Java AWT & Swing

Unit IV & Unit V — Comprehensive Lecture Notes

Topics: Frame · Color · Fonts · Layout Managers · Event Handling · EDM · Mouse & Keyboard · Adapter Classes · Inner Classes · Swing Components

Core Java Programming | B.Sc / BCA / MCA

UNIT IV — AWT: Abstract Window Toolkit

1. Introduction to AWT

The Abstract Window Toolkit (AWT) is Java's original platform-dependent windowing, graphics, and user-interface widget toolkit. Introduced in Java 1.0, AWT provides the foundational classes for building Graphical User Interfaces (GUI) in Java. It is part of the java.awt package and enables developers to create windows, buttons, text fields, menus, and other GUI components.

AWT components are called 'heavyweight' components because they rely on the native operating system's windowing system. This means that each AWT component has a corresponding native peer object managed by the underlying OS. As a result, the appearance of AWT applications may differ across platforms — a button on Windows may look different from the same button on macOS or Linux.

> Key Package: java.awt — Contains all AWT classes. java.awt.event — Contains all event-handling classes.

2. Frame Class

2.1 What is a Frame?

A Frame is a top-level window with a title bar, borders, and optional menu bar. It is the primary container used to create standalone AWT applications. The Frame class is part of the java.awt package and extends the Window class, which itself extends the Container class.

The Frame provides the main application window that holds all other GUI components. When you create an AWT application, you typically start by creating a Frame object, setting its properties, adding components to it, and making it visible.

2.2 Frame Class Hierarchy

Understanding the class hierarchy helps in knowing what capabilities are inherited:

- java.lang.Object
    - java.awt.Component — Provides basic display and event-handling capabilities
        - java.awt.Container — Allows adding other components

java.awt.Window — A top-level window with no borders or menu bar

java.awt.Frame — A Window with title bar, border, and (optionally) menu bar

2.3 Creating a Frame

There are two common ways to create a Frame: by instantiating the Frame class directly or by extending it. Extending Frame is the preferred approach for building a real application, as it allows you to define the entire GUI within the constructor.

```
import java.awt.*;

// Method 1: Extending Frame
class MyApp extends Frame {
    MyApp(String title) {
        super(title);                     // Set window title
        setSize(600, 400);                // Width x Height in pixels
        setLocation(200, 100);            // X, Y position on screen
        setLayout(new FlowLayout());      // Set layout manager
        setBackground(Color.LIGHT_GRAY);  // Background color
        add(new Button("Click Me"));      // Add component
        setVisible(true);                 // Make visible (MUST be last)
    }
    public static void main(String[] args) {
        new MyApp("My AWT Application");
    }
}
```

2.4 Important Frame Methods

Method	Description
setTitle(String)	Sets the title displayed in the title bar of the frame.
setSize(int w, int h)	Sets the width and height of the frame in pixels.
setLocation(int x, int y)	Positions the frame at (x, y) on screen; (0,0) is top-left.
setVisible(boolean)	Makes the frame visible (true) or hidden (false).
setResizable(boolean)	Enables/disables the user's ability to resize the frame.
setLayout(LayoutManager)	Sets the layout manager to arrange components.
setBackground(Color)	Sets the background color of the frame's content area.
add(Component)	Adds a component to the frame.
dispose()	Destroys the frame and releases all its resources.

3. Color in AWT

3.1 The Color Class

The java.awt.Color class encapsulates colors using the RGB (Red-Green-Blue) or RGBA (with Alpha for transparency) color models. Every component in AWT can have a foreground color (text/drawing color) and a background color. The Color class provides numerous predefined constants as well as the ability to define custom colors.

3.2 Predefined Color Constants

The Color class includes the following built-in constants for commonly used colors:

Constant	Constant	Constant	Constant
Color.RED	Color.GREEN	Color.BLUE	Color.YELLOW
Color.ORANGE	Color.PINK	Color.CYAN	Color.MAGENTA
Color.BLACK	Color.WHITE	Color.GRAY	Color.LIGHT_GRAY

3.3 Custom Colors & Usage

You can create custom colors by specifying RGB values (each ranging from 0 to 255) in the Color constructor. An optional fourth parameter, alpha, controls transparency (0 = fully transparent, 255 = fully opaque).

```
// Creating custom colors
Color customOrange = new Color(255, 165, 0);       // RGB
Color semiTransparent = new Color(0, 128, 255, 128); // RGBA

// Applying colors to components
Button btn = new Button("Submit");
btn.setBackground(Color.BLUE);      // Background
btn.setForeground(Color.WHITE);     // Text color

// Getting color components
int r = customOrange.getRed();     // Returns 255
int g = customOrange.getGreen();   // Returns 165
int b = customOrange.getBlue();    // Returns 0

// Darker / brighter variations
Color darker  = customOrange.darker();
Color lighter = customOrange.brighter();
```

4. Fonts in AWT

4.1 The Font Class

The java.awt.Font class represents fonts used to render text in AWT components. A Font object is characterized by three attributes: the font name (family), the style (Plain, Bold, Italic, or Bold+Italic), and the size (in points). Fonts are used with Component.setFont() or directly in Graphics context when drawing text on a Canvas.

4.2 Font Styles

The Font class provides the following style constants:

- Font.PLAIN — Regular, undecorated text.
- Font.BOLD — Bold (thicker stroke) text.
- Font.ITALIC — Slanted italic text.
- Font.BOLD + Font.ITALIC — Combined bold and italic (simply add the two constants).

