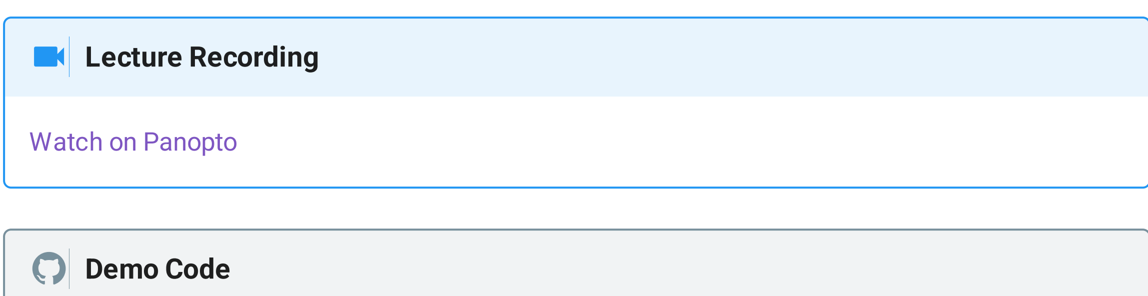
Week 8: Sprint 1 & Introduction to 2D Graphics (Wednesday, February 25, 2026)



Demo Code

Clone the lecture examples: TCSS305-Into-GUI

Related Assignment

This lecture covers concepts for Group Project Sprint 1.

Lecture Preview 5:25

Today's agenda:

1. **Rest of the quarter** — Where we are and where we're headed
2. **Sprint 1 requirements** — Q&A with client hat on
3. **Menu systems** — Pointer to the JMenu guide (self-study)
4. **2D graphics** — Coordinate systems, shapes, and rendering with `Graphics2D`

Admin Notes 0:07

Registration: Spring registration is open. TCSS 342 (Data Structures) has one large section. TCSS 371 has one section with ~35 seats.

TCSS 390 Workshop: The 342/343 workshops have been combined to ensure enrollment hits the minimum of 10. Meets one day a week (Wednesdays, 5:50–7:50 PM). Workshop students will need to meet with their team outside of class for an additional hour or two per week. Data structures workshop will involve more coding activities and data structure analysis — good practice for interview-style questions.

Test 2: Results look a little better than Test 1 on average — likely no curve needed. Tests will be returned Friday or Monday.

Assignment 2 & 3 Grades: Complete blind spot — these have not been graded yet. At least one set of grades will be out this weekend, ideally both.

Finals Week: No comprehensive final exam. The course grade is based on the three tests given during the quarter. Sprint 3 is due Sunday night of Week 10 (before finals week starts). During finals week, a group project write-up/reflection will be due by the Wednesday final time slot.

Rest of the Quarter 6:35

Week	Monday	Wednesday	Friday
8 (this week)	Group meetings	2D drawing (today)	Animation & model-view intro
9	Group meetings	Observer pattern	Observer pattern continued
10	Group meetings	Wrap-up (Java 8 functional if time)	**Test 3**

- Every Monday for the rest of the quarter: **group meetings with demos**. Have a laptop ready with your Tetris running — no fumbling with branches.
- Only ~4 real content lectures remain. Reading the guides on your own is more important than ever.
- Sprint 3 due **Sunday night, Week 10** (before finals week).

Sprint 1 Requirements 🕓 13:51

Sprint 1 has two related guides:

1. **Building Menus with JMenu** — Self-study. Not covered in lecture. You are expected to read and learn this on your own.
2. **Custom Painting with Java 2D** — Covered today in lecture. The guide reinforces what we discuss.

> **Learning on Your Own**
>
> You're late enough in this course that you should be getting comfortable learning material independently. That's one of the most important skills you can develop — figuring stuff out on your own is exactly what you'll do on the job.

Checkstyle in Sprints 🕓 15:47

- **Sprint 1 and Sprint 2** are considered **development code** — Checkstyle and IntelliJ warnings do not need to be fully cleaned up.
- **Sprint 3** is considered **production code** — all linting warnings must be resolved before submission.

