a3 # gui # java-fundamentals

Swing API Basics

TCSS 305 Programming Practicum

Before you write your first GUI application, it helps to understand where Swing came from, why we use it in this course, and how its class hierarchy fits together. This guide orients you to the landscape—later guides will dive into layout managers, event handling, and custom painting.

---

1 A Brief History of GUI in Java

Java has had three major approaches to building graphical user interfaces. Understanding this history explains some of the design decisions you will encounter in Swing's API.

1.1 AWT: The First Attempt (1996)

The **Abstract Window Toolkit (AWT)** shipped with Java 1.0. AWT used a "heavyweight" approach: every Java component (button, text field, scrollbar) was backed by a corresponding native OS widget. When you created a `Button` in AWT, Java asked the operating system to create a real native button.

This had a significant drawback: **your application looked and behaved differently on every platform.** A button on Windows looked like a Windows button. The same button on macOS looked like a macOS button. Worse, layout and sizing could shift between platforms because native widgets had different dimensions. The promise of "write once, run anywhere" fell apart at the GUI layer.

AWT also had a limited component set. If the native OS didn't provide a widget, AWT didn't either.

1.2 Swing: Pure Java (1998)

Swing arrived with Java 1.2 and took a fundamentally different approach. Instead of delegating to native OS widgets, Swing draws its own components using Java 2D graphics.

This is called a "lightweight" approach—there are no native peers.

The result: **Swing applications look the same on every platform** (unless you explicitly choose a platform-specific look and feel). A `JButton` on Windows renders identically to a `JButton` on Linux because Java itself draws every pixel.

> **Note**
>
> Swing is built on top of AWT, not a replacement for it. Swing uses AWT's event system, window management, and drawing infrastructure. You will see `java.awt` imports in Swing applications—that's normal and expected.

1.3 JavaFX: The Modern Attempt (2014)

JavaFX was introduced as the "next generation" GUI toolkit for Java. It supports CSS styling, FXML layout files, a scene graph rendering model, and richer multimedia capabilities.

However, JavaFX was **removed from the JDK starting with Java 11** (2018). It now lives as an external library called OpenJFX. While JavaFX is a fine choice for production applications, the fact that it requires an external dependency makes it less ideal for a course focused on core Java concepts.

---

2 Why Swing in TCSS 305?

You might wonder: if JavaFX is more modern, why learn Swing? The reasons are pragmatic.

Reason	Explanation
Part of the JDK	Swing ships with every JDK installation—no external libraries, no dependency management
Stable and mature	Swing has been stable since the early 2000s; APIs don't change between JDK versions
Well-documented	Decades of tutorials, textbooks, and Stack Overflow answers
Teaches core concepts	Event-driven programming, component hierarchies, and layout management work the same way in any GUI framework

Reason	Explanation
Sufficient for learning	You don't need CSS or scene graphs to learn how GUIs work

> **Important**
>
> Swing is not what you would choose for a new production desktop application in 2025. That is not the point. The point is learning **event-driven programming**, **component-based design**, and **the Model-View-Controller pattern**—concepts that transfer to any GUI framework, web or desktop.

> **Related Guide**
>
> For the theory behind event-driven programming — the paradigm that drives all GUI development — see the Event-Driven Programming guide.

---

3 Core Class Hierarchy

Swing's class hierarchy follows a clear pattern. Understanding this hierarchy helps you reason about what any component can do.

3.1 The Inheritance Chain

Every visible Swing widget descends from the same chain of classes:

```
java.lang.Object
  └── java.awt.Component          ← Base class for ALL visual elements
        └── java.awt.Container    ← Can hold other Components
              └── javax.swing.JComponent  ← Base for ALL Swing widgets
                    ├── javax.swing.JPanel
                    ├── javax.swing.JButton
                    ├── javax.swing.JLabel
                    ├── javax.swing.JTextField
                    └── ... (dozens more)
```

There is also one important top-level container that sits outside `JComponent`:

```
java.awt.Component
  └── java.awt.Container
```

```
        └── java.awt.Window
              └── java.awt.Frame
                    └── javax.swing.JFrame  ← Top-level application window
```

Notice that `JFrame` does **not** extend `JComponent`. It extends AWT's `Frame` class because it represents a native OS window. This is one of the places where Swing's "built on AWT" heritage shows through.

3.2 What Each Level Provides

Class	What It Provides
`Component`	Size, position, visibility, painting, event handling (`getWidth()`, `setVisible()`, `addMouseListener()`)
`Container`	Holds child components, layout management (`add()`, `setLayout()`)
`JComponent`	Borders, tooltips, double buffering, pluggable look and feel (`setBorder()`, `setToolTipText()`, `paintComponent()`)

---

4 The Big Three: JFrame, JPanel, and Components

Most Swing applications are built from three kinds of objects.