4.3 Logical Font Names

AWT defines logical font names that map to actual system fonts on different platforms, ensuring portability:

Logical Name	Typical Mapping
Serif	Times New Roman (Windows) / Times (macOS/Linux)
SansSerif	Arial (Windows) / Helvetica (macOS/Linux)
Monospaced	Courier New (Windows) / Courier (macOS/Linux)
Dialog	Default dialog font — typically Arial or similar
DialogInput	Monospaced font suitable for user input fields

```
// Creating Font objects
Font titleFont  = new Font("Serif",     Font.BOLD,          24);
Font bodyFont   = new Font("SansSerif", Font.PLAIN,         14);
Font codeFont   = new Font("Monospaced",Font.PLAIN,         13);
Font emphFont   = new Font("SansSerif", Font.BOLD + Font.ITALIC, 16);

// Applying fonts to components
Label title = new Label("Welcome");
title.setFont(titleFont);

TextField input = new TextField(20);
input.setFont(codeFont);

// Getting font metrics (for precise positioning)
FontMetrics fm = getFontMetrics(titleFont);
int textWidth  = fm.stringWidth("Hello");  // width in pixels
int textHeight = fm.getHeight();             // total line height
```

5. Layout Managers

5.1 What is a Layout Manager?

A Layout Manager is an object that implements the LayoutManager interface and controls how components are positioned and sized within a container (like Frame or Panel). Instead of specifying exact pixel coordinates for each component (absolute positioning), layout managers calculate positions automatically based on rules. This makes GUIs adaptable to different screen sizes and resolutions.

> Why use Layout Managers? They ensure your GUI remains functional and visually correct when the window is resized or displayed on different screen resolutions.

5.2 FlowLayout

FlowLayout is the default layout for Panel and Applet. It arranges components in a left-to-right, top-to-bottom flow — similar to how words flow in a paragraph. When there is no more space in a row, components wrap to the next line. Alignment can be set to LEFT, CENTER (default), or RIGHT.

```
// FlowLayout with center alignment and gaps
setLayout(new FlowLayout(FlowLayout.CENTER, 10, 10));
// Parameters: alignment, horizontal gap, vertical gap

add(new Button("Button 1"));
add(new Button("Button 2"));
add(new Button("Button 3"));
// Result: B1  B2  B3  (all in one row if space permits)
```

5.3 BorderLayout

BorderLayout is the default layout for Frame and Dialog. It divides the container into five distinct regions: NORTH (top), SOUTH (bottom), EAST (right), WEST (left), and CENTER. Only one component can occupy each region. The CENTER region takes all remaining space after the other regions are allocated. Unused regions are not allocated any space.

```
setLayout(new BorderLayout(5, 5)); // hgap=5, vgap=5

add(new Button("Header"),  BorderLayout.NORTH);
add(new Button("Footer"),  BorderLayout.SOUTH);
add(new Button("Left"),    BorderLayout.WEST);
add(new Button("Right"),   BorderLayout.EAST);
add(new TextArea(),         BorderLayout.CENTER); // Main content area
```

5.4 GridLayout

GridLayout arranges components in a rectangular grid of equal-sized cells. You specify the number of rows and columns. Components are added left-to-right, top-to-bottom. All cells have the same size — determined by the largest component. This layout is ideal for calculator buttons, keyboard layouts, or any uniform grid of components.

```
// 3 rows, 3 columns, 5px gaps
setLayout(new GridLayout(3, 3, 5, 5));

```

```
for (int i = 1; i <= 9; i++) {
    add(new Button(String.valueOf(i)));
}
// Result: neatly arranged 3x3 grid of buttons
```

5.5 CardLayout

CardLayout treats the container as a stack of cards where only one card (panel of components) is visible at a time. It is useful for creating tabbed interfaces, wizards, or multi-step forms without using the JTabbedPane component. You can navigate between cards programmatically using first(), last(), next(), previous(), and show() methods.

```
CardLayout card = new CardLayout();
Panel container = new Panel(card);

container.add(new Panel() {{ add(new Label("Page 1")); }}, "page1");
container.add(new Panel() {{ add(new Label("Page 2")); }}, "page2");

card.show(container, "page2"); // Switch to page 2
card.next(container);          // Go to next card
card.first(container);         // Go to first card
```

5.6 GridBagLayout

GridBagLayout is the most powerful and flexible layout manager in AWT. It arranges components in a grid, but — unlike GridLayout — cells can have different sizes, and components can span multiple rows or columns. Each component's layout is controlled by a GridBagConstraints object that specifies position, size, alignment, padding, and weight (how space is distributed when the window is resized).

```
setLayout(new GridBagLayout());
GridBagConstraints gbc = new GridBagConstraints();

gbc.gridx = 0;  gbc.gridy = 0;           // Cell position
gbc.gridwidth = 2;                        // Span 2 columns
gbc.fill = GridBagConstraints.HORIZONTAL; // Fill horizontally
gbc.insets = new Insets(5, 5, 5, 5);     // External padding
add(new TextField("Name"), gbc);

gbc.gridx = 0;  gbc.gridy = 1;
gbc.gridwidth = 1;
gbc.weightx = 0.3;  // This column gets 30% of extra space
add(new Label("Email:"), gbc);
```

Event Handling in Java AWT

6. Event Handling

6.1 What is an Event?

An event is an object that describes a state change in a source component. Events are generated when a user interacts with a GUI component — for example, clicking a button, typing in a text field, moving the mouse, pressing a key, or selecting an item from a list. In Java AWT, all events are represented as objects that inherit from the java.util.EventObject class.

Events carry information about the interaction — who generated it (the source), what happened (event type), and any associated data (like mouse coordinates or key codes). The event-handling mechanism in Java follows a well-defined design called the Event Delegation Model.

6.2 Event Sources

An event source is any component that can generate events. When a user interacts with a component, the component creates an event object and passes it to registered listener objects. Every AWT component can be an event source for one or more event types:

Event Source	Event Class	Triggered When
Button	ActionEvent	User clicks the button.
TextField	ActionEvent	User presses Enter in the field.
Checkbox	ItemEvent	User checks or unchecks the box.
Choice / List	ItemEvent	User selects an item.
Scrollbar	AdjustmentEvent	User moves the scrollbar.
Any Component	MouseEvent	Mouse is clicked, pressed, released, entered, exited.
Any Component	KeyEvent	A key is pressed, released, or typed.
Frame / Window	WindowEvent	Window is opened, closed, minimized, etc.

