a3 # group-project # gui

Custom Painting with Java 2D

TCSS 305 Programming Practicum

Most Swing applications are built from standard components — buttons, labels, text fields — laid out by layout managers. But what happens when you need to render something that no standard component provides? A game board, a data visualization, a drawing canvas? That is where **custom painting** comes in: you override a single method and take control of every pixel.

This guide picks up where the Swing API Basics guide left off. There, we promised a later guide on custom painting. Here it is.

---

1 The paintComponent Method

1.1 Why Override paintComponent

Custom painting in Swing means creating a subclass of `JPanel` and overriding its `paintComponent(Graphics)` method. Every time Swing needs to draw your panel — when the window first appears, when it is resized, when something calls `repaint()` — Swing calls this method and hands you a `Graphics` object. You use that object to draw whatever you want.

```
import java.awt.Graphics;
import javax.swing.JPanel;

public class GamePanel extends JPanel {

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);

        // Your custom drawing code goes here
        g.drawString("Hello, Java 2D!", 50, 50);
    }
}
```

paintComponent vs paintComponents — One Letter Changes Everything

`paintComponent` (singular) is the method you override for custom drawing. `paintComponents` (with an 's') is a completely different method — it delegates painting to child components. If you accidentally override `paintComponents`, your custom drawing will fight with Swing's child-painting mechanism and produce confusing results. Always override `paintComponent` (singular).

1.2 The Painting Contract

The first line inside your `paintComponent` override should always be:

```
super.paintComponent(g);
```

This call does essential housekeeping — most importantly, it **clears the panel's background** by filling it with the panel's background color. If you skip this call, remnants of previous frames will linger on screen, producing visual artifacts. Every Swing tutorial and textbook emphasizes this rule.

1.3 Never Call paintComponent Directly

You never call `paintComponent` yourself. Instead, you call `repaint()`, and Swing schedules the paint operation on the Event Dispatch Thread at an appropriate time. This is a fundamental part of Swing's architecture — you request a repaint, and Swing decides when and how to execute it.

```
// GOOD: Request a repaint — Swing will call paintComponent for you
myPanel.repaint();

// BAD: Never do this — bypasses Swing's painting system
myPanel.paintComponent(myPanel.getGraphics());  // Don't do this!
```

---

2 The Graphics2D Object

The `paintComponent` method receives a `Graphics` parameter, but `Graphics` is the original AWT drawing API from 1996. The modern drawing API is `Graphics2D`, which extends `Graphics` with far more capabilities — antialiasing, stroke control, shape objects, and gradient paints.

Swing always passes a `Graphics2D` instance to `paintComponent`, even though the parameter type is `Graphics`. You just need to cast it:

```
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);

    Graphics2D g2d = (Graphics2D) g;
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                         RenderingHints.VALUE_ANTIALIAS_ON);

    // Now use g2d for all your drawing
}
```

This Is a Safe Cast

You might wonder: if casting can throw a `ClassCastException`, why not use `instanceof` first? Because this cast is **guaranteed to succeed**. Swing's internal painting system always creates a `Graphics2D` object and passes it to `paintComponent` — this has been the case since Java 1.2 (1998). The parameter type is `Graphics` rather than `Graphics2D` only because `paintComponent` was designed for backward compatibility with the original AWT API.

A "safe cast" means the runtime type is guaranteed by the framework's contract, not just by hope. You are not guessing that the object *might* be a `Graphics2D` — Swing *promises* it is one. Using `instanceof` here would be defensive against a scenario that cannot happen, which adds noise without value. Save `instanceof` for situations where the runtime type is genuinely uncertain.

2.1 Why Enable Antialiasing

Without antialiasing, shapes and lines are drawn with hard pixel boundaries, producing visible "staircase" effects on diagonal and curved edges. Enabling antialiasing tells the renderer to blend edge pixels with the background, producing smooth, professional-looking output.

The difference is especially visible on circles, ellipses, and diagonal lines. There is almost no reason to leave antialiasing off — the performance cost is negligible for typical 2D rendering.

