Test 3 Prep

Topics & Study Guide

The following topics may appear on Test 3. Use the linked guides to review.

Topic	Guide	Key Sections
Layout Managers (FlowLayout, BorderLayout, GridLayout, composite layouts)	Swing Layout Managers	All sections — defaults, region rules, nesting
JFrame setup (extend JPanel, pack, EXIT_ON_CLOSE, invokeLater)	Swing API Basics	JFrame boilerplate, EDT
2D coordinate system (origin top-left, +Y down)	Custom Painting with Java 2D	Coordinate system, drawing off-screen
Shapes & bounding boxes (Rectangle2D, Ellipse2D, bounding box top-left corner)	Custom Painting with Java 2D	Shapes, bounding boxes, circle as ellipse
paintComponent (override, super call, cast to Graphics2D, never call directly)	Custom Painting with Java 2D	paintComponent pattern, anti-aliasing
Drawing operations (setPaint, fill, draw, setStroke, painter's algorithm)	Custom Painting with Java 2D	Fill vs draw, stroke, layering
Grid-to-pixel mapping	Custom Painting with Java 2D	Grid mapping section
Animation loop (render → change → rerender)	Animation with javax.swing.Timer	Animation concepts
Shapes as instance fields (not local variables in paintComponent)	Animation with javax.swing.Timer	Instance fields, shape mutability
javax.swing.Timer (not java.util.Timer; fires on EDT; start/stop/setDelay)	Animation with javax.swing.Timer	Timer API, game loop pattern, speed control

Topic	Guide	Key Sections
Pause/unpause (stop/start), Thread.sleep on EDT is bad	Animation with javax.swing.Timer	Pause pattern, EDT safety
Observer pattern (loose coupling, subject/observer, addObserver/removeObserver)	The Observer Pattern	Problem, solution, loose coupling
PropertyChangeListener & PropertyChangeSupport	The Observer Pattern	PCL interface, PCS composition, firePropertyChange
MVC pattern (Model, View, Controller roles and data flow)	Model-View-Controller (MVC)	Roles, data flow, pure views, container
Sealed interfaces & pattern matching (exhaustive switch, no default)	Model-View-Controller (MVC)	Sealed events, switch expressions
KeyListener / KeyAdapter (keyPressed vs keyTyped vs keyReleased)	Handling Key Events	Method distinctions, focus requirements
Focus requirements (setFocusable, requestFocusInWindow, buttons steal focus)	Handling Key Events	Focus section
Map-based key bindings (Map<Integer, Runnable>, getOrDefault)	The Strategy Pattern	Map-based dispatch, method references
Swing menus (setJMenuBar, JMenu extends JMenuItem, exit via WindowEvent)	Building Menus with JMenuBar	Menu bar setup, exit pattern, option dialogs
Design patterns overview (Gang of Four, three categories, Singleton)	Introduction to Design Patterns	History, categories, Singleton
Strategy pattern (encapsulate algorithms, Map<Key, Runnable>)	The Strategy Pattern	Pattern definition, modern Java implementation
Lambda expressions & method references (functional interfaces, SAM)	Introduction to Lambda Expressions	Functional interfaces, method references

Practice Problems

Want More Practice?

These are sample problems — one of each type you'll see on the test. Want more? **Form a study group!** Each person writes their own version of one of these problem types, then exchange and challenge each other. Writing problems is one of the best ways to deepen your understanding.

Test 3 has **6 types of problems**. Below is one example of each with a solution you can reveal when ready.

Section 1: Fill in the Blank

> **Instructions:** Fill in the blank with the correct term.

The default layout manager for a `JPanel` is _____, which arranges components in horizontal rows from left to right and wraps to the next row when space runs out.

Answer

Section 2: Visual Bug Detective

> **Instructions:** For the `paintComponent` method below, a table describes the **intended** visual output. Predict whether each shape renders as described. Write **YES** if the shape appears exactly as described, or **NO** if the code produces something different. The described output represents the **correct specification** — if the answer is NO, the code has a bug.

