

Topic 6

Event Driven Programming and the GUI

Learning Outcomes

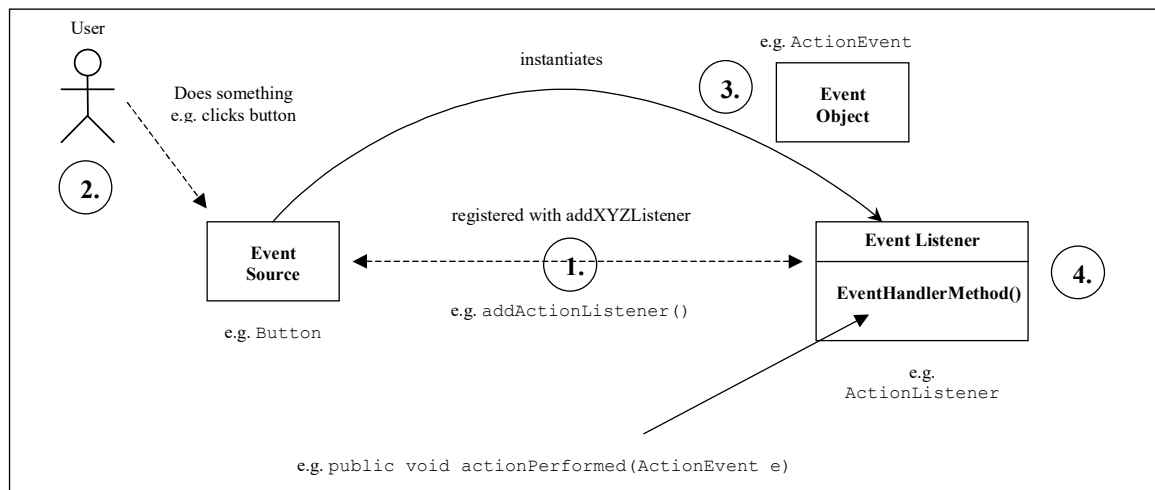
- **Describe and Document Diagrammatically** the components of the AWT Delegation based Event Handling Mechanism and how they interact together to form a structured approach to event management
- **Demonstrate** how the different constructs such as *Event Source*, *Event Object*, *Event Listener* and *Event Handler Method* relate to specific AWT/Swing classes by **programming** your own event handling code
- **Describe** the difference between an *Event Listener Interface* and an *Event Adapter Class* so you can **identify** when to **apply** them appropriately in code
- **Demonstrate** code application of specific interfaces: `KeyListener`, `Mouse/MouseMotionListener`, `FocusListener` and `WindowListener`
- **Describe** the syntax and semantics of Inner Classes and Anonymous Inner classes in Java
- **Demonstrate** the different ways Inner Classes can be used to support event handling by writing Java code
- **Explain** the pros and cons of each approach so you can **apply** them in a specific code scenario

AWT - Sources, Listeners and Events

- Delegation event model in AWT 1.1 (and later)
 - Event sources (usually `Components`) fire *events*
 - Events are listened for and acted upon by *event listeners*
 - Listeners are registered with a source by invoking the component's `addXYZListener(XYZListener)` method
 - Can add multiple listeners to an event source
 - Notification order is undefined
 - Can register single listener with multiple sources (careful you don't reduce cohesion!)
 - i.e. only do this if all sources expect same response (e.g. menu item and toolbar button)
 - Event listeners implement *event handling methods*
 - are passed an instance of an *event object* (an instance of `java.awt.Event`)
 - contains info. about the event and a reference to the event source

Diagram of Event Interaction

1. A listener is registered with an event source [e.g `button.addActionListener(new ActionListener())`]
2. The user interacts with the event source (e.g. clicks button with mouse)
3. The event source instantiates an event object containing information about the event
4. The event source passes this event object to the event listener (i.e. invokes the event handler method of its registered listener and passes it the event object as a parameter)