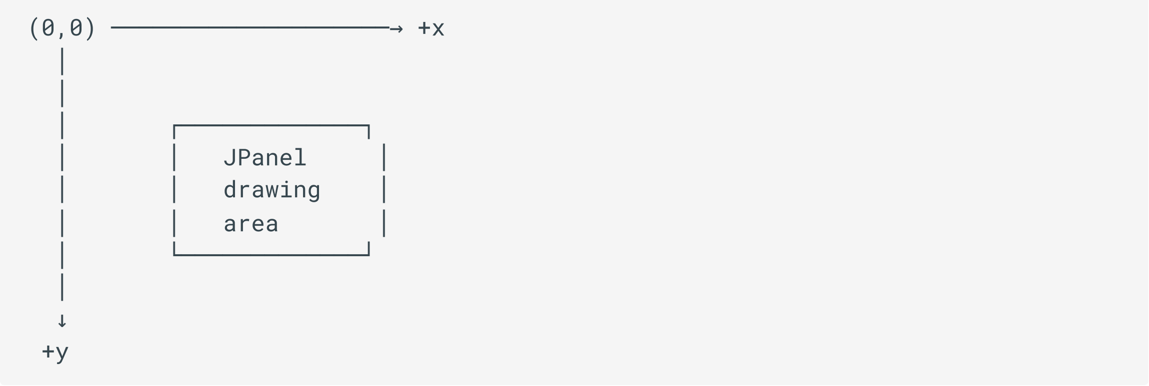
---

3 The Coordinate System

Before you draw anything, you need to understand where things will appear.

3.1 Origin and Axes

The coordinate system for a Swing component places the **origin (0, 0) at the top-left corner** of the component. The x-axis increases to the right, and the y-axis increases **downward** — the opposite of what you may be used to from math class.



+y goes DOWN, not up!

This is the single most common source of "my drawing is upside down" bugs. In math class, increasing y moves **up**. In computer graphics, increasing y moves **down**. Every pixel, every shape, every coordinate you compute must account for this. If something appears at the bottom when you expected the top (or vice versa), check your y-axis assumptions first.

This "y goes down" convention is standard across virtually all computer graphics systems — not just Java.

3.2 Panel Dimensions

You can query the panel's current size at any time using `getWidth()` and `getHeight()`. These return the panel's dimensions in pixels, and they update when the panel is resized.

```
int panelWidth = getWidth();     // Current width in pixels
int panelHeight = getHeight();   // Current height in pixels
```

Coordinates Are Relative to the Component

All coordinates in `paintComponent` are relative to the component being painted, **not** the screen. The point (0, 0) is the top-left corner of your `JPanel`, regardless of where the panel sits on screen or within a `JFrame`.

4 Drawing vs Filling

The Graphics2D API distinguishes between two kinds of rendering operations:

- `draw` **methods** render **outlines** — the border of a shape with no fill
- `fill` **methods** render **solid shapes** — the interior is filled with the current color

```
// Draw just the outline of a rectangle
g2d.drawRect(50, 50, 200, 100);

// Fill the entire interior of a rectangle
g2d.fillRect(300, 50, 200, 100);
```

4.1 The Painter's Algorithm

Drawing order matters. Things drawn later appear **on top of** things drawn earlier — just like painting on a canvas. This is called the **painter's algorithm**: later paint covers earlier paint.

```
// The blue rectangle is drawn first — it will be underneath
g2d.setColor(Color.BLUE);
g2d.fillRect(50, 50, 200, 150);

// The red rectangle is drawn second — it appears on top
g2d.setColor(Color.RED);
g2d.fillRect(100, 100, 200, 150);
```

If you want a border around a filled shape, draw the fill first, then the outline on top:

```
// Step 1: Fill the shape
g2d.setColor(Color.YELLOW);
g2d.fillRect(50, 50, 200, 100);

// Step 2: Draw the outline on top
g2d.setColor(Color.BLACK);
g2d.drawRect(50, 50, 200, 100);
```

---

5 The Stroke API

When you call a `draw` method (any method that renders an outline), the **stroke** determines how that outline looks — its thickness, how line ends are capped, and how corners are joined.

