GUI Event Handling

What is an Event?

- GUI components communicate with the rest of the applications through events.
- The source of an event is the component that causes that event to occur.
- The listener of an event is an object that receives the event and processes it appropriately.

Handling Events

- Every time the user types a character or clicks the mouse, an event occurs.
- Any object can be notified of any particular event.
- To be notified for an event,
  - The object has to be registered as an event listener on the appropriate event source.
  - The object has to implement the appropriate interface.

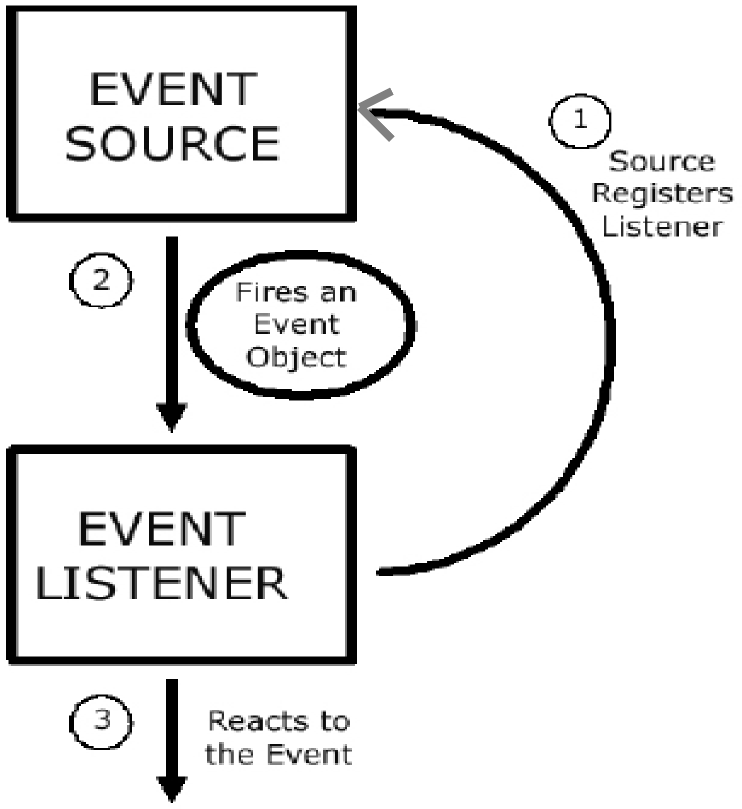
An example of Event Handling

```
public class SwingApplication implements
   ActionListener {
   ...
   JButton button = new JButton("I'm a Swing
   button!");
   button.addActionListener(this);
   ....
   public void actionPerformed(ActionEvent e) {
   numClicks++;
   label.setText(labelPrefix + numClicks);
   }
}
```

The Event Handling process

- When an event is triggered, the JAVA runtime first determines its source and type.
- If a listener for this type of event is registered with the source, an event object is created.
- For each listener to this type of an event, the JAVA runtime invokes the appropriate event handling method to the listener and passes the event object as the parameter.

The Event Handling Process (contd..)



What does an Event Handler require?

- It just looks for 3 pieces of code!
- First, in the declaration of the event handler class, one line of code must specify that the class implements either a listener interface or extends a class that implements a listener interface.

```
public class DemoClass implements
ActionListener {
```

What does an Event Handler require? (contd..)

- Second, it looks for a line of code which registers an instance of the event handler class as a listener of one or more components because, as mentioned earlier, the object must be registered as an event listener.

```
anyComponent.addActionListener(instanceOf
DemoClass);
```

What does an Event Handler require? (contd..)

- Third, the event handler must have a piece of code that implements the methods in the listener interface.

```
public void actionPerformed(ActionEvent e) {
   ...//code that reacts to the action...
 }
```

Types of Events

- Below, are some of the many kinds of events, swing components generate.

Act causing Event	Listener Type
User clicks a button, presses Enter, typing in text field	ActionListener
User closes a frame	WindowListener
Clicking a mouse button, while the cursor is over a component	MouseListener

Types of Events (contd..)