The Star Import Trap 🕓 17:02

GUI classes tend to have many imports (`JPanel` , `JFrame` , `JButton` , `JTextField` , layout managers, etc.). You might be tempted to use star imports (`import javax.swing.*`), but Checkstyle prohibits them.

Here's the trap: if you have too many individual imports, Checkstyle warns about that too. You might think "I'll just use a star import to fix it" — but then Checkstyle rejects the star import. It's circular.

The real fix: Too many imports means your class has too much responsibility. Decompose it.

- Separate class for the menu bar
- Separate class for each major panel region
- A coordinator class that assembles the panels

Don't Use Star Imports

Import each class individually. It shows intent — "I need *these specific classes* and no others." If you hit the too-many-imports warning, that's a signal to break up your class, not to reach for `*`.

Layout and Design Decisions 🕓 20:55

The client's specific requirements for Sprint 1 are minimal:

Region	Requirements
Board	Background must be red. Tetrominos made of equal-sided squares. Board must fit all pieces (20x10). Pieces can't hang off edges.
Next Piece	Background must be blue. Tetromino must be centered. Made of equal-sided units.
Info Region	Background must be green. Can be empty for this sprint.
Overall	No specific placement requirements. No color requirements for tetrominos. No outline requirements.

Sprint Colors Are Temporary

The red/blue/green backgrounds are only for Sprint 1 so the client can see the regions. Don't use these colors in your final product.

You are the artistic developer. As a group, decide how to lay out the window. `BorderLayout` will likely be your primary layout manager, but you'll probably need nested panels too. This can change between sprints — that's the beauty of iterative development.

