lectures # week-7

Week 7: Sprint Zero & Layout Managers (Wednesday, February 18, 2026)

Lecture Recording

Watch on Panopto

Related Assignment

This lecture covers Sprint Zero setup for the Group Project. Layout manager concepts are covered in the Swing Layout Managers guide.

Demo Code

Clone the lecture examples: TCSS305-Into-GUI

Lecture Preview 0:00

Today's agenda:

1. **Test 2 preview** — What's on Friday's test, question format
2. **Sprint Zero** — Group project setup, GitHub Classroom, starter code, model API
3. **Layout managers** — FlowLayout, BorderLayout, GridLayout
4. **Composite layouts** — Combining layout managers for complex UIs

---

Admin Notes — Test 2 Preview 1:14

Test 2 is Friday. Same format as Test 1: single page of handwritten notes allowed (double-sided).

What's On the Test 3:46

From Assignment 2 guides:

- Inheritance and polymorphism
- Comparator and Comparable
- Unit testing (still testable — you won't write test cases, but you'll read code and determine if test cases pass or fail)

From Assignment 3 guides:

- Event-driven programming
- Adding event handlers
- Introduction to lambda expressions
- Handling mouse events

NOT on this test:

- Swing API Basics
- Swing Layout Managers (today's new material)

Question Format 7:14

The unit testing questions have changed from Test 1:

- **Test 1 format:** Here's code with a bug — find and fix the bug
- **Test 2 format:** Here's code and test cases — tell me which test cases **pass** and which **fail**

> **Read Carefully**
>
> There will be 4-6 code/test-case questions. For at least one, the answer is that **all tests pass** (the code is correct). Don't assume every question has a bug. Hand-trace the code carefully.

Fill-in-the-blank questions will test terminology. Know your vocabulary — being precise with terms like "instance variable" matters more than ever.

> **Study Strategy**
>
> Get with your group and have each member write practice questions, then trade and study. The practice problems on Discord show the style of questions to expect.

Sprint Zero 11:20

Groups should be forming now. If you're in a group of five, expect that one member may not respond. If you're in a group of four and only two have responded, notify the instructor.

Repository Setup 12:29

Use GitHub Classroom like previous assignments, but this time it's a **group repository**:

1. One group member creates the team in GitHub Classroom
2. Other members click the same link and join the existing team
3. Don't join the wrong team

Starter Code 14:17

The repository comes with three packages: **model**, **view**, and a sandbox class.

Package	Purpose
`model`	Keep empty for Sprints 0-2. The Tetris game model is provided as a JAR file.
`view`	All your GUI code goes here
Sandbox	Runnable class that creates a game model object and demonstrates the JAR works

The model source code is viewable but not editable. The sandbox class shows how to instantiate and interact with the model.

Working with the Model API 16:15

The backend API documentation is published on GitHub Pages with full Javadoc.

Key classes and interfaces:

Class/Interface	Role
`GameControls`	**Interface** that defines the API — this is your contract for interacting with the model
`TetrisGame`	**Concrete class** that implements `GameControls`

For Sprints 0-2, you primarily interact with `GameControls`. Listening to model events (via the Observer pattern) will be introduced at the end of next week.

Sprint Planning 20:46

Sprint 1 requirements will be posted Friday afternoon (during Test 2 — no in-class introduction).

Suggested timeline:

- Wrap up Sprint Zero by Sunday
- Sunday: quick 30-minute group meeting to review Sprint 1 requirements and plan who does what

> **Monday Group Meetings**
>
> Monday's class time will be used for 10-minute group meetings with the instructor (8 groups, 10 minutes each). Sign-up sheet will be posted on Discord. Three lecture periods will be replaced by group meetings throughout the project — use the extra time for self-study and reading.

Layout Managers 23:04

Two approaches for placing components on a GUI:

Absolute Layout 25:13

Place components at specific pixel coordinates (e.g., button A at pixel 10,10; button B at pixel 50,10).

Problems with absolute layout:

- **Cross-platform issues** — Button sizes differ between macOS and Windows, causing overlaps
- **Resolution differences** — Different screen resolutions and scaling break fixed positions
- **No resize support** — When users resize the window, components stay at their original positions and look out of place

Layout Managers 27:45

Layout managers are **objects** that define **rules** for how a panel arranges its components. Components show up where the layout manager's rules say they should.

Strengths: Adapts to resizing, different operating systems, different screen resolutions.

Trade-off: You must understand each layout manager's rules. If you don't know the rules, you'll never get the GUI to look the way you want.

We'll cover three common layout managers: **FlowLayout**, **BorderLayout**, and **GridLayout**.

FlowLayout 30:42

FlowLayout is the **default layout manager for JPanel**.

Rules 31:37

Rule	Description
Order	Components appear in the order they are added
Direction	Laid out in horizontal rows, left to right
Wrapping	When a row runs out of horizontal space, components flow to the next row
Sizing	Respects each component's preferred size — does NOT resize components
Alignment	Centered by default; can be changed to leading or trailing
Capacity	Unlimited — add as many components as you want

Alignment: Leading vs. Trailing

Instead of "left" and "right," use `FlowLayout.LEADING` and `FlowLayout.TRAILING`. These respect the internationalization of the OS — in a right-to-left language, leading means right-aligned.

