Chapter 9: Graphics and User Interfaces

Object Oriented Programming with Java

Graphics programming | JFrame and JPanel | Drawing with Graphics | Point, Color, Font | Layout managers | Mouse events | Buttons and text components | Sample apps

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/java/slides/Chapter_09_Graphics_and_User_Interfaces.html

Learning Outcomes

Build a window

Create windows and panels with `JFrame` and `JPanel`, add components, show and close the window.

Draw

Override `paintComponent` and draw shapes and text with `Graphics`, using screen coordinates.

Lay out

Arrange components with `BorderLayout`, `FlowLayout` and `GridLayout`.

Respond to events

Handle mouse and button events with listener interfaces, adapter classes and lambdas.

Start up safely

Explain why the GUI is built on the event thread with `SwingUtilities.invokeLater`.

Write applications

Read and modify a drawing program, a text converter and a change calculator.

Graphics Programming

- Java offers a powerful, complete toolkit for building **graphical user interfaces** (GUIs): windows, buttons, text fields, drawings.
- With it you can write **interactive, visual** applications. Three ideas are key:

Components

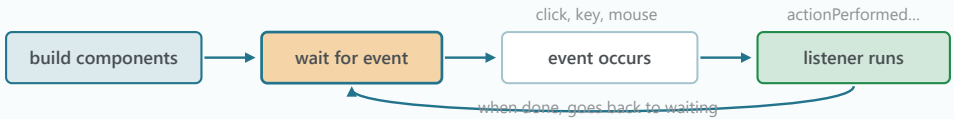
The **building blocks** of the interface: windows, buttons, labels, text fields. Each is an object of some class.

Event handling

Captures user actions such as mouse clicks and key presses, and calls the methods that respond to them.

Painting

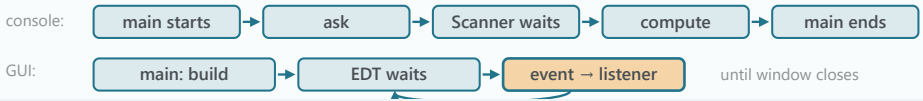
Draws shapes, updates the screen and **renders** the components.



Analogy: a GUI program is like a shop waiting for customers. The window display (components) is set up, then the program **waits**; whenever a customer touches something (an event), the right clerk (a listener method) steps in.

Console vs GUI Programs

	Console program	GUI program
Flow	Sequential: `main` runs top to bottom; the program asks the questions	**Event-driven**: the program waits for the user's move
Input	Lines read from the keyboard with `Scanner`	Buttons, text fields, mouse; listener methods are called
Output	`System.out.println`	Component text changes or the panel is repainted
End	The program ends when `main` does	Runs until the window closes, even after `main` returns
Library	`java.util`, `java.io`	`java.awt`, `javax.swing`, `java.awt.event`



Rule: in a GUI program, “what happens next?” is decided not by the order of the code but by **the user's next action**. Most of the code is listeners of the form “when this happens, do that”.

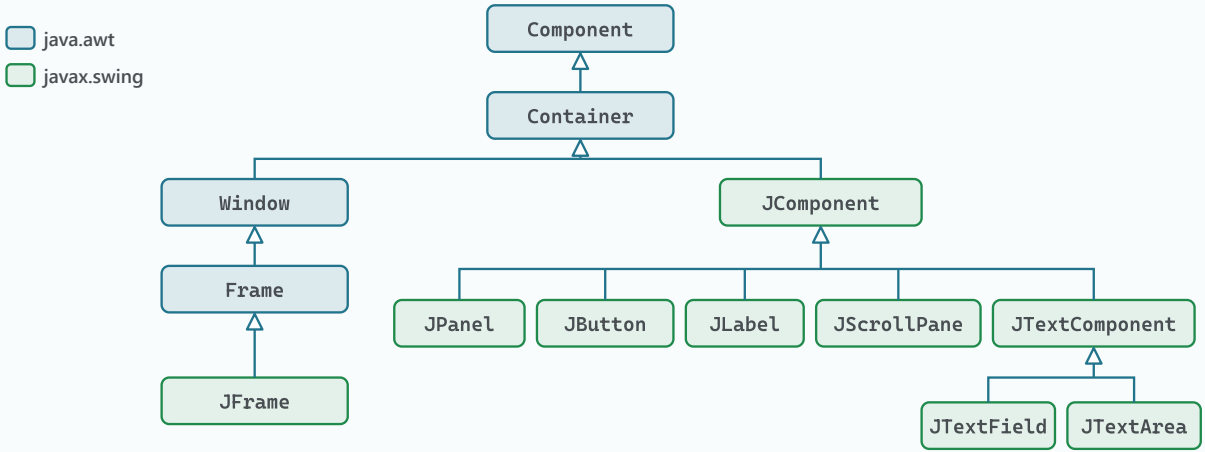
AWT and Swing

AWT (Abstract Window Toolkit), `java.awt`

Java's first graphics library. Today it is used mostly for its shared classes: `Graphics`, `Color`, `Font`, `Point`, layout managers and event classes (`java.awt.event`).

Swing, `javax.swing`

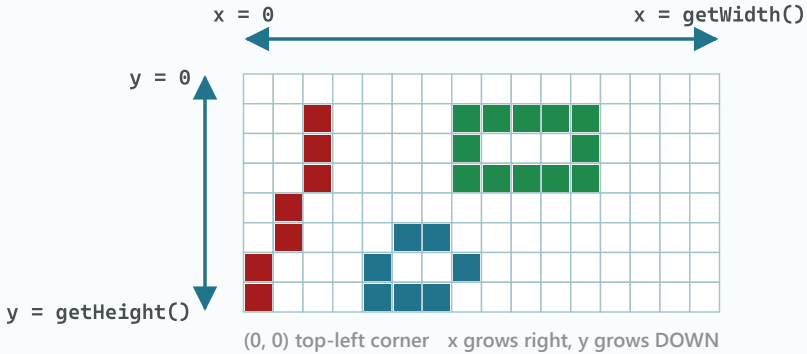
Built on top of AWT; Java draws the components itself. Class names start with **J**: `JFrame`, `JPanel`, `JButton`, `JLabel`...



Hollow triangle: "extends" (inheritance). `JFrame` is an AWT window; the other Swing components extend `JComponent` (`JButton` via `AbstractButton`).

Building a GUI App

1. **Add components** to the interface using Java's graphics classes.
2. Define **event-handling** methods that respond to user actions.
3. **Render the visuals** with methods such as `paint()` or `paintComponent()`.



The screen is a **grid of pixels**. Coordinates are integers measured from the component's **top-left corner**. Unlike the axes in math, **y grows downward**: `(0, 0)` is the top-left pixel, `(getWidth() - 1, getHeight() - 1)` the bottom-right one.

Basic Graphics Concepts

Canvas

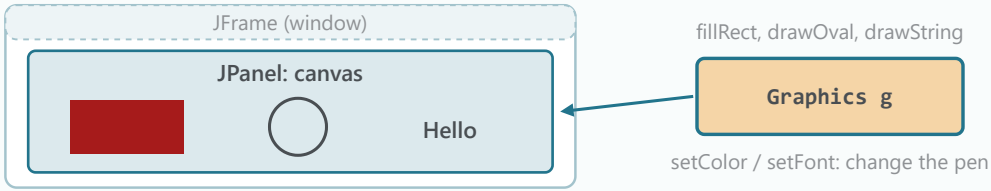
A surface you can draw on. Swing usually uses a `JPanel` (AWT has `Canvas`): we extend `JPanel` and draw inside it.

The `Graphics` class

The "pen" that does the drawing. Methods: `drawLine()`, `drawRect()`, `fillOval()`, `drawString()`, `drawImage()`...

`Color` and `Font`

`Color` sets the pen's color and `Font` its typeface: `g.setColor(Color.RED)`, `g.setFont(new Font("Serif", Font.BOLD, 24))`.



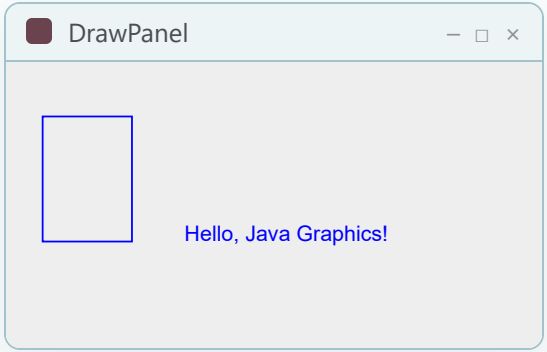
The pen analogy: the `Graphics` object is the pen in the programmer's hand. `setColor` swaps the pen; until you swap it again, everything is drawn **in the same color**.

Drawing with paintComponent

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

class DrawPanel extends JPanel {
    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);     // clear background
        g.setColor(Color.BLUE);
        g.drawRect(20, 30, 50, 70);  // x, y, w, h
        g.drawString("Hello, Java Graphics!", 100, 100);
    }
}
```

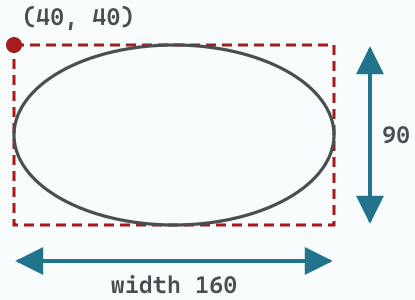
- **We never call** `paintComponent` ourselves: Swing calls it when the window first opens, when it is resized or when `repaint()` is requested, and hands us a ready `Graphics g`.
- `panel.getGraphics()` also returns a graphics object, but anything drawn with it is **erased** the next time the window repaints. For lasting drawings, use `paintComponent`.



Result with a 300 × 160 panel. The y of `drawString` is the text's **baseline** (the bottom of the letters).

Graphics Drawing Methods

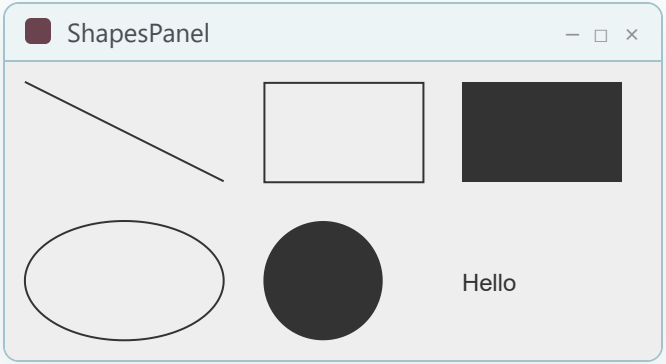
Method	What it does
`drawLine(x1, y1, x2, y2)`	Line between two points
`drawRect(x, y, w, h)`	Rectangle outline; (x, y) is the top-left corner
`fillRect(x, y, w, h)`	Filled rectangle
`drawOval(x, y, w, h)`	Ellipse that fits this rectangle (circle if w = h)
`fillOval(x, y, w, h)`	Filled ellipse / circle
`drawString(s, x, y)`	Text; y is the baseline
`setColor(c)` / `setFont(f)`	Color / font of later drawings
`drawImage(img, x, y, null)`	Draws an image at (x, y)



Ellipses and circles are given by their **bounding rectangle**: the top-left corner, not the center. A circle with center (cx, cy) and radius r: `fillOval(cx - r, cy - r, 2r, 2r)`.

Shapes: Code and Result

```
class ShapesPanel extends JPanel {
    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawLine(10, 10, 110, 60);
        g.drawRect(130, 10, 80, 50);
        g.fillRect(230, 10, 80, 50);
        g.drawOval(10, 80, 100, 60);
        g.fillOval(130, 80, 60, 60);
        g.drawString("Hello", 230, 115);
    }
}
```

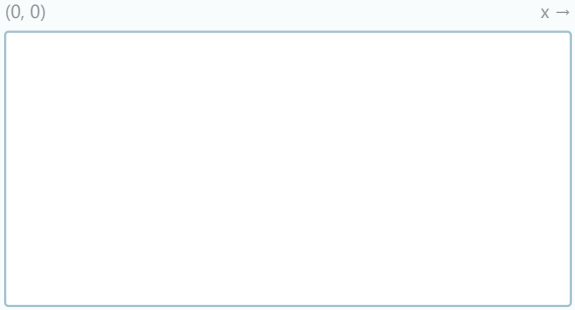


With no color set, the pen uses the panel's **foreground color** (dark gray in Swing). `draw…` draws outlines, `fill…` filled shapes.

Drawings are painted **on top of each other**; later ones cover earlier ones.

Shapes: Drawing Order, Step by Step

```
1  class ShapesPanel extends JPanel {
2      @Override
3      public void paintComponent(Graphics g) {
▶          super.paintComponent(g);
5          g.drawLine(10, 10, 110, 60);
6          g.drawRect(130, 10, 80, 50);
7          g.fillRect(230, 10, 80, 50);
8          g.drawOval(10, 80, 100, 60);
9          g.fillOval(130, 80, 60, 60);
10         g.drawString("Hello", 230, 115);
11     }
12 }
```



Swing calls this method itself.

`super.paintComponent(g)` **clears** the canvas with the background color.

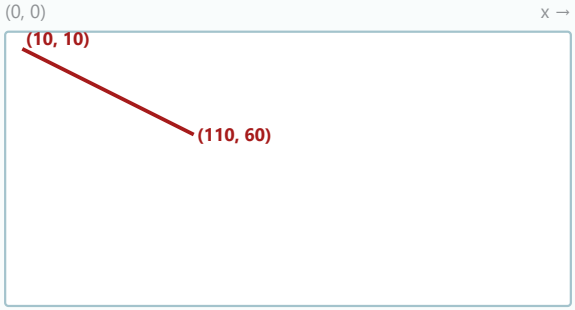
Step 1 / 8

VARIABLES

`g = Graphics (dark gray pen)`

Shapes: Drawing Order, Step by Step

```
1  class ShapesPanel extends JPanel {
2      @Override
3      public void paintComponent(Graphics g) {
4          super.paintComponent(g);
▶          g.drawLine(10, 10, 110, 60);
6          g.drawRect(130, 10, 80, 50);
7          g.fillRect(230, 10, 80, 50);
8          g.drawOval(10, 80, 100, 60);
9          g.fillOval(130, 80, 60, 60);
10         g.drawString("Hello", 230, 115);
11     }
12 }
```



A line between two points. Since y grows downward, the line runs right and **down**.

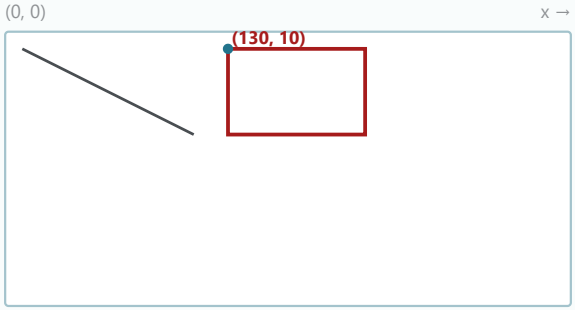
Step 2 / 8

VARIABLES

`x1 = 10` `y1 = 10` `x2 = 110` `y2 = 60`

Shapes: Drawing Order, Step by Step

```
1  class ShapesPanel extends JPanel {
2      @Override
3      public void paintComponent(Graphics g) {
4          super.paintComponent(g);
5          g.drawLine(10, 10, 110, 60);
▶          g.drawRect(130, 10, 80, 50);
7          g.fillRect(230, 10, 80, 50);
8          g.drawOval(10, 80, 100, 60);
9          g.fillOval(130, 80, 60, 60);
10         g.drawString("Hello", 230, 115);
11     }
12 }
```



`drawRect` : (x, y) is the **top-left corner**; only the outline is drawn.

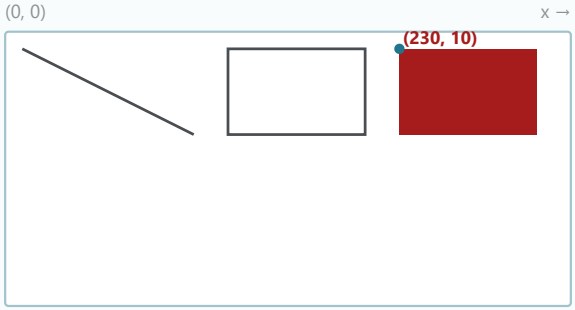
Step 3 / 8

VARIABLES

`x = 130` `y = 10` `w = 80` `h = 50`

Shapes: Drawing Order, Step by Step

```
1  class ShapesPanel extends JPanel {
2      @Override
3      public void paintComponent(Graphics g) {
4          super.paintComponent(g);
5          g.drawLine(10, 10, 110, 60);
6          g.drawRect(130, 10, 80, 50);
▶          g.fillRect(230, 10, 80, 50);
8          g.drawOval(10, 80, 100, 60);
9          g.fillOval(130, 80, 60, 60);
10         g.drawString("Hello", 230, 115);
11     }
12 }
```



`fillRect`: a rectangle of the same size, filled with the pen color.

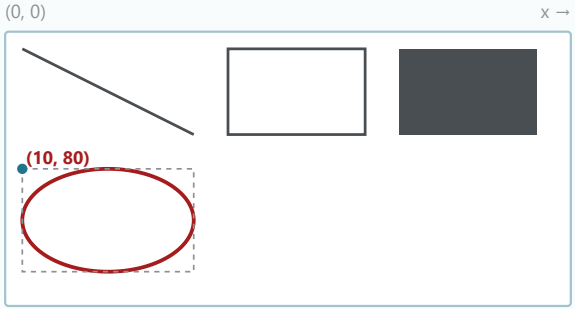
Step 4 / 8

VARIABLES

`x = 230` `y = 10` `w = 80` `h = 50`

Shapes: Drawing Order, Step by Step

```
1  class ShapesPanel extends JPanel {
2      @Override
3      public void paintComponent(Graphics g) {
4          super.paintComponent(g);
5          g.drawLine(10, 10, 110, 60);
6          g.drawRect(130, 10, 80, 50);
7          g.fillRect(230, 10, 80, 50);
▶          g.drawOval(10, 80, 100, 60);
9          g.fillOval(130, 80, 60, 60);
10         g.drawString("Hello", 230, 115);
11     }
12 }
```



`drawOval` : the ellipse that **fits** the dashed box. (10, 80) is the box's corner, not the ellipse's.

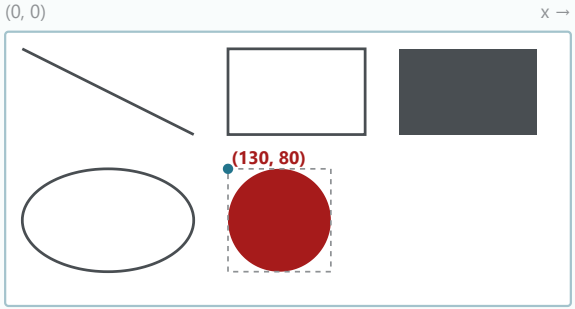
Step 5 / 8

VARIABLES

`x = 10` `y = 80` `w = 100` `h = 60`

Shapes: Drawing Order, Step by Step