6.3 Event Listeners

An event listener is an interface that defines one or more callback methods that are called when a specific event occurs. To handle an event, you must:

1. Implement the appropriate listener interface (e.g., ActionListener, MouseListener).
2. Override all abstract methods declared in the interface.
3. Register the listener with the source component using an addXxxListener() method.

When the user triggers an event, the AWT event thread automatically calls the corresponding method in the registered listener.

6.4 The Event Delegation Model (EDM)

The Event Delegation Model (introduced in Java 1.1) is the standard mechanism for handling events in Java GUI programming. In this model, the responsibility of handling events is delegated from the event source to one or more listener objects. This cleanly separates UI logic (the component) from application logic (the handler).

The EDM works through three key participants:

- Event Source: The component that generates the event (e.g., a Button).
- Event Object: Carries information about the event (e.g., ActionEvent).
- Event Listener: The object that receives and processes the event.

> The EDM replaced the older 'inheritance-based' event model from Java 1.0, making event handling more flexible, efficient, and maintainable.

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

// Complete EDM Example
class EDMDemo extends Frame implements ActionListener {
    Label result;

    EDMDemo() {
        setLayout(new FlowLayout());
        setSize(300, 200);

        Button btn = new Button("Click Me");
        result = new Label("Waiting...");

        // Step 3: Register listener with source
        btn.addActionListener(this);

        add(btn);
        add(result);
        setVisible(true);
    }

    // Step 1+2: Implement listener interface & override method
    @Override
    public void actionPerformed(ActionEvent e) {
        result.setText("Button clicked at: " + e.getWhen());
    }
}
```

7. Handling Mouse and Keyboard Events

7.1 MouseListener Interface

The MouseListener interface handles mouse button events. It declares five abstract methods that must all be overridden when implementing this interface. Mouse events are generated for any component the mouse interacts with, making it essential for drawing applications, game development, and interactive UIs.

Method	When It's Called
mouseClicked(MouseEvent e)	Mouse button was pressed and released at the same position.
mousePressed(MouseEvent e)	Mouse button was pressed down (not yet released).
mouseReleased(MouseEvent e)	Mouse button was released.
mouseEntered(MouseEvent e)	Cursor moved into the component's area.
mouseExited(MouseEvent e)	Cursor moved out of the component's area.

7.2 MouseMotionListener Interface

MouseMotionListener handles mouse movement events — separate from button events. This interface is used for drag-and-drop, tracking cursor position, or drawing freehand paths.

- mouseMoved(MouseEvent e) — Called when the mouse is moved without any button pressed.
- mouseDragged(MouseEvent e) — Called when the mouse is moved with a button pressed (dragging).

7.3 Useful MouseEvent Methods

```
public void mouseClicked(MouseEvent e) {
    int x = e.getX();              // X coordinate relative to component
    int y = e.getY();              // Y coordinate relative to component
    int button = e.getButton();    // MouseEvent.BUTTON1/BUTTON2/BUTTON3
    int clicks = e.getClickCount(); // 1 = single, 2 = double click
    boolean isDouble = (clicks == 2);

    System.out.println("Clicked at (" + x + "," + y + ")");
    if (isDouble) System.out.println("Double-clicked!");
}
```

7.4 KeyListener Interface

The KeyListener interface handles keyboard events. It declares three methods. Key events are generated for the component that currently has keyboard focus. Every AWT component can receive keyboard events once it has focus (typically gained by clicking on it or using Tab to navigate).

Method	When It's Called
keyPressed(KeyEvent e)	A key on the keyboard is pressed down.
keyReleased(KeyEvent e)	A previously pressed key is released.

Method	When It's Called
keyTyped(KeyEvent e)	A Unicode character was generated by pressing a key. Not called for action keys (F1, arrows, etc.).

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

class KeyDemo extends Frame implements KeyListener {
    Label info = new Label("");

    KeyDemo() {
        add(info);
        addKeyListener(this); // Register on frame itself
        setFocusable(true);   // Frame must be focusable
        setVisible(true);
    }

    public void keyPressed(KeyEvent e) {
        int code = e.getKeyCode();          // Virtual key code
        String name = KeyEvent.getKeyText(code); // Human-readable name

        if (code == KeyEvent.VK_ESCAPE) System.exit(0); // ESC to quit
        if (e.isShiftDown() && code == KeyEvent.VK_A)
            info.setText("Shift+A pressed!");
    }
    public void keyReleased(KeyEvent e) { }
    public void keyTyped(KeyEvent e) {
        info.setText("Typed: " + e.getKeyChar());
    }
}
```

8. Adapter Classes

8.1 The Problem with Listener Interfaces

Every Listener interface may declare multiple abstract methods. For example, MouseListener has five methods and WindowListener has seven methods. When a class implements a listener interface, it must provide concrete implementations for ALL declared methods — even if you only need one or two of them. This results in unnecessarily verbose code with many empty method bodies cluttering the class.

```
// Without Adapter — Must implement ALL 7 WindowListener methods
class MyFrame extends Frame implements WindowListener {
    public void windowOpened(WindowEvent e)    { } // not needed
    public void windowIconified(WindowEvent e){ } // not needed
    public void windowDeiconified(WindowEvent e){}// not needed
    public void windowActivated(WindowEvent e){ } // not needed
    public void windowDeactivated(WindowEvent e){}// not needed
    public void windowClosed(WindowEvent e)    { } // not needed
    public void windowClosing(WindowEvent e) {    // ONLY this is needed
        System.exit(0);
    }
}
```

8.2 Solution: Adapter Classes

Java provides Adapter classes as a convenience. Each Adapter class is an abstract class that implements a corresponding Listener interface and provides default (empty) implementations for all its methods. You simply extend the Adapter class and override only the methods you actually need, keeping your code clean and concise.