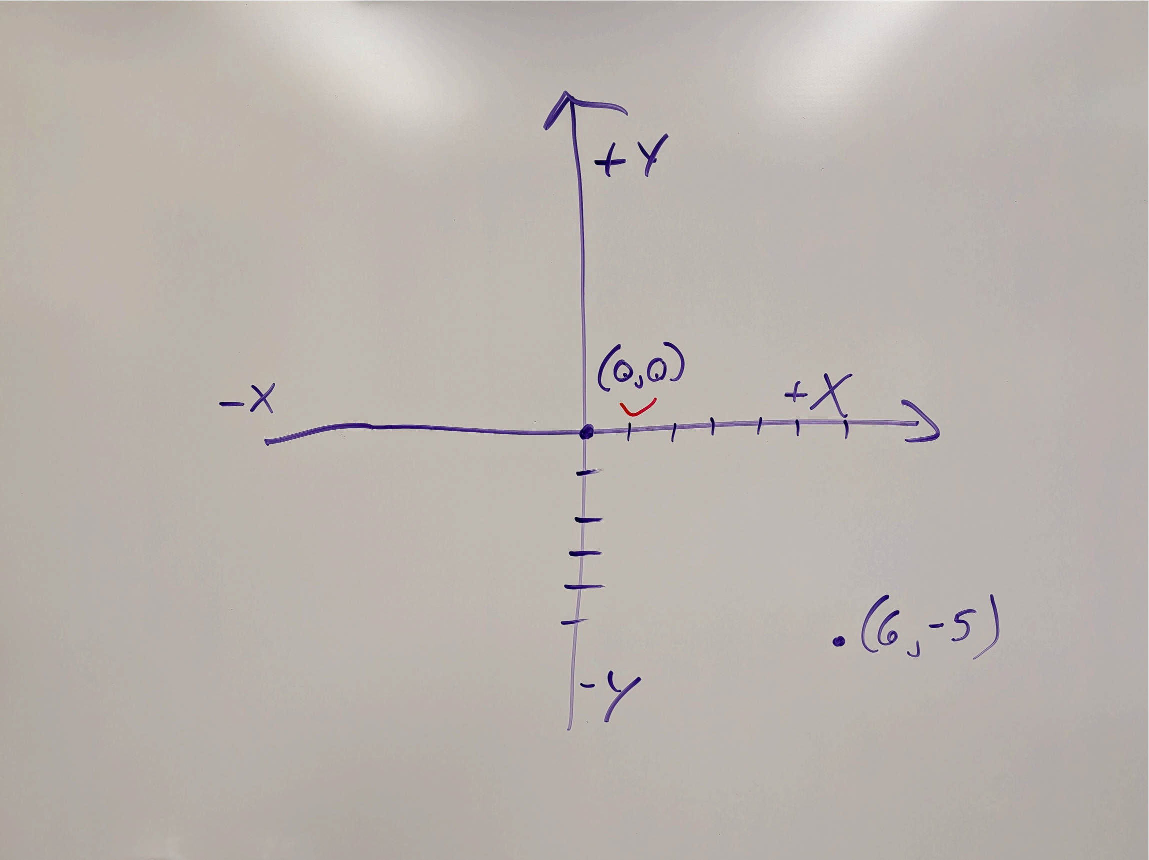
Student Question

Q: Should we try to nail down all requirements with the client early, or iterate?

A: There's no single answer — it depends on the system and the client. But a good rule of thumb: **don't do work you're not getting paid for.** If there's no specific requirement, don't build it. The scope of work defines what you deliver. Sprints let you iterate and get client feedback along the way.

Java 2D Graphics Coordinate System 🕓 28:37

Math Coordinate System 🕓 32:08



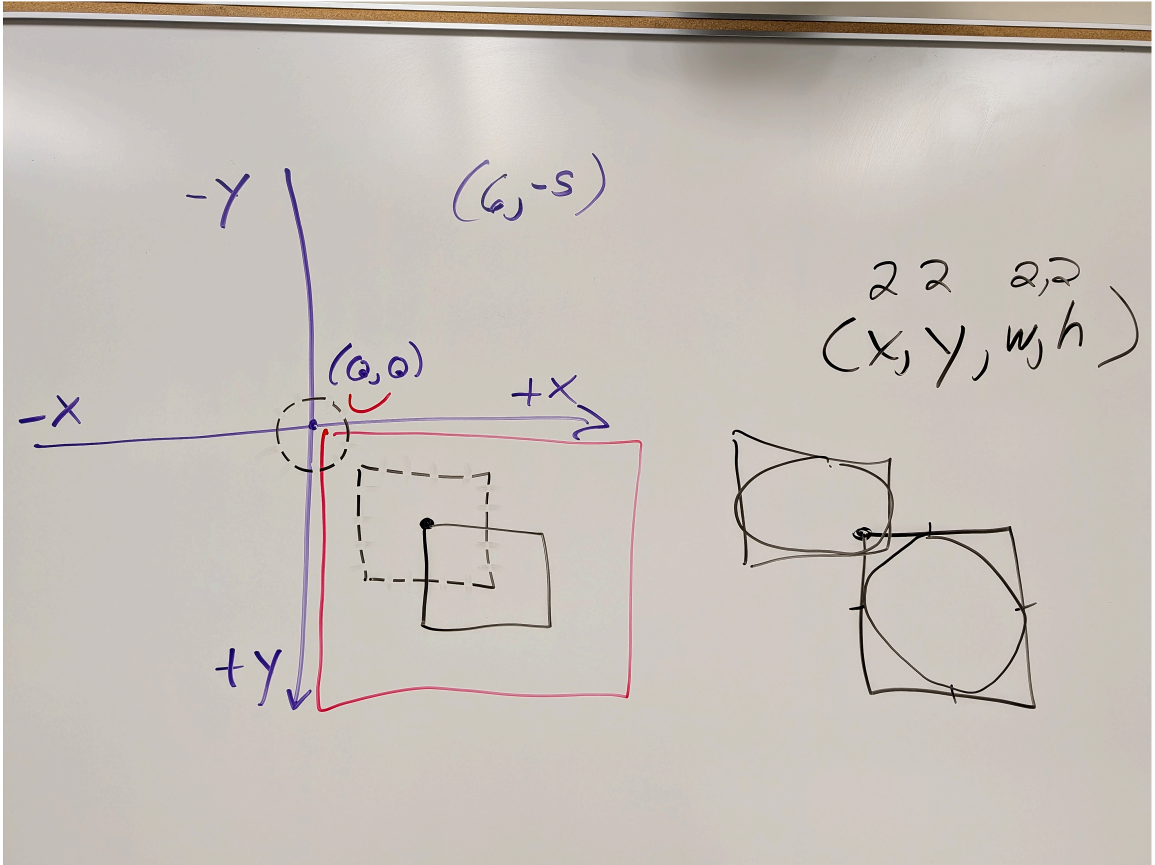
In the Cartesian plane from math class:

