Java GUI programming

On event-driven programming

- a user interface combines three functions: input, output, and data handling
- in batch-oriented processing
  - the control of execution is within the program
  - the program reads more information when it needs it, and prints out results when they are ready
- in event-driven GUI processing
  - processing happens as reactions to the interactive manipulation of visual components

On event-driven programming (cont.)

- code is executed when events are generated by user actions, such as
  - mouse movements,
  - keystrokes, or
  - clicking buttons
- note the reversal of control, so called Hollywood principle
  - “don’t call us, we’ll call you”
- Java uses a delegation-based model to register listeners and handle events:
  - a source component (say, a button) notifies its listener objects by calling the handlers implemented by the listeners

AWT vs. Swing

Java provides two sets of GUI components ("controls")

- AWT (Abstract Windowing Toolkit) component library that was a part of Java 1.0 and modified in Java 1.1
  - the so-called event delegation model was already part of the AWT library version 1.1
- Swing component library, the GUI part of the JFC (Java Foundation Classes),
  - was available in JDK 1.2 and later versions
**AWT peer components**

- every AWT component has a **peer component** in the native windowing system
- every GUI component automatically assumes the appearance determined by the host window system
- AWT components are tied directly to the local platform’s GUI capabilities
- unfortunately, the behaviour of native components differs in annoying respects, even if only minimum common capabilities are used
  - "write once, test everywhere"
- AWT components were **heavyweight, platform dependent, and inflexible**

**AWT design patterns**

- **Composite** design pattern organizes components into tree-like hierarchical structures
  - each node represents a component
  - all nodes implement the same interface
    - polymorphism ensures clients traverse all nodes uniformly
  - AWT class **Component**
    - provides **paint** for drawing on screen
  - AWT class **Container** is an aggregate
    - provides **add** for adding components
- **Bridge** design pattern: peer organization

**Swing components**

- Swing components are written, manipulated, and displayed completely in Java
- Swing components have **platform-independent**, look-and-feel
  - Swing provides three built-in look-and-feel style packages: (1) Windows, (2) UNIX-like Motif, and (3) Java (Metal)
  - the Windows style is available only in MS Windows platforms (for copyright reasons)
Relationship between AWT and top-level Swing components

- A GUI component is a visible object in a graphical user interface.
- A GUI container is an area where components can be placed.
- The GUI class library is organized into an inheritance hierarchy rooted at `java.awt.Component`.
- The top-level heavyweight components depend on the peer model of AWT.
  - `javax.swing.JWindow`, `javax.swing.JFrame`, `javax.swing.JDialog`, and `javax.swing.JApplet` extend the AWT components `java.awt.Window`, `java.awt.Frame`, `java.awt.Dialog`, and `java.applet.Applet`, respectively.

Swing components derived from `JComponent`

- Swing lightweight components are rooted at `javax.swing.JComponent`.
  - That itself extends `java.awt.Container`.
    - For generality and flexibility.
    - Reuses code and design form AWT.
  - `JComponent` is both "component" and "composite" in the Composite design pattern.
  - Lightweight components are implemented entirely in Java, having no peers within the window system of the platform.

GUI library organization

- A GUI component is a visible object in a graphical user interface.
- A GUI container is an area where components can be placed.
- The GUI class library is organized into an inheritance hierarchy rooted at `java.awt.Component`.
- The top-level heavyweight components depend on the peer model of AWT.
  - `javax.swing.JWindow`, `javax.swing.JFrame`, `javax.swing.JDialog`, and `javax.swing.JApplet` extend the AWT components `java.awt.Window`, `java.awt.Frame`, `java.awt.Dialog`, and `java.applet.Applet`, respectively.

Swing components

- Swing lightweight components are rooted at `javax.swing.JComponent`.
  - That itself extends `java.awt.Container`.
    - For generality and flexibility.
    - Reuses code and design form AWT.
  - `JComponent` is both "component" and "composite" in the Composite design pattern.
  - Lightweight components are implemented entirely in Java, having no peers within the window system of the platform.
Model-View-Controller (MVC)

• Swing components are internally based on the MVC (Model-View-Controller) architecture
• the component is divided into three separate objects:
  – view: how it looks (output/display)
  – model: what state it is in (data)
  – controller: what user input it accepts and what it does (input/events)
• the MVC architecture originates from the Smalltalk-80 system, developed during 70's
  – in Smalltalk, MVC was used as an architectural model at the application level: the data (model) is made independent of the UI (view and controller)

Model-View-Controller (cont.)