Adapter Class	Implements
MouseAdapter	MouseListener (5 methods)
MouseMotionAdapter	MouseMotionListener (2 methods)
KeyAdapter	KeyListener (3 methods)
WindowAdapter	WindowListener (7 methods)
FocusAdapter	FocusListener (2 methods)
ComponentAdapter	ComponentListener (4 methods)

```java
// With Adapter — Override only what you need
class MyFrame extends Frame {
    MyFrame() {
        // Anonymous WindowAdapter — only close behavior needed
        addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                dispose(); // Close window gracefully
                System.exit(0);
            }
        });

        // Anonymous MouseAdapter — only click tracking needed
        addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                System.out.println("Clicked at: " + e.getX() + "," + e.getY());
            }
        });
    }
}
```

9. Inner Classes for Event Handling

9.1 What are Inner Classes?

An inner class is a class defined within another class. In the context of event handling, inner classes allow you to define listener logic close to the component it handles, improving code readability and organization. Inner classes have access to all private members (fields and methods) of their enclosing class — a significant advantage for GUI programming where event handlers often need to modify the outer class's UI.

9.2 Types of Inner Classes Used in Event Handling

Named Inner Classes

A named inner class is defined with a class keyword inside the outer class. It is reusable and can be applied to multiple components. Named inner classes are appropriate when the event-handling logic is complex or shared.

```
class MainFrame extends Frame {
    Label statusBar = new Label("Ready");

    // Named inner class implementing listener
    class ButtonHandler implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent e) {
            statusBar.setText("Button: " + e.getActionCommand());
        }
    }

    MainFrame() {
        Button b1 = new Button("Save");
        Button b2 = new Button("Load");
        ButtonHandler handler = new ButtonHandler();
        b1.addActionListener(handler); // Reuse same handler
        b2.addActionListener(handler);
    }
}
```

Anonymous Inner Classes

An anonymous inner class is defined and instantiated in a single expression, without a name. It is ideal for short, one-off event handlers. This is the most common style seen in older Java GUI code.

```
Button btn = new Button("Delete");
btn.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        // Logic directly here — no need for a separate class
        System.out.println("Delete clicked");
    }
});
```

Lambda Expressions (Java 8+)

For functional interfaces (interfaces with exactly one abstract method, like ActionListener), Java 8 introduced lambda expressions — the most concise way to write event handlers. Lambdas effectively replace anonymous inner classes for simple event handling.

```
// Lambda replaces anonymous inner class completely
Button btn = new Button("Submit");
btn.addActionListener(e -> {
    System.out.println("Submitted! Command: " + e.getActionCommand());
    System.out.println("Source: " + e.getSource());
});
```

```
// Even shorter for single expression:
btn.addActionListener(e -> System.out.println("Done!"));
```

UNIT V — Swing: Java's Modern GUI Toolkit

10. Introduction to Swing

10.1 What is Swing?

Swing is a comprehensive, platform-independent GUI toolkit introduced in Java 1.2 as an extension of AWT. It is part of the Java Foundation Classes (JFC) and is available in the javax.swing package. Swing provides a rich set of components (often called widgets) that are entirely written in Java, making them 'lightweight' — they don't rely on native OS windowing components.

Because Swing components are drawn entirely by Java, they look and behave identically across all platforms. Moreover, Swing supports Pluggable Look and Feel (PLAF), allowing the entire appearance of the application to be switched at runtime to match the native OS style (Windows, macOS, Motif) or to use Swing's default 'Metal' look, or any custom look and feel.

10.2 Key Differences: AWT vs Swing

Feature	AWT	Swing
Component type	Heavyweight (native OS peers)	Lightweight (pure Java rendering)
Package	java.awt	javax.swing
Class prefix	Button, TextField, etc.	JButton, JTextField, etc.
Look & Feel	Platform-native only	Pluggable (Metal, Nimbus, System)
Rich components	Basic set only	Extensive (JTable, JTree, JSpinner, etc.)
Architecture	No MVC	MVC (Model-View-Controller)
Double buffering	Must implement manually	Built-in, flicker-free rendering
Accessibility	Limited	Full accessibility support

10.3 Swing Component Hierarchy

All Swing components extend JComponent (except top-level containers like JFrame and JDialog), which itself extends java.awt.Container. This means Swing components can also use AWT layout managers and event listeners.

- java.lang.Object
  - java.awt.Component
    - java.awt.Container

javax.swing.JComponent (base for all non-top-level Swing components)
JButton, JLabel, JTextField, JTextArea, JList, JComboBox, JPanel, ...

- javax.swing.JFrame (extends java.awt.Frame directly)
- javax.swing.JDialog (extends java.awt.Dialog directly)
- javax.swing.JWindow (extends java.awt.Window directly)

11. Top-Level Containers

11.1 JFrame

JFrame is the primary top-level window for Swing applications. It is the Swing equivalent of AWT's Frame. JFrame includes a content pane — a JPanel that serves as the actual container for all components. Unlike AWT Frame, components should be added to the content pane using getContentPane().add() (though in modern Java, calling add() on the JFrame itself works as a shortcut).

One critical feature of JFrame is the default close operation, which specifies what happens when the user clicks the window's close (X) button:

- JFrame.DO_NOTHING_ON_CLOSE — Ignore the close button click.
- JFrame.HIDE_ON_CLOSE — Hide the window but keep the application running.
- JFrame.DISPOSE_ON_CLOSE — Dispose the window; application continues if other windows exist.
- JFrame.EXIT_ON_CLOSE — Call System.exit(0) and terminate the entire application.