4.1 JFrame: The Top-Level Window

A `JFrame` represents the application window—the thing with a title bar, minimize/maximize buttons, and a close button. You typically create one `JFrame` per application.

```
import javax.swing.JFrame;

public final class MyApp {

    private MyApp() {
        // Utility class — prevent instantiation
    }

    public static void main(String[] args) {
        final JFrame frame = new JFrame("My Application");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(800, 600);
```

```
        frame.setLocationRelativeTo(null);  // Center on screen
        frame.setVisible(true);
    }
}
```

> **Always Set the Close Operation**
>
> If you forget `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)`, closing the window hides it but does **not** terminate the program. Your application will continue running invisibly in the background. This is a common source of confusion.

4.2 JPanel: The Organizer

A `JPanel` is a container that groups components together. Think of it as a blank rectangular area that you fill with buttons, labels, text fields, or even other panels.

```
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;

final JPanel panel = new JPanel();
panel.add(new JLabel("Name:"));
panel.add(new JTextField(20));
panel.add(new JButton("Submit"));
```

4.3 Components: The Building Blocks

Swing provides dozens of ready-made components. Here are the ones you will encounter most often.

Component	Purpose	Example Usage
`JButton`	Clickable button	Trigger actions
`JLabel`	Display text or images	Show status messages
`JTextField`	Single-line text input	Enter a file name
`JTextArea`	Multi-line text input	Display log output
`JCheckBox`	Toggle option	Enable/disable a feature

Component	Purpose	Example Usage
`JComboBox`	Drop-down selection	Choose a color
`JSlider`	Slide to select a value	Adjust line thickness
`JColorChooser`	Color picker dialog	Select a drawing color
`JOptionPane`	Simple dialog boxes	Show a message or ask yes/no
`JMenuBar` / `JMenu` / `JMenuItem`	Menu system	File > Save, Edit > Undo

---

5 Containers as Components: The Composition Pattern

Here is one of Swing's most powerful ideas: **a `JPanel` is both a container and a component.** It can hold other components, and it can itself be added to another container.

This means you can nest panels inside panels to build complex layouts:

```
// A toolbar panel with buttons
final JPanel toolbar = new JPanel();
toolbar.add(new JButton("Draw"));
toolbar.add(new JButton("Erase"));
toolbar.add(new JButton("Clear"));

// A status panel at the bottom
final JPanel statusBar = new JPanel();
statusBar.add(new JLabel("Ready"));

// A drawing panel in the center (custom JPanel subclass)
final DrawingPanel canvas = new DrawingPanel();

// Assemble everything into the frame
frame.add(toolbar, BorderLayout.NORTH);
frame.add(canvas, BorderLayout.CENTER);
frame.add(statusBar, BorderLayout.SOUTH);
```

This is the **Composite pattern** in action—the same pattern you practiced with composition in Assignment 2. A container treats a single component and a group of components uniformly. You build complex UIs by composing simple pieces.

> **Tip**
>
> When your layout gets complicated, resist the temptation to put everything in one panel. Instead, break the UI into logical sections—each in its own `JPanel`—and then combine those panels. This makes the layout easier to reason about and easier to change later.

---

6 General Swing Patterns

Swing programming follows a consistent rhythm. Once you recognize the pattern, building GUIs becomes methodical rather than mysterious.

6.1 Create, Configure, Add

The basic workflow for any component is:

1. **Create** the component
2. **Configure** it (set text, size, color, behavior)
3. **Add** it to a container

```
// 1. Create
final JButton button = new JButton("Click Me");

// 2. Configure
button.setToolTipText("Performs an action");
button.setEnabled(true);

// 3. Add
panel.add(button);
```

6.2 Layout Managers Control Positioning

You do not position components with pixel coordinates in Swing. Instead, you assign a **layout manager** to each container, and the layout manager decides where components go.

```
panel.setLayout(new BorderLayout());   // North/South/East/West/Center
panel.setLayout(new FlowLayout());     // Left to right, wrapping
panel.setLayout(new GridLayout(3, 2)); // 3 rows, 2 columns
```

> **Related Guide**
>
> Layout managers are a substantial topic on their own. See the Swing Layout Managers guide for detailed coverage of `BorderLayout`, `FlowLayout`, `GridLayout`, and `BoxLayout`.

6.3 Event Listeners Respond to User Interaction

Components generate **events** when the user interacts with them. You register **listeners** to respond to those events:

```
button.addActionListener(e -> System.out.println("Button clicked!"));
```

> **Related Guides**
>
> - For wiring up button events with `ActionListener`, see the Adding Event Handlers guide.
> - For `MouseListener`, `MouseMotionListener`, and `MouseAdapter`, see the Handling Mouse Events guide.
> - For lambda expression syntax used in the example above, see the Introduction to Lambda Expressions guide.