• in Swing, a version of MVC is used internally at the component level
  – the outer appearance (view) of a component is made independent of the data (model), and is changeable even at run time
• AWT and Swing do not determine the architecture of the whole application
  – the programmer has the responsibility of program modularization
  – can put data into graphical components (bad style), or represent it separately (a better way)

Internal MVC components within JButton

- User Interface (UI): View
- Controller: Behavior
- Model: State
- View: Outward appearance

- Mouse, Keyboard, Events
- Java Method Calls
- Change state
- Look and feel

often, interactions between controller and view, too

- JButton
- Model
- Controller
- View
- SetsState
- DeterminesLook
- HandlesEvents
- Responds
- JavaMethodCall
- Event
Component Organization in GUIs

- within an application, its GUI components are organized into a containment hierarchy rooted at some top-level window-like component
- the top-level Swing classes - JWindow (undecorated frame), JFrame, JDialog, and JApplet - use an implicit content pane as their component container:
  
  ```java
  add (component);  // into its content pane
  ```
- javax.swing.JPanel and other containers are used to organize components into an appropriate containment hierarchy as needed by the application

Layout managers

- a layout manager is an object that manages the size and arrangement of the components in a container
  
  - relative positions of the elements are specified, not their absolute coordinates (as with MS Windows)
  - the positions and sizes of the element will be automatically adjusted when the outlook or size of components is changed, or the whole window is resized

Layout managers (cont.)

- AWT and Swing provide a number of built-in layouts, including:
  
  - flow, border, grid, and box layouts
  - JFrame uses the border layout (north, center,..)
  - JPanel uses the flow layout (row-by-row)
    
    ```java
    container.setLayout (new FlowLayout ());
    ```
  
- you can write your own layout manager that implements LayoutManager (or LayoutManager2)
  
- layout managers are based on the Strategy design pattern
Default layout managers for some containers

<table>
<thead>
<tr>
<th>Container</th>
<th>Layout Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>JApplet</td>
<td>BorderLayout (on its content pane)</td>
</tr>
<tr>
<td>JBox</td>
<td>BoxLayout</td>
</tr>
<tr>
<td>JDialog</td>
<td>BorderLayout (on its content pane)</td>
</tr>
<tr>
<td>JFrame</td>
<td>BorderLayout (on its content pane)</td>
</tr>
<tr>
<td>JPanel</td>
<td>FlowLayout</td>
</tr>
<tr>
<td>JWindow</td>
<td>BorderLayout (on its content pane)</td>
</tr>
</tbody>
</table>

GUI event processing

- graphical user interfaces generate events when the user interacts with the GUI
- information about a GUI event is stored in a special created object of a class that extends AWTEvent
- the use of event listeners in Java event handling is known as the event delegation model:
  - the processing of an event is delegated to a particular object in the program
  - when a GUI component is created, an appropriate event listener is registered with it
  - this event listener accepts responsibility for handling the component's events

GUI event processing (cont.)

- event sources: buttons, mouse, checkboxes, etc.
- event listeners: any classes interested in handling certain kinds of events
- a listener must
  - implement an appropriate listener interface
  - inform the source that it is interested in handling a certain type of events (registration)
- a listener may listen to several sources and different types of events
  - the source may be the listener for other events
- listeners can be full-fledged classes, inner classes, or anonymous inner classes

GUI event processing (cont.)

```java
component.addXListener (aListener);
```

- listeners may be restricted to only one, or there may be multiple listeners
  - the source component may also be a listener for the events generated by itself (is good/bad idea?)
- when an event occurs, the GUI system notifies the registered listeners
  - by calling each listener's appropriate event handling method (callback function):
    ```java
    void eventHandler (EventType event) { ... }
    ```
GUI event processing (cont.)

class MyClass implements EventTypeListener
{
    
    public void eventHandler(EventType e) {
        // processing of event ...
    }
}

• often event handlers are defined as objects of anonymous inner classes

   . . new EventTypeListener () {
        public void eventHandler(EventType e) {
            can access enclosing object }}

GUI event processing (cont.)