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

public class MySwingApp extends JFrame {

    public MySwingApp() {
        setTitle("My Swing Application");
        setSize(600, 400);
        setLocationRelativeTo(null);  // Center on screen
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Content pane configuration
        Container cp = getContentPane();
        cp.setLayout(new BorderLayout());
        cp.setBackground(Color.WHITE);

        cp.add(new JLabel("Hello Swing!", JLabel.CENTER), BorderLayout.CENTER);
        cp.add(new JButton("Click"), BorderLayout.SOUTH);

        setVisible(true);  // Always last
    }

    public static void main(String[] args) {
        // Best practice: Create GUI on Event Dispatch Thread
        SwingUtilities.invokeLater(() -> new MySwingApp());
    }
}
```

11.2 JDialog

JDialog is used to create dialog boxes — secondary windows that appear over the main window to present information or collect user input. JDialog can be modal (blocks input to other windows until dismissed) or non-modal. The JOptionPane class provides convenient static methods for creating standard message, input, and confirmation dialogs without manually constructing a JDialog.

```
// Modal dialog (blocks parent window)
JDialog dialog = new JDialog(parentFrame, "Settings", true); // true = modal
dialog.setSize(300, 200);
dialog.setLocationRelativeTo(parentFrame);
dialog.add(new JLabel("Configure settings here"));
dialog.setVisible(true); // Execution pauses here until dialog closes

// Using JOptionPane shortcuts
JOptionPane.showMessageDialog(null, "File saved!", "Info",
    JOptionPane.INFORMATION_MESSAGE);

int choice = JOptionPane.showConfirmDialog(null,
    "Are you sure you want to delete?", "Confirm",
    JOptionPane.YES_NO_OPTION);
if (choice == JOptionPane.YES_OPTION) performDelete();

String name = JOptionPane.showInputDialog("Enter your name:");
```

11.3 JWindow

JWindow is a top-level container that, unlike JFrame, has no title bar, borders, or system menu. It is typically used for splash screens displayed when an application is loading, or for custom popup windows that need a completely custom appearance. JWindow requires a parent JFrame to be associated with it.

```
JFrame parent = new JFrame();
JWindow splash = new JWindow(parent);
splash.setSize(400, 300);
splash.setLocationRelativeTo(null); // Center screen

// Add content (logo, progress bar, etc.)
splash.add(new JLabel("Loading Application...", JLabel.CENTER));
splash.setVisible(true);

// Simulate loading
Thread.sleep(3000);
splash.dispose();
parent.setVisible(true); // Show main window
```

11.4 JPanel

JPanel is the most versatile and commonly used intermediate-level Swing container. It is a lightweight container used to group other components together and apply a specific layout to that group. JPanels can be nested within each other to create complex layouts. Unlike top-level containers, JPanel does not have a title bar or borders (though you can add a Border using setBorder()).

```java
// Creating a form panel
JPanel formPanel = new JPanel(new GridLayout(3, 2, 10, 10));
formPanel.setBorder(BorderFactory.createTitledBorder("User Info"));
formPanel.setBackground(new Color(240, 248, 255));

formPanel.add(new JLabel("Name:"));
formPanel.add(new JTextField(20));
formPanel.add(new JLabel("Email:"));
formPanel.add(new JTextField(20));
formPanel.add(new JLabel("Phone:"));
formPanel.add(new JTextField(20));

// Nested panels for complex layout
JPanel buttonPanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
buttonPanel.add(new JButton("Cancel"));
buttonPanel.add(new JButton("Submit"));

frame.add(formPanel, BorderLayout.CENTER);
frame.add(buttonPanel, BorderLayout.SOUTH);
```

12. Button Components

12.1 JButton

JButton is the standard push button in Swing. It generates ActionEvents when clicked. JButton is significantly more feature-rich than AWT's Button — it supports icons (images), HTML formatted text for labels, keyboard mnemonics (underlined shortcut keys), and full customization via painting and look-and-feel.

```
// Basic JButton
JButton saveBtn = new JButton("Save");

// JButton with icon
JButton iconBtn = new JButton("Open File", new ImageIcon("folder.png"));
iconBtn.setHorizontalTextPosition(JButton.RIGHT); // Text to right of icon

// HTML formatted label
JButton htmlBtn = new JButton("<html><b>Bold</b> & <i>Italic</i></html>");

// Keyboard mnemonic (Alt+S triggers the button)
saveBtn.setMnemonic(KeyEvent.VK_S); // Underlines 'S' in label

// Styling
saveBtn.setBackground(new Color(33, 150, 243));
saveBtn.setForeground(Color.WHITE);
saveBtn.setFont(new Font("Arial", Font.BOLD, 14));
saveBtn.setToolTipText("Click to save your file (Alt+S)");
saveBtn.setBorderPainted(false); // Remove default border
saveBtn.setFocusPainted(false);  // Remove focus ring

// Disable a button
saveBtn.setEnabled(false); // Grayed out, not clickable

// Event handling
saveBtn.addActionListener(e -> saveFile());
```

12.2 JToggleButton

JToggleButton is a button that maintains a selected/deselected state — like a light switch. When pressed, it stays pressed (selected) until clicked again (deselected). It fires both ActionEvents and ItemEvents. JCheckBox and JRadioButton are actually subclasses of JToggleButton.

```
JToggleButton toggleBtn = new JToggleButton("OFF");

toggleBtn.addItemListener(e -> {
    if (toggleBtn.isSelected()) {
        toggleBtn.setText("ON");
        toggleBtn.setBackground(Color.GREEN);
    } else {
        toggleBtn.setText("OFF");
        toggleBtn.setBackground(Color.RED);
    }
});

// Programmatically set state
toggleBtn.setSelected(true);  // Force ON state
```

```
boolean isOn = toggleBtn.isSelected(); // Check state
```

12.3 JCheckBox

JCheckBox represents a boolean choice — checked or unchecked. Multiple checkboxes are independent of each other; any number can be checked simultaneously. JCheckBox is typically used when a user needs to make multiple independent selections from a list of options.