Act causing Event	**Listener Type**
User moving the mouse over a component	MouseMotionListener
Component becomes visible	ComponentListener
Table or list selection changes	ListSelectionListener

The Event classes

- An event object has an event class as its reference data type.
- The Event object class
  - Defined in the java.util package.
- The AWT Event class
  - An immediate subclass of EventObject.
  - Defined in java.awt package.
  - Root of all AWT based events.

Event Listeners

- Event listeners are the classes that implement the <type>Listener interfaces.

  Example:

  1. ActionListener receives action events
  2. MouseListener receives mouse events.

The following slides give you a brief overview on some of the listener types.

The ActionListener Method

- It contains exactly one method.

  Example:

  ```
  public void actionPerformed(ActionEvent e)
  ```

  The above code contains the handler for the ActionEvent e that occurred.

The MouseListener Methods

- Event handling when the mouse is clicked.

  `public void mouseClicked(MouseEvent e)`

- Event handling when the mouse enters a component.

  `public void mouseEntered(MouseEvent e)`

- Event handling when the mouse exits a component.

  `public void mouseExited(MouseEvent e)`

The MouseListener Methods (contd..)

- Event handling when the mouse button is pressed on a component.

```
public void mousePressed(MouseEvent e)
```

- Event handling when the mouse button is released on a component.

```
public void mouseReleased(MouseEvent e)
```

The MouseMotionListener Methods

- Invoked when the mouse button is pressed over a component and dragged. Called several times as the mouse is dragged

  ```
  public void mouseDragged(MouseEvent e)
  ```

- Invoked when the mouse cursor has been moved onto a component but no buttons have been pushed.

  ```
  public void mouseMoved(MouseEvent e)
  ```

The WindowListener Methods

- Invoked when the window object is opened.
  ```
  public void windowOpened(WindowEvent e)
  ```
- Invoked when the user attempts to close the window object from the object's system menu.
  ```
  public void windowClosing(WindowEvent e)
  ```

The WindowListener Methods (contd..)

- Invoked when the window object is closed as a result of calling dispose (release of resources used by the source).

```
public void windowClosed(WindowEvent e)
```

- Invoked when the window is set to be the active window.

```
public void windowActivated(WindowEvent e)
```

The WindowListener Methods (contd..)

- Invoked when the window object is no longer the active window

```
public void windowDeactivated(WindowEvent e)
```