```
1  class ShapesPanel extends JPanel {
2      @Override
3      public void paintComponent(Graphics g) {
4          super.paintComponent(g);
5          g.drawLine(10, 10, 110, 60);
6          g.drawRect(130, 10, 80, 50);
7          g.fillRect(230, 10, 80, 50);
8          g.drawOval(10, 80, 100, 60);
▶          g.fillOval(130, 80, 60, 60);
10         g.drawString("Hello", 230, 115);
11     }
12 }
```



Width = height, so it is a filled **circle**: center (160, 110), radius 30.

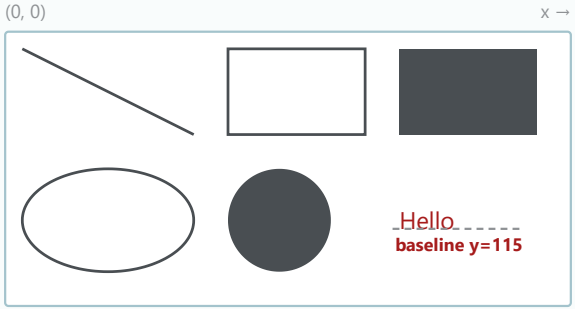
Step 6 / 8

VARIABLES

`x = 130` `y = 80` `w = 60` `h = 60`

Shapes: Drawing Order, Step by Step

```
1  class ShapesPanel extends JPanel {
2      @Override
3      public void paintComponent(Graphics g) {
4          super.paintComponent(g);
5          g.drawLine(10, 10, 110, 60);
6          g.drawRect(130, 10, 80, 50);
7          g.fillRect(230, 10, 80, 50);
8          g.drawOval(10, 80, 100, 60);
9          g.fillOval(130, 80, 60, 60);
▶          g.drawString("Hello", 230, 115);
11     }
12 }
```



In `drawString`, y is the text's **baseline**: the letters sit above 115.

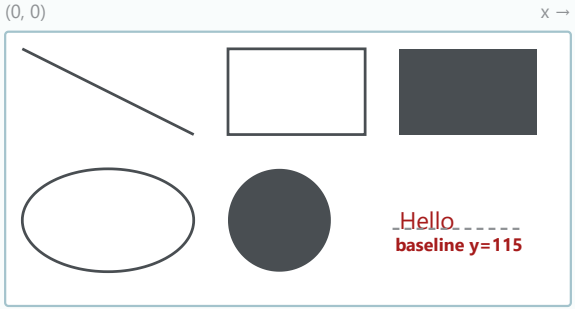
Step 7 / 8

VARIABLES

`s = "Hello"` `x = 230` `y = 115`

Shapes: Drawing Order, Step by Step

```
1  class ShapesPanel extends JPanel {
2      @Override
3      public void paintComponent(Graphics g) {
4          super.paintComponent(g);
5          g.drawLine(10, 10, 110, 60);
6          g.drawRect(130, 10, 80, 50);
7          g.fillRect(230, 10, 80, 50);
8          g.drawOval(10, 80, 100, 60);
9          g.fillOval(130, 80, 60, 60);
10         g.drawString("Hello", 230, 115);
▶      }
12 }
```



The method ends; every time the window repaints (resize, `repaint()`), everything is drawn again **from scratch** in the same order.

Step 8 / 8

VARIABLES

—

The JFrame Class

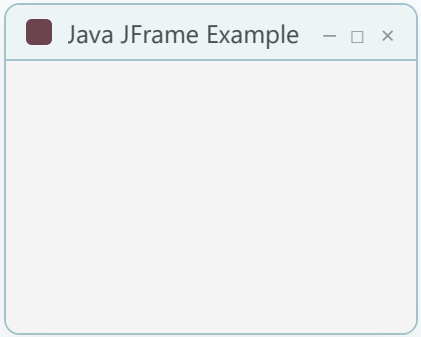
Creates a **top-level window** with a title bar and minimize and close buttons. Every other component lives inside a window.

Code	Purpose
`JFrame frame = new JFrame("Title");`	Creates a window with the given title (not yet visible)
`frame.add(component);`	Adds a component to the window (its content pane)
`frame.setSize(400, 300);`	Width and height (pixels, borders included)
`frame.pack();`	Sizes the window to its components' preferred sizes
`frame.setLocation(x, y);`	Position on screen; `setLocationRelativeTo(null)` centers it
`frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`	Ends the program when the window closes
`frame.setVisible(true);`	Makes the window visible (usually called last)

A First Window

```java
import javax.swing.JFrame;
import javax.swing.SwingUtilities;

public class JFrameDemo {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            JFrame frame = new JFrame("Java JFrame Example");
            frame.setSize(400, 300);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
        });
    }
}
```

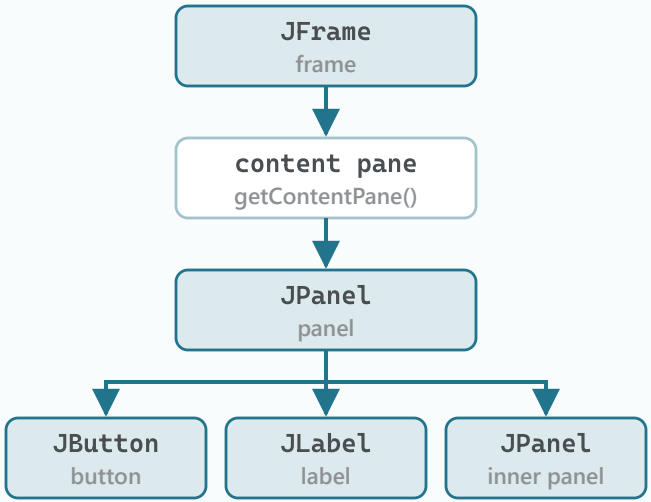


An empty 400 × 300 window (scaled).

- The GUI is built on the **event thread** with `invokeLater` (see The Event Thread).
- Forget `setVisible(true)` and the window exists but stays **invisible**.
- Without `EXIT_ON_CLOSE` the default is `HIDE_ON_CLOSE`: the window hides, but the program **keeps running in the background**.
- Without a size, the window opens at nearly 0 × 0, just a title bar.

The JPanel Class

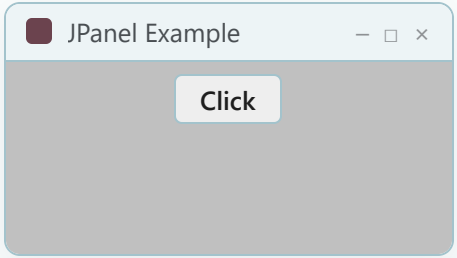
- A **container** in the `javax.swing` package: it holds other GUI elements.
- Used to **group and arrange** the components inside a window; nesting several panels builds complex layouts.
- Customize color, border and size with `setBackground()`, `setBorder()`, `setPreferredSize()`.
- The **canvas** for our own drawings, animations and graphics: extend `JPanel` and override `paintComponent`.
- Helps split the code up: each panel can be its own class.



JPanel Example

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

public class PanelDemo {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            JPanel panel = new JPanel();
            panel.setBackground(Color.LIGHT_GRAY);
            panel.add(new JButton("Click"));
            JFrame frame = new JFrame("JPanel Example");
            frame.add(panel);
            frame.setSize(300, 150);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
        });
    }
}
```

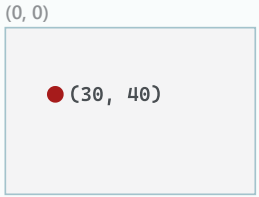


`Color.LIGHT_GRAY` = (192, 192, 192). The button sits at the top center of the panel.

Why top center? `JPanel` 's default layout manager is **`FlowLayout`** : it lays out components at their preferred size, row by row, centered. The `JFrame` 's content pane uses **`BorderLayout`** , so the panel fills the whole window.

The Point Class: an (x, y) Point

```
▶ Point p = new Point(30, 40);
2 double px = p.getX();
3 double py = p.getY();
4 System.out.println(px + " " + py);
5 System.out.println(p.x + " " + p.y);
6 System.out.println(p.distance(0, 0));
7 p.setLocation(60, 80);
8 System.out.println(p);
9 p.translate(5, -10);   // 5 right, 10 up
10 System.out.println(p);
11 System.out.println(Point.distance(0, 0, 3, 4));
```



A `Point` object is created; its fields `x` and `y` are `public int`.

Step 1 / 11

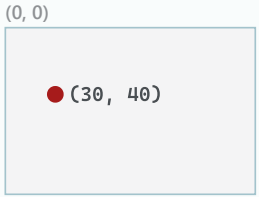
OUTPUT

VARIABLES

`p = (30, 40)`

The Point Class: an (x, y) Point

```
 1 Point p = new Point(30, 40);
 2 double px = p.getX();
 3 double py = p.getY();
 4 System.out.println(px + " " + py);
 5 System.out.println(p.x + " " + p.y);
 6 System.out.println(p.distance(0, 0));
 7 p.setLocation(60, 80);
 8 System.out.println(p);
 9 p.translate(5, -10);   // 5 right, 10 up
10 System.out.println(p);
11 System.out.println(Point.distance(0, 0, 3, 4));
```



`getX()` returns a `double`: 30.0.

Step 2 / 11

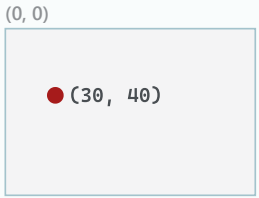
OUTPUT

VARIABLES

`p = (30, 40)` `px = 30.0`

The Point Class: an (x, y) Point

```
1  Point p = new Point(30, 40);
2  double px = p.getX();
▶  double py = p.getY();
4  System.out.println(px + " " + py);
5  System.out.println(p.x + " " + p.y);
6  System.out.println(p.distance(0, 0));
7  p.setLocation(60, 80);
8  System.out.println(p);
9  p.translate(5, -10);   // 5 right, 10 up
10 System.out.println(p);
11 System.out.println(Point.distance(0, 0, 3, 4));
```



`getY()` : 40.0.

Step 3 / 11

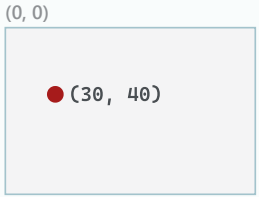
OUTPUT

VARIABLES

`p = (30, 40)` `px = 30.0` `py = 40.0`

The Point Class: an (x, y) Point

```
1  Point p = new Point(30, 40);
2  double px = p.getX();
3  double py = p.getY();
▶  System.out.println(px + " " + py);
5  System.out.println(p.x + " " + p.y);
6  System.out.println(p.distance(0, 0));
7  p.setLocation(60, 80);
8  System.out.println(p);
9  p.translate(5, -10);   // 5 right, 10 up
10 System.out.println(p);
11 System.out.println(Point.distance(0, 0, 3, 4));
```



`double` values print with a decimal point.

Step 4 / 11

OUTPUT

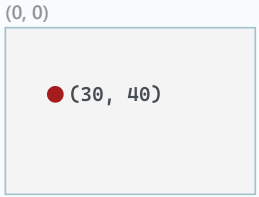
```
30.0 40.0
```

VARIABLES

p = (30, 40) px = 30.0 py = 40.0

The Point Class: an (x, y) Point

```
1  Point p = new Point(30, 40);
2  double px = p.getX();
3  double py = p.getY();
4  System.out.println(px + " " + py);
▶  System.out.println(p.x + " " + p.y);
6  System.out.println(p.distance(0, 0));
7  p.setLocation(60, 80);
8  System.out.println(p);
9  p.translate(5, -10);   // 5 right, 10 up
10 System.out.println(p);
11 System.out.println(Point.distance(0, 0, 3, 4));
```



Accessing the fields directly gives `int`s.

Step 5 / 11

OUTPUT

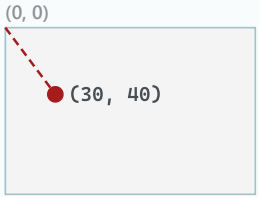
```
30.0 40.0
30 40
```

VARIABLES

p = (30, 40) px = 30.0 py = 40.0

The Point Class: an (x, y) Point

```
1  Point p = new Point(30, 40);
2  double px = p.getX();
3  double py = p.getY();
4  System.out.println(px + " " + py);
5  System.out.println(p.x + " " + p.y);
▶  System.out.println(p.distance(0, 0));
7  p.setLocation(60, 80);
8  System.out.println(p);
9  p.translate(5, -10);   // 5 right, 10 up
10 System.out.println(p);
11 System.out.println(Point.distance(0, 0, 3, 4));
```



Distance to the origin: $\sqrt{(30^2 + 40^2)} = 50.0$.

Step 6 / 11

OUTPUT

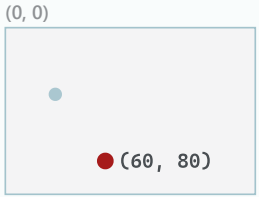
```
30.0 40.0
30 40
50.0
```

VARIABLES

p = **(30, 40)** px = **30.0** py = **40.0**

The Point Class: an (x, y) Point

```
1  Point p = new Point(30, 40);
2  double px = p.getX();
3  double py = p.getY();
4  System.out.println(px + " " + py);
5  System.out.println(p.x + " " + p.y);
6  System.out.println(p.distance(0, 0));
▶  p.setLocation(60, 80);
8  System.out.println(p);
9  p.translate(5, -10);   // 5 right, 10 up
10 System.out.println(p);
11 System.out.println(Point.distance(0, 0, 3, 4));
```



`setLocation` moves the point. The variables keep the old values.

OUTPUT

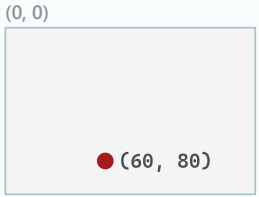
```
30.0 40.0
30 40
50.0
```

VARIABLES

p = (60, 80) px = 30.0 py = 40.0

The Point Class: an (x, y) Point

```
1  Point p = new Point(30, 40);
2  double px = p.getX();
3  double py = p.getY();
4  System.out.println(px + " " + py);
5  System.out.println(p.x + " " + p.y);
6  System.out.println(p.distance(0, 0));
7  p.setLocation(60, 80);
▶  System.out.println(p);
9  p.translate(5, -10);   // 5 right, 10 up
10 System.out.println(p);
11 System.out.println(Point.distance(0, 0, 3, 4));
```



`toString()` prints the class name and fields.

Step 8 / 11

OUTPUT

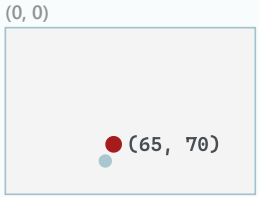
```
30.0 40.0
30 40
50.0
java.awt.Point[x=60,y=80]
```

VARIABLES

p = (60, 80) px = 30.0 py = 40.0

The Point Class: an (x, y) Point

```
 1 Point p = new Point(30, 40);
 2 double px = p.getX();
 3 double py = p.getY();
 4 System.out.println(px + " " + py);
 5 System.out.println(p.x + " " + p.y);
 6 System.out.println(p.distance(0, 0));
 7 p.setLocation(60, 80);
 8 System.out.println(p);
▶ p.translate(5, -10);   // 5 right, 10 up
10 System.out.println(p);
11 System.out.println(Point.distance(0, 0, 3, 4));
```



`translate(dx, dy)` shifts it relatively. y decreased: the point moved **up**.

Step 9 / 11

OUTPUT

```
30.0 40.0
30 40
50.0
java.awt.Point[x=60,y=80]
```

VARIABLES

p = (65, 70) px = 30.0 py = 40.0

The Point Class: an (x, y) Point

```
1  Point p = new Point(30, 40);
2  double px = p.getX();
3  double py = p.getY();
4  System.out.println(px + " " + py);
5  System.out.println(p.x + " " + p.y);
6  System.out.println(p.distance(0, 0));
7  p.setLocation(60, 80);
8  System.out.println(p);
9  p.translate(5, -10);   // 5 right, 10 up
▶  System.out.println(p);
11 System.out.println(Point.distance(0, 0, 3, 4));
```



The new position is printed.

OUTPUT

```
30.0 40.0
30 40
50.0
java.awt.Point[x=60,y=80]
java.awt.Point[x=65,y=70]
```

VARIABLES

p = (65, 70) px = 30.0 py = 40.0

The Point Class: an (x, y) Point

```
1  Point p = new Point(30, 40);
2  double px = p.getX();
3  double py = p.getY();
4  System.out.println(px + " " + py);
5  System.out.println(p.x + " " + p.y);
6  System.out.println(p.distance(0, 0));
7  p.setLocation(60, 80);
8  System.out.println(p);
9  p.translate(5, -10);   // 5 right, 10 up
10 System.out.println(p);
▶  System.out.println(Point.distance(0, 0, 3, 4));
```



`static` `distance(x1, y1, x2, y2)` : from (0, 0) to (3, 4) is 5.0.

Step 11 / 11

OUTPUT

```
30.0 40.0
30 40
50.0
java.awt.Point[x=60,y=80]
java.awt.Point[x=65,y=70]
5.0
```

VARIABLES

`p = (65, 70)` `px = 30.0` `py = 40.0`

The Color Class

- Provides **coloring** for the interface and drawings (`java.awt.Color`).
- **Predefined constants**: `Color.RED` , `Color.GREEN` , `Color.WHITE` , `Color.LIGHT_GRAY` ...
- **RGB**: red, green and blue components from 0 to 255: `new Color(int red, int green, int blue)` .
- `darker()` multiplies the components by 0.7, `brighter()` divides them by 0.7 (max 255): each returns a **new** color object.


`Color.RED` (255, 0, 0)


`Color.BLUE` (0, 0, 255)


`Color.LIGHT_GRAY` (192, 192, 192)


`new Color(255, 128, 0)` orange


`orange.darker()` (178, 89, 0)


`Color.RED.darker()` (178, 0, 0)


`Color.GREEN.darker()` (0, 178, 0)


`new Color(100, 50, 0).brighter()` (142, 71, 0)

```
Color orange = new Color(255, 128, 0);
System.out.println(orange);
System.out.println(orange.getGreen());
System.out.println(Color.RED.darker());
```

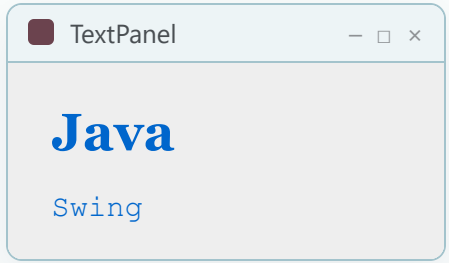
```
java.awt.Color[r=255,g=128,b=0]
128
java.awt.Color[r=178,g=0,b=0]
```

The Font Class

Sets the typeface, style and size of text: `new Font(name, style, size)`.

