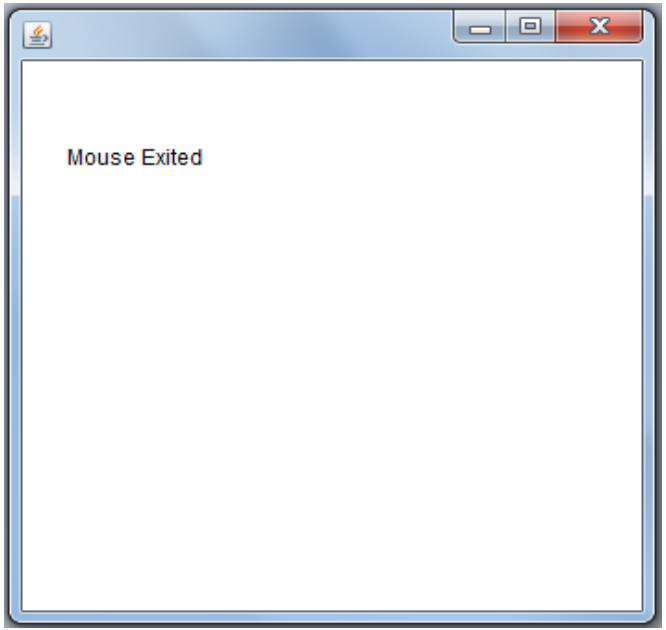
public void addItemListener(ItemListener a){}

Using the Delegation Event Model

```
import java.awt.*;
import java.awt.event.*;
public class MouseListener1 extends Frame implements MouseListener
{
  Label M;
  MouseListener1()
  {
    addMouseListener(this);
    M=new Label();
    M.setBounds(30,70,150,30);
    add(M);
    setSize(350,350);
    setLayout(null);
    setVisible(true);
  }
  public void mouseClicked(MouseEvent e)
  {
    M.setText("Mouse Clicked");
  }
  public void mouseEntered(MouseEvent e)
  {
    M.setText("Mouse Entered");
  }
  public void mouseExited(MouseEvent e)
  {
    M.setText("Mouse Exited");
  }
  public void mousePressed(MouseEvent e)
  {
    M.setText("Mouse Pressed");
  }
  public void mouseReleased(MouseEvent e)
  {
    M.setText("Mouse Released");
  }
  public static void main(String[] args)
  {
    new MouseListener1();
  }
}
```

Using the Delegation Event Model (Cont…)

- **Output**

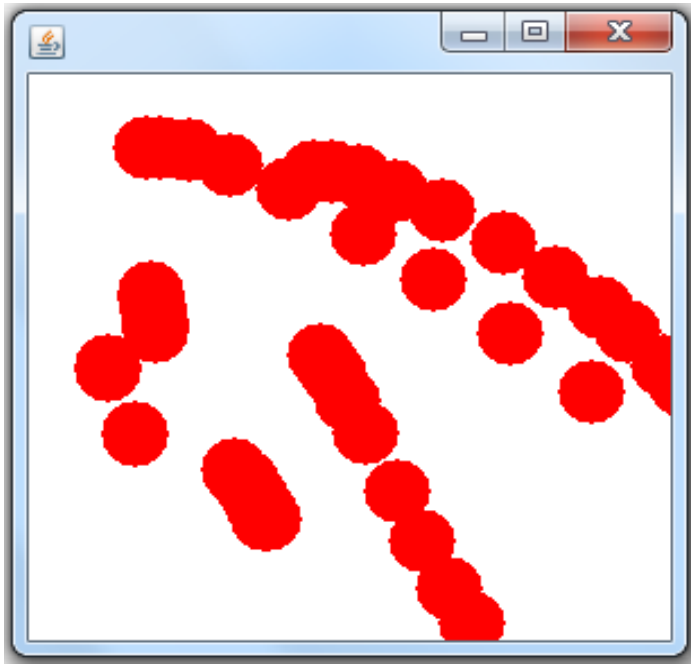


Using the Delegation Event Model (Cont...)

```
import java.awt.*;
import java.awt.event.*;
public class MouseMotionListener1 extends Frame implements MouseMotionListener
{
  MouseMotionListener1()
  {
    addMouseMotionListener(this);
    setSize(300,300);
    setLayout(null);
    setVisible(true);
  }
  public void mouseDragged(MouseEvent e)
  {
    Graphics g=getGraphics();
    g.setColor(Color.RED);
    g.fillOval(e.getX(),e.getY(),30,30);
  }
  public void mouseMoved(MouseEvent e)
  {
  }
  public static void main(String[] args)
  {
    new MouseMotionListener1();
  }
}
```