Components as Event Sources

- All Java components inherit the following listener registration methods (addXYZListener) from their parent class java.awt.Component:

- void addFocusListener(FocusListener)
- void addKeyListener(KeyListener)
- void addMouseListener(MouseListener)
- void addMouseMotionListener(MouseMotionListener)
- void addInputMethodListener(InputMethodListener)
- void addComponentListener(ComponentListener)

- Each AWT component also has specialised listener registration methods:

- e.g. (J)MenuItem and (J)Button have addActionListener methods
- Usage: Source.addXYZListener(XYZListener listener)

Listeners

- java.awt.event package defines many interfaces for different types of listeners.
- Each interface defines methods that are called when a specific event occurs e.g.
 - ActionListener defines a lone method:

```
void actionPerformed(ActionEvent)
```
 - invoked when an action event occurs within the source (component) that registered the ActionListener with
`Component.addActionListener(ActionListener l)`
- Instances of classes that are to act as event listeners must implement one or more of the listener interfaces

Event Objects

- `EventObject` is a simple class
 - base class for all of the `AWTEvent` classes
 - keeps track of its event source, i.e. the object that triggered the event
 - use `EventObject.getSource()` method to return the event source [must be converted from `Object` to correct type e.g `Window`
`w=(Window)e.getSource()`]
- Declare constants to provide additional info about the event
 - `FOCUS_LOST`, `FOCUS_GAINED` etc.
 - use methods such as `AWTEvent.getID()` to return constants
- Provide methods to access event properties
 - e.g `ActionEvent` class
 - `String getActionCommand();`
 - `int getModifiers();`

Event Handling Example

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

public class ButtonTest extends Applet {
    public void init() {
        Button button = new Button("Press Me");
        button.addMouseListener(new
            ButtonMouseListener());
        add(button);
    }
}
```

Event Handling Example (cont)

```
class ButtonMouseListener implements MouseListener {  
    public void mouseEntered(MouseEvent event) {  
        System.out.println("Mouse Entered Button");  
    }  
    public void mouseExited(MouseEvent event) {  
        System.out.println("Mouse Exited Button");  
    }  
    public void mousePressed (MouseEvent event) { }  
    public void mouseClicked (MouseEvent event) { }  
    public void mouseReleased(MouseEvent event) { }  
}
```

AWT Adapters

- Most AWT Listener interfaces provide more than one method which must be implemented
- Can be tedious if only one or two are of interest
- e.g. interested only in `WindowListener.windowClosing()` but not the other six Window Events
- Consequently AWT provides Adapters for each of the Listener interfaces that provide multiple methods
- Adapters provide no-op (empty) implementations of the listeners
- => Specific methods can be overridden and handled and the others ignored
- Adapters are classes not interfaces (i.e. they are extended not implemented) therefore only one adapter can be used per class

Adapter Example

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

public class ButtonTest2 extends Applet {
    public void init() {
        Button button = new Button("Press Me");
        button.addMouseListener(new
            ButtonMouseListener());
        add(button);
    }
}
```

Adapter Example (contd.)

```
class ButtonMouseListener extends MouseAdapter {
    public void mouseEntered(MouseEvent event) {
        System.out.println("Mouse Entered Button");
    }
    public void mouseExited(MouseEvent event) {
        System.out.println("Mouse Exited Button");
    }
}
```

Focus Events

- Maximum of one component can have focus at any one time
- Keyboard events are sent to the focused component
- Typically have a prominent appearance
- Qualified as `FOCUS_LOST` or `FOCUS_GAINED` events
- Focus event can be permanent or temporary
 - permanent focus event is when focus is deliberately changed
 - temporary event occurs when containing object (usually window) loses focus
 - `FocusEvent.isTemporary()`
- Gain focus by:
 - interacting with the component
 - invoking `Component.requestFocus()`
 - typing TAB or SHIFT-TAB
- Use `FocusEvent` and `FocusListener` to handle the focus
- Default focus order is order components were added but can modify
- see <http://docs.oracle.com/javase/tutorial/uiswing/misc/focus.html>