Parameter	Values
name	`"Serif"`, `"SansSerif"`, `"Monospaced"`, `"Dialog"` (logical names available on every system) or a real font such as `"Arial"`
style	`Font.PLAIN` (0), `Font.BOLD` (1), `Font.ITALIC` (2), or both: `Font.BOLD | Font.ITALIC`
size	Integer point size, e.g. 24

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(new Color(0, 102, 204));
    g.setFont(new Font("Serif", Font.BOLD, 24));
    g.drawString("Java", 20, 40);
    g.setFont(new Font("Monospaced", Font.PLAIN, 14));
    g.drawString("Swing", 20, 70);
}
```

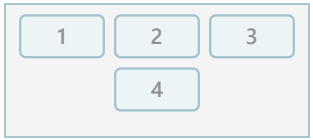


Components such as `JLabel` use the same object: `label.setFont(f)`. To derive from an existing font: `f.deriveFont(Font.ITALIC, 18f)`.

Layout Managers

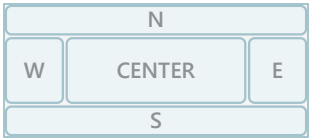
A **layout manager** computes the **size and position** of the components in a container by rule: `panel.setLayout(new GridLayout(2, 3));`. We use one instead of hard-coding coordinates because the layout is recomputed automatically when the window grows or the font or screen resolution changes.

FlowLayout



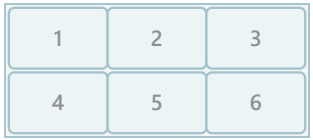
Left to right; wraps when a row fills. **JPanel default**.

BorderLayout



Five regions. **Default for a JFrame's content pane**.

GridLayout



A grid of equal cells: `new GridLayout(2, 3)`.

BoxLayout



A single row or column: `BoxLayout.Y_AXIS`.

The more flexible but complex `GridBagLayout` builds a grid of cells of different sizes. Nesting panels, each with a simple manager, is usually enough.

How FlowLayout Fills Its Rows

```
▶ int[] w = {90, 60, 110, 70, 80};
▶ JPanel panel = new JPanel();   // FlowLayout(CENTER, 5, 5)
▶ panel.setSize(260, 120);
4 for (int i = 0; i < w.length; i++) {
5     JButton b = new JButton("B" + i);
6     b.setPreferredSize(new Dimension(w[i], 30));
7     panel.add(b);
8 }
9 panel.doLayout();              // compute positions
10 for (Component c : panel.getComponents())
11     System.out.println(c.getX() + ", " + c.getY());
```



`JPanel`'s default manager is `FlowLayout`: centered, with a 5-pixel gap between components. The dashed lines are the margins.

Step 1 / 9

VARIABLES

`w = [90, 60, 110, 70, 80]`

`panel = 260×120`

OUTPUT

How FlowLayout Fills Its Rows

```
1  int[] w = {90, 60, 110, 70, 80};
2  JPanel panel = new JPanel();   // FlowLayout(CENTER, 5, 5)
3  panel.setSize(260, 120);
4  for (int i = 0; i < w.length; i++) {
▶      JButton b = new JButton("B" + i);
▶      b.setPreferredSize(new Dimension(w[i], 30));
▶      panel.add(b);
8  }
9  panel.doLayout();              // compute positions
10 for (Component c : panel.getComponents())
11     System.out.println(c.getX() + ", " + c.getY());
```



`add` only appends the button to a list; its position is still (0, 0), its size 0 × 0.

Step 2 / 9

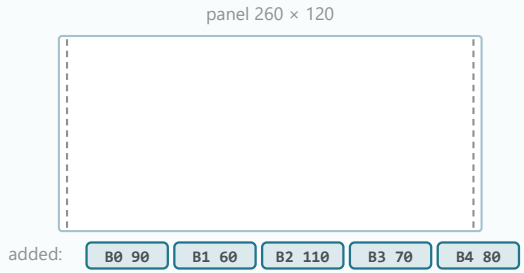
VARIABLES

`i = 0` `b = B0 (90×30)`

OUTPUT

How FlowLayout Fills Its Rows

```
1  int[] w = {90, 60, 110, 70, 80};
2  JPanel panel = new JPanel();   // FlowLayout(CENTER, 5,
   5)
3  panel.setSize(260, 120);
▶  for (int i = 0; i < w.length; i++) {
▶      JButton b = new JButton("B" + i);
▶      b.setPreferredSize(new Dimension(w[i], 30));
▶      panel.add(b);
8  }
9  panel.doLayout();              // compute positions
10 for (Component c : panel.getComponents())
11     System.out.println(c.getX() + ", " + c.getY());
```



Loop done: five buttons added, none placed. Placement happens only when the layout is computed.

Step 3 / 9

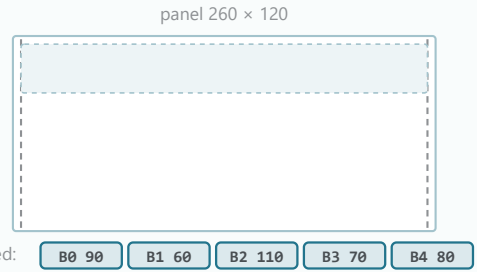
VARIABLES

`i = 4` `b = B4 (80×30)`

OUTPUT

How FlowLayout Fills Its Rows

```
1  int[] w = {90, 60, 110, 70, 80};
2  JPanel panel = new JPanel();   // FlowLayout(CENTER, 5, 5)
3  panel.setSize(260, 120);
4  for (int i = 0; i < w.length; i++) {
5      JButton b = new JButton("B" + i);
6      b.setPreferredSize(new Dimension(w[i], 30));
7      panel.add(b);
8  }
▶  panel.doLayout();              // compute positions
10 for (Component c : panel.getComponents())
11     System.out.println(c.getX() + ", " + c.getY());
```



`doLayout()` : usable width 260 − 2 × 5 = 250. B0 and B1 fit: 155 pixels.

Step 4 / 9

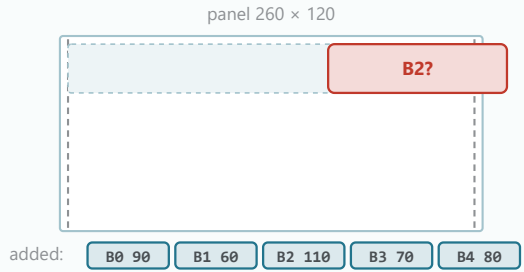
VARIABLES

`row = 1` `width = 90+5+60=155`

OUTPUT

How FlowLayout Fills Its Rows

```
1  int[] w = {90, 60, 110, 70, 80};
2  JPanel panel = new JPanel();   // FlowLayout(CENTER, 5,
   5)
3  panel.setSize(260, 120);
4  for (int i = 0; i < w.length; i++) {
5      JButton b = new JButton("B" + i);
6      b.setPreferredSize(new Dimension(w[i], 30));
7      panel.add(b);
8  }
▶  panel.doLayout();              // compute positions
10 for (Component c : panel.getComponents())
11     System.out.println(c.getX() + ", " + c.getY());
```



Adding B2 would make the row 270 pixels: it doesn't fit, so it goes to **the next row**.

Step 5 / 9

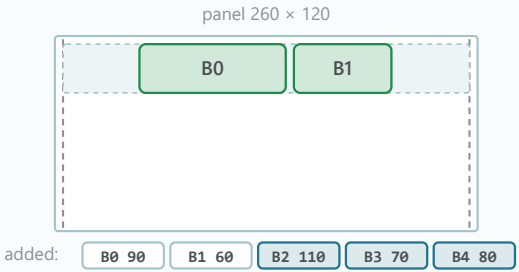
VARIABLES

`row = 1` `try = 155+5+110 = 270 > 250`

OUTPUT

How FlowLayout Fills Its Rows

```
1  int[] w = {90, 60, 110, 70, 80};
2  JPanel panel = new JPanel();   // FlowLayout(CENTER, 5,
   5)
3  panel.setSize(260, 120);
4  for (int i = 0; i < w.length; i++) {
5      JButton b = new JButton("B" + i);
6      b.setPreferredSize(new Dimension(w[i], 30));
7      panel.add(b);
8  }
▶  panel.doLayout();              // compute positions
10 for (Component c : panel.getComponents())
11     System.out.println(c.getX() + ", " + c.getY());
```



Row 1 is centered: B0 x = 52, B1 x = 52 + 90 + 5 = 147, y = 5.

Step 6 / 9

VARIABLES

`row = 1` `x0 = 5+(250−155)/2 = 52`

OUTPUT

How FlowLayout Fills Its Rows

```
1  int[] w = {90, 60, 110, 70, 80};
2  JPanel panel = new JPanel();   // FlowLayout(CENTER, 5,
   5)
3  panel.setSize(260, 120);
4  for (int i = 0; i < w.length; i++) {
5      JButton b = new JButton("B" + i);
6      b.setPreferredSize(new Dimension(w[i], 30));
7      panel.add(b);
8  }
▶  panel.doLayout();              // compute positions
10 for (Component c : panel.getComponents())
11     System.out.println(c.getX() + ", " + c.getY());
```



Row 2: B2 and B3 (185 pixels); adding B4 would make 270. Centering: x0 = 5 + (250 − 185)/2 = 37.

Step 7 / 9

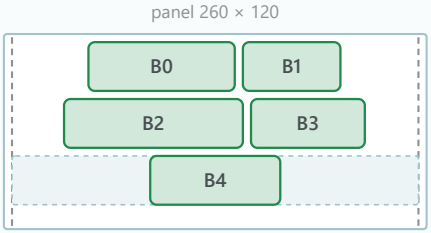
VARIABLES

`row = 2` `width = 110+5+70 = 185`

OUTPUT

How FlowLayout Fills Its Rows

```
1  int[] w = {90, 60, 110, 70, 80};
2  JPanel panel = new JPanel();   // FlowLayout(CENTER, 5,
   5)
3  panel.setSize(260, 120);
4  for (int i = 0; i < w.length; i++) {
5      JButton b = new JButton("B" + i);
6      b.setPreferredSize(new Dimension(w[i], 30));
7      panel.add(b);
8  }
▶  panel.doLayout();              // compute positions
10 for (Component c : panel.getComponents())
11     System.out.println(c.getX() + ", " + c.getY());
```



B4 sits alone on row 3; y = 5 + 30 + 5 + 30 + 5 = 75.

Step 8 / 9

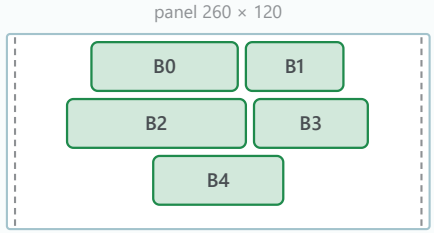
VARIABLES

`row = 3` `x0 = 5+(250−80)/2 = 90`

OUTPUT

How FlowLayout Fills Its Rows

```
1 int[] w = {90, 60, 110, 70, 80};
2 JPanel panel = new JPanel();   // FlowLayout(CENTER, 5,
  5)
3 panel.setSize(260, 120);
4 for (int i = 0; i < w.length; i++) {
5     JButton b = new JButton("B" + i);
6     b.setPreferredSize(new Dimension(w[i], 30));
7     panel.add(b);
8 }
9 panel.doLayout();              // compute positions
▶ for (Component c : panel.getComponents())
▶     System.out.println(c.getX() + ", " + c.getY());
```



The computed bounds are printed. If the panel widens, the rows are recomputed and the buttons move.

Step 9 / 9

VARIABLES

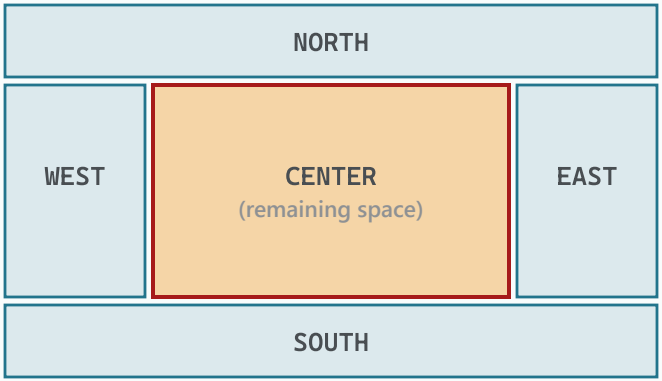
`c = B0 … B4`

OUTPUT

```
52, 5
147, 5
37, 40
152, 40
90, 75
```

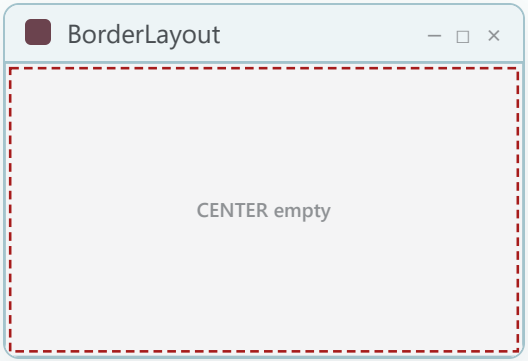
The BorderLayout Class

- Places components in **five regions**: `NORTH`, `SOUTH`, `WEST`, `EAST`, `CENTER`.
- Good for components of different priority: toolbar on top, status bar at the bottom, main content in the middle.
- A component added without a region goes to the **center**: `add(b)` = `add(b, BorderLayout.CENTER)`.
- Each region holds **one** component; adding a second to the same region replaces the first.
- North and south take the full width, west and east the remaining height; **the center fills everything left**.
- `setHgap()` and `setVgap()` set the horizontal and vertical gaps between regions (pixels).



BorderLayout: Filling the Regions

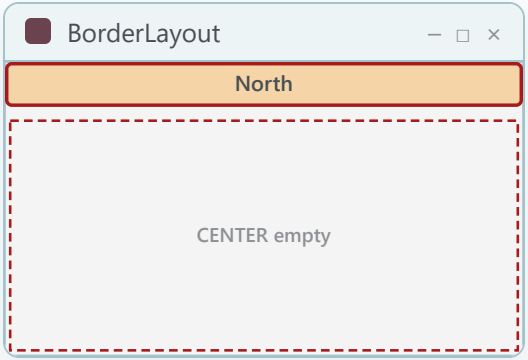
```
JPanel panel = new JPanel();
BorderLayout layout = new BorderLayout();
layout.setHgap(5);
layout.setVgap(5);
panel.setLayout(layout);
panel.add(new JButton("North"), BorderLayout.NORTH);
panel.add(new JButton("South"), BorderLayout.SOUTH);
panel.add(new JButton("West"), BorderLayout.WEST);
panel.add(new JButton("East"), BorderLayout.EAST);
panel.add(new JButton("Center"), BorderLayout.CENTER);
```



The panel will use `BorderLayout` with 5-pixel gaps between regions. No components yet.

BorderLayout: Filling the Regions

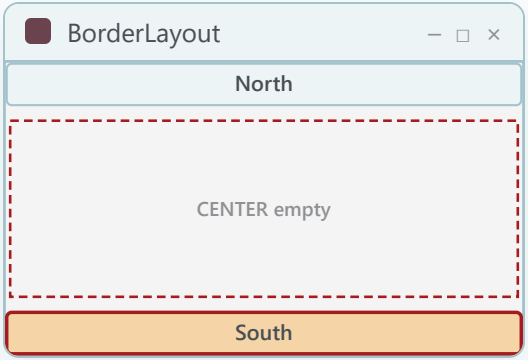
```
1  JPanel panel = new JPanel();
2  BorderLayout layout = new BorderLayout();
3  layout.setHgap(5);
4  layout.setVgap(5);
5  panel.setLayout(layout);
▶  panel.add(new JButton("North"), BorderLayout.NORTH);
7  panel.add(new JButton("South"), BorderLayout.SOUTH);
8  panel.add(new JButton("West"), BorderLayout.WEST);
9  panel.add(new JButton("East"), BorderLayout.EAST);
10 panel.add(new JButton("Center"), BorderLayout.CENTER);
```



The North button takes the **full width**; its height is the button's preferred height.

BorderLayout: Filling the Regions

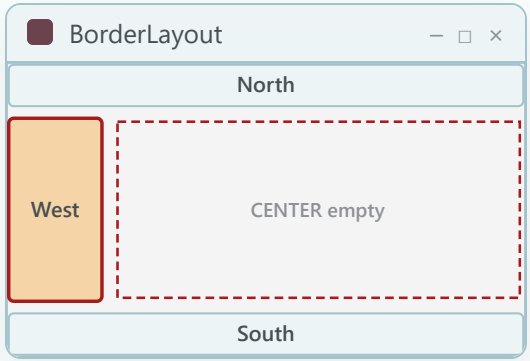
```
1  JPanel panel = new JPanel();
2  BorderLayout layout = new BorderLayout();
3  layout.setHgap(5);
4  layout.setVgap(5);
5  panel.setLayout(layout);
6  panel.add(new JButton("North"), BorderLayout.NORTH);
▶  panel.add(new JButton("South"), BorderLayout.SOUTH);
8  panel.add(new JButton("West"), BorderLayout.WEST);
9  panel.add(new JButton("East"), BorderLayout.EAST);
10 panel.add(new JButton("Center"), BorderLayout.CENTER);
```



South also spans the full width, at the bottom.

BorderLayout: Filling the Regions

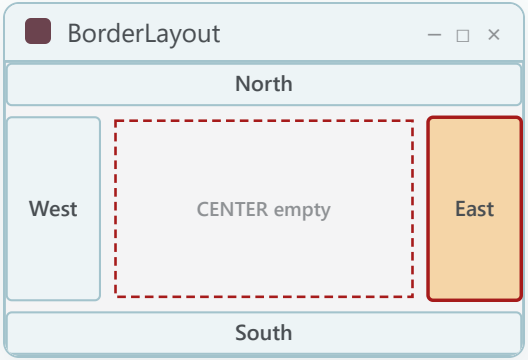
```
1  JPanel panel = new JPanel();
2  BorderLayout layout = new BorderLayout();
3  layout.setHgap(5);
4  layout.setVgap(5);
5  panel.setLayout(layout);
6  panel.add(new JButton("North"), BorderLayout.NORTH);
7  panel.add(new JButton("South"), BorderLayout.SOUTH);
▶  panel.add(new JButton("West"), BorderLayout.WEST);
9  panel.add(new JButton("East"), BorderLayout.EAST);
10 panel.add(new JButton("Center"), BorderLayout.CENTER);
```



West takes the height between north and south; its width fits its text.

BorderLayout: Filling the Regions

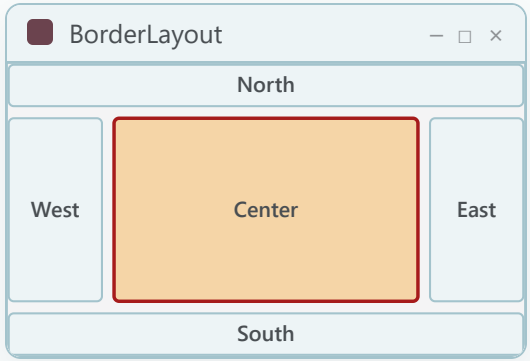
```
1  JPanel panel = new JPanel();
2  BorderLayout layout = new BorderLayout();
3  layout.setHgap(5);
4  layout.setVgap(5);
5  panel.setLayout(layout);
6  panel.add(new JButton("North"), BorderLayout.NORTH);
7  panel.add(new JButton("South"), BorderLayout.SOUTH);
8  panel.add(new JButton("West"), BorderLayout.WEST);
▶  panel.add(new JButton("East"), BorderLayout.EAST);
10 panel.add(new JButton("Center"), BorderLayout.CENTER);
```



East goes to the right edge.

BorderLayout: Filling the Regions

