events # group-project # gui

Animation with javax.swing.Timer

TCSS 305 Programming Practicum

GUI applications that move, animate, or tick on a schedule need a way to execute code at regular intervals. In Swing, there is exactly one correct tool for this job: `javax.swing.Timer`. It fires `ActionEvent`s on the Event Dispatch Thread, which means your callback can safely update GUI components without any threading gymnastics. This guide explains how the Timer works, how it drives game loops, and why the alternatives are wrong.

Demo Project

The code examples in this guide come from the demo project: TCSS305-Game

Related Guide

This guide builds on the repaint cycle concepts from the Custom Painting with Java2D guide. Understanding how `repaint()` schedules paint requests is important context for animation.

---

1 What Is javax.swing.Timer?

`javax.swing.Timer` is a timer that fires one or more `ActionEvent`s at a specified interval. The critical detail: **it fires those events on the Event Dispatch Thread (EDT)**.

This matters because Swing is single-threaded. All GUI updates--setting text, repainting panels, changing colors--must happen on the EDT. If you modify a component from a background thread, you get unpredictable behavior: garbled rendering, race conditions, or outright crashes.

Because `javax.swing.Timer` fires on the EDT, your `ActionListener` callback can update the GUI directly. No `SwingUtilities.invokeLater()` wrapper. No synchronization. It just works.

⚠ Two Timers in Java -- Use the Right One

Java has two classes named `Timer`:

Class	Fires On	Safe for GUI Updates?
`javax.swing.Timer`	Event Dispatch Thread	Yes
`java.util.Timer`	Background thread	No

If you import the wrong one, your code will compile and appear to work--until it doesn't. Always use `javax.swing.Timer` for anything that touches Swing components.

---

2 Basic Usage

A `Timer` takes two arguments: a delay in milliseconds and an `ActionListener` to call each time the timer fires.

```
// Fire every 1000ms (1 second)
Timer timer = new Timer(1000, e -> advanceGame());
timer.start();
```

That's it. Every second, `advanceGame()` runs on the EDT. When you want to stop:

```
timer.stop();
```

Because `ActionListener` is a functional interface (one abstract method), the lambda syntax works cleanly here--unlike `MouseListener`, which has five methods and requires an adapter.

In the demo project, the NPC timer in `GameController` follows this exact pattern. The timer field is declared with its delay and callback, then started when a new game begins:

```
/** The NPC timer delay in milliseconds (1 second). */
private static final int NPC_TIMER_DELAY = 1000;

/** Timer that drives NPC movement. */
private final Timer myNPCTimer;

// In the constructor:
myNPCTimer = new Timer(NPC_TIMER_DELAY, theEvent -> npcTimerTick());
```

Every second, the timer fires and calls `npcTimerTick()`, which advances the NPC on the game board. The timer is created once and reused--there is no reason to construct a new `Timer` each time a game starts.

---

3 Timer and the Game Loop

A `javax.swing.Timer` is the standard way to drive a game loop in Swing. Each tick of the timer advances the game state by one step. In Tetris, that means dropping the current piece down one row. The pattern is straightforward:

```
// In the GUI layer -- create a timer that calls one model method per tick
final Timer gameTimer = new Timer(1000, e -> myGameControls.step());
```

The timer's `ActionListener` calls a **single method** on the model API. That's all it should do. The model handles the game logic internally--collision detection, line clearing, piece spawning--and fires `PropertyChangeEvent`s to notify the view of changes. The view's `PropertyChangeListener`s react by repainting.

3.1 Concrete Example: NPC Timer

The demo project uses this same pattern for NPC movement. In `GameController`, a `Timer` calls `npcTimerTick()` once per second:

```
private void npcTimerTick() {
    myGame.moveNPC();
    if (myStunCounter > 0) {
        myStunCounter--;
        if (myStunCounter == 0 && myGameBoardPanel != null) {
            myGameBoardPanel.clearStunVisual();
        }
    }
}
```

Notice that the core game action--`myGame.moveNPC()`--is a single call to the model. The model method fires `PropertyChangeEvent`s that cause the view to repaint. The stun counter is lightweight bookkeeping managed by the controller, not game logic leaking into the view.

This is the same pattern you will use for the Tetris timer in Sprint 2: one model call per tick, with the Observer pattern handling all view updates.