Key Events

- Key Events fired when a key is pressed or released in a component that has the focus
- `KeyListener` interface has `keyPressed()`, `keyReleased()` and `keyTyped()` methods
- Each key on the keyboard has a unique key code
 - Returned by `KeyEvent.getKeyCode()` (e.g. `KeyEvent.VK_A`, `KeyEvent.VK_F1` etc. - see API docs for full list of key codes)
 - For key typed events, `keyCode` is `KeyEvent.VK_UNDEFINED`
- Keys also map to a UNICODE (16 bit) character
 - `KeyEvent.getKeyChar()`
 - Returns the character associated with the key in this event. For example, the key-typed event for shift + "a" returns the value for "A"
 - If no valid Unicode character exists for the key then it returns `KeyEvent.CHAR_UNDEFINED`

Mouse and Mouse Motion Events

- Mouse events and mouse motion events are distinguished via separate `MouseListener` and `MouseMotionListener` interfaces
- There is also a separate `MouseWheelListener` that was added in Java 1.4
- Mouse moved and mouse dragged are mouse motion events
- The remaining events (enter/exit, pressed/released and clicked) are mouse events
- Both share the same event class `MouseEvent`
 - can identify position of cursor: `getPoint()` or `getX()`, `getY()`
 - can determine key modifiers if any (SHIFT, CTRL etc.) `getModifiers()` or `getModifiersEx()` (since Java 1.4)
 - can determine click count (can be more than two!) `getClickCount()`

Window Events

- Fired by instances of `java.awt.Window` (and subclasses such as `(J)Frame` and `(J)Dialog`)
- Signify that a window has been *activated/deactivated*, *iconified/deiconified*, *opened/closed*, or closing
- `windowClosing()` is commonly overridden for a main frame window to prompt before exit
- `ComponentListener` also useful for checking for movement, resizing etc.

AWT/Swing Events and Threads

- AWT provides a dedicated thread called the “Event Handler Thread” or “UI Thread” on which all event processing takes place
 - All event handler methods are processed on this thread
- When this thread is running due to an event being handled no other UI processing will take place (including any user interaction such as menu or component interaction etc.)
- Therefore event handler methods must execute quickly otherwise the UI will appear to lock up
- Any lengthy processing initiated in an event handler (or on the UI thread through other means) must run in its own background thread
- All UI updates such as adding components, laying out, (re)drawing etc. must also be done on the UI thread
- Therefore if UI updating is done anywhere other than an event handler method it should be delegated to the UI thread using the following method:

```
public static void invokeLater(Runnable doRun)
```
- Threads are covered in more detail towards the end of this course!

Multi-Threaded Code Example for AWT Swing

To call a *worker* method on a separate thread ..

```
new Thread()  
{  
    @Override  
    public void run()  
    {  
        //call worker methods on separate thread  
    }  
}.start();
```

to update the GUI from a non UI thread ..

```
SwingUtilities.invokeLater(new Runnable()  
{  
    @Override  
    public void run()  
    {  
        // do GUI update on UI thread  
    }  
});
```

Inner Classes

Version 1.1 of the JDK introduced the notion of inner classes

- often used with event handling
- (potentially) simplify the relationship between event sources and listeners

Non-static (standard) inner classes:

- have direct access to the implementation of their enclosing class (including private attributes and methods)
- are automatically initialised with a reference to the instance of the outer class that created it [usually the reference is invisible and implicit e.g.
`someMethodCall()` but can be made explicit with `EnclosingClassName.this.someMethodCall()`]

Inner Classes

- Inner classes may be named or anonymous
- Static inner classes are similar to C++ nested classes (no reference to outer class)
- Can be hidden from other classes in a package (private)
- See `sadi.topic6.inner.fundamentals.*`;
- The **next five slides** show five variations on the relationship between event sources (`ThreeDButton` in this case) and listeners and shows how inner classes can be used as part of expressing this relationship