```
1 JPanel panel = new JPanel();
2 BorderLayout layout = new BorderLayout();
3 layout.setHgap(5);
4 layout.setVgap(5);
5 panel.setLayout(layout);
6 panel.add(new JButton("North"), BorderLayout.NORTH);
7 panel.add(new JButton("South"), BorderLayout.SOUTH);
8 panel.add(new JButton("West"), BorderLayout.WEST);
9 panel.add(new JButton("East"), BorderLayout.EAST);
▶ panel.add(new JButton("Center"), BorderLayout.CENTER);
```



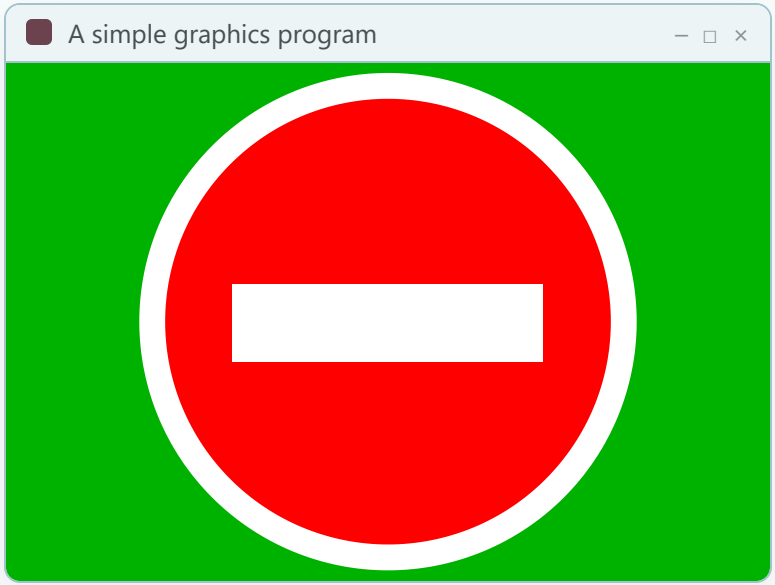
Center fills **all the remaining space**. When the window grows, the center grows most.

No Entry Sign

Goal: a panel that draws the “no entry” traffic sign. The shape is four layers painted on top of each other:

1. Green background (`setBackground`)
2. Large white circle (outer ring)
3. Slightly smaller red circle
4. White horizontal bar in the middle

Every measurement is **computed** from `getWidth()` and `getHeight()` : when the window grows, the sign grows too and stays centered.



The paintComponent Code

```
public class NoEntrySign extends JPanel {
    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        var c = new Point(getWidth() / 2, getHeight() / 2); // center
        var r = Math.min(getWidth() / 2, getHeight() / 2) - 5; // radius
        var r2 = (int) (r * 0.9);   // inner radius
        var bw = (int) (r2 * 1.4);  // bar width
        var bh = (int) (r2 * 0.35); // bar height
        g.setColor(Color.WHITE);
        g.fillOval(c.x - r, c.y - r, 2 * r, 2 * r);
        g.setColor(Color.RED);
        g.fillOval(c.x - r2, c.y - r2, r2 * 2, r2 * 2);
        g.setColor(Color.WHITE);
        g.fillRect(c.x - bw / 2, c.y - bh / 2, bw, bh);
    }
}
```

`var` (Java 10+): the compiler infers the type from the right-hand side; here `c` is a `Point`, the others `int`. Classic form: `Point c = new Point(…); int r = …;`. The `(int)` cast **drops** the fractional part. Imports needed: `java.awt.*` and `javax.swing.*`.

Drawing the Sign: Tracing `paintComponent`

```
▶ super.paintComponent(g);
2  var c = new Point(getWidth() / 2, getHeight() / 2);
3  var r = Math.min(getWidth() / 2, getHeight() / 2) - 5;
4  var r2 = (int) (r * 0.9);
5  var bw = (int) (r2 * 1.4);
6  var bh = (int) (r2 * 0.35);
7  g.setColor(Color.WHITE);
8  g.fillOval(c.x - r, c.y - r, 2 * r, 2 * r);
9  g.setColor(Color.RED);
10 g.fillOval(c.x - r2, c.y - r2, r2 * 2, r2 * 2);
11 g.setColor(Color.WHITE);
12 g.fillRect(c.x - bw / 2, c.y - bh / 2, bw, bh);
```



The panel is 384 × 261 pixels. `super.paintComponent` paints the background color (green).

Step 1 / 9

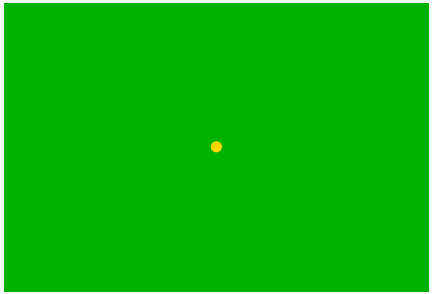
VARIABLES

`getWidth() = 384`

`getHeight() = 261`

Drawing the Sign: Tracing `paintComponent`

```
1  super.paintComponent(g);
2  var c = new Point(getWidth() / 2, getHeight() / 2);
3  var r = Math.min(getWidth() / 2, getHeight() / 2) - 5;
4  var r2 = (int) (r * 0.9);
5  var bw = (int) (r2 * 1.4);
6  var bh = (int) (r2 * 0.35);
7  g.setColor(Color.WHITE);
8  g.fillOval(c.x - r, c.y - r, 2 * r, 2 * r);
9  g.setColor(Color.RED);
10 g.fillOval(c.x - r2, c.y - r2, r2 * 2, r2 * 2);
11 g.setColor(Color.WHITE);
12 g.fillRect(c.x - bw / 2, c.y - bh / 2, bw, bh);
```



Center: (384 / 2, 261 / 2) = (192, 130). Integer division: 130.5 → 130.

Step 2 / 9

VARIABLES

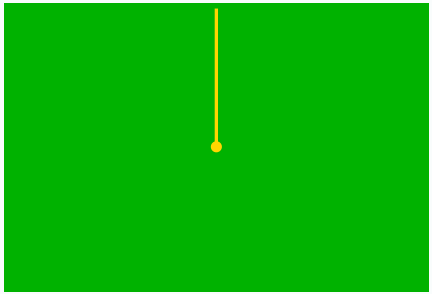
`getWidth() = 384`

`getHeight() = 261`

`c = (192, 130)`

Drawing the Sign: Tracing `paintComponent`

```
1  super.paintComponent(g);
2  var c = new Point(getWidth() / 2, getHeight() / 2);
▶  var r = Math.min(getWidth() / 2, getHeight() / 2) - 5;
4  var r2 = (int) (r * 0.9);
5  var bw = (int) (r2 * 1.4);
6  var bh = (int) (r2 * 0.35);
7  g.setColor(Color.WHITE);
8  g.fillOval(c.x - r, c.y - r, 2 * r, 2 * r);
9  g.setColor(Color.RED);
10 g.fillOval(c.x - r2, c.y - r2, r2 * 2, r2 * 2);
11 g.setColor(Color.WHITE);
12 g.fillRect(c.x - bw / 2, c.y - bh / 2, bw, bh);
```



Radius: min(192, 130) − 5 = 125. Based on the shorter side, with a 5-pixel margin.

Step 3 / 9

VARIABLES

`getWidth()` = **384**

`getHeight()` = **261**

`c` = **(192, 130)**

`r` = **125**

Drawing the Sign: Tracing `paintComponent`

```
1  super.paintComponent(g);
2  var c = new Point(getWidth() / 2, getHeight() / 2);
3  var r = Math.min(getWidth() / 2, getHeight() / 2) - 5;
▶  var r2 = (int) (r * 0.9);
5  var bw = (int) (r2 * 1.4);
6  var bh = (int) (r2 * 0.35);
7  g.setColor(Color.WHITE);
8  g.fillOval(c.x - r, c.y - r, 2 * r, 2 * r);
9  g.setColor(Color.RED);
10 g.fillOval(c.x - r2, c.y - r2, r2 * 2, r2 * 2);
11 g.setColor(Color.WHITE);
12 g.fillRect(c.x - bw / 2, c.y - bh / 2, bw, bh);
```



Inner radius: (int) 112.5 = 112.

Step 4 / 9

VARIABLES

`getWidth() = 384`

`getHeight() = 261` `c = (192, 130)`

`r = 125` `r2 = 112`

Drawing the Sign: Tracing `paintComponent`

```
1  super.paintComponent(g);
2  var c = new Point(getWidth() / 2, getHeight() / 2);
3  var r = Math.min(getWidth() / 2, getHeight() / 2) - 5;
4  var r2 = (int) (r * 0.9);
▶  var bw = (int) (r2 * 1.4);
6  var bh = (int) (r2 * 0.35);
7  g.setColor(Color.WHITE);
8  g.fillOval(c.x - r, c.y - r, 2 * r, 2 * r);
9  g.setColor(Color.RED);
10 g.fillOval(c.x - r2, c.y - r2, r2 * 2, r2 * 2);
11 g.setColor(Color.WHITE);
12 g.fillRect(c.x - bw / 2, c.y - bh / 2, bw, bh);
```



Bar width: (int) (112 × 1.4) = (int) 156.8 = 156.

Step 5 / 9

VARIABLES

`getWidth() = 384`

`getHeight() = 261` `c = (192, 130)`

`r = 125` `r2 = 112` `bw = 156`

Drawing the Sign: Tracing `paintComponent`

```
1  super.paintComponent(g);
2  var c = new Point(getWidth() / 2, getHeight() / 2);
3  var r = Math.min(getWidth() / 2, getHeight() / 2) - 5;
4  var r2 = (int) (r * 0.9);
5  var bw = (int) (r2 * 1.4);
▶  var bh = (int) (r2 * 0.35);
7  g.setColor(Color.WHITE);
8  g.fillOval(c.x - r, c.y - r, 2 * r, 2 * r);
9  g.setColor(Color.RED);
10 g.fillOval(c.x - r2, c.y - r2, r2 * 2, r2 * 2);
11 g.setColor(Color.WHITE);
12 g.fillRect(c.x - bw / 2, c.y - bh / 2, bw, bh);
```



Bar height: (int) 39.2 = 39.

Step 6 / 9

VARIABLES

`getWidth() = 384`

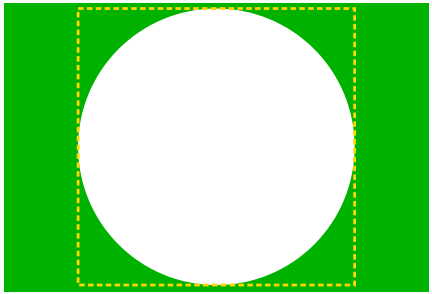
`getHeight() = 261` `c = (192, 130)`

`r = 125` `r2 = 112` `bw = 156`

`bh = 39`

Drawing the Sign: Tracing `paintComponent`

```
1  super.paintComponent(g);
2  var c = new Point(getWidth() / 2, getHeight() / 2);
3  var r = Math.min(getWidth() / 2, getHeight() / 2) - 5;
4  var r2 = (int) (r * 0.9);
5  var bw = (int) (r2 * 1.4);
6  var bh = (int) (r2 * 0.35);
▶  g.setColor(Color.WHITE);
▶  g.fillOval(c.x - r, c.y - r, 2 * r, 2 * r);
9  g.setColor(Color.RED);
10 g.fillOval(c.x - r2, c.y - r2, r2 * 2, r2 * 2);
11 g.setColor(Color.WHITE);
12 g.fillRect(c.x - bw / 2, c.y - bh / 2, bw, bh);
```



`fillOval(67, 5, 250, 250)` : the white circle; its bounding square is dashed.

Step 7 / 9

VARIABLES

`getWidth()` = **384**

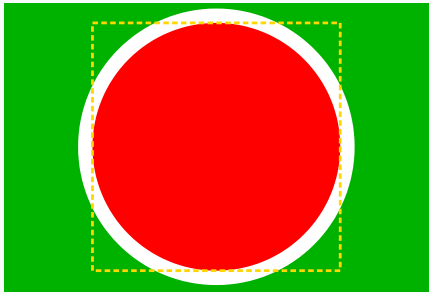
`getHeight()` = **261** `c` = **(192, 130)**

`r` = **125** `r2` = **112** `bw` = **156**

`bh` = **39**

Drawing the Sign: Tracing `paintComponent`

```
1  super.paintComponent(g);
2  var c = new Point(getWidth() / 2, getHeight() / 2);
3  var r = Math.min(getWidth() / 2, getHeight() / 2) - 5;
4  var r2 = (int) (r * 0.9);
5  var bw = (int) (r2 * 1.4);
6  var bh = (int) (r2 * 0.35);
7  g.setColor(Color.WHITE);
8  g.fillOval(c.x - r, c.y - r, 2 * r, 2 * r);
▶  g.setColor(Color.RED);
▶  g.fillOval(c.x - r2, c.y - r2, r2 * 2, r2 * 2);
11 g.setColor(Color.WHITE);
12 g.fillRect(c.x - bw / 2, c.y - bh / 2, bw, bh);
```



`fillOval(80, 18, 224, 224)` : the red circle is painted over the white one, leaving a 13-pixel white ring.

Step 8 / 9

VARIABLES

`getWidth()` = **384**

`getHeight()` = **261**

`c` = **(192, 130)**

`r` = **125**

`r2` = **112**

`bw` = **156**

`bh` = **39**

Drawing the Sign: Tracing `paintComponent`

```
1  super.paintComponent(g);
2  var c = new Point(getWidth() / 2, getHeight() / 2);
3  var r = Math.min(getWidth() / 2, getHeight() / 2) - 5;
4  var r2 = (int) (r * 0.9);
5  var bw = (int) (r2 * 1.4);
6  var bh = (int) (r2 * 0.35);
7  g.setColor(Color.WHITE);
8  g.fillOval(c.x - r, c.y - r, 2 * r, 2 * r);
9  g.setColor(Color.RED);
10 g.fillOval(c.x - r2, c.y - r2, r2 * 2, r2 * 2);
▶  g.setColor(Color.WHITE);
▶  g.fillRect(c.x - bw / 2, c.y - bh / 2, bw, bh);
```



`fillRect(114, 111, 156, 39)` : the white bar goes on top. The sign is done.

Step 9 / 9

VARIABLES

`getWidth()` = **384**

`getHeight()` = **261**

`c` = **(192, 130)**

`r` = **125**

`r2` = **112**

`bw` = **156**

`bh` = **39**

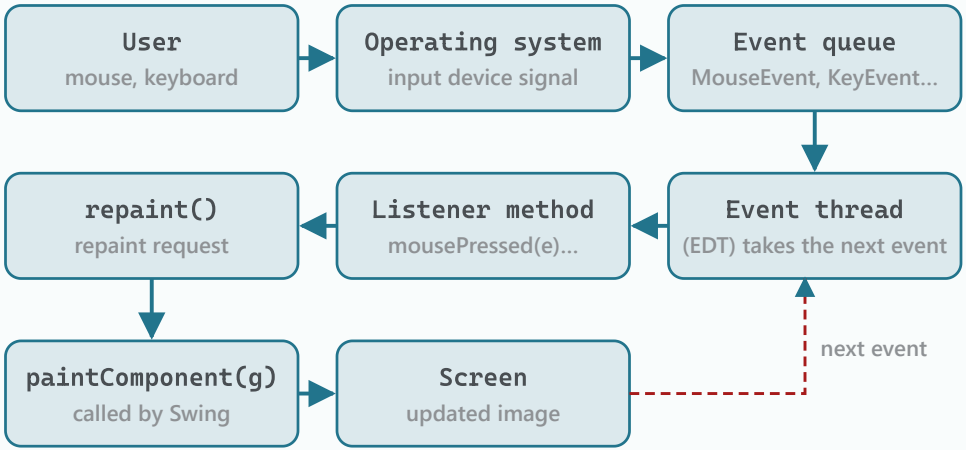
main and the Window

```
public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> {
        var panel = new NoEntrySign();
        panel.setBackground(Color.GREEN.darker());
        var frame = new JFrame("A simple graphics program");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(panel, BorderLayout.CENTER);
        frame.setVisible(true);
    });
}
```

- `SwingUtilities.invokeLater(() -> { … })` builds the GUI on the **event thread**.
- `() -> { … }` is a **lambda expression** (Java 8+): “a piece of code to run later”.
- `Color.GREEN.darker()` = (0, 178, 0): a bit darker than pure green.
- The panel is added to the **center** of the content pane; as the only component, it fills the whole area.
- In a 400 × 300 window, minus the borders and title bar, the panel is about 384 × 261 (varies by OS).

Interactive Computer Graphics

- **Computer graphics** is the creation and processing of visual content. In **interactive** graphics the user changes the image with input devices such as the mouse and keyboard: drawing programs, games, maps.
- Graphics libraries draw, color and transform graphical objects; interaction happens through the **event loop** below.



A listener doesn't paint the screen directly: it updates the data and **requests a repaint** with `repaint()`. Swing calls `paintComponent` at the right moment.

Listener Interface and Adapter

INTERFACE: ALL FIVE METHODS MUST BE WRITTEN

```
class ClickListener implements MouseListener {
    public void mouseClicked(MouseEvent e) {
        System.out.println("click");
    }
    public void mousePressed(MouseEvent e) { }
    public void mouseReleased(MouseEvent e) { }
    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
}
```

`MouseAdapter` : OVERRIDE ONLY WHAT YOU NEED

```
class ClickListener extends MouseAdapter {
    @Override
    public void mouseClicked(MouseEvent e) {
        System.out.println("click");
    }
}
// panel.addMouseListener(new ClickListener());
```

- `MouseAdapter` is a ready-made class that implements `MouseListener` (and `MouseMotionListener`, `MouseWheelListener`) with **empty bodies**. A subclass writes only the method it cares about.
- Motion events live in a separate interface: `MouseMotionListener` (`mouseDragged`, `mouseMoved`); its adapter is `MouseMotionAdapter`. Register with `addMouseListener(…)`, `addMouseMotionListener(…)`.
- The `new MouseAdapter() { … }` form is an **anonymous inner class**: a nameless, single-use subclass.

Mouse Event Methods

Method	Interface	When is it called?
`mousePressed(e)`	`MouseListener`	The moment a mouse button is pressed
`mouseReleased(e)`	`MouseListener`	When the button is released
`mouseClicked(e)`	`MouseListener`	Press and release **in the same spot** (pressed → released → clicked)
`mouseEntered(e)` / `mouseExited(e)`	`MouseListener`	The pointer enters / leaves the component
`mouseMoved(e)`	`MouseMotionListener`	Moving with no button pressed
`mouseDragged(e)`	`MouseMotionListener`	Moving with a button held down (dragging)
`mouseWheelMoved(e)`	`MouseWheelListener`	The wheel is turned



Every method takes a `MouseEvent e` : it tells **where, with which button and with which keys held** the event happened.

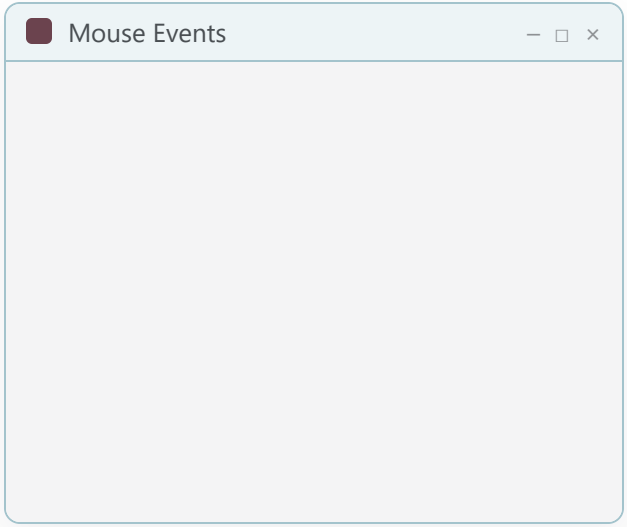
The MouseEvent Class

Method	Returns
`getX()`, `getY()`	Pointer position relative to the component
`getPoint()`	The same position as a `Point`
`getButton()`	`BUTTON1` left, `BUTTON2` middle, `BUTTON3` right
`getClickCount()`	Number of consecutive clicks (2: double-click)
`isShiftDown()`, `isCtrlDown()`, `isAltDown()`	Is the key held down right now?