The panel is 400×400. Assume anti-aliasing is enabled.

```
public class NextPiecePanel extends JPanel {
    // T-piece stored as (col, row) pairs
    // Model convention: row 0 is the BOTTOM of the piece
    private final int[][] myPiece = { {0, 0}, {1, 0}, {2, 0}, {1, 1} };
    private static final int BLOCK = 30;
    private static final int MAX_ROW = 1;  // highest row in this piece

    @Override
```

```
    protected void paintComponent(final Graphics theGraphics) {
        super.paintComponent(theGraphics);
        final Graphics2D g2d = (Graphics2D) theGraphics;
        g2d.setPaint(Color.CYAN);

        for (final int[] block : myPiece) {
            final int x = block[0] * BLOCK;
            final int y = (MAX_ROW - block[1]) * BLOCK;
            g2d.fill(new Rectangle2D.Double(x, y, BLOCK, BLOCK));
        }
    }
}
```

The T-piece should render with the stem pointing UP:

#	Intended Shape	Position	Size	Style	Your Answer
1	Three blocks in a horizontal row	y = 30 (second row from top)	30×30 each	Filled cyan	____
2	One stem block centered above the row	(30, 0) — top row	30×30	Filled cyan	____
3	Overall T-shape points upward (stem on top)	Top-left area of panel	90×60 total	—	____

✓ **Answer** ›

Section 3: Bug Detective — Test Prediction

> **Instructions:** For the method below, predict whether each JUnit 5 test will **PASS** or **FAIL**. The method may be correct or may contain bugs. **All tests shown are correct specifications** — if

> a test fails, it's because the code has a bug, not the test.
>
> The `...` in the parameter `final Point... thePoints` is Java's **varargs** syntax — it lets the method accept zero or more `Point` arguments, which are received as an array. If you're unfamiliar, see the Oracle varargs tutorial.

```
public record Point(int x, int y) { }
```

```
/**
 * Converts grid positions to pixel-sized rectangles.
 *
 * @param theBlockSize the width and height of each block in pixels
 * @param thePoints the grid positions to convert
 * @return a list of Rectangle2D, one per point
 */
public static List<Rectangle2D> buildBlocks(final int theBlockSize,
                                            final Point... thePoints) {
    final List<Rectangle2D> result = new ArrayList<>();
    for (final Point p : thePoints) {
        result.add(new Rectangle2D.Double(
            p.y() * theBlockSize,
            p.x() * theBlockSize,
            theBlockSize, theBlockSize));
    }
    return result;
}
```

Test 1:

```
final var blocks = buildBlocks(30, new Point(0, 0));
assertEquals(0.0, blocks.get(0).getX());
```

PASS / FAIL:____

Test 2:

```
final var blocks = buildBlocks(30, new Point(2, 5));
assertEquals(60.0, blocks.get(0).getX());
```

PASS / FAIL:____

Test 3:

```
final var blocks = buildBlocks(30, new Point(2, 5));
assertEquals(150.0, blocks.get(0).getY());
```

PASS / FAIL:____

Test 4:

```
final var blocks = buildBlocks(25, new Point(1, 1));
assertEquals(25.0, blocks.get(0).getWidth());
```

PASS / FAIL: ____

Test 5:

```
final var blocks = buildBlocks(30, new Point(0, 0), new Point(1, 0));
assertEquals(2, blocks.size());
```

PASS / FAIL: ____

✓ **Answer** ›

Section 4: Listener Trace — Key Events

> **Instructions:** You are given a listener implementation and a sequence of user actions. In **Part A**, identify which listener method fires for each action (or "none"). In **Part B**, write the complete console output.

```
public class GameKeyListener extends KeyAdapter {
    private static final Runnable DO_NOTHING = () -> { };
    private final Map<Integer, Runnable> myKeyMap = new HashMap<>();

    public GameKeyListener() {
        myKeyMap.put(KeyEvent.VK_LEFT, () -> System.out.println("left"));
        myKeyMap.put(KeyEvent.VK_RIGHT, () -> System.out.println("right"));
        myKeyMap.put(KeyEvent.VK_SPACE, () -> System.out.println("drop"));
    }

    @Override
    public void keyPressed(final KeyEvent theEvent) {
        myKeyMap.getOrDefault(theEvent.getKeyCode(), DO_NOTHING).run();
    }
}
```

Registration: `panel.addKeyListener(new GameKeyListener());` Panel has `setFocusable(true)` and `requestFocusInWindow()`.