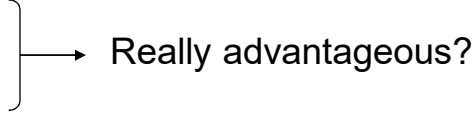
Separate Listener (using `EventObject.getSource()`)

- **See** `sadi.topic6.listener.separate.*;`
- **Advantages:**
 - self contained and cohesive
 - can have multiple sources use a single instance of a listener if desirable
 - listener is not ‘hard coded’ to the source
 - ⇒ source can be modified independently of listener
- **Disadvantages:**
 - must call `getSource()` in every method that needs access to the event source and cast is fragile
 - may be cumbersome to create separate classes for very small one line operations (not really true with good IDE)

Separate Listener (alternative approach constructor parameter)

- **See** `sadi.topic6.listener.alternative.*;`
- **Advantages:**
 - maintains reference to source [does not need to call `getSource()`]
 - source/listener type checking done at compile time
 - source can be modified independently of listener
- **Disadvantages:**
 - Less efficient since we must construct separate listener instances for each source even if functionality the same

Combined Source/Listener

- **See** `sadi.topic6.listener.combined.*;`
- **Advantages:**
 - no separate class is necessary
 - less lines of code

Really advantageous?
- **Disadvantages:**
 - reduced cohesion (and harder to identify coupling)
 - reduced extensibility (cannot modify source independently of listener)
 - classes may become large and more difficult to maintain
 - less efficient (one listener instance per source)

Think carefully before using this approach!

Named Inner Class

- **See** `sadi.topic6.inner.named.*;`
- **Advantages:**
 - can automatically reference event source (because it is the enclosing class)
 - does not need to call `getSource()` or maintain a reference (as in the two separate examples)
 - maintains moderate encapsulation and cohesion
- **Disadvantages:**
 - non-obvious coupling between source/listener
 - more difficult to share coding responsibilities (since they are in the same code module)
 - assumes a one to one mapping between source and listener instances (inefficient)

Anonymous Inner Class

- See `sadi.topic6.inner.anonymous.*;`
- Advantages:
 - provides a syntactic shortcut by combining the instantiation with the definition
 - convenient for small handler methods
 - can automatically reference event source (because it is the enclosing class)
 - does not need to call `getSource()` or maintain a reference (as in the two separate examples)
- Disadvantages:
 - code clarity and cohesion may be reduced (since coupling is implicit)
 - more difficult to share coding responsibilities (since they are in the same code module)
 - assumes a one to one mapping between source and listener instances
 - anonymous classes can only be instantiated once (but can be assigned to a variable!)

Property Change Listeners

- Properties:
 - According to the Java specifications a **property** is an attribute that has appropriately named getter and setter methods
 - <https://docs.oracle.com/javase/tutorial/javabeans/writing/properties.html>
 - e.g. `int propertyName;`
 - `public int getPropertyName()`
 - `public void setPropertyName(int propertyName)`
 - A *bound* property fires property change events when it changes
- `java.beans.PropertyChangeSupport`
 - A helper class which implements/manages listener behaviour for a class with properties
 - The event source class (containing the properties of interest) maintains an instance of the support class and forwards appropriate calls to it (add/remove listeners, fire property change etc.)
 - See API docs for details and tutorial link above

Property Change Listeners (contd.)

- Client (Observer) classes can listen for all property changes or just a specific property
(see `PropertyChangeSupport.addPropertyChangeListener(...)` overrides)
- The `java.beans.PropertyChangeListener` interface has a single event handler method that takes the event object type `java.beans.PropertyChangeEvent` containing the event source, `oldValue`, `newValue` etc.

```
public void propertyChange(PropertyChangeEvent event)
```
- NOTE: the property classes are older and do not support generics so all properties are considered to be of type `java.lang.Object` although it is possible to work around this using generics where appropriate or simply calling the source bean `getter()` to retrieve a value with a specific type upon receiving property change events