• in AWT, every event object is given an “event id”
  – the event id to determines the type of the listener to which the event should be dispatched and the event handler method to call
  – the original GUI target component is stored into the event object as the source of the event

• event-listener interfaces may provide multiple methods that separate events at different levels of detail (pressed, released, clicked)

• for such interfaces, an adapter class provides default empty implementations for every method
  – the programmer may then override only a selected method or methods and leave others as they are defined (i.e., empty)

Example: mouse events

public class MouseFrame extends JFrame
{
    public MouseFrame () {
        setSize (250, 300); setTitle ("Mouse moves");
        addMouseMotionListener (new MouseMotionAdapter () {
            public void mouseMoved(MouseEvent e) {
                System.out.println ("Mouse moved to [" + e.getX () + ", " + e.getY () + "]");
            }
        });
    }
}
Event objects and listener classes

<table>
<thead>
<tr>
<th>Event type</th>
<th>Listener type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionEvent</td>
<td>ActionListener</td>
</tr>
<tr>
<td>WindowEvent</td>
<td>WindowListener</td>
</tr>
<tr>
<td>MouseEvent</td>
<td>MouseListener</td>
</tr>
<tr>
<td>KeyEvent</td>
<td>KeyListener</td>
</tr>
<tr>
<td>ItemEvent</td>
<td>ItemListener</td>
</tr>
</tbody>
</table>

- XListener is an interface
- XAdapter is a class that implements the corresponding listener interface

GUI event processing (cont.)

- never do any **lengthy work** or **waiting** in event handlers
  - otherwise, your user interface freezes up, as events queue up waiting for delivery
  - instead, deliver fast a signal or data, and, if necessary, create a new **worker thread** to do any heavy stuff

- the Java event model is based on the **Observer** design pattern
  - the dependencies and data flows between the MVC participants (model -> view, etc.) are often realized using **Observers**

Creating a simplest visible frame (window)

```java
import javax.swing.JFrame;
public class SimpleFrame extends JFrame {
    public static void main(String[] args) {
        new SimpleFrame().setVisible(true);
    }
}
```

- **main** creates a window (with default properties), and then exits but leaves the window behind
  - only some built-in default behaviour: the window can be resized, moved, iconized, and closed
- closing a window may only **hide** the window; the application may still exist but it's now invisible

Terminating a graphical application

```java
import javax.swing.*;
import java.awt.event.*;
public class ClosingFrame extends JFrame {
    public static void main(String[] args) {
        JFrame frame = new ClosingFrame();
        frame.setSize(300, 200);
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);   // ignores event itself
            }
        });
        frame.setVisible(true); frame.toFront();
    }
}
```
Terminating a graphical application (cont.)

- as before, the main method creates a window (ClosingFrame), but now defines some properties (size), and then leaves the window behind to process user input as before
- to make sure that the application really terminates, an anonymous handler object is created to process the closing of the window
  -- when called on, the handler simply exits
  -- the default behaviour (just hiding) can also be changed by calling setDefaultCloseOperation (see below)

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

public class ButtonFrame extends JFrame {
    private JButton button = new JButton("Push");
    private JPanel panel = new JPanel();
    . .
    public static void main (String [] args) {
        JFrame frame = new ButtonFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible (true);   frame.pack ();
    . .
```

Adding components to a window (cont.)

- the main method creates a ButtonFrame window
- pack () causes the window to be (down) sized to fit the preferred size and layouts of its components; the window is then automatically redrawn
  -- JFrame uses the default layout BorderLayout (manages north, west, east, south, and center)
- a JPanel is placed into the content pane at its center (default), and a JButton is placed into the panel
  -- the placement into a panel uses FlowLayout (row-by-row)
- an anonymous handler object is created to process the clicks of the button

```
    . .
    public ButtonFrame () // constructor
    {
        panel.add (button);  // uses flow layout
        add (panel);         // default: CENTER
        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                String text = button.getText () + ", again";
                button.setText (text);
                pack (); }});
    }
```

Adding components to a window (cont.)

- a JPanel is placed into the content pane at its center (default), and a JButton is placed into the panel
  -- the placement into a panel uses FlowLayout (row-by-row)
- an anonymous handler object is created to process the clicks of the button