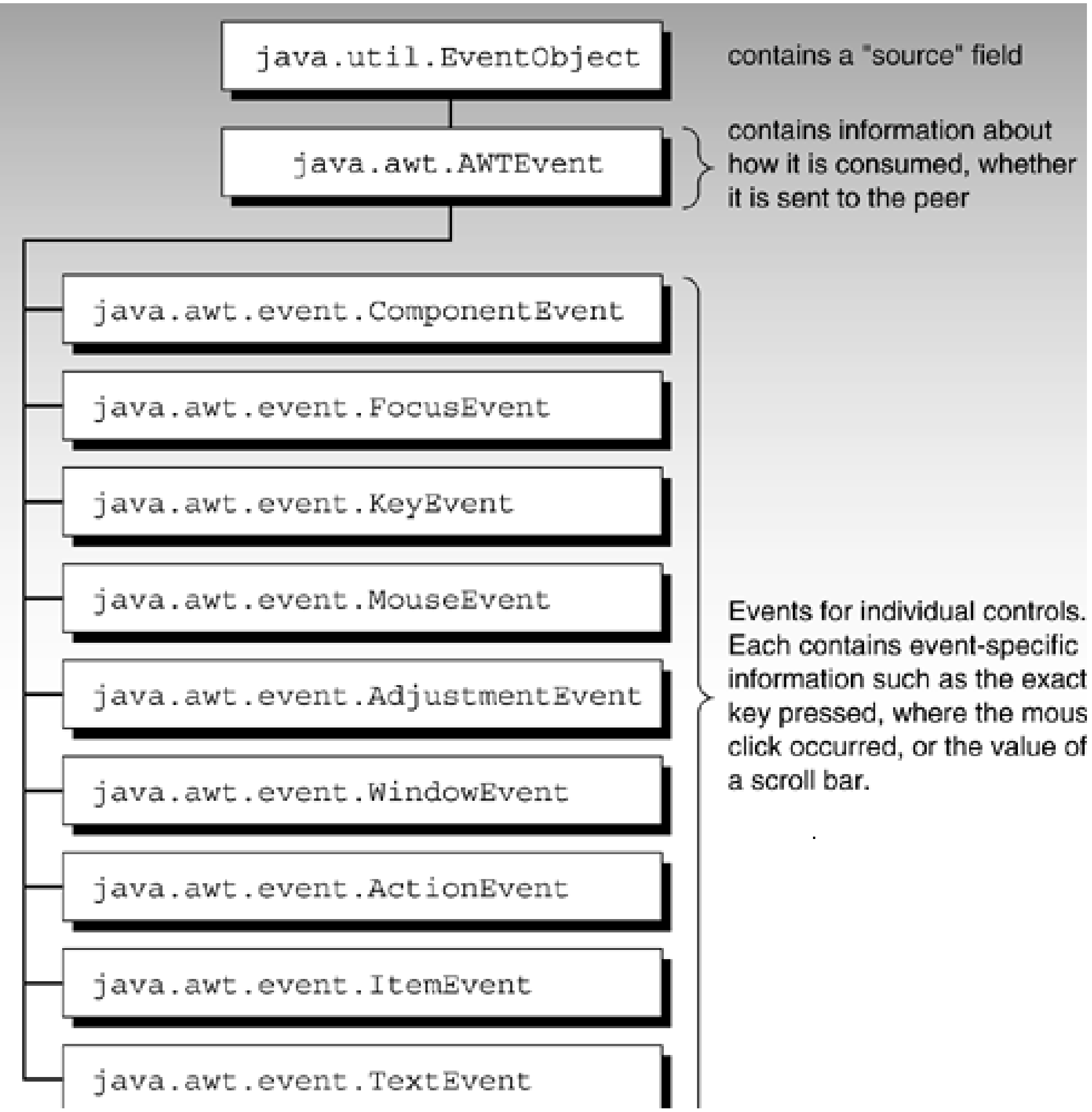
- Invoked when the window is minimized.

```
public void windowIconified(WindowEvent e)
```

- Invoked when the window is changed from the minimized state to the normal state.

```
public void windowDeconified(WindowEvent e)
```

Hierarchy of event **objects**


java.util.EventObject
contains a "source" field
java.awt.AWTEvent
contains information about how it is consumed, whether it is sent to the peer
java.awt.event.ComponentEvent
java.awt.event.FocusEvent
java.awt.event.KeyEvent
java.awt.event.MouseEvent
java.awt.event.AdjustmentEvent
java.awt.event.WindowEvent
java.awt.event.ActionEvent
java.awt.event.ItemEvent
java.awt.event.TextEvent
Events for individual controls. Each contains event-specific information such as the exact key pressed, where the mous click occurred, or the value of a scroll bar.

Note: The number of event objects is much greater then specified in diagram…Due to space constraints, only some of them are represented in the figure

Courtesy: Safari.oreilly.com

Additional Listener Types

- Change Listener
- Container Listener
- Document Listener
- Focus Listener
- Internal Frame Listener
- Item Listener
- Key Listener
- Property Change Listener
- Table Model Listener

The main purpose of the last few slides is to give you an idea as to how you can use event handlers in your programs. It is beyond the scope of the O'Reilly book to cover every event handler. See the JAVA tutorials for more information.

Adapter classes for Event Handling.

- Why do you need adapter classes?
  - Implementing all the methods of an interface involves a lot of work.
  - If you are interested in only using some methods of the interface.
- Adapter classes
  - Built-in in JAVA
  - Implement all the methods of each listener interface with more than one method.
  - Implementation of all empty methods

Adapter classes - an Illustration.