5.1 Setting Line Thickness

The default stroke is 1 pixel wide, which is often too thin for game graphics or visualizations. Use `BasicStroke` to control line width:

```
// Set a 3-pixel-wide stroke
g2d.setStroke(new BasicStroke(3));
g2d.drawRect(50, 50, 200, 100);  // This outline is now 3 pixels wide

// Set a thicker stroke for emphasis
g2d.setStroke(new BasicStroke(6));
g2d.drawOval(300, 50, 150, 150);  // This circle outline is 6 pixels wide
```

> ⚠ **Stroke Width Extends Outside the Shape**
>
> A stroke is centered on the shape's boundary — **half the width draws outside the shape's coordinates and half draws inside**. For example, with a stroke width of 9: 4 pixels extend outside the shape, 4 pixels extend inside, and 1 pixel sits on the boundary itself. This means a `drawRect(50, 50, 200, 100)` with a stroke of 9 actually paints pixels starting at x=46, y=46. If your shape is near the edge of the panel, a thick stroke can be clipped or appear to "bleed" outside where you expect.

5.2 Stroke Is Stateful

Like `Color`, the stroke is a **stateful property** of the graphics context. Once you set it, it applies to all subsequent `draw` calls until you change it. There is no "reset" — you explicitly set a new stroke when you want a different thickness.

5.3 Cap and Join Styles

`BasicStroke` offers options beyond just width. Two that matter for clean rendering:

- **Cap styles** control how line ends look: `BasicStroke.CAP_ROUND` (rounded ends), `BasicStroke.CAP_SQUARE` (square ends extending past the endpoint), `BasicStroke.CAP_BUTT` (flat ends at the endpoint)
- **Join styles** control how corners look: `BasicStroke.JOIN_ROUND` (rounded corners), `BasicStroke.JOIN_MITER` (sharp corners), `BasicStroke.JOIN_BEVEL` (flat corners)

```
// A 4-pixel stroke with rounded caps and rounded joins
g2d.setStroke(new BasicStroke(4,
    BasicStroke.CAP_ROUND,
    BasicStroke.JOIN_ROUND));
```

These are useful for clean grid lines and polished borders.

Stroke Only Affects draw Calls

The stroke controls outlines rendered by `draw` methods. `fill` methods ignore the stroke entirely — they fill the interior of a shape with no regard for stroke width. This is a common source of confusion: setting a thick stroke does not make filled shapes larger.

5.4 Common Pattern: Fill Then Stroke

A common technique for rendering shapes with visible borders: fill the shape for its solid color, then draw the same shape with a contrasting stroke for its border.

```
// A blue square with a thick black border
g2d.setColor(Color.BLUE);
g2d.fillRect(50, 50, 100, 100);

g2d.setColor(Color.BLACK);
g2d.setStroke(new BasicStroke(3));
g2d.drawRect(50, 50, 100, 100);
```

---

6 The Shape API — Rectangles and Ellipses

So far we have used convenience methods like `drawRect` and `fillRect`. The Graphics2D API also provides a more powerful **Shape-based** approach using objects from the `java.awt.geom` package.

6.1 Creating Shape Objects

The two shapes you will use most often:

```
import java.awt.geom.Rectangle2D;
import java.awt.geom.Ellipse2D;

// A rectangle at (50, 50) with width 200 and height 100
Rectangle2D rect = new Rectangle2D.Double(50, 50, 200, 100);

// An ellipse bounded by the same box — inscribed in a 200x100 rectangle
Ellipse2D ellipse = new Ellipse2D.Double(50, 200, 200, 100);
```

The `Ellipse2D` constructor takes the same parameters as `Rectangle2D` — it defines the **bounding box** that contains the ellipse. A circle is simply an ellipse whose bounding box is a square.

The (x, y) of an Ellipse Is NOT Inside the Ellipse

The `(x, y)` you pass to `Ellipse2D.Double` is the **top-left corner of the bounding box**, not a point on or inside the ellipse. The ellipse touches the bounding box at the midpoint of each side — so the actual ellipse does not reach the corner at `(x, y)`. If you draw both the bounding rectangle and the ellipse with the same parameters, the ellipse is inscribed inside the rectangle, touching only at the four midpoints of each side.

Why `Rectangle2D.Double`? What Is That Syntax?