```
panel.addMouseListener(new MouseAdapter() {
    @Override
    public void mousePressed(MouseEvent e) {
        String button = "left";
        if (e.getButton() == MouseEvent.BUTTON3) {
            button = "right";
        }
        System.out.println(button + " (" + e.getX()
                + ", " + e.getY() + ")"
                + (e.isShiftDown() ? " + Shift" : ""));
    }
});
```

Represents the state after a mouse event. Coordinates are relative **to the component, not the screen**: the panel's top-left corner is (0, 0).

Mouse Events Playground



Clicking and dragging on the panel fire events (also with Shift or Ctrl held).

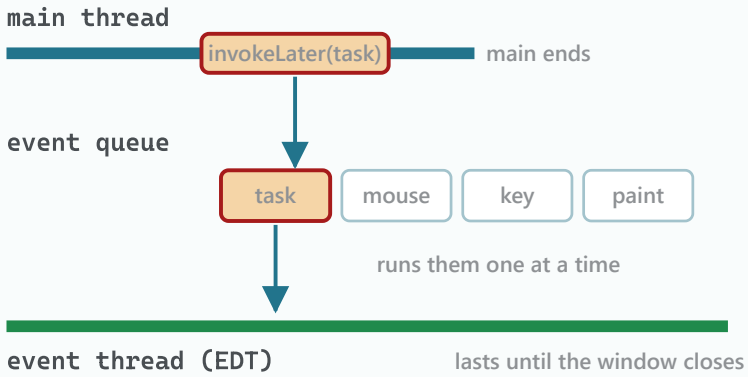
JAVA CALL AND `MouseEvent` VALUES

```
(no events yet)
```

Browser events are shown as their Java counterparts. A click fires three events: **pressed** → **released** → **clicked**; a drag fires many `mouseDragged` events. A right click is `BUTTON3`.

SwingUtilities and the Event Thread

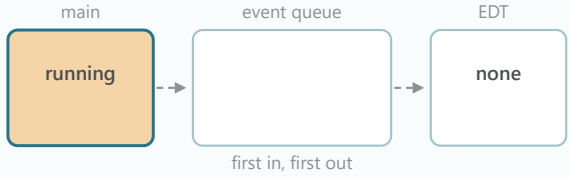
- `javax.swing.SwingUtilities` holds helper methods.
- Swing components are **not thread-safe**: if several threads touch them at once, they can become inconsistent.
- Rule: components are created and modified only on the **Event Dispatch Thread** (EDT). Listener methods already run on the EDT.
- Analogy: the kitchen has a single cook (the EDT); everyone else leaves an **order ticket** (a `Runnable`) instead of cooking.



Method	Behavior
`invokeLater(task)`	Queues the task and **returns immediately** (asynchronous). The standard way to start a GUI.
`invokeAndWait(task)`	**Waits until** the task finishes on the EDT (synchronous). Called from the EDT itself, it throws an `Error` (the EDT can't wait for itself).

The Event Queue: `invokeLater` and `invokeAndWait`

```
1  public static void main(String[] a) throws Exception {
▶    System.out.println("1 main");
3    SwingUtilities.invokeLater(
4        () -> System.out.println("3 task A"));
5    SwingUtilities.invokeLater(
6        () -> System.out.println("4 task B"));
7    System.out.println("2 main goes on");
8    SwingUtilities.invokeAndWait(() ->
   System.out.println(
9        "5 task C, EDT? " +
   SwingUtilities.isEventDispatchThread()));
10   System.out.println("6 main, EDT? "
11       + SwingUtilities.isEventDispatchThread());
12 }
```



The program starts on the `main` thread. The event thread (EDT) doesn't exist yet.

Step 1 / 9

VARIABLES

`thread = main`

OUTPUT

```
1 main
```

The Event Queue: `invokeLater` and `invokeAndWait`

```
1  public static void main(String[] a) throws Exception {
2    System.out.println("1 main");
▶    SwingUtilities.invokeLater(
▶        () -> System.out.println("3 task A"));
5    SwingUtilities.invokeLater(
6        () -> System.out.println("4 task B"));
7    System.out.println("2 main goes on");
8    SwingUtilities.invokeAndWait(() ->
   System.out.println(
9        "5 task C, EDT? " +
   SwingUtilities.isEventDispatchThread()));
10   System.out.println("6 main, EDT? "
11       + SwingUtilities.isEventDispatchThread());
12 }
```



`invokeLater` queues task A and **returns at once**. The EDT starts.

Step 2 / 9

VARIABLES

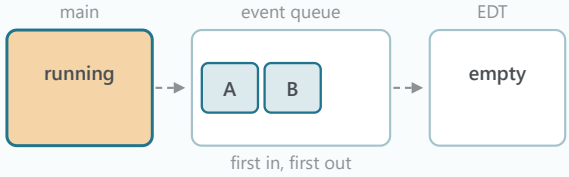
`queue = [A]`

OUTPUT

```
1 main
```

The Event Queue: `invokeLater` and `invokeAndWait`

```
1  public static void main(String[] a) throws Exception {
2    System.out.println("1 main");
3    SwingUtilities.invokeLater(
4        () -> System.out.println("3 task A"));
▶    SwingUtilities.invokeLater(
▶        () -> System.out.println("4 task B"));
7    System.out.println("2 main goes on");
8    SwingUtilities.invokeAndWait(() ->
   System.out.println(
9        "5 task C, EDT? " +
   SwingUtilities.isEventDispatchThread()));
10   System.out.println("6 main, EDT? "
11       + SwingUtilities.isEventDispatchThread());
12 }
```



Task B is queued too; `main` still doesn't wait.

Step 3 / 9

VARIABLES

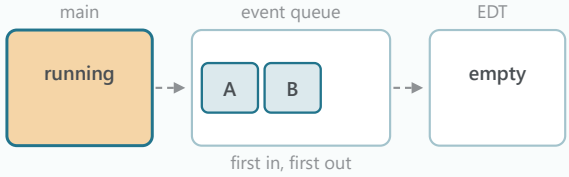
`queue = [A, B]`

OUTPUT

```
1 main
```

The Event Queue: `invokeLater` and `invokeAndWait`

```
1  public static void main(String[] a) throws Exception {
2    System.out.println("1 main");
3    SwingUtilities.invokeLater(
4        () -> System.out.println("3 task A"));
5    SwingUtilities.invokeLater(
6        () -> System.out.println("4 task B"));
▶    System.out.println("2 main goes on");
8    SwingUtilities.invokeAndWait(() ->
   System.out.println(
9        "5 task C, EDT? " +
   SwingUtilities.isEventDispatchThread()));
10   System.out.println("6 main, EDT? "
11       + SwingUtilities.isEventDispatchThread());
12 }
```



`main` carries on. (In this run the EDT hasn't picked up a task yet; the two threads are independent.)

Step 4 / 9

VARIABLES

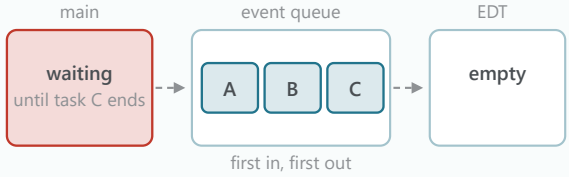
`queue = [A, B]`

OUTPUT

```
1 main
2 main goes on
```

The Event Queue: `invokeLater` and `invokeAndWait`

```
1  public static void main(String[] a) throws Exception {
2    System.out.println("1 main");
3    SwingUtilities.invokeLater(
4        () -> System.out.println("3 task A"));
5    SwingUtilities.invokeLater(
6        () -> System.out.println("4 task B"));
7    System.out.println("2 main goes on");
▶    SwingUtilities.invokeAndWait(() ->
     System.out.println(
▶        "5 task C, EDT? " +
     SwingUtilities.isEventDispatchThread()));
10   System.out.println("6 main, EDT? "
11       + SwingUtilities.isEventDispatchThread());
12 }
```



`invokeAndWait` queues task C and **blocks** `main`.

Step 5 / 9

VARIABLES

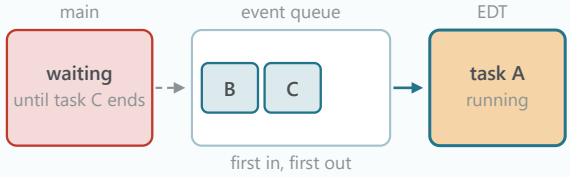
`queue = [A, B, C]`

OUTPUT

```
1 main
2 main goes on
```

The Event Queue: `invokeLater` and `invokeAndWait`

```
1  public static void main(String[] a) throws Exception {
2    System.out.println("1 main");
3    SwingUtilities.invokeLater(
▶        () -> System.out.println("3 task A"));
5    SwingUtilities.invokeLater(
6        () -> System.out.println("4 task B"));
7    System.out.println("2 main goes on");
8    SwingUtilities.invokeAndWait(() ->
   System.out.println(
9        "5 task C, EDT? " +
   SwingUtilities.isEventDispatchThread()));
10   System.out.println("6 main, EDT? "
11       + SwingUtilities.isEventDispatchThread());
12 }
```



The EDT runs task A from the head of the queue (the thread is named `AWT-EventQueue-0`).

Step 6 / 9

VARIABLES

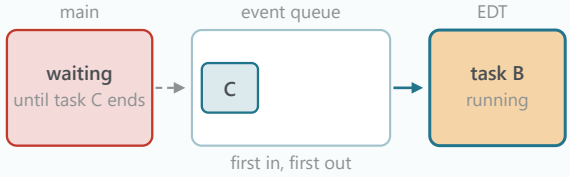
`queue = [B, C]`

OUTPUT

```
1 main
2 main goes on
3 task A
```

The Event Queue: `invokeLater` and `invokeAndWait`

```
1  public static void main(String[] a) throws Exception {
2    System.out.println("1 main");
3    SwingUtilities.invokeLater(
4        () -> System.out.println("3 task A"));
5    SwingUtilities.invokeLater(
▶        () -> System.out.println("4 task B"));
7    System.out.println("2 main goes on");
8    SwingUtilities.invokeAndWait(() ->
   System.out.println(
9        "5 task C, EDT? " +
   SwingUtilities.isEventDispatchThread()));
10   System.out.println("6 main, EDT? "
11       + SwingUtilities.isEventDispatchThread());
12 }
```



Order is preserved: task B runs after task A.

Step 7 / 9

VARIABLES

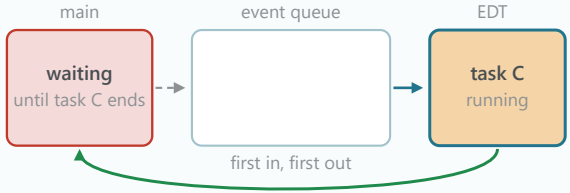
`queue = [C]`

OUTPUT

```
1 main
2 main goes on
3 task A
4 task B
```

The Event Queue: `invokeLater` and `invokeAndWait`

```
1  public static void main(String[] a) throws Exception {
2    System.out.println("1 main");
3    SwingUtilities.invokeLater(
4        () -> System.out.println("3 task A"));
5    SwingUtilities.invokeLater(
6        () -> System.out.println("4 task B"));
7    System.out.println("2 main goes on");
▶    SwingUtilities.invokeAndWait(() ->
     System.out.println(
▶        "5 task C, EDT? " +
     SwingUtilities.isEventDispatchThread()));
10   System.out.println("6 main, EDT? "
11       + SwingUtilities.isEventDispatchThread());
12 }
```



Task C also runs on the EDT (`true`). When it finishes, the waiting `main` wakes up.

Step 8 / 9

VARIABLES

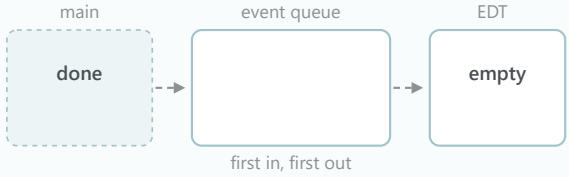
`queue = []`

OUTPUT

```
1 main
2 main goes on
3 task A
4 task B
5 task C, EDT? true
```

The Event Queue: `invokeLater` and `invokeAndWait`

```
1  public static void main(String[] a) throws Exception {
2    System.out.println("1 main");
3    SwingUtilities.invokeLater(
4        () -> System.out.println("3 task A"));
5    SwingUtilities.invokeLater(
6        () -> System.out.println("4 task B"));
7    System.out.println("2 main goes on");
8    SwingUtilities.invokeAndWait(() ->
   System.out.println(
9        "5 task C, EDT? " +
   SwingUtilities.isEventDispatchThread()));
▶    System.out.println("6 main, EDT? "
▶        + SwingUtilities.isEventDispatchThread());
12 }
```



`main` is not the EDT (`false`). That's why components are touched only from the EDT.

Step 9 / 9

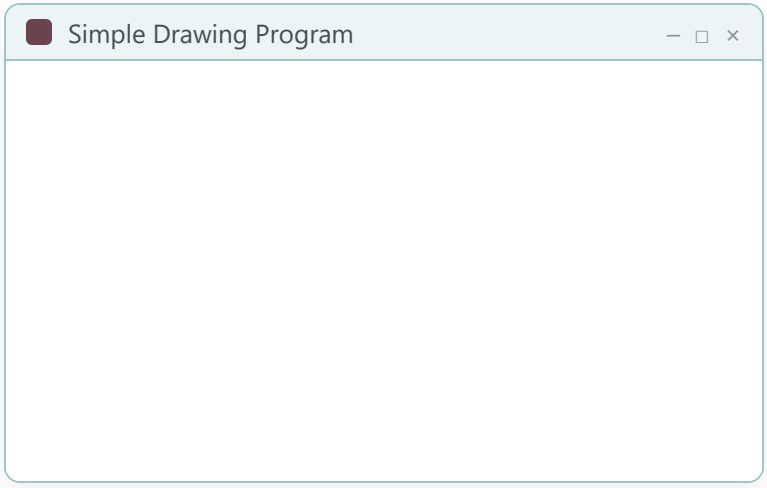
VARIABLES

`thread = main`

OUTPUT

```
1 main
2 main goes on
3 task A
4 task B
5 task C, EDT? true
6 main, EDT? false
```

A Simple Drawing Program



A program for drawing by holding down the mouse button and dragging.

- **On press** (`mousePressed`) a new curve starts.
- **While dragging** (`mouseDragged`) points are appended to the curve and a `repaint` is requested.
- **When painting** (`paintComponent`) the consecutive points of each curve are joined with lines.

The program stores the drawing not as an image but as **lists of points**; the screen is redrawn from this data every time.

Storing the Curves

```
public class BasitDrawPanel extends JPanel {
    private List<List<Point>> curves = new ArrayList<>();

    public BasitDrawPanel() {
        addMouseListener(new MouseAdapter() {
            @Override
            public void mousePressed(MouseEvent e) {
                var curve = new ArrayList<Point>();
                curve.add(new Point(e.getX(), e.getY()));
                curves.add(curve);
            }
        });
        addMouseMotionListener(new MouseMotionAdapter() {
            @Override
            public void mouseDragged(MouseEvent e) {
                curves.get(curves.size() - 1)
                      .add(new Point(e.getX(), e.getY()));
                repaint(0, 0, getWidth(), getHeight());
            }
        });
    }
}
```



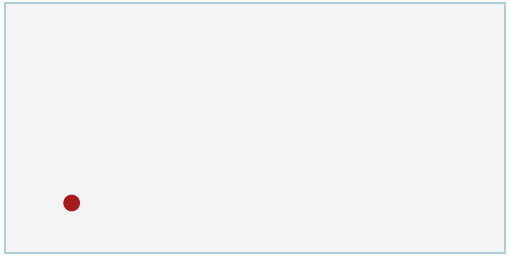
`curve 0:` `(40,120)` `(60,60)` `(80,120)`

`curve 1:` `(50,95)` `(70,95)`

- `List<List<Point>>` : a **list of lists**. The outer list holds the curves; each inner list holds one curve's points.
- The last curve: `curves.get(curves.size() - 1)` . The last index is `size() - 1` ; `get(size())` throws `IndexOutOfBoundsException` .

Tracing the Mouse Events