- Consider, you create a class that implements a MouseListener interface, where you require only a couple of methods to be implemented. If your class directly implements the MouseListener, you *must* implement all five methods of this interface.
- Methods for those events you don't care about can have empty bodies

Illustration (contd..)

```
public class MyClass implements MouseListener {
   ... someObject.addMouseListener(this);
   /* Empty method definition. */
   public void mousePressed(MouseEvent e) { }
   /* Empty method definition. */
   public void mouseReleased(MouseEvent e) { }
   /* Empty method definition. */
   public void mouseEntered(MouseEvent e) { }
   /* Empty method definition. */
   public void mouseExited(MouseEvent e) { }
   public void mouseClicked(MouseEvent e) {
   //Event listener implementation goes here...
   }
 }
```

Illustration (contd..)

- What is the result?
  - The resulting collection of empty bodies can make the code harder to read and maintain.
- To help you avoid implementing empty bodies, the API generally includes an *adapter* class for each listener interface with more than one method. For example, the MouseAdapter class implements the MouseListener interface.

How to use an Adapter class?

```
/* Using an adapter class
 */
 public class MyClass extends MouseAdapter {
 ....
    someObject.addMouseListener(this);
 ....
    public void mouseClicked(MouseEvent e) {
      ...//Event listener implementation goes
         // here
    }
 }
```

Using Inner classes for Event Handling

- Consider that you want to use an adapter class but you don’t want your public class to inherit from the adapter class.
- For example, you write an applet with some code to handle mouse events. As you know, JAVA does not permit multiple inheritance and hence your class cannot extend both the Applet and MouseAdapter classes.

Using Inner classes (contd..)

- Use a class inside your Applet subclass that extends the MouseAdapter class.

```
public class MyClass extends Applet { ...
 someObject.addMouseListener(new
                          MyAdapter());
 ...
  class MyAdapter extends MouseAdapter {   public
 void mouseClicked(MouseEvent e) {
//Event listener implementation here... }
 }
}
```

Creating GUI applications with Event Handling.

- Guidelines:
  1. Create a GUI class
     - Describes the appearance of your GUI application.
  2. Create a class implementing the appropriate listener interface
     - May refer to the same class as step 1.

Creating GUI applications with Event Handling (contd..)

3. In the implementing class
   - Override all methods of the appropriate listener interface.
   - Describe in each method how you want to handle the events.
   - May give empty implementations for the methods you don’t need.

Creating GUI applications with Event Handling (contd..)

4. Register the listener object with the source
    - The object is an instantiation of the listener class specified in step 2.
    - Use the *add<Type>Listener* method.

Design Considerations

- The most important rule to keep in mind about event listeners is that they must execute quickly. Because, all drawing and event-listening methods are executed in the same thread, a slow event listener might make the program seem unresponsive. So, consider the performance issues also when you create event handlers in your programs.

Design Considerations

- You can have choices on how the event listener has to be implemented. Because, one particular solution might not fit in all situations.

  For example, you might choose to implement separate classes for different types of listeners. This might be a relatively easy architecture to maintain, but many classes can also result in reduced performance .

Common Event-Handling Issues

1. You are trying to handle certain events from a component, but it doesn’t generate the events it should.
   - Make sure you have registered the right kind of listener to detect the events.
   - Make sure you have registered the listener on the right object.
   - Make sure you have implemented the event handler correctly, especially, the method signatures.