Customizing FlowLayout 58:01

FlowLayout objects are **immutable** — you can't change the alignment of an existing one. To customize, create a new FlowLayout and set it on the panel:

```
setLayout(new FlowLayout(FlowLayout.LEADING));
```

The constructor also accepts horizontal and vertical gap parameters to control spacing between components.

JFrame Boilerplate 39:46

Class Design — Extend JPanel 39:55

Three approaches for structuring GUI classes:

Approach	Description
Composition	Class *has* a JFrame/JPanel (no inheritance)
Extend JFrame	Class *is* a window
Extend JPanel	Class *is* a panel (recommended)

Why extend JPanel? Panels can be nested inside other panels for reusable, composite layouts. JFrames can't be nested inside JFrames. If you create three demo classes as JPanels, you can combine them into one larger layout. If they're JFrames, you can't.

Main Method & Event Dispatch Thread 46:49

Start GUI applications on the **Event Dispatch Thread (EDT)**:

```
public static void main(final String[] theArgs) {
    SwingUtilities.invokeLater(FlowLayoutDemo::createAndShowGUI);
}
```

This is **boilerplate** — use it in every Swing application.

Create and Show GUI Pattern 48:23

The `createAndShowGUI` method follows a standard pattern:

```
private static void createAndShowGUI() {
    final FlowLayoutDemo mainPanel = new FlowLayoutDemo();
    final JFrame window = new JFrame("Flow Layout Demo");
    window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    window.setContentPane(mainPanel);
    window.pack();
    window.setVisible(true);
}
```

Key points:

Line	Purpose
`setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)`	Without this, closing the window doesn't terminate the program. Always include it.
`setContentPane(mainPanel)`	A JFrame holds **one** content pane (a JPanel). This replaces the default with yours.
`pack()`	Sizes the window to fit its components. Call this **last** before `setVisible`.
`setVisible(true)`	The window won't appear until you explicitly show it.

Missing EXIT_ON_CLOSE

Without `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)`, your application keeps running after you close the window. You'll end up with 30 invisible instances running in the background.

JFrame.add() Shortcut

JFrame has an `add()` method, but you're actually adding to the JFrame's internal JPanel, not to the JFrame itself. Subtle but important distinction.

Static Initializer Block 42:42

A `static` block (no method name) runs code to initialize `static final` variables whose values can't be determined at compile time:

```
private static final String[] BUTTON_LABELS;
static {
    BUTTON_LABELS = new String[NUM_BUTTONS];
    for (int i = 0; i < NUM_BUTTONS; i++) {
        BUTTON_LABELS[i] = "Button " + i;
    }
}
```

This executes at runtime before any objects are instantiated — technically it may **lazy load**, executing only when the class is first referenced.

BorderLayout 1:00:27

BorderLayout manages exactly **five regions**: North, South, East, West, and Center. You specify which region when adding a component.

Region Rules 1:03:46

Region	Horizontal	Vertical
North	Stretches to full width of panel	Respects component's preferred height
South	Stretches to full width of panel	Respects component's preferred height
East	Respects component's preferred width	Stretches between North and South (or panel edge if absent)

Region	Horizontal	Vertical
West	Respects component's preferred width	Stretches between North and South (or panel edge if absent)
Center	Stretches to fill all remaining horizontal space	Stretches to fill all remaining vertical space

Key differences from FlowLayout:

- **Does NOT respect component sizes** (except as noted per region above)
- **Maximum 5 components** (FlowLayout has no limit)
- **Must specify region** when adding: `add(component, BorderLayout.NORTH)`
- Adding a second component to the same region **replaces** the first

> ⚠ **No Default Region**
>
> If you add a component without specifying a region, it defaults to Center — but always be explicit.

Nesting Panels for Control 1:05:03

If you add a button directly to the North, it stretches across the entire width. To keep the button at its natural size, **nest it in a JPanel** (which uses FlowLayout by default):

```
JPanel northPanel = new JPanel(); // FlowLayout by default
northPanel.add(new JButton("North"));
add(northPanel, BorderLayout.NORTH);
```

The JPanel stretches to fill the North region, but the button inside it stays at its preferred size thanks to FlowLayout.

> 🔥 **Most Useful Layout Manager**
>
> BorderLayout with its five regions is arguably the **most powerful** layout manager. Most application windows use a BorderLayout for the overall structure, with other layout managers nested in each region.

GridLayout 1:14:57

GridLayout arranges components in a grid of **rows and columns**.

Rule	Description
Order	Components fill left-to-right across each row, then down to the next row
Sizing	ALL components are resized to the **same size** — the max width and max height of any component
Overflow	Adding more components than rows × columns creates new rows dynamically
Does NOT respect component preferred sizes	

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

Composite Layouts 1:17:06

Real GUIs combine multiple layout managers by nesting panels:

- **Outer panel:** BorderLayout for overall structure
- **Center region:** GridLayout for a grid of components
- **South region:** FlowLayout for buttons (Send, Cancel)
- **North region:** Component added directly (text field stretches to fill)

The key insight: **a JPanel is both a container and a component**. A JPanel can hold components and be added to another JPanel's layout. This lets you build complex, responsive layouts from simple building blocks.

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