```
1  addMouseListener(new MouseAdapter() {
2      @Override
3      public void mousePressed(MouseEvent e) {
▶          var curve = new ArrayList<Point>();
▶          curve.add(new Point(e.getX(), e.getY()));
▶          curves.add(curve);
7      }
8  });
9  addMouseMotionListener(new MouseMotionAdapter() {
10     @Override
11     public void mouseDragged(MouseEvent e) {
12         curves.get(curves.size() - 1)
13                .add(new Point(e.getX(), e.getY()));
14         repaint(0, 0, getWidth(), getHeight());
15     }
16 });
```



curve 0: (40,120)

mousePressed: pressed at (40, 120): a new inner list is created, gets its first point and goes into the outer list.

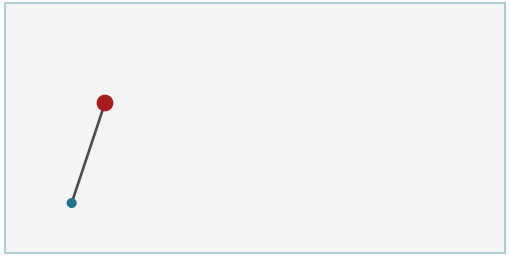
Step 1 / 6

VARIABLES

`e.getX()` = **40** `e.getY()` = **120** `curves.size()` = **1**

Tracing the Mouse Events

```
1  addMouseListener(new MouseAdapter() {
2      @Override
3      public void mousePressed(MouseEvent e) {
4          var curve = new ArrayList<Point>();
5          curve.add(new Point(e.getX(), e.getY()));
6          curves.add(curve);
7      }
8  });
9  addMouseMotionListener(new MouseMotionAdapter() {
10     @Override
11     public void mouseDragged(MouseEvent e) {
▶          curves.get(curves.size() - 1)
▶                .add(new Point(e.getX(), e.getY()));
▶          repaint(0, 0, getWidth(), getHeight());
15     }
16 });
```



curve 0: (40,120) (60,60)

mouseDragged: dragged to (60, 60): a point is added to the last curve; `repaint` requests a redraw and the line appears.

Step 2 / 6

VARIABLES

`e.getX() = 60` `e.getY() = 60` `curves.size() = 1`

Tracing the Mouse Events

```
1  addMouseListener(new MouseAdapter() {
2      @Override
3      public void mousePressed(MouseEvent e) {
4          var curve = new ArrayList<Point>();
5          curve.add(new Point(e.getX(), e.getY()));
6          curves.add(curve);
7      }
8  });
9  addMouseMotionListener(new MouseMotionAdapter() {
10     @Override
11     public void mouseDragged(MouseEvent e) {
▶          curves.get(curves.size() - 1)
▶                .add(new Point(e.getX(), e.getY()));
▶          repaint(0, 0, getWidth(), getHeight());
15     }
16 });
```



curve 0: (40,120) (60,60) (80,120)

mouseDragged: dragged to (80, 120): the same curve grows. In reality dozens of `mouseDragged` events arrive per second.

Step 3 / 6

VARIABLES

`e.getX() = 80` `e.getY() = 120` `curves.size() = 1`

Tracing the Mouse Events

```
▶ addMouseListener(new MouseAdapter() {
2     @Override
3     public void mousePressed(MouseEvent e) {
4         var curve = new ArrayList<Point>();
5         curve.add(new Point(e.getX(), e.getY()));
6         curves.add(curve);
7     }
8 });
9 addMouseMotionListener(new MouseMotionAdapter() {
10    @Override
11    public void mouseDragged(MouseEvent e) {
12        curves.get(curves.size() - 1)
13              .add(new Point(e.getX(), e.getY()));
14        repaint(0, 0, getWidth(), getHeight());
15    }
16 });
```



curve 0: `(40,120)` `(60,60)` `(80,120)`

mouseReleased: released. We didn't write `mouseReleased`, so `MouseAdapter`'s **empty body** runs and nothing happens.

Step 4 / 6

VARIABLES

`e.getX() = 80` `e.getY() = 120` `curves.size() = 1`

Tracing the Mouse Events

```
1  addMouseListener(new MouseAdapter() {
2      @Override
3      public void mousePressed(MouseEvent e) {
▶          var curve = new ArrayList<Point>();
▶          curve.add(new Point(e.getX(), e.getY()));
▶          curves.add(curve);
7      }
8  });
9  addMouseMotionListener(new MouseMotionAdapter() {
10     @Override
11     public void mouseDragged(MouseEvent e) {
12         curves.get(curves.size() - 1)
13                .add(new Point(e.getX(), e.getY()));
14         repaint(0, 0, getWidth(), getHeight());
15     }
16 });
```



mousePressed: pressed again at (50, 95): a **second** curve starts. The old curve stays as it is.

Step 5 / 6

VARIABLES

`e.getX()` = **50** `e.getY()` = **95**

`curves.size()` = **2**

Tracing the Mouse Events

```
1  addMouseListener(new MouseAdapter() {
2      @Override
3      public void mousePressed(MouseEvent e) {
4          var curve = new ArrayList<Point>();
5          curve.add(new Point(e.getX(), e.getY()));
6          curves.add(curve);
7      }
8  });
9  addMouseMotionListener(new MouseMotionAdapter() {
10     @Override
11     public void mouseDragged(MouseEvent e) {
▶          curves.get(curves.size() - 1)
▶                .add(new Point(e.getX(), e.getY()));
▶          repaint(0, 0, getWidth(), getHeight());
15     }
16 });
```



curve 0: (40,120) (60,60) (80,120)

curve 1: (50,95) (70,95)

mouseDragged: dragged to (70, 95): the point again goes to the **last** curve, curve 1.

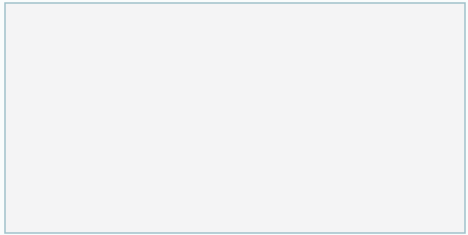
Step 6 / 6

VARIABLES

`e.getX()` = **70** `e.getY()` = **95** `curves.size()` = **2**

Drawing the Curves: `paintComponent`

```
1  @Override
2  public void paintComponent(Graphics g) {
▶      super.paintComponent(g);
4      for (var c : curves) {
5          var prev = c.get(0);
6          for (var p : c) {
7              g.drawLine(prev.x, prev.y, p.x, p.y);
8              prev = p;
9          }
10     }
11 }
```



The background is cleared; nothing of the previous drawing remains.

Step 1 / 8

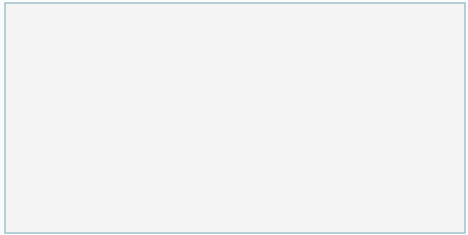
VARIABLES

—

OUTPUT

Drawing the Curves: `paintComponent`

```
1  @Override
2  public void paintComponent(Graphics g) {
3      super.paintComponent(g);
▶      for (var c : curves) {
▶          var prev = c.get(0);
6          for (var p : c) {
7              g.drawLine(prev.x, prev.y, p.x, p.y);
8              prev = p;
9          }
10     }
11 }
```



The first curve; `prev` is its first point.

Step 2 / 8

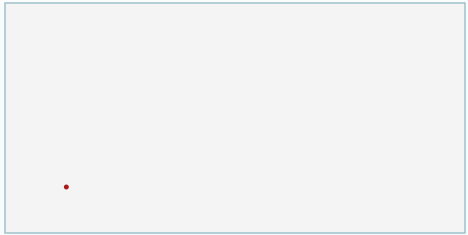
VARIABLES

`c = curve 0` `prev = (40,120)`

OUTPUT

Drawing the Curves: `paintComponent`

```
1  @Override
2  public void paintComponent(Graphics g) {
3      super.paintComponent(g);
4      for (var c : curves) {
5          var prev = c.get(0);
           for (var p : c) {
               g.drawLine(prev.x, prev.y, p.x, p.y);
               prev = p;
9          }
10     }
11 }
```



The inner loop starts at the **first point**: it joins itself, so a single pixel is drawn. Then `prev = p`.

Step 3 / 8

VARIABLES

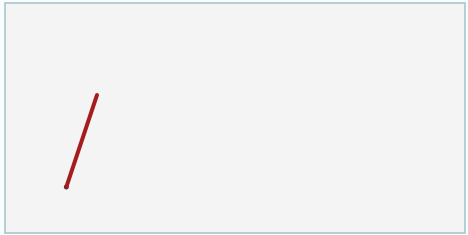
`c` = **curve 0** | `prev` = **(40,120)** | `p` = **(40,120)**

OUTPUT

```
drawLine(40, 120, 40, 120)
```

Drawing the Curves: `paintComponent`

```
1  @Override
2  public void paintComponent(Graphics g) {
3      super.paintComponent(g);
4      for (var c : curves) {
5          var prev = c.get(0);
▶          for (var p : c) {
▶              g.drawLine(prev.x, prev.y, p.x, p.y);
▶              prev = p;
9          }
10     }
11 }
```



The line (40,120) → (60,60).

Step 4 / 8

VARIABLES

`c` = **curve 0** | `prev` = **(40,120)** | `p` = **(60,60)**

OUTPUT

```
drawLine(40, 120, 40, 120)
drawLine(40, 120, 60, 60)
```

Drawing the Curves: `paintComponent`

```
1  @Override
2  public void paintComponent(Graphics g) {
3      super.paintComponent(g);
4      for (var c : curves) {
5          var prev = c.get(0);
▶          for (var p : c) {
▶              g.drawLine(prev.x, prev.y, p.x, p.y);
▶              prev = p;
9          }
10     }
11 }
```



(60,60) → (80,120). Curve 0 is done.

Step 5 / 8

VARIABLES

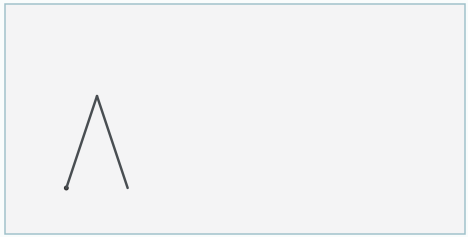
`c = curve 0` `prev = (60,60)` `p = (80,120)`

OUTPUT

```
drawLine(40, 120, 40, 120)
drawLine(40, 120, 60, 60)
drawLine(60, 60, 80, 120)
```

Drawing the Curves: `paintComponent`

```
1  @Override
2  public void paintComponent(Graphics g) {
3      super.paintComponent(g);
▶      for (var c : curves) {
▶          var prev = c.get(0);
6          for (var p : c) {
7              g.drawLine(prev.x, prev.y, p.x, p.y);
8              prev = p;
9          }
10     }
11 }
```



The outer loop moves to the second curve; `prev` is reset to its first point, so the two curves are **not joined**.

Step 6 / 8

VARIABLES

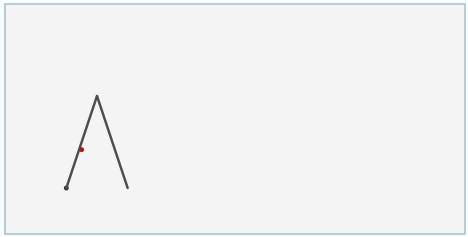
`c = curve 1` `prev = (50,95)`

OUTPUT

```
drawLine(40, 120, 40, 120)
drawLine(40, 120, 60, 60)
drawLine(60, 60, 80, 120)
```

Drawing the Curves: `paintComponent`

```
1  @Override
2  public void paintComponent(Graphics g) {
3      super.paintComponent(g);
4      for (var c : curves) {
5          var prev = c.get(0);
▶          for (var p : c) {
▶              g.drawLine(prev.x, prev.y, p.x, p.y);
▶              prev = p;
9          }
10     }
11 }
```



Again a single pixel.

Step 7 / 8

VARIABLES

`c = curve 1` `prev = (50,95)` `p = (50,95)`

OUTPUT

```
drawLine(40, 120, 40, 120)
drawLine(40, 120, 60, 60)
drawLine(60, 60, 80, 120)
drawLine(50, 95, 50, 95)
```

Drawing the Curves: `paintComponent`

```
1  @Override
2  public void paintComponent(Graphics g) {
3      super.paintComponent(g);
4      for (var c : curves) {
5          var prev = c.get(0);
▶          for (var p : c) {
▶              g.drawLine(prev.x, prev.y, p.x, p.y);
▶              prev = p;
9          }
10     }
11 }
```

VARIABLES

`c = curve 1` `prev = (50,95)` `p = (70,95)`

OUTPUT

```
drawLine(40, 120, 40, 120)
drawLine(40, 120, 60, 60)
drawLine(60, 60, 80, 120)
drawLine(50, 95, 50, 95)
drawLine(50, 95, 70, 95)
```



(50,95) → (70,95). Five `drawLine` calls complete the drawing.

Step 8 / 8

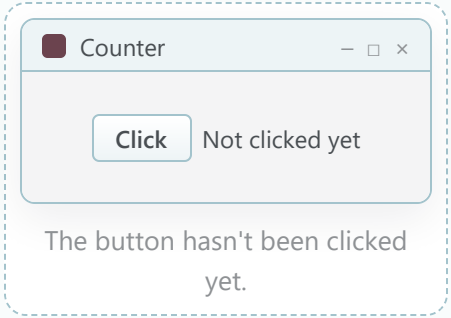
main and repaint

```
public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> {
        var frame = new JFrame("Simple Drawing Program");
        frame.getContentPane().add(new BasitDrawPanel(),
                BorderLayout.CENTER);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);
        frame.setVisible(true);
    });
}
```

- `repaint()` does **not draw right away**: it queues a "repaint" request. Back-to-back requests may be merged.
- `repaint(x, y, w, h)` asks to redraw only that rectangle. Swing **clips** everything else: `g` can't paint outside the region, so there is less work. `repaint(0, 0, getWidth(), getHeight())` = `repaint()`.
- Forget `super.paintComponent(g)` and the background isn't cleared; leftovers of the old image may remain.
- We **never call** `paintComponent` ourselves; we change the data and request `repaint()`.

The JButton Class

- Creates clickable **buttons**, customized with text, icons and event handlers.
- `setText()` sets the text, `setIcon()` the icon; `setEnabled(false)` disables it.
- `addActionListener()` adds a listener that runs on click (`ActionListener`, whose single method is `actionPerformed`). Adding listeners is the **Observer** pattern: the button is the subject, the listeners are observers.

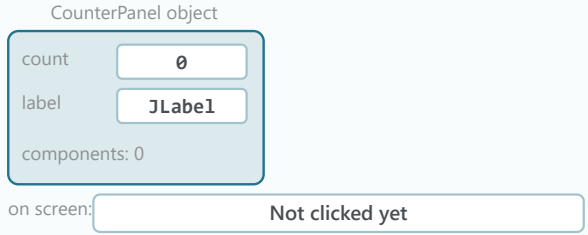



```
class CounterPanel extends JPanel {
    private int count = 0;
    private JLabel label = new JLabel("Not clicked yet");

    CounterPanel() {
        JButton button = new JButton("Click");
        button.addActionListener(e -> {
            count++;
            label.setText(count + " click(s)");
        });
        add(button);
        add(label);
    }
}
```

Counter Button: A Click, Step by Step

```
1  class CounterPanel extends JPanel {
▶      private int count = 0;
▶      private JLabel label = new JLabel("Not clicked
   yet");
4
5      CounterPanel() {
6          JButton button = new JButton("Click");
7          button.addActionListener(e -> {
8              count++;
9              label.setText(count + " click(s)");
10         });
11         add(button);
12         add(label);
13     }
14 }
```



As the object is created, its fields are initialized first: `count` is 0, the label is ready.

Step 1 / 8

VARIABLES

`count = 0` `label = JLabel`

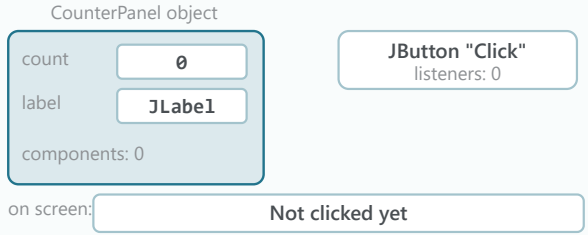
CALL STACK

`new CounterPanel()`

`main()`

Counter Button: A Click, Step by Step

```
1  class CounterPanel extends JPanel {
2      private int count = 0;
3      private JLabel label = new JLabel("Not clicked
   yet");
4
5      CounterPanel() {
▶          JButton button = new JButton("Click");
7          button.addActionListener(e -> {
8              count++;
9              label.setText(count + " click(s)");
10         });
11         add(button);
12         add(label);
13     }
14 }
```



The constructor creates a button; `button` is a local variable.

Step 2 / 8

VARIABLES

`button = JButton("Click")`

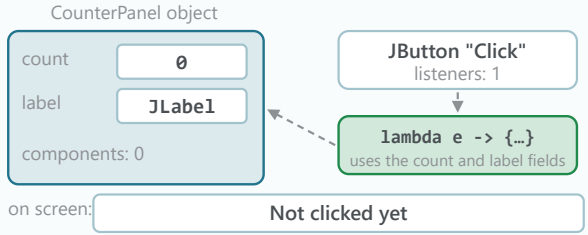
CALL STACK

`CounterPanel()`

`main()`

Counter Button: A Click, Step by Step

```
1  class CounterPanel extends JPanel {
2      private int count = 0;
3      private JLabel label = new JLabel("Not clicked
   yet");
4
5      CounterPanel() {
6          JButton button = new JButton("Click");
▶          button.addActionListener(e -> {
▶              count++;
▶              label.setText(count + " click(s)");
▶          });
11         add(button);
12         add(label);
13     }
14 }
```



The lambda is **not run**, only registered with the button. Its body will run on a click.

Step 3 / 8

VARIABLES

`listeners = 1`

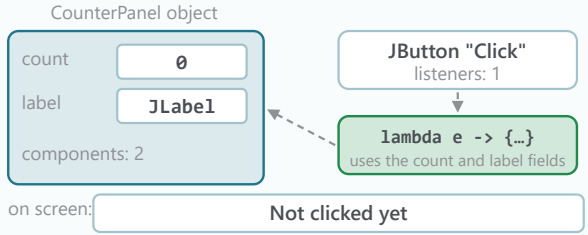
CALL STACK

`CounterPanel()`

`main()`

Counter Button: A Click, Step by Step

```
1  class CounterPanel extends JPanel {
2      private int count = 0;
3      private JLabel label = new JLabel("Not clicked
   yet");
4
5      CounterPanel() {
6          JButton button = new JButton("Click");
7          button.addActionListener(e -> {
8              count++;
9              label.setText(count + " click(s)");
10         });
▶          add(button);
▶          add(label);
13     }
14 }
```



The button and label are added to the panel (`FlowLayout` : side by side). The constructor ends.

Step 4 / 8

VARIABLES

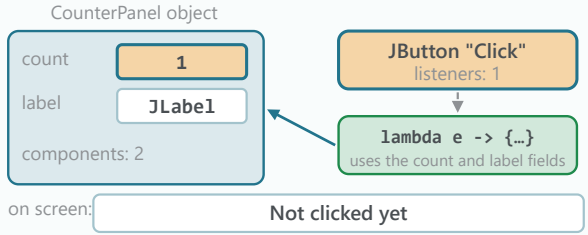
`components = 2`

CALL STACK

`main()`

Counter Button: A Click, Step by Step