```
JCheckBox javaCheck  = new JCheckBox("Java",   true);  // Pre-checked
JCheckBox pythonCheck= new JCheckBox("Python", false);
JCheckBox cppCheck   = new JCheckBox("C++",    false);

// ItemListener to respond to state changes
javaCheck.addItemListener(e -> {
    if (javaCheck.isSelected()) {
        System.out.println("Java selected");
    } else {
        System.out.println("Java deselected");
    }
});

// Read all checkbox states
List<String> selected = new ArrayList<>();
if (javaCheck.isSelected())   selected.add("Java");
if (pythonCheck.isSelected()) selected.add("Python");
if (cppCheck.isSelected())    selected.add("C++");
System.out.println("Selected: " + selected);
```

12.4 JRadioButton & ButtonGroup

JRadioButton represents a choice that is mutually exclusive within a group — only one radio button in a group can be selected at a time, like selecting a single option from multiple choices. To enforce mutual exclusivity, JRadioButton objects must be added to a ButtonGroup. The ButtonGroup manages selection logic; it does not visually group the buttons (for visual grouping, add them to a JPanel).

```
// Create radio buttons
JRadioButton maleRB   = new JRadioButton("Male");
JRadioButton femaleRB = new JRadioButton("Female");
JRadioButton otherRB  = new JRadioButton("Other");
maleRB.setSelected(true); // Default selection

// CRITICAL: Group them for mutual exclusivity
ButtonGroup genderGroup = new ButtonGroup();
genderGroup.add(maleRB);
genderGroup.add(femaleRB);
genderGroup.add(otherRB);

// Add to panel for visual grouping
JPanel genderPanel = new JPanel();
genderPanel.setBorder(BorderFactory.createTitledBorder("Gender"));
genderPanel.add(maleRB);
genderPanel.add(femaleRB);
genderPanel.add(otherRB);

// Find which radio button is selected
maleRB.addActionListener(e -> System.out.println("Male selected"));
```

```
// Get selected value programmatically
String gender = maleRB.isSelected() ? "Male" :
                femaleRB.isSelected() ? "Female" : "Other";
```

13. Text and Display Components

13.1 JLabel

JLabel is a non-interactive display component for showing text, images, or a combination of both. It does not respond to user input and does not generate events (other than mouse events if a listener is added). JLabel supports HTML-formatted text, which enables rich text formatting like bold, italic, colors, and bullet lists within a label. It is widely used for form field labels, status indicators, and displaying images.

```
// Text-only label
JLabel nameLabel = new JLabel("Full Name:");
nameLabel.setFont(new Font("Arial", Font.BOLD, 14));
nameLabel.setForeground(new Color(33, 33, 33));

// Image label
JLabel logoLabel = new JLabel(new ImageIcon("logo.png"));

// Combined: icon + text
JLabel combo = new JLabel("Status: OK", new ImageIcon("green.png"),
                          JLabel.LEFT);
combo.setIconTextGap(8); // Space between icon and text

// HTML-formatted label
JLabel richLabel = new JLabel(
    "<html><h2>Welcome!</h2><p>Select an option below.</p></html>");

// Alignment
JLabel centered = new JLabel("Centered", JLabel.CENTER);
centered.setVerticalAlignment(JLabel.TOP);
```

13.2 JTextField

JTextField is a single-line text input component that allows users to type and edit text. It is the most common text input component in Swing forms. JTextField fires ActionEvents when the user presses Enter, making it easy to handle form submission. getText() retrieves the current text, and setText() sets text programmatically.

```
JTextField nameField = new JTextField(20);  // 20 columns wide
JTextField emailField = new JTextField("placeholder@example.com");

// Retrieve and set text
String name = nameField.getText().trim();
nameField.setText("");          // Clear the field
nameField.requestFocusInWindow(); // Focus programmatically

// Make read-only (display-only)
JTextField readOnly = new JTextField("Read-only text");
readOnly.setEditable(false);
readOnly.setBackground(new Color(245, 245, 245));

// Listen for Enter key
nameField.addActionListener(e -> {
    System.out.println("Submitted: " + nameField.getText());
});
```

```
// Listen for every character typed
nameField.getDocument().addDocumentListener(new DocumentListener() {
    public void insertUpdate(DocumentEvent e)  { validate(); }
    public void removeUpdate(DocumentEvent e)  { validate(); }
    public void changedUpdate(DocumentEvent e) { validate(); }
});
```

13.3 JTextArea

JTextArea is a multi-line text editing component. Unlike JTextField, JTextArea allows multiple lines of input and is suitable for longer text such as descriptions, notes, comments, and log output. JTextArea itself does not include scrollbars — it must be wrapped in a JScrollPane to allow scrolling. Text wrapping is configurable.

```
// Create JTextArea: 8 rows, 40 columns
JTextArea textArea = new JTextArea(8, 40);
textArea.setFont(new Font("Monospaced", Font.PLAIN, 13));

// Enable word wrapping
textArea.setLineWrap(true);     // Wrap to next line
textArea.setWrapStyleWord(true);// Wrap at word boundaries

// Add scrollable container (IMPORTANT)
JScrollPane scrollPane = new JScrollPane(textArea);
scrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);

// Manipulate content
textArea.setText("Initial content");
textArea.append("\nAppended line");
String text = textArea.getText();
textArea.setCaretPosition(textArea.getDocument().getLength()); // Scroll to end

// Make read-only
textArea.setEditable(false);

frame.add(scrollPane); // Add ScrollPane, not textArea directly!
```

14. Selection Components

14.1 JList

JList displays a scrollable list of items from which the user can select one or more items. JList uses a ListModel to hold its data, but it can also be initialized with an array or Vector for convenience. The selection mode determines whether the user can select a single item, a contiguous range of items, or non-contiguous items (multiple individual selections with Ctrl+click).