Keep the Timer Callback Thin

The timer callback should call **one method** on the model. Do not put game logic in the callback. Do not call `repaint()` from the callback. The model fires property changes; the view listens and repaints. This separation is the Observer pattern at work.

```
// GOOD: One model method call
Timer timer = new Timer(delay, e -> myGameControls.step());

// BAD: Game logic leaking into the view
Timer timer = new Timer(delay, e -> {
    if (!myBoard.isGameOver()) {          // Don't check state
here
        myBoard.moveDown();               // Don't call internal
methods
        myPanel.repaint();                // Don't repaint from
here
    }
});
```

---

4 Controlling the Timer

`javax.swing.Timer` provides a small, focused API for controlling its behavior:

Method	Purpose
`start()`	Begin firing events at the specified interval
`stop()`	Stop firing events
`isRunning()`	Check whether the timer is currently active
`setDelay(int ms)`	Change the interval between events
`getDelay()`	Get the current interval
`restart()`	Stop and start the timer, resetting the delay clock
`setInitialDelay(int ms)`	Set a different delay before the *first* event fires

4.1 Mapping GameState to Timer Control

Game state changes drive timer behavior. When the view receives a `PropertyChangeEvent` for the game state, it responds:

```
// In a PropertyChangeListener reacting to game state changes
switch (newState) {
    case RUNNING -> gameTimer.start();
    case PAUSED  -> gameTimer.stop();
    case OVER    -> gameTimer.stop();
}
```

In the demo project, the NPC timer restarts cleanly when a new game begins. The `GameController` listens for game events and uses `restart()` to reset the timer from wherever it was:

```
case final GameEvent.NewGameEvent newGameEvent -> {
    enableValidDirections(newGameEvent.validMoves());
    myStunCounter = 0;
    myNPCTimer.restart();
}
```

`restart()` is equivalent to calling `stop()` then `start()`--it resets the timer's internal clock so the full delay elapses before the next fire. This is the right call when starting a new game because you want a clean one-second delay before the first NPC move, regardless of where the previous timer cycle was.

4.2 Speeding Up with Levels

As the player clears lines and advances levels, the game should speed up. The timer makes this easy:

```
// Reduce delay as level increases
final int newDelay = BASE_DELAY - (level * SPEED_INCREMENT);
gameTimer.setDelay(Math.max(newDelay, MINIMUM_DELAY));
```

The `Math.max()` ensures you never set a zero or negative delay. A minimum delay of around 100-150ms keeps the game playable at high levels.

---

5 Common Mistakes

Mistakes That Will Cost You Debugging Time

Using `java.util.Timer` instead of `javax.swing.Timer`. The wrong import compiles without errors. Your timer callback runs on a background thread, and GUI updates silently corrupt Swing's internal state. The symptoms are intermittent and maddening.

Putting game logic in the timer callback. The callback should call one method on the model. If you find yourself writing `if` statements or calling `repaint()` inside the timer's `ActionListener`, you are bypassing the Observer pattern and coupling the view to model internals.

Not stopping the timer when the game ends or pauses. A running timer keeps firing. If the game is over but the timer is still ticking, the model keeps receiving step commands. Always stop the timer when the game state is no longer `RUNNING`.

Forgetting to restart after a pause. Pausing stops the timer; resuming must restart it. If your pause/unpause logic only toggles a boolean flag without calling `start()` and `stop()`, the timer either never stops or never resumes.

---

6 Why Not Thread.sleep()?

Students sometimes attempt a game loop using `Thread.sleep()`:

```
// DO NOT DO THIS -- blocks the EDT and freezes the GUI
while (gameRunning) {
    advanceGame();
    repaint();
    Thread.sleep(1000);  // Freezes everything for 1 second
}
```

This freezes the entire application. `Thread.sleep()` on the EDT means no mouse events, no keyboard events, no repainting, no button clicks--nothing--for the duration of the sleep. The window becomes unresponsive and the OS may flag it as "not responding."

`javax.swing.Timer` solves this by **scheduling** callbacks rather than blocking. Between timer fires, the EDT is free to process user input, repaint requests, and other events. The GUI stays responsive.

Summary

Concept	Key Point
javax.swing.Timer	Fires `ActionEvent`s on the EDT at a regular interval--safe for GUI updates
java.util.Timer	Fires on a background thread--**not safe** for Swing
Basic pattern	`new Timer(delay, e -> model.step())` then `start()`
Game loop	Timer callback calls one model method; view updates via PropertyChangeListeners
Controlling the timer	`start()`, `stop()`, `restart()`, `setDelay()`, `isRunning()`
Speeding up	`setDelay()` with a calculated interval based on level, clamped to a minimum
Thread.sleep()	Blocks the EDT, freezes the GUI--never use for animation

---

Further Reading

External Resources

- Oracle Java Tutorial: How to Use Swing Timers - Official guide to javax.swing.Timer
- Oracle Java Tutorial: Concurrency in Swing - Understanding the EDT and thread safety

---

References

Primary Texts:

- Horstmann, C. S. (2022). *Core Java, Volume I: Fundamentals* (12th ed.). Oracle Press. Chapter 12: User Interface Components with Swing.

Language Documentation:

- Oracle JDK 25: `javax.swing.Timer` -- Timer that fires ActionEvents at specified intervals on the EDT
- Oracle JDK 25: `java.awt.event.ActionListener` -- Functional interface for receiving action events

Tutorials:

- Oracle Java Tutorial: How to Use Swing Timers -- Official Swing Timer tutorial

---

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