```
1  class CounterPanel extends JPanel {
2      private int count = 0;
3      private JLabel label = new JLabel("Not clicked
   yet");
4
5      CounterPanel() {
6          JButton button = new JButton("Click");
7          button.addActionListener(e -> {
▶              count++;
9              label.setText(count + " click(s)");
10         });
11         add(button);
12         add(label);
13     }
14 }
```



The user clicks: the EDT calls `actionPerformed`, the lambda body. `count` is a **field**, so it can be incremented.

Step 5 / 8

VARIABLES

`count = 1`

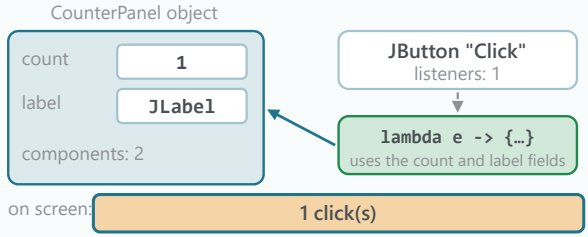
CALL STACK

`actionPerformed(e)`

EDT

Counter Button: A Click, Step by Step

```
1  class CounterPanel extends JPanel {
2      private int count = 0;
3      private JLabel label = new JLabel("Not clicked
   yet");
4
5      CounterPanel() {
6          JButton button = new JButton("Click");
7          button.addActionListener(e -> {
8              count++;
▶              label.setText(count + " click(s)");
10         });
11         add(button);
12         add(label);
13     }
14 }
```



`setText` changes the label's text; Swing repaints the label.

Step 6 / 8

VARIABLES

`count = 1`

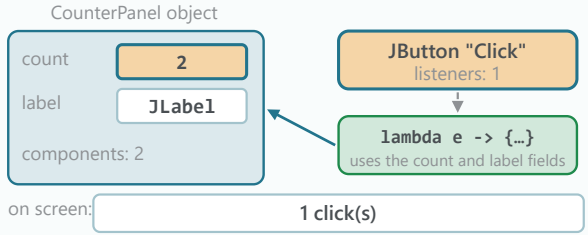
CALL STACK

`actionPerformed(e)`

EDT

Counter Button: A Click, Step by Step

```
1  class CounterPanel extends JPanel {
2      private int count = 0;
3      private JLabel label = new JLabel("Not clicked
   yet");
4
5      CounterPanel() {
6          JButton button = new JButton("Click");
7          button.addActionListener(e -> {
▶              count++;
9              label.setText(count + " click(s)");
10         });
11         add(button);
12         add(label);
13     }
14 }
```



Second click: the same lambda is called again; the field becomes 2.

Step 7 / 8

VARIABLES

`count = 2`

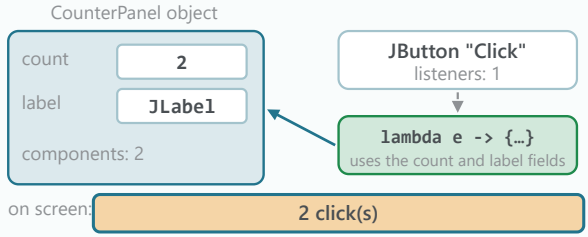
CALL STACK

`actionPerformed(e)`

EDT

Counter Button: A Click, Step by Step

```
1  class CounterPanel extends JPanel {
2      private int count = 0;
3      private JLabel label = new JLabel("Not clicked
   yet");
4
5      CounterPanel() {
6          JButton button = new JButton("Click");
7          button.addActionListener(e -> {
8              count++;
▶              label.setText(count + " click(s)");
10         });
11         add(button);
12         add(label);
13     }
14 }
```



The label updates. Between clicks the program runs no code at all; it just **waits for events**.

Step 8 / 8

VARIABLES

`count = 2`

CALL STACK

`actionPerformed(e)`

EDT

ActionListener: Anonymous Class and Lambda

ANONYMOUS INNER CLASS (CLASSIC)

```
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        count++;
        label.setText("Count: " + count);
    }
});
```

LAMBDA EXPRESSION (JAVA 8+)

```
button.addActionListener(e -> {
    count++;
    label.setText("Count: " + count);
});

// one expression: no braces needed
button.addActionListener(e -> System.exit(0));
```

- `ActionListener` has just **one** abstract method (a functional interface), so the compiler accepts the lambda `e -> …` as the body of `actionPerformed`.
- `MouseListener` has five methods, so it can't be a lambda; use an adapter class there.
- A lambda **cannot modify a local variable**: `int count = 0; … e -> count++` won't compile. Mutable values such as counters must be **fields**.

TEXT COMPONENTS

JLabel and JTextField

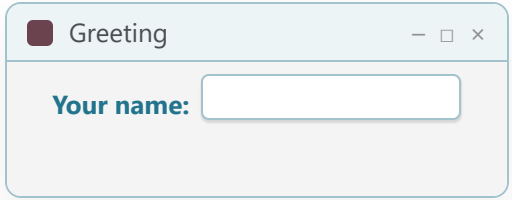
`JLabel`: a label

Displays text or an image; the user can't change it. `setText()` sets the text, `setIcon()` the image, `setFont()` the font, `setForeground()` the text color.

`JTextField`: a one-line text field

Lets the user **enter text**; forms collect data with it. `getText()` / `setText()` read and write, `setColumns()` sets the visible width in columns, `addActionListener()` fires on Enter.

```
JLabel label = new JLabel("Your name:");
label.setFont(new Font("SansSerif", Font.BOLD, 14));
label.setForeground(Color.BLUE);
JTextField nameField = new JTextField(12);
JLabel greeting = new JLabel(" ");
nameField.addActionListener(e -> greeting.setText(
        "Hello, " + nameField.getText() + "!"));
```



Type a name and press Enter to see the greeting.

TEXT COMPONENTS

JTextArea and JScrollPane

`JTextArea` : multi-line text

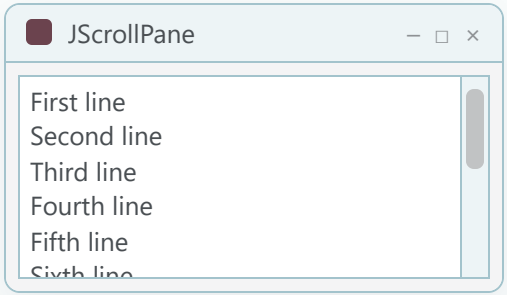
Content can be read, set and updated. `new JTextArea(rows, cols)` fixes the visible size up front. `getText()`, `setText()`, `append()` (add at the end), `setEditable(false)` (read-only), `setLineWrap(true)` (wrap lines).

`JScrollPane` : a scroll pane

When the component inside (text area, table, image) doesn't fit, it adds **scroll bars automatically** and adapts as the content changes size. Set its size with `setPreferredSize()`, its content with `setViewportView()`.

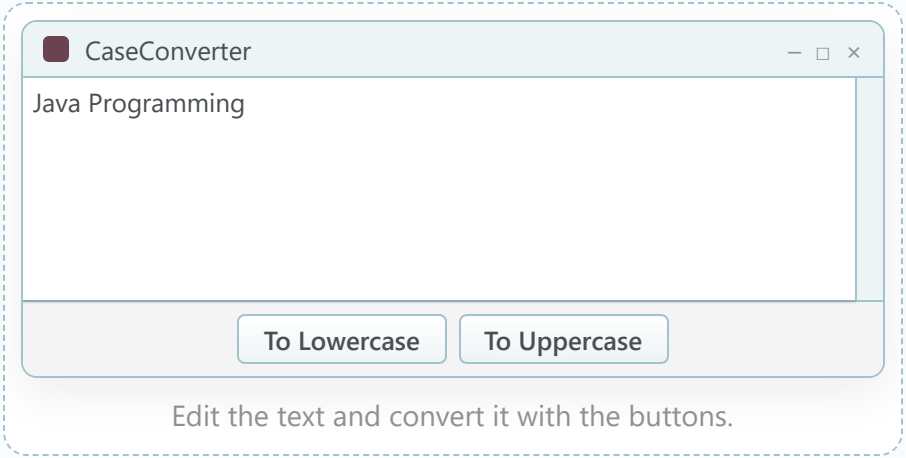
```
JTextArea area = new JTextArea(5, 20);
area.setText("First line\n");
area.append("Second line\n");
JScrollPane scroll = new JScrollPane(area);
System.out.print(area.getText());
System.out.println(area.getLineCount());
```

```
First line
Second line
3
```



CASE CONVERTER

Text Converter: The Window



Edit the text and convert it with the buttons.

- In the `CENTER`, an 8-row × 50-column `JTextArea` inside a `JScrollPane`.
- At the bottom (`SOUTH`), a `JPanel` holding two buttons (`FlowLayout`: centered).
- `pack()` computes the window size from these components' preferred sizes.
- The buttons convert the text with `toLowerCase()` / `toUpperCase()` and write it back.

Note: `toUpperCase()` uses the machine's locale. On an English-locale machine `"istanbul"` → `"ISTANBUL"`; on a Turkish one, `"İSTANBUL"`. To pin the result: `toUpperCase(Locale.ROOT)` or `toUpperCase(Locale.forLanguageTag("tr"))`. This demo uses English rules.

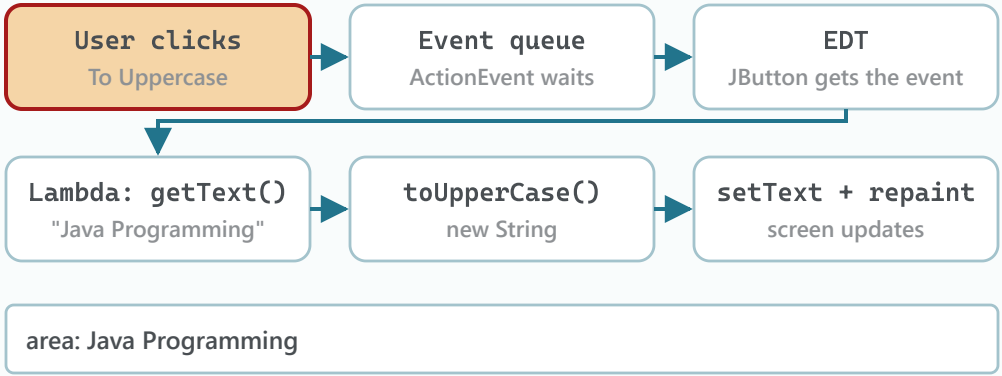
Text Converter: The Code

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

public class CaseConverter {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            var area = new JTextArea("Java Programming", 8, 50);
            var lowerButton = new JButton("To Lowercase");
            var upperButton = new JButton("To Uppercase");
            lowerButton.addActionListener(e ->
                    area.setText(area.getText().toLowerCase()));
            upperButton.addActionListener(e ->
                    area.setText(area.getText().toUpperCase()));
            var panel = new JPanel();
            panel.add(lowerButton);
            panel.add(upperButton);
            var frame = new JFrame("CaseConverter");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.getContentPane().add(new JScrollPane(area),
                    BorderLayout.CENTER);
            frame.getContentPane().add(panel, BorderLayout.SOUTH);
            frame.pack();
            frame.setVisible(true);
        });
    }
}
```

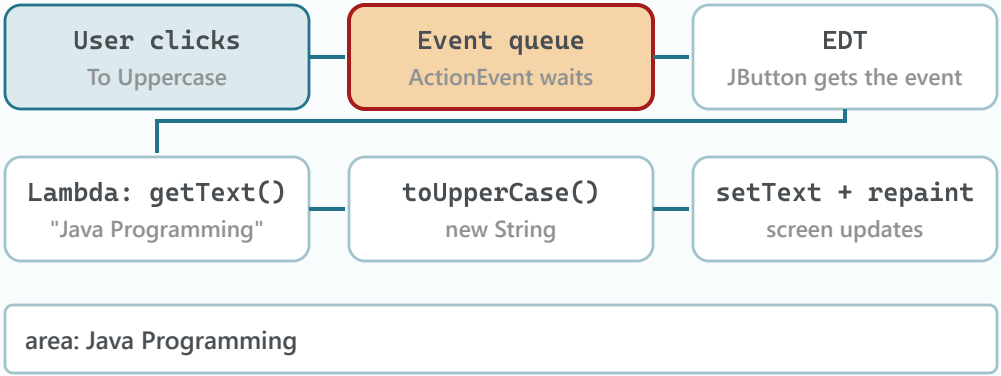
- The whole GUI is built inside one lambda, on the EDT.
- `area` is a local variable, but the lambda only **reads** it (never reassigns it), so it may use it (effectively final).
- `toUpperCase()` returns a new `String`; since `String` is immutable, the result must be **written back** with `setText`.
- Write `BorderLayout.CENTER` in full; a bare `CENTER` won't compile.

What Happens on a Click?



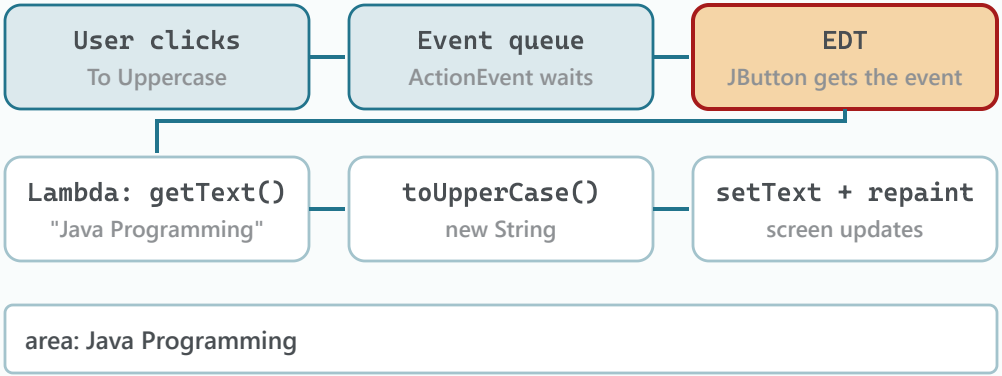
The user clicks “To Uppercase”; the operating system passes this to Java.

What Happens on a Click?



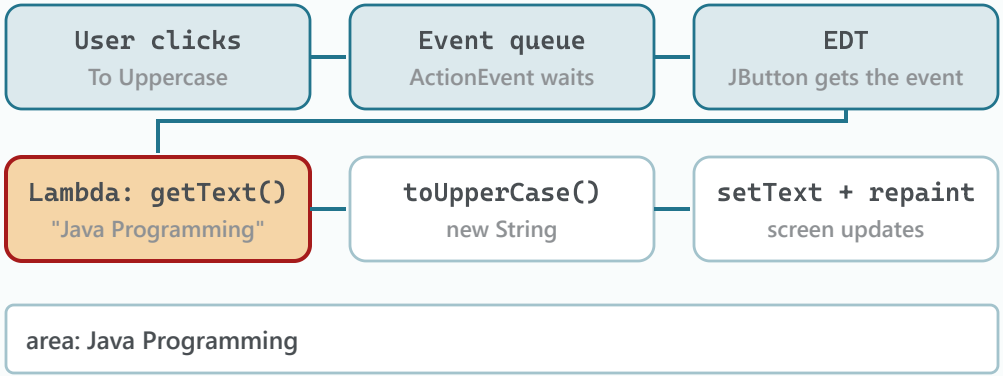
Java creates an event object and puts it on the **event queue**.

What Happens on a Click?



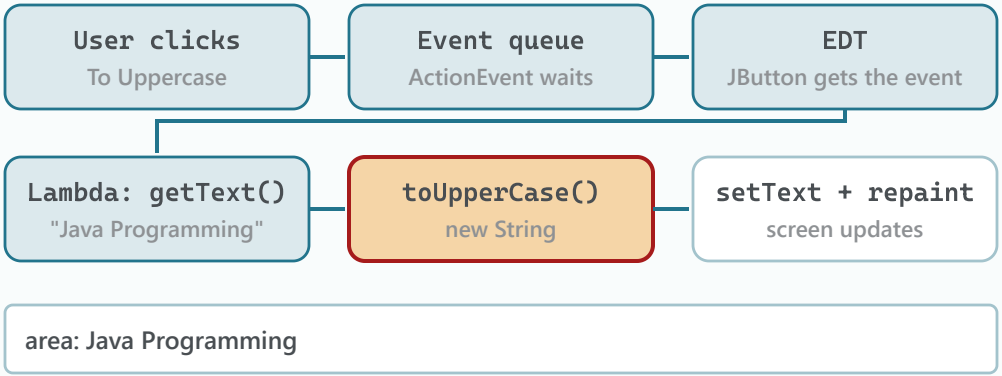
The EDT takes the next event; the button calls `actionPerformed` on every registered `ActionListener`, so our lambda runs.

What Happens on a Click?



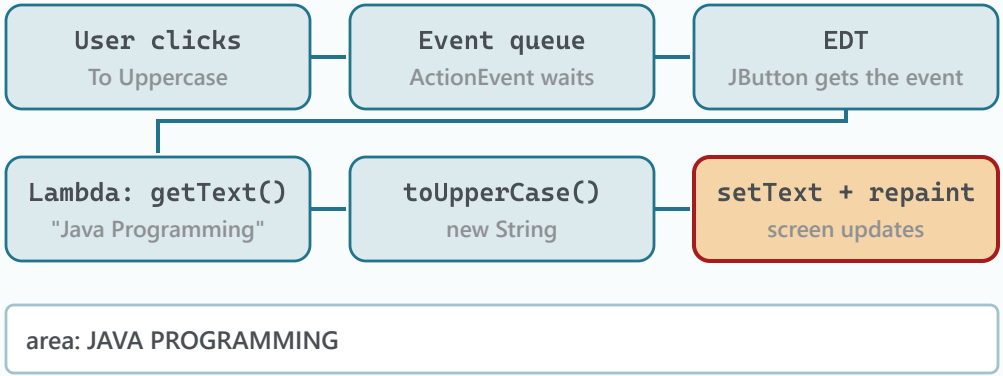
`area.getText()` reads the current text.

What Happens on a Click?



`toUpperCase()` makes a **new** string; the old one is unchanged.