Using the Delegation Event Model (Cont…)

- **Output**

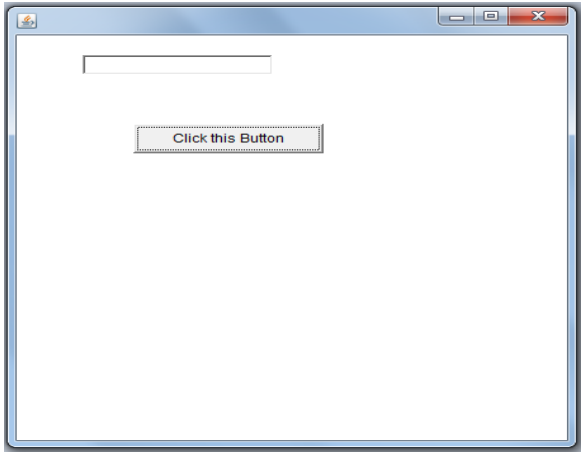


Using the Delegation Event Model (Cont…)

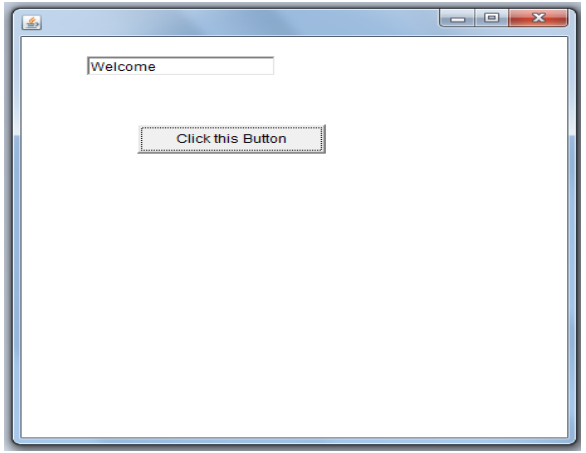
```
import java.awt.*;
import java.awt.event.*;
class EventHandling1 extends Frame implements ActionListener
{
  TextField tf;
  EventHandling1()
  {

    tf=new TextField();     //create components
    tf.setBounds(60,50,150,20);
    Button b=new Button("Click this Button");
    b.setBounds(100,120,150,30);

    b.addActionListener(this);    //register listener and passing current instance

    add(b);      //add components and set size, layout and visibility
    add(tf);
    setSize(450,450);
    setLayout(null);
    setVisible(true);
  }
  public void actionPerformed(ActionEvent e)
  {
    tf.setText("Welcome");
  }
  public static void main(String args[])
  {
    new EventHandling1();
  }
}
```

Using the Delegation Event Model (Cont…)

- **Output**
  - Before clicking the Button



  - After clicking the Button



Using the Delegation Event Model (Cont…)