`Rectangle2D.Double` looks unusual — it is not a method call or a package path. `Double` is a **static inner class** inside `Rectangle2D`. The outer class `Rectangle2D` is abstract; you cannot instantiate it directly. Instead, you instantiate one of its inner classes: `Rectangle2D.Double` (which stores coordinates as `double` values) or `Rectangle2D.Float` (which uses `float`). The same pattern applies to `Ellipse2D.Double`, `Line2D.Double`, and other shapes in `java.awt.geom`. Use the `.Double` version — `double` is Java's default floating-point type and avoids narrowing issues.

6.2 Drawing and Filling Shapes

Once you have a Shape object, pass it to `g2d.draw()` or `g2d.fill()`:

```
g2d.setColor(Color.GREEN);
g2d.fill(rect);    // Fill the rectangle with green

g2d.setColor(Color.BLACK);
g2d.draw(rect);    // Draw the rectangle's outline in black

g2d.setColor(Color.RED);
g2d.fill(ellipse); // Fill the ellipse with red
```

6.3 Why Use Shape Objects

Shape objects are preferred over the primitive `drawRect` / `fillRect` convenience methods for several reasons:

Approach	Advantage
`g2d.fillRect(50, 50, 200, 100)`	Quick for one-off drawing
`Rectangle2D rect = new Rectangle2D.Double(...)`	Reusable — draw and fill the same shape; store in collections; pass to methods; use for hit testing

When you are rendering a game board with many cells, creating Shape objects and storing them in a data structure is far more manageable than passing raw coordinate values everywhere.

6.4 Other Shapes

The `java.awt.geom` package includes many more shapes — `Line2D`, `Arc2D`, `Path2D`, `RoundRectangle2D` — but rectangles and ellipses cover the vast majority of game rendering needs. You can explore the others as needed.

---

7 Colors and Gradients

7.1 Setting Colors

The `setColor` method sets the color for all subsequent draw and fill operations:

```
// Using predefined constants
g2d.setColor(Color.RED);
g2d.fillRect(50, 50, 100, 100);

// Using a custom RGB color
g2d.setColor(new Color(64, 128, 255));  // A medium blue
g2d.fillRect(200, 50, 100, 100);
```

The `Color` class provides constants for common colors: `Color.RED`, `Color.BLUE`, `Color.GREEN`, `Color.BLACK`, `Color.WHITE`, `Color.GRAY`, `Color.YELLOW`, `Color.CYAN`, `Color.MAGENTA`, `Color.ORANGE`, `Color.PINK`.

For custom colors, use `new Color(r, g, b)` where each value ranges from 0 to 255.

Set Color Before Every Draw/Fill Call

The graphics context is **stateful** — the current color persists until you change it. If you forget to set a color before a drawing operation, you get whatever color was set by the previous operation. This is the most common source of "why is everything the same color?" bugs. Make it a habit to set the color immediately before each draw or fill call.

7.2 Gradient Paints

For visual polish, `GradientPaint` creates a smooth transition between two colors:

```
import java.awt.GradientPaint;

// Gradient from blue (at top) to white (at bottom)
GradientPaint gradient = new GradientPaint(
    0, 0, Color.BLUE,          // Start point and color
    0, getHeight(), Color.WHITE // End point and color
);

g2d.setPaint(gradient);
g2d.fillRect(0, 0, getWidth(), getHeight());
```

Gradients are not required for course assignments but can make visualizations and game panels look more polished.

---

8 repaint() and the Paint Cycle

Understanding how painting works is essential for writing correct Swing code.

8.1 The Repaint Request

You never call `paintComponent` directly. Instead, you call `repaint()`:

```
// Something changed in the model — request a visual update
myPanel.repaint();
```

`repaint()` does **not** paint immediately. It posts a paint request to Swing's event queue. Swing will call `paintComponent` on the Event Dispatch Thread when it processes that request.

8.2 Coalescing

If multiple `repaint()` calls arrive before Swing processes the first one, Swing may **coalesce** them into a single paint operation. This is an optimization — there is no point in painting the same panel three times when one paint with the latest state is sufficient.