```
// Initialize with array
String[] fruits = { "Apple", "Banana", "Cherry", "Date", "Elderberry" };
JList<String> fruitList = new JList<>(fruits);
```

```
// Selection modes
fruitList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
// or: SINGLE_INTERVAL_SELECTION, MULTIPLE_INTERVAL_SELECTION

// Visual settings
fruitList.setFixedCellHeight(25); // Uniform row height
fruitList.setVisibleRowCount(5);  // Visible rows without scrolling

// CRITICAL: Wrap in JScrollPane for scrollability
JScrollPane listScroll = new JScrollPane(fruitList);

// Get selections
String selected = fruitList.getSelectedValue();     // Single selection
List<String> multi = fruitList.getSelectedValuesList(); // Multiple

// Listen for selection changes
fruitList.addListSelectionListener(e -> {
    if (!e.getValueIsAdjusting()) { // Ignore intermediate events
        System.out.println("Selected: " + fruitList.getSelectedValue());
    }
});

// Dynamic list with DefaultListModel
DefaultListModel<String> model = new DefaultListModel<>();
model.addElement("New Item");
model.removeElement("Apple");
JList<String> dynamicList = new JList<>(model);
```

14.2 JComboBox

JComboBox presents a compact drop-down selection list. It shows the currently selected item and, when clicked, expands to reveal all available options. It saves screen space compared to JList. JComboBox can be either non-editable (selection-only) or editable (user can type a custom value in addition to choosing from the list). It fires ActionEvents when the selection changes.

```
// Non-editable JComboBox
String[] countries = { "India", "USA", "UK", "Germany", "Japan" };
JComboBox<String> countryBox = new JComboBox<>(countries);
countryBox.setSelectedIndex(0);  // Select first item by default

// Editable ComboBox (user can type custom values)
JComboBox<String> editableBox = new JComboBox<>(countries);
editableBox.setEditable(true);

// Get selected value
String selected = (String) countryBox.getSelectedItem();
int selectedIdx = countryBox.getSelectedIndex();

// Dynamic items
countryBox.addItem("Australia");
countryBox.removeItem("UK");
countryBox.removeAllItems(); // Clear all

// Event handling
countryBox.addActionListener(e -> {
    String choice = (String) countryBox.getSelectedItem();
    System.out.println("Selected: " + choice);
});

// ItemListener (fires twice: once for deselect, once for select)
countryBox.addItemListener(e -> {
    if (e.getStateChange() == ItemEvent.SELECTED) {
        System.out.println("New selection: " + e.getItem());
    }
});
```

14.3 JScrollPane

JScrollPane is a container that automatically provides vertical and/or horizontal scrollbars for any component that may have more content than can be displayed in the available space. It is most commonly used with JTextArea, JList, JTable, and JTree. The scroll policy for each scrollbar can be set to ALWAYS, AS_NEEDED (default), or NEVER.

```
JTextArea ta = new JTextArea(5, 40);
JList<String> list = new JList<>(new String[]{"Item 1","Item 2","..."});

// Wrap any component in JScrollPane
JScrollPane scrollPane = new JScrollPane(ta);

// Configure scroll policies
scrollPane.setVerticalScrollBarPolicy(
    JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);  // Always show
scrollPane.setHorizontalScrollBarPolicy(
    JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED); // Show only when needed

// Scrollable JList
JScrollPane listScroll = new JScrollPane(list,
    JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
    JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);

// Add corner component (e.g., a resize grip)
scrollPane.setCorner(JScrollPane.LOWER_RIGHT_CORNER,
    new JPanel()); // Custom corner

