Event Handling

Events, Event Sources, and Event Classes

1. **Events**:
   - **Definition**: An event is an object that describes a change in the state of a source. It occurs when the user interacts with a graphical user interface (GUI) component.
   - **Types of Events**:
     - **ActionEvent**: Generated when a button is pressed, a list item is selected, or a menu item is selected.
     - **MouseEvent**: Generated when the mouse is clicked, pressed, released, moved, or dragged.
     - **KeyEvent**: Generated when a key is pressed, released, or typed.
     - **FocusEvent**: Generated when a component gains or loses keyboard focus.
     - **WindowEvent**: Generated when a window is opened, closed, activated, deactivated, iconified, or deiconified.
2. **Event Sources**:
   - **Definition**: An event source is an object that generates an event. This object recognizes that an event has occurred and notifies the event listeners.
   - **Common Event Sources**:
     - Buttons, Text Fields, Windows, Lists, Checkboxes, etc.

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

public class EventSourceDemo extends Frame implements ActionListener {
    Button button;

    public EventSourceDemo() {
        button = new Button("Click Me");
        button.setBounds(50, 100, 80, 30);
        button.addActionListener(this);
        add(button);
        setSize(200, 200);
        setLayout(null);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        System.out.println("Button Clicked");
    }

    public static void main(String[] args) {
```

```
        new EventSourceDemo();
    }
}
```

3. **Event Classes**:
    - **Hierarchy**: All event classes are derived from `java.util.EventObject` which is the superclass of all events.
    - **Common Event Classes**:
        - `ActionEvent`
        - `MouseEvent`
        - `KeyEvent`
        - `FocusEvent`
        - `WindowEvent`

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

public class EventClassDemo extends Frame implements ActionListener,
MouseListener {
    Button button;

    public EventClassDemo() {
        button = new Button("Click Me");
        button.setBounds(50, 100, 80, 30);
        button.addActionListener(this);
        button.addMouseListener(this);
        add(button);
        setSize(200, 200);
        setLayout(null);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        System.out.println("Action Event: Button Clicked");
    }

    public void mouseClicked(MouseEvent e) {
        System.out.println("Mouse Event: Button Clicked");
    }

    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}

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

```
    }
}
```

Event Listeners and Delegation Event Model

1. **Event Listeners**:
   - **Definition**: An event listener is an object that is notified when an event occurs. It has methods that are invoked when an event occurs.
   - **Common Event Listener Interfaces**:
     - `ActionListener`: Contains the method `actionPerformed(ActionEvent e)`.
     - `MouseListener`: Contains methods for mouse events such as `mouseClicked`, `mousePressed`, `mouseReleased`, `mouseEntered`, and `mouseExited`.
     - `KeyListener`: Contains methods for key events such as `keyPressed`, `keyReleased`, and `keyTyped`.
     - `FocusListener`: Contains methods `focusGained` and `focusLost`.
     - `WindowListener`: Contains methods for window events such as `windowOpened`, `windowClosing`, `windowClosed`, `windowIconified`, `windowDeiconified`, `windowActivated`, and `windowDeactivated`.

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

public class EventListenerDemo extends Frame implements ActionListener
{
    Button button;

    public EventListenerDemo() {
        button = new Button("Click Me");
        button.setBounds(50, 100, 80, 30);
        button.addActionListener(this);
        add(button);
        setSize(200, 200);
        setLayout(null);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        System.out.println("Button Clicked");
    }

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

```
    }
}
```

2. **Delegation Event Model**:
    - **Concept**: The delegation event model is based on event listeners that handle events generated by event sources. In this model:
        - A source generates an event and sends it to one or more listeners.
        - The listener waits until it receives an event.
        - Once an event is received, the listener processes the event and returns.
    - **Advantages**:
        - The event handling mechanism is clean and efficient.
        - The model is highly flexible and reusable.
    - **Steps to Use Delegation Event Model**:
        1. **Implement Listener Interface**: Implement one or more listener interfaces in the listener class.
        2. **Register Listener**: Register the listener object with an event source using methods like `addActionListener`, `addMouseListener`, etc.
        3. **Event Handling Method**: Override the event handling method defined in the listener interface to handle the event.

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

public class DelegationEventModelDemo extends Frame implements
ActionListener {
    Button button;

    public DelegationEventModelDemo() {
        button = new Button("Click Me");
        button.setBounds(50, 100, 80, 30);
        button.addActionListener(this);  // Registering the listener
        add(button);
        setSize(200, 200);
        setLayout(null);
        setVisible(true);
    }

    @Override
    public void actionPerformed(ActionEvent e) {  // Event handling
method
        System.out.println("Button Clicked");
    }

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