This means `repaint()` is a **request**, not a command. Students often expect `repaint()` to be synchronous — to immediately trigger painting and block until painting is finished. It does not. It is asynchronous.

8.3 Double Buffering

`JPanel` is **double-buffered by default**. This means Swing draws to an off-screen buffer first, then copies the completed image to the screen in one operation. The result: flicker-free rendering with no extra work on your part.

If you have used graphics APIs in other languages that required you to manage double buffering manually, Swing handles it for you.

Looking Ahead: Animation with javax.swing.Timer

Understanding the repaint cycle will be critical when we add animations with `javax.swing.Timer`. The Timer fires events that trigger `repaint()`, and knowing that Swing coalesces and schedules paints — rather than executing them immediately — is key to smooth animation. The Timer guide will build directly on the concepts from this section.

---

9 Drawing Text

Custom panels often need to render text — scores, labels, status messages, or game-over overlays.

9.1 Basic Text Drawing

```
g2d.setColor(Color.BLACK);
g2d.drawString("Score: 100", 20, 30);
```

The `drawString` method takes a string and an (x, y) position. **The (x, y) specifies the baseline of the text**, not the top-left corner. The baseline is the line that letters sit on —

descenders (like 'g', 'p', 'y') extend below it.

```
         ┌── This is the top of the text (ascent)
   Score │
   ──────┤── This is the baseline (x, y) position
         │
         └── Descenders hang below the baseline
```

9.2 Setting Fonts

Set the font before calling `drawString`:

```
g2d.setFont(new Font("SansSerif", Font.BOLD, 24));
g2d.drawString("GAME OVER", 100, 200);

g2d.setFont(new Font("Monospaced", Font.PLAIN, 14));
g2d.drawString("Press R to restart", 100, 230);
```

The `Font` constructor takes three arguments:

1. **Font family** — `"SansSerif"`, `"Serif"`, `"Monospaced"` are guaranteed on all platforms
2. **Style** — `Font.PLAIN`, `Font.BOLD`, `Font.ITALIC`, or combined with `Font.BOLD | Font.ITALIC`
3. **Size** — Font size in points

Like `Color` and `Stroke`, the font is **stateful** on the graphics context. Set it before drawing, and it stays until changed.

---

10 Transparency

Transparency allows you to draw semi-transparent shapes — useful for ghost pieces, overlays, highlights, and fade effects.

10.1 Alpha Values

The `Color` constructor has a four-argument form that includes an **alpha** channel:

```
// new Color(red, green, blue, alpha)
// Alpha: 0 = fully transparent, 255 = fully opaque

Color semiTransparentRed = new Color(255, 0, 0, 128);  // 50% transparent red
```

```
g2d.setColor(semiTransparentRed);
g2d.fillRect(50, 50, 200, 200);
```

With an alpha of 128 (roughly 50%), the red rectangle will blend with whatever is behind it.

10.2 Layering Transparent Shapes

Transparency combined with the painter's algorithm lets you create layered effects:

```
// Draw a solid blue background
g2d.setColor(Color.BLUE);
g2d.fillRect(50, 50, 200, 200);

// Draw a semi-transparent yellow overlay
g2d.setColor(new Color(255, 255, 0, 100));
g2d.fillRect(100, 100, 200, 200);
```

The overlapping region will show a blended color.

Reset to Opaque After Transparent Drawing

Remember to set back to a fully opaque color after drawing transparent elements. If you forget, all subsequent drawing operations will be semi-transparent — a subtle bug that can be hard to spot.

---

11 Mapping Model Coordinates to Pixels

In real applications, your data model rarely uses pixel coordinates. A game board might use grid positions (column 4, row 7). A chart might use data values (temperature 72.3 at time step 15). Your `paintComponent` method must **translate model coordinates into pixel coordinates**.

11.1 The Core Problem

Suppose your model uses a grid with a certain number of columns and rows. The panel has a width and height in pixels. You need to map between these two coordinate systems every time you paint.

11.2 Calculating Block Size

First, calculate how many pixels each grid cell occupies:

```
int gridColumns = 10;  // From your model
int gridRows = 20;     // From your model

int blockWidth = getWidth() / gridColumns;
int blockHeight = getHeight() / gridRows;
```

This divides the panel evenly into a grid of cells.