6.4 The Event Dispatch Thread (EDT)

All Swing code must run on a single thread called the **Event Dispatch Thread (EDT)**. This is a critical rule: if you create or modify Swing components from the wrong thread, you will encounter intermittent, hard-to-diagnose bugs.

The standard way to ensure your startup code runs on the EDT:

```
public static void main(String[] args) {
    EventQueue.invokeLater(MyApp::createAndShowGui);
}

private static void createAndShowGui() {
    final JFrame frame = new JFrame("My Application");
    // ... set up frame and components ...
    frame.setVisible(true);
}
```

`EventQueue.invokeLater()` schedules the `Runnable` to execute on the EDT. You will see this pattern in every well-written Swing application.

> **Note**
>
> Thread safety in Swing is a topic we will revisit later. For now, just follow the pattern: wrap your GUI startup code in `EventQueue.invokeLater()` and you will be fine.

6.5 The Standard Startup Sequence

Most Swing applications follow this startup sequence:

```
import java.awt.EventQueue;
import javax.swing.JFrame;

public final class MyApp {

    private MyApp() {
        // Utility class – prevent instantiation
    }

    public static void main(String[] args) {
        EventQueue.invokeLater(MyApp::createAndShowGui);
    }

    private static void createAndShowGui() {
        final JFrame frame = new JFrame("My Application");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Add panels and components to frame here

        frame.pack();                       // Size frame to fit contents
        frame.setLocationRelativeTo(null);  // Center on screen
        frame.setVisible(true);             // Show the window
    }
}
```

Two methods deserve special attention:

- **`pack()`** — Asks the frame to size itself based on the preferred sizes of its contents. This is almost always better than calling `setSize()` with hard-coded dimensions, because it adapts to the actual components.
- **`setVisible(true)`** — Makes the window appear. Always call this **last**, after all components have been added. If you call it too early, the window may appear briefly empty before components are painted.

Gen AI & Learning: GUI Code and AI Assistants

GUI code tends to be highly formulaic—creating components, setting properties, adding them to containers. AI coding assistants are quite good at generating this boilerplate. However, understanding the *structure* (why `JPanel` goes inside `JFrame`, why `pack()` comes before `setVisible()`, why everything runs on the EDT) is what allows you to debug and extend the generated code. An AI can scaffold a window for you in seconds, but only your understanding of the component hierarchy lets you fix it when the layout doesn't look right.

Summary

Concept	Key Point
AWT	Java's original GUI toolkit (1996); used native OS widgets; looked different on each platform
Swing	Pure Java GUI toolkit (1998); draws its own components; consistent cross-platform appearance
JavaFX	Modern toolkit (2014); removed from JDK in Java 11; now an external library
Why Swing	Ships with the JDK, stable, well-documented, sufficient for learning GUI concepts
JFrame	Top-level application window; one per application
JPanel	Container for organizing components; also a component itself
JComponent	Base class for all Swing widgets; provides borders, tooltips, painting
Composite pattern	Panels inside panels—build complex UIs from simple pieces
Create, Configure, Add	The rhythm of Swing programming
Layout managers	Control component positioning; no manual pixel placement

Concept	Key Point
Event listeners	Respond to user interaction (clicks, key presses, mouse movement)
EDT	All Swing code runs on the Event Dispatch Thread; use `EventQueue.invokeLater()`
`pack()` + `setVisible(true)`	Standard startup sequence: size to contents, then show

Further Reading

External Resources

- Oracle Swing Tutorial: Getting Started - Official tutorial for building your first Swing application
- Oracle Swing Tutorial: Using Top-Level Containers - How JFrame, JDialog, and content panes work
- Oracle Swing Tutorial: Using Swing Components - Reference for all standard Swing components
- Baeldung: Introduction to Java Swing - Concise overview with practical examples

References

Primary Texts:

- Horstmann, C. S. (2022). *Core Java, Volume I: Fundamentals* (12th ed.). Oracle Press. Chapter 10: Graphical User Interface Programming — Frames, Components, and Layout Management. Chapter 11: User Interface Components with Swing — Event Handling.
- Reges, S., & Stepp, M. (2020). *Building Java Programs* (5th ed.). Pearson. Chapter 14: Graphical User Interfaces — Introduction to Swing.
- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley. Item 73: Throw exceptions appropriate to the abstraction (relevant to event-driven error handling).

Language Documentation:

- Oracle JDK 25: javax.swing Package — Official API reference for Swing

- Oracle JDK 25: JFrame — JFrame class documentation
- Oracle JDK 25: JPanel — JPanel class documentation
- Oracle JDK 25: JComponent — JComponent class documentation

Tutorials:

- Oracle Swing Tutorial — Comprehensive official guide to Swing programming
- Oracle Trail: Creating a GUI with Swing — Step-by-step learning path

---

This guide is part of TCSS 305 Programming Practicum, School of Engineering and Technology, University of Washington Tacoma.