- **+X** goes right, **-X** goes left
- **+Y** goes up, **-Y** goes down
- Origin **(0, 0)** is at the center

The point **(6, -5)** would be in the fourth quadrant (bottom-right).

And if you draw (0, 0) on a whiteboard, you are *required* to add the little smiley face — the zeros are the eyes, the comma is the nose. Notice which direction the origin is looking... that's foreshadowing.

Computer Graphics Coordinate System 🕓 35:17



In most computer graphics systems (including Java 2D):

- **+X** goes right, **-X** goes left (same as math)
- **+Y** goes **down**, **-Y** goes **up** (flipped from math!)
- Origin **(0, 0)** is at the **top-left corner** of the panel

The visible region of the panel is the bottom-right quadrant of a full Cartesian plane. This means:

- Shapes drawn at (0, 0) appear in the top-left corner
- The point (6, -5) would be off-screen above the window

> **Drawing Off-Screen**
>
> You *can* draw shapes at negative coordinates or beyond the panel bounds. This is useful for animation — instead of setting a shape to `null` and adding null checks, just position it off-screen until you need it visible.

Shapes and Bounding Boxes 39:36

Every rectangle and ellipse in Java 2D is defined by four values: **(x, y, width, height)**.

- **(x, y)** is the **top-left corner of the bounding box** (not the center of the shape)
- **(width, height)** define the size of the bounding box

How Ellipses Work 43:58

An ellipse is inscribed inside its bounding box rectangle. The ellipse touches the midpoint of each side of the bounding box. This means:

- A circle is just an ellipse where width = height
- The (x, y) point that defines the ellipse is **outside the ellipse itself** — it's at the top-left corner of the bounding box

Collision Detection with Circles

Because a circle's position is defined by its bounding box corner (not its center), collision detection between circles requires extra math. Two bounding boxes can overlap while the circles inside them don't actually touch.

The `paintComponent` Method 47:00

Every Swing component has a 2D graphics rendering system built into it. The Event Dispatch Thread (EDT) calls a method on each component to tell it to render itself.

To do custom drawing, extend `JPanel` and override `paintComponent`:

```
@Override
protected void paintComponent(final Graphics theGraphics) {
    super.paintComponent(theGraphics);
    final Graphics2D g2d = (Graphics2D) theGraphics;

    // Turn on anti-aliasing
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                         RenderingHints.VALUE_ANTIALIAS_ON);

    // Draw shapes here...
}
```

There are several critical gotchas in this boilerplate:

`paintComponent` vs `paintComponents` 🕓 49:05

Override **`paintComponent`** (singular). There is also a method called `paintComponents` (plural) — a holdover from the older AWT painting system. If you accidentally override `paintComponents`, your shapes won't render and you'll have a debugging nightmare. The `@Override` annotation won't save you here since both are valid overrides.

Solution: Don't memorize it. Keep a working example and copy the boilerplate every time.

Always Call `super.paintComponent` 🕓 51:34

If you skip `super.paintComponent(theGraphics)`, your panel will *probably* look *mostly* okay — but you'll get ghosting artifacts. Common symptom: opening a `JMenu` and closing it leaves the menu visually stuck on screen. `super.paintComponent` clears the background for a fresh canvas.