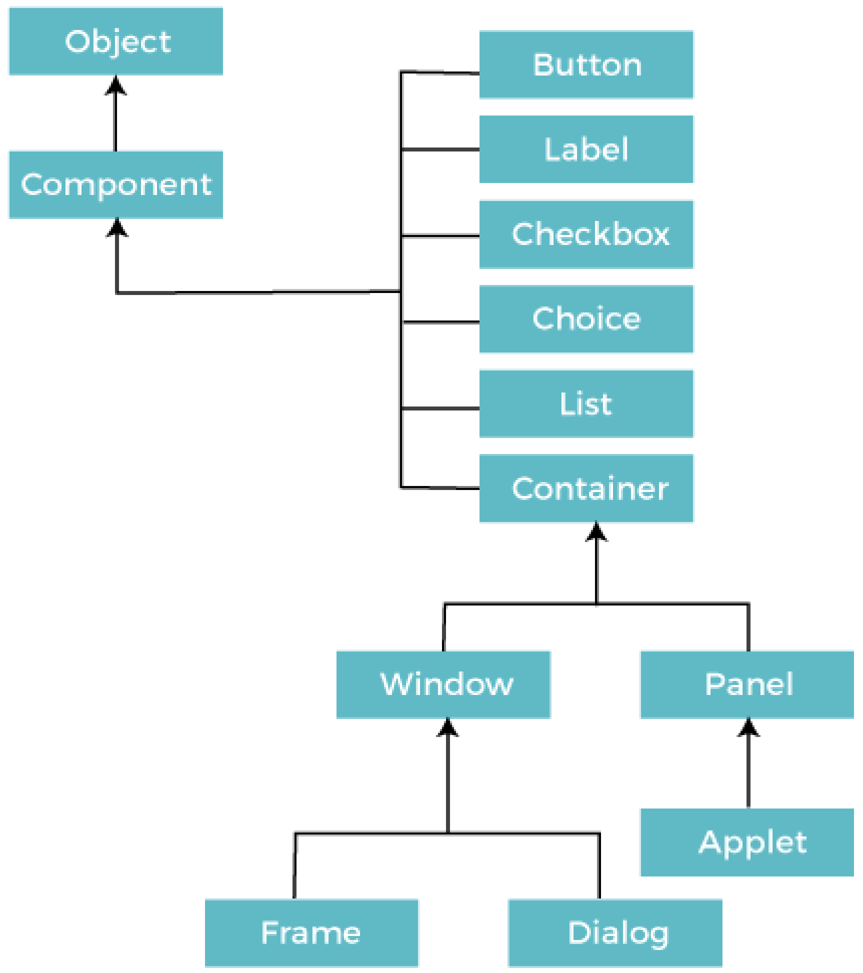
Common Event-Handling Issues (contd..)

2. Your combo box isn’t generating low level events like focus events.
    - Since combo boxes are compound components, i.e., components implemented using multiple components, combo-boxes do not fire the low-level events that simple components fire.

Common Event-Handling Issues (contd..)

3. The document for an editor pane is not triggering document events.
   - The document instance for an editor pane might change when loading text from a URL. Thus your listeners might be listening for events on an unused document.
   - Hence, make sure that the code adjusts for possible changes to the document if your program dynamically loads text into an editor pane.

Java AWT Hierarchy



AWT Example by Inheritance

```
import java.awt.*;
class First extends Frame{
First(){
Button b=new Button("click me");
b.setBounds(30,100,80,30);// setting button position
add(b);//adding button into frame
setSize(300,300);//frame size 300 width and 300 height
setLayout(null);//no layout now bydefault BorderLayout
setVisible(true);//now frame willbe visible, bydefault not
visible
}
public static void main(String args[]){
First f=new First();
}
}
```

By creating the object of Frame class (**association**)

```
import java.awt.*;
class First2{
First2(){
Frame f=new Frame();
Button b=new Button("click me");
b.setBounds(30,50,80,30);
f.add(b);
f.setSize(300,300);
f.setLayout(null);
f.setVisible(true);}
public static void main(String args[]){
First2 f=new First2();
}
}
```

AWT Button

1.	Button( )	It constructs a new button with an empty string i.e. it has no label.
2.	Button (String text)	It constructs a new button with given string as its label.

Label Fields

Constructors

1. Label(): It constructs an empty label.
2. Label(String text): It constructs a label with the given string (left-justified by default).
3. Label(String text, int alignment): It constructs a label with the specified string and the specified alignment.

Methods

1. void setText(String text) It sets the texts for label with the specified text.
2. void setAlignment(int alignment) It sets the alignment for label with the specified alignment.
3. String getText() It gets the text of the label
4. int getAlignment() It gets the current alignment of the label.