frame.add(scrollPane); // Always add the ScrollPane, not the wrapped component
```

15. Complete Swing Application Example

Student Registration Form

The following complete example demonstrates a realistic Swing application that incorporates JFrame, JPanel with layout managers, JLabel, JTextField, JPasswordField, JTextArea, JCheckBox, JRadioButton with ButtonGroup, JComboBox, JButton, JScrollPane, and event handling with ActionListener and ItemListener.

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

public class StudentRegistration extends JFrame {

    // Form components
    private JTextField nameField, emailField;
    private JPasswordField passField;
    private JComboBox<String> courseBox;
    private JRadioButton maleRB, femaleRB;
    private JCheckBox termsCheck;
    private JTextArea addressArea;
    private JButton submitBtn, resetBtn;
    private JLabel statusLabel;

    public StudentRegistration() {
        setTitle("Student Registration");
        setSize(500, 550);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        buildUI();
        setVisible(true);
    }

    private void buildUI() {
        JPanel mainPanel = new JPanel(new BorderLayout(10, 10));
        mainPanel.setBorder(new EmptyBorder(20, 20, 20, 20));

        // Form panel
        JPanel formPanel = new JPanel(new GridLayout(8, 2, 8, 8));
        formPanel.setBorder(BorderFactory.createTitledBorder(
            BorderFactory.createLineBorder(new Color(33, 150, 243)),
            "Registration Details"));

        // Name
        formPanel.add(new JLabel("Full Name: *"));
        nameField = new JTextField(20);
        formPanel.add(nameField);

        // Email
        formPanel.add(new JLabel("Email: *"));
        emailField = new JTextField(20);
        formPanel.add(emailField);

        // Password
        formPanel.add(new JLabel("Password: *"));
        passField = new JPasswordField(20);
        formPanel.add(passField);

        // Course
        formPanel.add(new JLabel("Course: *"));
        courseBox = new JComboBox<>(new String[]{
            "-- Select --","BCA","MCA","B.Sc CS","B.Tech CS"});
        formPanel.add(courseBox);

        // Gender
        formPanel.add(new JLabel("Gender: *"));
        maleRB = new JRadioButton("Male"); maleRB.setSelected(true);
        femaleRB = new JRadioButton("Female");
        ButtonGroup bg = new ButtonGroup();
        bg.add(maleRB); bg.add(femaleRB);
        JPanel gPanel = new JPanel(new FlowLayout(FlowLayout.LEFT, 0, 0));
        gPanel.add(maleRB); gPanel.add(femaleRB);
        formPanel.add(gPanel);

        // Address
        formPanel.add(new JLabel("Address:"));
        addressArea = new JTextArea(3, 20);
        addressArea.setLineWrap(true);
        addressArea.setWrapStyleWord(true);
        formPanel.add(new JScrollPane(addressArea));

        // Terms
        formPanel.add(new JLabel(""));
        termsCheck = new JCheckBox("I agree to Terms & Conditions");
        formPanel.add(termsCheck);

        // Buttons
        submitBtn = new JButton("Register");
        submitBtn.setBackground(new Color(33, 150, 243));
        submitBtn.setForeground(Color.WHITE);
        submitBtn.setFont(new Font("Arial", Font.BOLD, 14));
        submitBtn.setEnabled(false); // Disabled until terms accepted

        resetBtn = new JButton("Reset");

        // Enable submit only when terms accepted
        termsCheck.addItemListener(e ->
            submitBtn.setEnabled(termsCheck.isSelected()));

        // Submit action
        submitBtn.addActionListener(e -> {
            String name  = nameField.getText().trim();
            String email = emailField.getText().trim();
            String course = (String) courseBox.getSelectedItem();

            if (name.isEmpty() || email.isEmpty()) {
                JOptionPane.showMessageDialog(this,
                    "Name and Email are required!",
                    "Validation Error", JOptionPane.ERROR_MESSAGE);
                return;
            }
            if (course.equals("-- Select --")) {
                JOptionPane.showMessageDialog(this, "Please select a course!");
                return;
            }
            String gender = maleRB.isSelected() ? "Male" : "Female";
            JOptionPane.showMessageDialog(this,
                "Registered!\nName: " + name + "\nCourse: " + course +
                "\nGender: " + gender,
                "Success", JOptionPane.INFORMATION_MESSAGE);
        });

        // Reset action
        resetBtn.addActionListener(e -> {
            nameField.setText(""); emailField.setText("");
            passField.setText(""); addressArea.setText("");
            courseBox.setSelectedIndex(0);
            maleRB.setSelected(true);
            termsCheck.setSelected(false);
        });

        JPanel btnPanel = new JPanel(new FlowLayout(FlowLayout.CENTER, 15, 5));
        btnPanel.add(submitBtn);
        btnPanel.add(resetBtn);

        statusLabel = new JLabel("All fields marked * are mandatory.",
            JLabel.CENTER);
        statusLabel.setForeground(Color.GRAY);

        mainPanel.add(formPanel,   BorderLayout.CENTER);
        mainPanel.add(btnPanel,    BorderLayout.SOUTH);
        mainPanel.add(statusLabel, BorderLayout.NORTH);
        add(mainPanel);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(StudentRegistration::new);
    }
}
```

16. Quick Reference Summary

16.1 AWT Components Summary

Component	Key Methods / Features	Common Use Case
Frame	setTitle, setSize, setLayout, setVisible	Main application window
Color	new Color(r,g,b), predefined constants	Style any component
Font	new Font(name, style, size), FontMetrics	Text styling
FlowLayout	FlowLayout(align, hgap, vgap)	Simple horizontal flow
BorderLayout	add(comp, NORTH/SOUTH/EAST/WEST/CENTER)	Main window layout
GridLayout	new GridLayout(rows, cols, hgap, vgap)	Uniform grids

Component	Key Methods / Features	Common Use Case
ActionListener	actionPerformed(ActionEvent e)	Button, TextField Enter
MouseListener	mouseClicked/Pressed/Released/Entered/Exited	Mouse interaction
KeyListener	keyPressed/Released/Typed	Keyboard input
WindowAdapter	windowClosing, windowOpened, etc.	Close operations

16.2 Swing Components Summary

Component	Key Features	Common Use Case
JFrame	ContentPane, setDefaultCloseOperation	Main app window
JDialog	Modal/non-modal, JOptionPane shortcuts	Pop-up dialogs
JWindow	No title bar / border	Splash screens
JPanel	Nested containers, setBorder	Group & organize components
JButton	Icons, HTML text, mnemonics	User actions
JToggleButton	isSelected(), setSelected()	ON/OFF switches
JCheckBox	isSelected(), ItemListener	Multi-choice selection
JRadioButton	ButtonGroup for mutual exclusion	Single-choice selection
JLabel	HTML support, icons, alignment	Display text/images
JTextField	getText/setText, columns, DocumentListener	Single-line input
JTextArea	lineWrap, wrapStyleWord, append()	Multi-line input
JList	SelectionModel, getSelectedValue()	Scrollable item list
JComboBox	getSelectedItem, addItem, setEditable	Drop-down selection
JScrollPane	VERTICAL/HORIZONTAL_SCROLLBAR policies	Scroll any component

Best Practices for Swing Development

- Always create and update Swing components on the Event Dispatch Thread (EDT) using SwingUtilities.invokeLater().
- Use JOptionPane for simple dialogs instead of creating JDialog from scratch.
- Wrap JTextArea and JList in JScrollPane — never add them directly to a container.
- Use ButtonGroup with JRadioButton to enforce mutual exclusion.

- Prefer lambda expressions over anonymous inner classes for concise, readable event handlers.
- Separate business logic from UI code — avoid putting application logic directly in event listeners.
- Use SwingWorker for long-running background tasks to prevent the UI from freezing.

> Remember: Swing is thread-sensitive. Never update Swing components from background threads. Use SwingUtilities.invokeLater() or SwingWorker for safe UI updates from non-EDT threads.