Event Handling by Outer Class

Program of main class

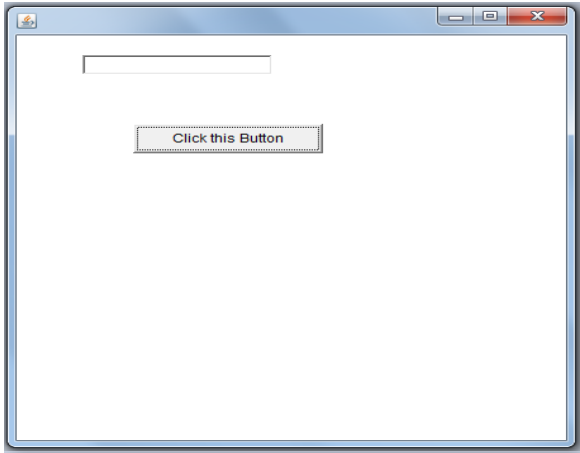
```
import java.awt.*;
import java.awt.event.*;
class Event2 extends Frame
{
  TextField tf;
  Event2()
  {

    tf=new TextField();     //create components
    tf.setBounds(60,50,150,20);
    Button b=new Button("Click this Button");
    b.setBounds(100,120,150,30);

    OuterClass oc=new OuterClass(this);

    b.addActionListener(oc);    //Passing current instance

    add(b);      //add components and set size, layout and visibility
    add(tf);
    setSize(350,350);
    setLayout(null);
    setVisible(true);
  }

  public static void main(String args[])
  {
    new Event2();
  }
}
```

Program of outer class

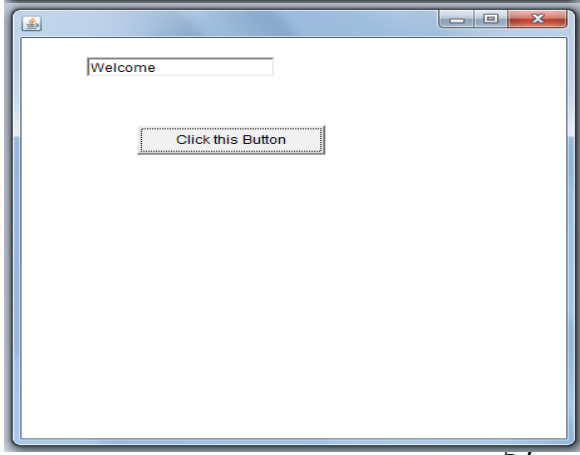
```
import java.awt.*;
import java.awt.event.*;
class OuterClass implements ActionListener
{
  Event2 obj;
  OuterClass(Event2 obj)
  {
    this.obj=obj;
  }
  public void actionPerformed(ActionEvent e)
  {
    obj.tf.setText("Welcome");
  }
}
```

Using the Delegation Event Model (Cont…)

- **Event Handling by Outer Class (Cont…)**
  - **Output**
    - Before clicking the Button



    - After clicking the Button



Using the Delegation Event Model (Cont…)

Event Handling by Adapter Class

- Java adapter classes provide the default implementation of listener interfaces.
- If we inherit the adapter class, we need not be forced to provide the implementation of all the methods of listener interfaces. So, it saves lines of codes and time.
- The adapter classes are found in java.awt.event, java.awt.dnd and javax.swing.event packages. java.awt.dnd adapter classes are given below:

Adapter class	Listener interface
DragSourceAdapter	DragSourceListener
DragTargetAdapter	DragTargetListener

Using the Delegation Event Model (Cont…)

Event Handling by Adapter Class (Cont…)

- java.awt.event adapter classes are given below:

Adapter class	Listener interface
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
HierarchyBoundsAdapter	HierarchyBoundsListener

Using the Delegation Event Model (Cont…)

- **Event Handling by Adapter Class (Cont…)**
  - javax.swing.event adapter classes are given below:

Adapter class	Listener interface
MouseInputAdapter	MouseInputListener
InternalFrameAdapter	InternalFrameListener

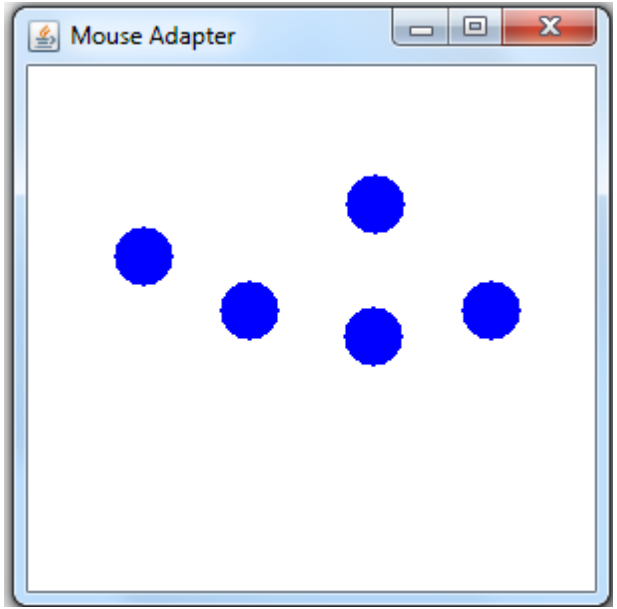
Using the Delegation Event Model (Cont…)