```
    }
}
```

Handling Mouse and Keyboard Events

1. **Handling Mouse Events**:
   - **MouseEvent**: Generated when mouse actions such as clicks, presses, releases, movements, and drags occur.
   - **MouseListener Interface**:
     - **Methods**:
       - `mouseClicked(MouseEvent e)` : Invoked when the mouse button is clicked (pressed and released) on a component.
       - `mousePressed(MouseEvent e)` : Invoked when a mouse button is pressed on a component.
       - `mouseReleased(MouseEvent e)` : Invoked when a mouse button is released on a component.
       - `mouseEntered(MouseEvent e)` : Invoked when the mouse enters a component.
       - `mouseExited(MouseEvent e)` : Invoked when the mouse exits a component.

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

public class MouseEventDemo extends Frame implements MouseListener {
    public MouseEventDemo() {
        setSize(300, 200);
        setLayout(new FlowLayout());
        addMouseListener(this);
        setVisible(true);
    }

    public void mouseClicked(MouseEvent e) {
        System.out.println("Mouse Clicked at (" + e.getX() + ", " +
e.getY() + ")");
    }

    public void mousePressed(MouseEvent e) {
        System.out.println("Mouse Pressed at (" + e.getX() + ", " +
e.getY() + ")");
    }
```

```
    public void mouseReleased(MouseEvent e) {
        System.out.println("Mouse Released at (" + e.getX() + ", " +
e.getY() + ")");
    }

    public void mouseEntered(MouseEvent e) {
        System.out.println("Mouse Entered");
    }

    public void mouseExited(MouseEvent e) {
        System.out.println("Mouse Exited");
    }

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

- **MouseMotionListener Interface**:
  - **Methods**:
    - `mouseDragged(MouseEvent e)` : Invoked when the mouse is dragged.
    - `mouseMoved(MouseEvent e)` : Invoked when the mouse is moved.

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

public class MouseMotionEventDemo extends Frame implements
MouseMotionListener {
    public MouseMotionEventDemo() {
        setSize(300, 200);
        setLayout(new FlowLayout());
        addMouseMotionListener(this);
        setVisible(true);
    }

    public void mouseDragged(MouseEvent e) {
        System.out.println("Mouse Dragged at (" + e.getX() + ", " +
e.getY() + ")");
    }

    public void mouseMoved(MouseEvent e) {
        System.out.println("Mouse Moved at (" + e.getX() + ", " +
e.getY() + ")");
    }

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