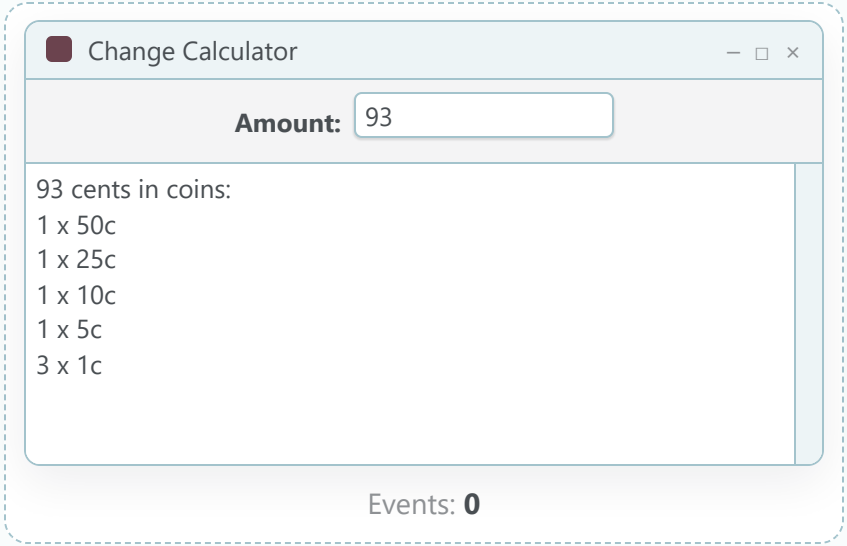
What Happens on a Click?



`setText` updates the area's model; the area repaints itself with `repaint`. The EDT moves on to the next event.

CHANGE CALCULATOR

Making Change



Shows how to pay the amount entered (in cents) with the fewest coins (50, 25, 10, 5, 1).

- There is no button: the report updates **by itself** on every keystroke.
- A `DocumentListener` attached to the text field's **document** (`Document`) makes this happen.
- Try: 93, 287, an empty field, `abc` , `-7` .

Fields and Constructor

```
public class ChangeCalculator extends JFrame {
    private JTextField amountField = new JTextField(12);
    private Document doc = amountField.getDocument();
    private JTextArea report = new JTextArea(8, 40);

    public ChangeCalculator() {
        var topPanel = new JPanel();
        topPanel.add(new JLabel("Amount:"));
        topPanel.add(amountField);
        getContentPane().add(topPanel, BorderLayout.NORTH);
        getContentPane().add(new JScrollPane(report),
                BorderLayout.CENTER);
        setBackground(Color.LIGHT_GRAY);
        report.setEditable(false);
        doc.addDocumentListener(new DocumentListener() {
            public void insertUpdate(DocumentEvent e) { updateReport(); }
            public void removeUpdate(DocumentEvent e) { updateReport(); }
            public void changedUpdate(DocumentEvent e) { updateReport(); }
        });
    }
```

- The class **extends** `JFrame`. Swing separates data from display (MVC): the `JTextField` is the **view**; the `Document` from `getDocument()` is the **model** holding the text.
- `DocumentListener`: `insertUpdate` (text added), `removeUpdate` (removed), `changedUpdate` (style changed; never called for plain text). It is an interface, so all three are written.

`updateReport`: 93 cents

```
1  void updateReport() {
2      try {
▶          var amount = Integer.parseInt(
▶                  doc.getText(0, doc.getLength()));
5          int[] coins = {50, 25, 10, 5, 1};
6          report.setText(amount
7                  + " cents in coins:\n");
8          for (int x : coins) {
9              report.append(amount / x + " x " + x + "c\n");
10             amount %= x;
11         }
12     } catch (NumberFormatException e) {
13         report.setText("Not an integer or out of range");
14     } catch (Exception e) {
15         report.setText(e.toString());
16     }
17 }
```



The field's text `"93"` is read and parsed as an integer.

Step 1 / 13

VARIABLES

`amount = 93`

OUTPUT

`updateReport`: 93 cents

```
1  void updateReport() {
2      try {
3          var amount = Integer.parseInt(
4                  doc.getText(0, doc.getLength()));
▶          int[] coins = {50, 25, 10, 5, 1};
6          report.setText(amount
7                  + " cents in coins:\n");
8          for (int x : coins) {
9              report.append(amount / x + " x " + x + "c\n");
10             amount %= x;
11         }
12     } catch (NumberFormatException e) {
13         report.setText("Not an integer or out of range");
14     } catch (Exception e) {
15         report.setText(e.toString());
16     }
17 }
```

50	25	10	5	1
cents	cents	cents	cents	cents
–	–	–	–	–

amount left: 93

The coin values, largest first, in an array.

Step 2 / 13

VARIABLES

`amount = 93`

`coins = [50, 25, 10, 5, 1]`

OUTPUT

`updateReport`: 93 cents

```
1  void updateReport() {
2      try {
3          var amount = Integer.parseInt(
4                  doc.getText(0, doc.getLength()));
5          int[] coins = {50, 25, 10, 5, 1};
▶          report.setText(amount
▶                  + " cents in coins:\n");
8          for (int x : coins) {
9              report.append(amount / x + " x " + x + "c\n");
10             amount %= x;
11         }
12     } catch (NumberFormatException e) {
13         report.setText("Not an integer or out of range");
14     } catch (Exception e) {
15         report.setText(e.toString());
16     }
17 }
```



`setText` starts the report **from scratch**: old content is erased.

Step 3 / 13

VARIABLES

`amount = 93`

`coins = [50, 25, 10, 5, 1]`

OUTPUT

```
93 cents in coins:
```

`updateReport`: 93 cents

```
1  void updateReport() {
2      try {
3          var amount = Integer.parseInt(
4                  doc.getText(0, doc.getLength()));
5          int[] coins = {50, 25, 10, 5, 1};
6          report.setText(amount
7                  + " cents in coins:\n");
8          for (int x : coins) {
▶              report.append(amount / x + " x " + x + "c\n");
10             amount %= x;
11         }
12     } catch (NumberFormatException e) {
13         report.setText("Not an integer or out of range");
14     } catch (Exception e) {
15         report.setText(e.toString());
16     }
17 }
```



x = 50: 93 / 50 = **1** (integer division); the line is appended.

Step 4 / 13

VARIABLES

`amount = 93` `x = 50`

OUTPUT

```
93 cents in coins:
1 x 50c
```

`updateReport`: 93 cents

```
1  void updateReport() {
2      try {
3          var amount = Integer.parseInt(
4                  doc.getText(0, doc.getLength()));
5          int[] coins = {50, 25, 10, 5, 1};
6          report.setText(amount
7                  + " cents in coins:\n");
8          for (int x : coins) {
9              report.append(amount / x + " x " + x + "c\n");
▶              amount %= x;
11         }
12     } catch (NumberFormatException e) {
13         report.setText("Not an integer or out of range");
14     } catch (Exception e) {
15         report.setText(e.toString());
16     }
17 }
```

50	25	10	5	1
cents	cents	cents	cents	cents
× 1	–	–	–	–

amount left: 43

Left: 93 % 50 = 43.

Step 5 / 13

VARIABLES

`amount = 43` `x = 50`

OUTPUT

```
93 cents in coins:
1 x 50c
```

`updateReport`: 93 cents

```
1  void updateReport() {
2      try {
3          var amount = Integer.parseInt(
4                  doc.getText(0, doc.getLength()));
5          int[] coins = {50, 25, 10, 5, 1};
6          report.setText(amount
7                  + " cents in coins:\n");
8          for (int x : coins) {
▶              report.append(amount / x + " x " + x + "c\n");
10             amount %= x;
11         }
12     } catch (NumberFormatException e) {
13         report.setText("Not an integer or out of range");
14     } catch (Exception e) {
15         report.setText(e.toString());
16     }
17 }
```

50	25	10	5	1
cents	cents	cents	cents	cents
× 1	× 1	–	–	–

amount left: 43

x = 25: 43 / 25 = **1** (integer division); the line is appended.

Step 6 / 13

VARIABLES

`amount = 43` `x = 25`

OUTPUT

```
93 cents in coins:
1 x 50c
1 x 25c
```

`updateReport`: 93 cents

```
1  void updateReport() {
2      try {
3          var amount = Integer.parseInt(
4                  doc.getText(0, doc.getLength()));
5          int[] coins = {50, 25, 10, 5, 1};
6          report.setText(amount
7                  + " cents in coins:\n");
8          for (int x : coins) {
9              report.append(amount / x + " x " + x + "c\n");
▶              amount %= x;
11         }
12     } catch (NumberFormatException e) {
13         report.setText("Not an integer or out of range");
14     } catch (Exception e) {
15         report.setText(e.toString());
16     }
17 }
```



Left: 43 % 25 = 18.

Step 7 / 13

VARIABLES

`amount = 18` `x = 25`

OUTPUT

```
93 cents in coins:
1 x 50c
1 x 25c
```

`updateReport`: 93 cents

```
1  void updateReport() {
2      try {
3          var amount = Integer.parseInt(
4                  doc.getText(0, doc.getLength()));
5          int[] coins = {50, 25, 10, 5, 1};
6          report.setText(amount
7                  + " cents in coins:\n");
8          for (int x : coins) {
▶              report.append(amount / x + " x " + x + "c\n");
10             amount %= x;
11         }
12     } catch (NumberFormatException e) {
13         report.setText("Not an integer or out of range");
14     } catch (Exception e) {
15         report.setText(e.toString());
16     }
17 }
```



x = 10: 18 / 10 = **1** (integer division); the line is appended.

Step 8 / 13

VARIABLES

`amount = 18` `x = 10`

OUTPUT

```
93 cents in coins:
1 x 50c
1 x 25c
1 x 10c
```

`updateReport`: 93 cents

```
1  void updateReport() {
2      try {
3          var amount = Integer.parseInt(
4                  doc.getText(0, doc.getLength()));
5          int[] coins = {50, 25, 10, 5, 1};
6          report.setText(amount
7                  + " cents in coins:\n");
8          for (int x : coins) {
9              report.append(amount / x + " x " + x + "c\n");
▶              amount %= x;
11         }
12     } catch (NumberFormatException e) {
13         report.setText("Not an integer or out of range");
14     } catch (Exception e) {
15         report.setText(e.toString());
16     }
17 }
```



Left: 18 % 10 = 8.

Step 9 / 13

VARIABLES

`amount = 8` `x = 10`

OUTPUT

```
93 cents in coins:
1 x 50c
1 x 25c
1 x 10c
```

`updateReport`: 93 cents

```
1  void updateReport() {
2      try {
3          var amount = Integer.parseInt(
4                  doc.getText(0, doc.getLength()));
5          int[] coins = {50, 25, 10, 5, 1};
6          report.setText(amount
7                  + " cents in coins:\n");
8          for (int x : coins) {
▶              report.append(amount / x + " x " + x + "c\n");
10             amount %= x;
11         }
12     } catch (NumberFormatException e) {
13         report.setText("Not an integer or out of range");
14     } catch (Exception e) {
15         report.setText(e.toString());
16     }
17 }
```



x = 5: 8 / 5 = **1** (integer division); the line is appended.

Step 10 / 13

VARIABLES

`amount` = **8** `x` = **5**

OUTPUT

```
93 cents in coins:
1 x 50c
1 x 25c
1 x 10c
1 x 5c
```

`updateReport`: 93 cents

```
1  void updateReport() {
2      try {
3          var amount = Integer.parseInt(
4                  doc.getText(0, doc.getLength()));
5          int[] coins = {50, 25, 10, 5, 1};
6          report.setText(amount
7                  + " cents in coins:\n");
8          for (int x : coins) {
9              report.append(amount / x + " x " + x + "c\n");
▶              amount %= x;
11         }
12     } catch (NumberFormatException e) {
13         report.setText("Not an integer or out of range");
14     } catch (Exception e) {
15         report.setText(e.toString());
16     }
17 }
```



Left: 8 % 5 = 3.

Step 11 / 13

VARIABLES

`amount = 3` `x = 5`

OUTPUT

```
93 cents in coins:
1 x 50c
1 x 25c
1 x 10c
1 x 5c
```

`updateReport`: 93 cents

```
1  void updateReport() {
2      try {
3          var amount = Integer.parseInt(
4                  doc.getText(0, doc.getLength()));
5          int[] coins = {50, 25, 10, 5, 1};
6          report.setText(amount
7                  + " cents in coins:\n");
8          for (int x : coins) {
▶              report.append(amount / x + " x " + x + "c\n");
10             amount %= x;
11         }
12     } catch (NumberFormatException e) {
13         report.setText("Not an integer or out of range");
14     } catch (Exception e) {
15         report.setText(e.toString());
16     }
17 }
```



x = 1: 3 / 1 = **3** (integer division); the line is appended.

VARIABLES

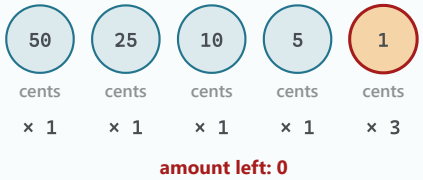
`amount = 3` `x = 1`

OUTPUT

```
93 cents in coins:
1 x 50c
1 x 25c
1 x 10c
1 x 5c
3 x 1c
```

`updateReport`: 93 cents

```
1  void updateReport() {
2      try {
3          var amount = Integer.parseInt(
4                  doc.getText(0, doc.getLength()));
5          int[] coins = {50, 25, 10, 5, 1};
6          report.setText(amount
7                  + " cents in coins:\n");
8          for (int x : coins) {
9              report.append(amount / x + " x " + x + "c\n");
▶              amount %= x;
11         }
12     } catch (NumberFormatException e) {
13         report.setText("Not an integer or out of range");
14     } catch (Exception e) {
15         report.setText(e.toString());
16     }
17 }
```



Left: 3 % 1 = 0.

Step 13 / 13

VARIABLES

`amount = 0` `x = 1`

OUTPUT

```
93 cents in coins:
1 x 50c
1 x 25c
1 x 10c
1 x 5c
3 x 1c
```

main and the Order of Events

```
public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> {
        var frame = new ChangeCalculator();
        frame.setTitle("Change Calculator");
        frame.pack();
        frame.setDefaultCloseOperation(
                JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    });
}
```

User / code	Event → report
types `9`	`insertUpdate` → 9 cents: 1 × 5, 4 × 1
types `3`	`insertUpdate` → the 93-cent report
deletes it all	`removeUpdate` → `parseInt("")`: "Not an integer..."
`abc` or `99999999999`	`NumberFormatException` → same message
`setText("287")`	**two** events: first `removeUpdate`, then `insertUpdate` → 5, 1, 1, 0, 2
`-7`	A valid number! 0, 0, 0, −1, −2: a check is missing

The first `catch` handles the expected error (non-numeric text, a value beyond the `int` range). The second is a general safety net; it is needed because `getText(0, n)` can throw the checked `BadLocationException`.

SUMMARY

Good Practices

Layout managers

Don't hard-code coordinates; with a layout manager the layout survives different screen sizes and resolutions.

Meaningful names

`upperButton`, `amountField` instead of `b1`, `t2`: code with dozens of components stays readable.

Think event-driven

Design around “what should happen when the user does this?”; one listener per action.

Separate the GUI from the logic

Keep rules such as making change in a separate method or class; it can be tested without a GUI and reused in another one.

Build the GUI on the EDT

The body of `main`: `SwingUtilities.invokeLater(() -> { … })`.

Don't block the EDT

A long job in a listener (a download, a big computation) **freezes** the window; move such work to a separate thread.

Summary

Concept	Essence
Packages	`java.awt` : Graphics, Color, Font, Point, layouts, events · `javax.swing` : the J… components
`JFrame`	The window: `setSize` / `pack` , `setDefaultCloseOperation(EXIT_ON_CLOSE)` , `setVisible(true)` last
`JPanel`	Container and canvas; `FlowLayout` by default
Drawing	Override `paintComponent(Graphics g)` , call `super.paintComponent(g)` first; `repaint()` on changes
Coordinates	(0, 0) top left; x right, y down; ovals are given by their bounding rectangle
Layout	`BorderLayout` (5 regions, content-pane default), `FlowLayout` , `GridLayout` , `BoxLayout`
Events	Listener interface + `add…Listener` ; adapters for multi-method interfaces; lambdas for single-method ones
Mouse	`MouseListener` , `MouseMotionListener` ; `MouseEvent` : `getX` , `getY` , `getButton`
EDT	Swing is single-threaded; `invokeLater` is asynchronous, `invokeAndWait` synchronous
Text	`JLabel` displays, `JTextField` one line, `JTextArea` many lines, `JScrollPane` scrolls

SUMMARY

Common Mistakes

Forgetting the import

Without `import javax.swing.*;` for `JFrame`: `cannot find symbol … class JFrame`.

No setVisible(true) or EXIT_ON_CLOSE

The window never appears, or the program keeps running in the background after it closes.

Misspelling the method name

`paintcomponent` is a new method that is never called. With `@Override`: `method does not override or implement a method from a supertype`.

Forgetting super.paintComponent(g)

The background isn't cleared; leftovers of old drawings stay on screen.

Not writing every interface method

`ClickListener is not abstract and does not override abstract method mouseExited(MouseEvent) in MouseListener`. Fix: `MouseAdapter`.

get(size()) for the last element

`IndexOutOfBoundsException: Index 1 out of bounds for length 1`. The last element is `size() - 1`.

Changing a local variable in a lambda

`e -> count++`: `local variables referenced from a lambda expression must be final or effectively final`. Fix: make the counter a field.

Writing a constant without its class

`add(comp, CENTER)`: `cannot find symbol … variable CENTER`. Use `BorderLayout.CENTER`.

SUMMARY

Practice Problems

1. If the no-entry-sign panel were 200 × 100 pixels, what would `r`, `r2`, `bw`, `bh` be?
2. What report does the change program show for `68`?
3. Where is the center of the ellipse drawn by `g.drawOval(50, 50, 100, 60)`?
4. In a panel with `BorderLayout`, button `A` and then button `B` are added to the same `NORTH` region. Which one shows in the north?
5. If the user clicks the panel once and releases, which `MouseListener` methods are called, in what order?
6. What color does `Color.RED.brighter()` give?

▶ **Show answers**

SUMMARY

Exercises

1. **Target**: draw 5 nested red and white rings that scale with the panel size.
2. **Counter window**: “−1”, “+1” and “Reset” buttons and a `JLabel` showing the value.
3. Add a **“Clear”** button to the drawing program.
4. In the drawing program, make a **right click** delete the last curve.
5. Make the change program print “Amount must be positive” for **negative** amounts.
6. **Temperature converter**: enter °C in a `JTextField`; a button shows °F.

▶ **Answers / hints**

References

- Oracle, *The Java Tutorials*: Creating a GUI With Swing; Performing Custom Painting; Laying Out Components Within a Container; Writing Event Listeners; Concurrency in Swing.
- Oracle, *Java SE 21 API Documentation*: `java.awt.Graphics`, `java.awt.Color`, `javax.swing.JFrame`, `javax.swing.SwingUtilities`.
- P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson (online chapters: Swing GUI Components; Graphics and Java 2D).
- K. Sierra, B. Bates and T. Gee, *Head First Java*, 3rd ed., O'Reilly (Chapters 14–15: GUI, Swing and events).
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., Pearson (Chapters 10–11: graphics programming, event handling, Swing).
- A. B. Downey and C. Mayfield, *Think Java*, 2nd ed., O'Reilly (Appendix C: Graphics).

The End

Chapter 9: Graphics and User Interfaces

Previous: Chapter 8: Input and Output · Next: Chapter 10: Threads