11.3 Converting Grid Position to Pixel Position

To find the pixel position for a given grid cell:

```
// Where does grid cell (column, row) start in pixels?
int pixelX = column * blockWidth;
int pixelY = row * blockHeight;

// Draw a block at that position
g2d.fillRect(pixelX, pixelY, blockWidth, blockHeight);
```

This is the fundamental formula: **pixel position = grid position * block size**.

11.4 Handling Y-Axis Differences

The Model's Y-Axis May Differ from the Screen's Y-Axis

In many models, row 0 is at the **bottom** (like a math coordinate system). But on screen, y = 0 is at the **top**. If your model uses a bottom-origin y-axis, you need to flip the row when converting to pixels:

```
// If row 0 is the bottom in the model but the top on screen:
int pixelY = (gridRows - 1 - row) * blockHeight;
```

Whether you need this flip depends entirely on your model's conventions. Always check which direction row numbers increase.

11.5 Why Recalculate on Every Paint

You might be tempted to calculate block sizes once and store them in a field. However, the panel can be **resized** at any time. If the user drags the window border, the panel dimensions change, and your block sizes must update accordingly.

The simplest correct approach: recalculate `blockWidth` and `blockHeight` at the top of `paintComponent` using `getWidth()` and `getHeight()`. This guarantees your rendering adapts to the current panel size, every frame.

```
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2d = (Graphics2D) g;
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                         RenderingHints.VALUE_ANTIALIAS_ON);

    // Recalculate block size based on current panel dimensions
    int blockWidth = getWidth() / gridColumns;
    int blockHeight = getHeight() / gridRows;

    // Now render the grid using these block sizes
    for (int row = 0; row < gridRows; row++) {
        for (int col = 0; col < gridColumns; col++) {
            int pixelX = col * blockWidth;
            int pixelY = row * blockHeight;
            // Draw each cell...
        }
    }
}
```

Summary

Concept	Key Point
`paintComponent`	Override this (not `paintComponents`!) on a JPanel
`super.paintComponent(g)`	Always call first to clear the background
Graphics2D	Cast from Graphics for antialiasing and Shape API
Coordinate system	(0,0) top-left, +y is down
draw vs fill	Outlines vs solid shapes
Stroke	Controls line thickness for `draw` calls
Color	Stateful — set before each draw/fill
`repaint()`	Request a repaint — never call `paintComponent` directly

Concept	Key Point
`drawString`	Baseline positioning, not top-left
Transparency	`new Color(r, g, b, alpha)` — 0 to 255
Grid-to-pixel	`pixel = gridPosition * blockSize`

Further Reading

External Resources

- Oracle: Performing Custom Painting — Official step-by-step tutorial for custom rendering in Swing
- Oracle: 2D Graphics Trail — Comprehensive coverage of shapes, colors, transforms, and text rendering
- Oracle: Painting in AWT and Swing — Deep dive into how and when Swing paints components

---

References

Primary Texts:

- Horstmann, C. S. (2022). *Core Java, Volume I: Fundamentals* (12th ed.). Oracle Press. Chapter 10: Graphical User Interface Programming — Custom painting, Graphics2D, and 2D shapes.
- Reges, S., & Stepp, M. (2020). *Building Java Programs* (5th ed.). Pearson. Chapter 3g (Supplement): Graphics — Introduction to drawing with Graphics objects.
- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley. Item 17: Minimize mutability — relevant to stateful graphics context patterns.

Language Documentation:

- Oracle JDK 25: Graphics2D — Official API reference for the Graphics2D class
- Oracle JDK 25: BasicStroke — Stroke configuration for line rendering
- Oracle JDK 25: Rectangle2D — Rectangle shape class

- Oracle JDK 25: Ellipse2D — Ellipse shape class
- Oracle JDK 25: Color — Color class including alpha transparency
- Oracle JDK 25: RenderingHints — Antialiasing and rendering quality controls

Tutorials:

- Oracle: Performing Custom Painting — Official Swing painting tutorial
- Oracle: 2D Graphics — Official Java 2D tutorial trail

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