```
    }
}
```

2. **Handling Keyboard Events**:
    - **KeyEvent**: Generated when a key is pressed, released, or typed.
    - **KeyListener Interface**:
        - **Methods**:
            - `keyPressed(KeyEvent e)` : Invoked when a key is pressed.
            - `keyReleased(KeyEvent e)` : Invoked when a key is released.
            - `keyTyped(KeyEvent e)` : Invoked when a key is typed.

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

public class KeyEventDemo extends Frame implements KeyListener {
    public KeyEventDemo() {
        setSize(300, 200);
        setLayout(new FlowLayout());
        addKeyListener(this);
        setVisible(true);
    }

    public void keyPressed(KeyEvent e) {
        System.out.println("Key Pressed: " + e.getKeyChar());
    }

    public void keyReleased(KeyEvent e) {
        System.out.println("Key Released: " + e.getKeyChar());
    }

    public void keyTyped(KeyEvent e) {
        System.out.println("Key Typed: " + e.getKeyChar());
    }

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

Adapter Classes and AWT Class Hierarchy

1. **Adapter Classes**:

- **Purpose**: Adapter classes provide default implementations of the methods in listener interfaces. They are useful when you want to override only a few methods from the listener interface.
- **Common Adapter Classes**:
  - `MouseAdapter` : Provides default implementations for `MouseListener` and `MouseMotionListener` methods.
  - `KeyAdapter` : Provides default implementations for `KeyListener` methods.
  - `WindowAdapter` : Provides default implementations for `WindowListener` methods.
  - `ComponentAdapter` : Provides default implementations for `ComponentListener` methods.

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

public class MouseAdapterDemo extends Frame {
    public MouseAdapterDemo() {
        setSize(300, 200);
        setLayout(new FlowLayout());
        addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                System.out.println("Mouse Clicked at (" + e.getX() + ",
" + e.getY() + ")");
            }
        });
        setVisible(true);
    }

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

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

public class KeyAdapterDemo extends Frame {
    public KeyAdapterDemo() {
        setSize(300, 200);
        setLayout(new FlowLayout());
        addKeyListener(new KeyAdapter() {
            public void keyPressed(KeyEvent e) {
                System.out.println("Key Pressed: " + e.getKeyChar());
            }
        });
        setVisible(true);
```

```
    }

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

2. **AWT Class Hierarchy**:
    - **Component Hierarchy**:
        - `Object`
            - `Component`
                - `Container`
                    - `Window`
                        - `Frame`
                        - `Dialog`
                - `Button`
                - `Label`
                - `TextField`
                - `TextArea`
                - `CheckBox`
                - `RadioButton`
                - `List`
                - `ScrollPane`
                - `Menu`
                    - `MenuBar`
                    - `MenuItem`
                - `Panel`
                - `Canvas`
    - **Important Classes**:
        - `Component` : Base class for all AWT components.
        - `Container` : A component that can contain other components.
        - `Window` : The top-level window.
        - `Frame` : A top-level window with a title.
        - `Dialog` : A pop-up window for dialogs.
        - `Panel` : A container for organizing components.
        - `Menu` : A pull-down menu.

```
import java.awt.*;

public class AWTClassHierarchyDemo {
    public static void main(String[] args) {
```

```
        Frame frame = new Frame("AWT Class Hierarchy Demo");
        Button button = new Button("Click Me");
        Panel panel = new Panel();
        panel.add(button);
        frame.add(panel);
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

User Interface Components

AWT provides various user interface components that are essential for building graphical user interfaces (GUIs). Here are some commonly used AWT components:

1. **Button**:
   - **Purpose**: Represents a push button in a GUI.
   - **Constructor**: `Button(String label)`
   - **Example**:

```
Button button = new Button("Click Me");
```

2. **Label**:
   - **Purpose**: Displays a short string or an image.
   - **Constructor**: `Label(String text)`
   - **Example**:

```
Label label = new Label("Hello, World!");
```

3. **TextField**:
   - **Purpose**: Allows the user to enter a single line of text.
   - **Constructor**: `TextField(String text, int columns)`
   - **Example**:

```
TextField textField = new TextField("Default Text", 20);
```

4. **TextArea**:
   - **Purpose**: Allows the user to enter multiple lines of text.
   - **Constructor**: `TextArea(String text, int rows, int columns)`
   - **Example**:

```
TextArea textArea = new TextArea("Default Text", 5, 30);
```

5. **CheckBox**:
   - **Purpose**: Represents a checkbox that can be selected or deselected.
   - **Constructor**: `Checkbox(String label)`
   - **Example**:

```
Checkbox checkbox = new Checkbox("Option");
```

6. **RadioButton**:
   - **Purpose**: Represents a radio button that can be selected among a group of radio buttons.
   - **Constructor**: `Checkbox(String label, boolean state, CheckboxGroup group)`
   - **Example**:

```
CheckboxGroup group = new CheckboxGroup();
Checkbox radioButton = new Checkbox("Option 1", group, false);
```

7. **List**:
   - **Purpose**: Displays a list of items that can be selected.
   - **Constructor**: `List(int rows, boolean multipleMode)`
   - **Example**:

```
List list = new List(4, true);
list.add("Item 1");
list.add("Item 2");
```