- **Event Handling by Adapter Class (Cont…)**

```
import java.awt.*;
import java.awt.event.*;
public class MouseAdapter1 extends MouseAdapter
{
  Frame f;
  MouseAdapter1()
  {
    f=new Frame("Mouse Adapter");
    f.addMouseListener(this);
    f.setSize(300,300);
    f.setLayout(null);
    f.setVisible(true);
  }
  public void mouseClicked(MouseEvent e)
  {
    Graphics g=f.getGraphics();
    g.setColor(Color.BLUE);
    g.fillOval(e.getX(),e.getY(),30,30);
  }

  public static void main(String[] args)
  {
    new MouseAdapter1();
  }
}
```

Using the Delegation Event Model (Cont…)

- **Event Handling by Adapter Class (Cont…)**
  - **Output**



Using the Delegation Event Model (Cont…)

Event Handling by Anonymous Inner Class

- When we create a class within a class without specifying a name, this is known as an anonymous inner class.
- Anonymous inner classes are declared and instantiated at the same time, using the new operator/keyword with the name of an existing class or interface.
- If we name a class, it will be subclassed. If we name an interface, the anonymous class extends java.lang.Object and implements the named interface. Example:

```
b.addActionListener(new ActionListener( )
{
  public void actionPerformed(ActionEvent e)
  {
    showStatus("Thanks for pushing my second button!");
  }
});
```

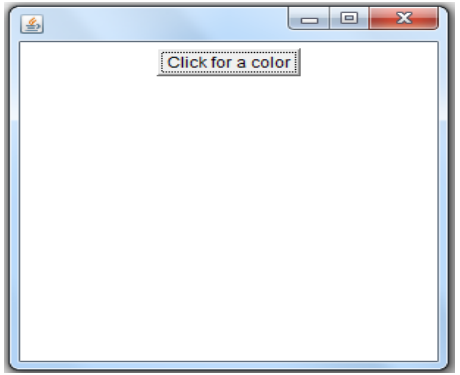
Using the Delegation Event Model (Cont…)

- **Event Handling by Anonymous Inner Class (Cont…)**

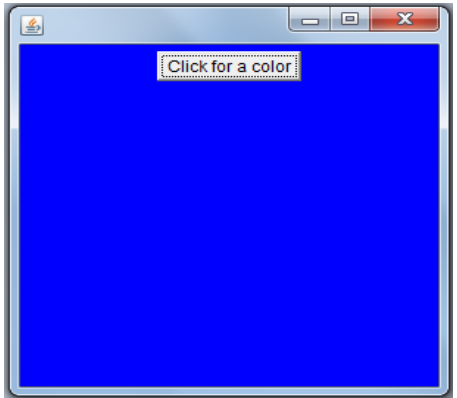
```
import java.awt.*;
import java.awt.event.*;

public class AnnonymousClass1 extends Frame
{
  Button b;
  public AnnonymousClass1()
  {
    setLayout(new FlowLayout());
    b = new Button("Click for a color");
    add(b);

    b.addActionListener(new ActionListener()
    {
      public void actionPerformed(ActionEvent e)
      {
        setBackground(Color.blue);
      }
    }
    );

    addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        dispose();
      }
    }
    );

    setSize(300, 300);
    setVisible(true);
  }
  public static void main(String args[])
  {
    new AnnonymousClass1();
  }
}
```

Using the Delegation Event Model (Cont…)

- **Event Handling by Anonymous Inner Class(Cont…)**
  - **Output**
    - Before clicking the Button



  - After clicking the Button




thank
you . . .

Slides are mainly prepared from Book. However, some Internet Links are also used.