Part A:

User Action	Method(s)
Presses the left arrow key	____

User Action	Method(s)
Presses the up arrow key	_____
Presses the spacebar	_____
Presses the letter 'W'	_____

Part B: What is the complete console output?

```
|
|
|
|
```

✓ **Answer** ›

Section 5: Listener Trace — Mouse Events

> **Instructions:** You are given a listener implementation and a sequence of user actions. In **Part A**, identify which listener method fires for each action (or "none"). In **Part B**, write the complete console output.

```
public class MyMouseListener extends MouseAdapter {
    private int myClickCount;

    @Override
    public void mousePressed(final MouseEvent theEvent) {
        System.out.println("down:" + theEvent.getX() + "," + theEvent.getY());
    }

    @Override
    public void mouseReleased(final MouseEvent theEvent) {
        System.out.println("up:" + theEvent.getX() + "," + theEvent.getY());
    }

    @Override
    public void mouseClicked(final MouseEvent theEvent) {
        myClickCount++;
        System.out.println("click#" + myClickCount);
    }
}
```

Registration:

```
final var ma = new MyMouseListener();
panel.addMouseListener(ma);
```

Part A:

User Action	Method(s)
Presses mouse at (100, 50)	_____
Releases mouse at (100, 50)	_____
Presses mouse at (200, 100)	_____
Drags mouse to (300, 150)	_____
Releases mouse at (300, 150)	_____

Part B: What is the complete console output?

```
|
|
|
|
```

✓ **Answer** ›

Section 6: Timer State Trace — Draw on Grid

> **Instructions:** Trace the timer ticks to determine field values at the given time. Then draw the shapes that `paintComponent` would render on the provided grid. Shade each cell and label it with the color. The grid origin (0, 0) is the **top-left** corner. X increases to the right, Y increases downward.

```
public class DiagonalPanel extends JPanel {
    private int myCol;
    private int myRow;
    private final Timer myTimer;
    private static final int BLOCK = 40; // each grid cell is 40×40 pixels

    public DiagonalPanel() {
        myTimer = new Timer(100, e -> step());
```

```
        myTimer.start();
    }

    private void step() {
        myCol++;
        myRow++;
        repaint();
    }

    @Override
    protected void paintComponent(final Graphics theGraphics) {
        super.paintComponent(theGraphics);
        final Graphics2D g2d = (Graphics2D) theGraphics;
        g2d.setPaint(Color.RED);
        g2d.fill(new Rectangle2D.Double(
            myCol * BLOCK, myRow * BLOCK, BLOCK, BLOCK));
    }
}
```

Part A: Fill in the field values.

Ticks	myCol	myRow
0 (start)	____	____
1	____	____
2	____	____
3	____	____

Part B: After **300ms** (3 ticks), draw the shape on the grid. Each cell is one block. Shade the cell(s) and label with the color.

```
     0   1   2   3   4   5
   +---+---+---+---+---+---+
0  |   |   |   |   |   |   |
   +---+---+---+---+---+---+
1  |   |   |   |   |   |   |
   +---+---+---+---+---+---+
2  |   |   |   |   |   |   |
   +---+---+---+---+---+---+
3  |   |   |   |   |   |   |
   +---+---+---+---+---+---+
4  |   |   |   |   |   |   |
   +---+---+---+---+---+---+
5  |   |   |   |   |   |   |
   +---+---+---+---+---+---+
```

Answer — Part A

Answer — Part B