8. **Choice**:
   - **Purpose**: Displays a dropdown menu of items.
   - **Constructor**: `Choice()`
   - **Example**:

```
Choice choice = new Choice();
choice.add("Option 1");
choice.add("Option 2");
```

9. **Scrollbar**:
   - **Purpose**: Provides a scrollable interface.
   - **Constructor**: `Scrollbar(int orientation, int value, int visibleAmount, int minimum, int maximum)`

- **Example**:

```
Scrollbar scrollbar = new Scrollbar(Scrollbar.VERTICAL, 0, 10, 0, 100);
```

10. **Canvas**:
    - **Purpose**: A blank area where custom drawing can be done.
    - **Constructor**: `Canvas()`
    - **Example**:

```
Canvas canvas = new Canvas();
```

11. **Panel**:
    - **Purpose**: A container used to group components.
    - **Constructor**: `Panel()`
    - **Example**:

```
Panel panel = new Panel();
```

12. **Frame**:
    - **Purpose**: Represents a top-level window with a title.
    - **Constructor**: `Frame(String title)`
    - **Example**:

```
Frame frame = new Frame("My Frame");
```

13. **Dialog**:
    - **Purpose**: A pop-up window used for dialogs.
    - **Constructor**: `Dialog(Frame owner, String title, boolean modal)`
    - **Example**:

```
Dialog dialog = new Dialog(frame, "Dialog Title", true);
```

Layout Managers

Layout Managers are used to arrange components within containers. They determine how components are laid out when the container is resized or when components are added. Here are some common layout managers:

1. **BorderLayout**:
   - **Purpose**: Arranges components in five regions: North, South, East, West, and Center.
   - **Example**:

```
Frame frame = new Frame();
frame.setLayout(new BorderLayout());
frame.add(new Button("North"), BorderLayout.NORTH);
frame.add(new Button("South"), BorderLayout.SOUTH);
frame.add(new Button("East"), BorderLayout.EAST);
frame.add(new Button("West"), BorderLayout.WEST);
frame.add(new Button("Center"), BorderLayout.CENTER);
```

2. **FlowLayout**:
   - **Purpose**: Arranges components in a left-to-right flow, much like lines of text in a paragraph.
   - **Example**:

```
Panel panel = new Panel();
panel.setLayout(new FlowLayout());
panel.add(new Button("Button 1"));
panel.add(new Button("Button 2"));
panel.add(new Button("Button 3"));
```

3. **GridLayout**:
   - **Purpose**: Arranges components in a grid of rows and columns, with each cell of the grid having the same size.
   - **Example**:

```
Panel panel = new Panel();
panel.setLayout(new GridLayout(3, 2));  // 3 rows, 2 columns
panel.add(new Button("Button 1"));
panel.add(new Button("Button 2"));
panel.add(new Button("Button 3"));
panel.add(new Button("Button 4"));
panel.add(new Button("Button 5"));
panel.add(new Button("Button 6"));
```

4. **CardLayout**:
   - **Purpose**: Allows multiple components to be stacked on top of each other, with only one component visible at a time. Components are accessed by their name or index.
   - **Example**:

```
Panel cardPanel = new Panel();
CardLayout cardLayout = new CardLayout();
cardPanel.setLayout(cardLayout);

cardPanel.add(new Button("Card 1"), "Card1");
cardPanel.add(new Button("Card 2"), "Card2");

cardLayout.show(cardPanel, "Card2");  // Show Card2
```

5. **GridBagLayout**:
   - **Purpose**: Provides a flexible grid-based layout manager with support for complex layouts. Components can span multiple rows and columns.
   - **Example**:

```
Panel panel = new Panel();
panel.setLayout(new GridBagLayout());
GridBagConstraints gbc = new GridBagConstraints();

gbc.gridx = 0;
gbc.gridy = 0;
panel.add(new Button("Button 1"), gbc);

gbc.gridx = 1;
gbc.gridy = 0;
panel.add(new Button("Button 2"), gbc);

gbc.gridx = 0;
gbc.gridy = 1;
gbc.gridwidth = 2; // Span across two columns
panel.add(new Button("Button 3"), gbc);
```