Cast to `Graphics2D` 🕓 53:06

The parameter type is `Graphics` (the older class), but in Swing, the actual object passed in is **always** a `Graphics2D` instance. This cast is guaranteed safe — no `instanceof` check needed. You want the `Graphics2D` reference to access the newer, richer API.

Anti-Aliasing 🕓 55:25

Without anti-aliasing, circles look like staircases. With it, edges are smooth. The performance cost is negligible for 2D rendering — always turn it on.

> **Boilerplate**
>
> Lines 74–80 of any `paintComponent` override are always the same: call `super`, cast to `Graphics2D`, enable anti-aliasing. Treat this as copy-paste boilerplate for every panel that does custom painting.

Drawing and Filling Shapes 🕓 58:32

Creating Shapes 🕓 59:26

`Shape` is a Java interface (not Swing-specific). Implementations include `Rectangle2D`, `Ellipse2D`, `Line2D`, `Path2D`, and more.

```
// Create a rectangle (x, y, width, height)
final Shape rectangle = new Rectangle2D.Double(x, y, width, height);

// Create an ellipse (same parameters — defines the bounding box)
final Shape ellipse = new Ellipse2D.Double(x, y, width, height);
```

Creating a shape does **not** render anything. It just defines the geometry.

Setting Paint and Stroke 🕓 1:03:07

Color: Use `setPaint()` instead of `setColor()`. Both work for solid colors, but `setPaint()` also supports gradients and patterns — use it by default so you don't have to change anything if you add gradients later.

```
g2d.setPaint(Color.RED);
```

Stroke: Defines the line appearance — thickness, solid vs. dashed, etc.

```
g2d.setStroke(new BasicStroke(9)); // 9-pixel wide line
```

> ⚠️ **Stroke Extends Beyond the Bounding Box**
>
> A stroke of 1 pixel draws exactly on the bounding box. A stroke of 9 pixels distributes 4 pixels inside and 4 pixels outside the bounding box (plus 1 on the line). For Tetris, a thick outline on tetrominos may overlap neighboring pieces or extend off-screen. Adjust your math accordingly.

`draw()` vs `fill()` 🕓 1:08:08

```
g2d.fill(ellipse);   // Fill the entire shape with the current paint
g2d.draw(rectangle);  // Draw just the outline using the current stroke and
paint
```

- **`fill()`** fills the interior of the shape
- **`draw()`** draws only the outline using the current stroke

You can do both on the same shape — fill it with one color, then draw the outline with another.

Painter's Rule (Draw Order Matters) 🕓 1:11:07

Think of it like painting on a real canvas. Each `draw()` or `fill()` call paints on top of whatever is already there. **The last shape drawn is on top.** If shapes don't overlap, order doesn't matter. If they do, the draw order determines which shape appears in front.

> **Student Question**
>
> Q: For `fill()`, do you need to set a stroke?
>
> A: No. `fill()` fills the entire shape interior — it doesn't use the stroke at all. Only `draw()` uses the stroke for the outline.

When Do Shapes Render? 🕓 1:12:48

The EDT calls `paintComponent` when a component needs to be rendered:

- When the component first appears on screen
- When the window is resized
- When the system requests a repaint

You should never call `paintComponent` directly. You don't have the right `Graphics` object. Instead, ask the EDT to repaint the component at its earliest convenience (we'll see how on Friday).

Resize = Rerender = Animation 🕓 1:14:14

If your shape positions are calculated relative to the panel size (e.g., centering a rectangle by using `getWidth()` and `getHeight()`), resizing the window causes the shapes to move. This is your first taste of animation:

1. **Render** the image
2. **Change** something (panel size changed)
3. **Re-render** the image
4. **Repeat**

That's all animation is — from Steamboat Willie to modern CGI. Render, change, re-render, loop.

Looking Ahead 1:17:52

On Friday:

- **Button-triggered animation** — making shapes move by clicking
- `javax.swing.Timer` — automatic ticking at a set interval to drive animation
- **Model-view separation** — keeping game logic separate from the GUI

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