lectures # week-6

Week 6: Lambda Expressions & Mouse Listeners (Wednesday, February 11, 2026)

Lecture Recording

Watch on Panopto

Related Assignment

This lecture covers concepts for Assignment A3.

Demo Code

Clone the lecture examples: TCSS305-Into-GUI

Lecture Preview [0:00]

Today's agenda:

1. **Assignment 3 Demo** — Undo and clear button behavior
2. **Software Design Discussion** — Loose coupling vs tight coupling
3. **ActionListener Review** — Why ActionListener, not MouseListener
4. **Lambda Expression Syntax** — Parameters, arrow operator, body rules
5. **Mouse Listeners** — MouseListener, MouseMotionListener, and the adapter pattern

---

Admin Notes [0:37]

Test 1 Results: Returned this Friday. Will go over the test and answer questions in class.

Test 2: Friday, February 20. Shorter than Test 1 due to classroom scheduling constraints. Sample questions will be posted in Discord over the weekend.

Group Project: Teams will be assigned and the group project introduced this Friday. Groups are assigned by the instructor — you do not pick your own teams.

Assignment 3: Undo & Clear [2:21]

The first two A3 requirements are implementing the **undo** button (curved arrow icon) and the **clear** button (trash can icon).

Clear erases everything from the canvas immediately.

Undo removes the most recently drawn shape. Each dot, line segment, or shape drawn on the canvas is a separate drawing. Undo removes them in reverse order — last drawn, first removed.

- There is no theoretical undo limit — if you drew 100,000 shapes, you can undo all 100,000
- There is no redo functionality
- Memory is the only practical constraint

> **Implementation Hint**
>
> Each button's action should be implementable as a **single line of code** (one lambda expression per button). Once you understand how to attach an ActionListener to a button and what model methods to call, the implementation is straightforward.

Software Design: Loose vs Tight Coupling [5:44]

> **Student Question**
>
> Q: Why should the backend keep track of all the state when the frontend is already showing it?
>
> A: The GUI is just a **representation** of state — its job is to show it, not store it. All shape data (color, width, coordinates) is stored in the model. The GUI is an "ignorant representation" that displays whatever the model tells it.

Why Not Put Everything in One Class? [6:49]

This application is small — it could be written in a single class, probably under 1,000 lines. Splitting it up actually creates *more* code (extra class headers, boilerplate). So why bother?

Two reasons:

1. **Scalability** — As the application grows (adding rectangles, ellipses, redo functionality), adding new features in a well-structured codebase means adding code in the *right place* and having everything just work. To add rectangle and ellipse drawing, you'd just add two model classes and one GUI controller — everything else works without change.
2. **Good software design** — Separation of responsibilities makes code debuggable, adaptable, and ready for change rather than fragile.

Loose Coupling vs Tight Coupling [11:14]

Loose Coupling	Tight Coupling
Changes on one side require minimal or no changes on the other	Changes break things across the entire application
New features plug in cleanly	New features require extensive rewrites
Code is adaptable	Code is fragile and static

The extreme example: Could you write the entire Linux operating system on a single line with no whitespace? Technically yes — the computer doesn't care about whitespace. But when you need to add a new feature, where does it go at character position 3,000,452?

> ⚠ **Technical Debt**
>
> Tightly coupled code isn't just debt you have to fix — it's debt where you might have to tear everything down and start from scratch. Good design means your software is built to grow.

Frontend vs Backend in This Context [15:01]

Term	Meaning in TCSS 305
Frontend / GUI / View	Everything shown on screen, including event handlers
Backend / Model	Non-GUI classes that store application state

This is different from web development where "backend" means a database sitting behind a web API. Here, the model classes store the shapes, their colors, their coordinates — everything that represents the application's state.

The Road Rage project follows the same pattern:

- **View package** — The GUI you see
- **Logic package** — Simulation logic (the `advance()` method)
- **Model package** — Cars, trucks, people — each with their own state

The design pattern for sending model updates to the GUI will be covered starting late next week.

ActionListener Review [18:48]

Quick review from Monday's lecture:

- **ActionListener** is an **interface** with one abstract method: `actionPerformed(ActionEvent e)`
- One abstract method makes it a **functional interface**
- You attach an ActionListener to a button via `addActionListener()`
- When the button is clicked, the button calls `actionPerformed()` on your listener object
- The `ActionEvent` object encapsulates the entire event (which component, when it happened)

Why ActionListener Instead of MouseListener? [20:03]

A button can be activated by:

- Mouse click
- Space bar (when focused)

- Keyboard navigation (accessibility)

An `ActionListener` says: "I don't care **how** this button was clicked — mouse, spacebar, or other means — it's been activated. Handle it."

A `MouseListener` would only catch mouse clicks, breaking accessibility. Not all users interact with a GUI using a mouse.

Student Question

Q: Is the MouseListener only used when you want mouse-specific actions?

A: Pretty much. Use MouseListener when you need mouse-specific information — like **where** the click happened (x, y coordinates). ActionListener doesn't provide location data. For a drawing canvas where you need click coordinates, you'd use a MouseListener. For buttons, always use ActionListener.

Built-in Button Behavior [24:39]

When you click a JButton, it visually depresses — the background changes, the outline shifts. But we never wrote that code. The `JButton` class has a built-in ActionListener that provides visual feedback when clicked. Since the method is `addActionListener` (not `setActionListener`), our listener is **added alongside** the built-in one.

Functional Interfaces and Lambda Context [29:25]

When can you use a lambda expression? When the **expected type** is a **functional interface**.

Look at `addActionListener(...)`:

- The parameter type is `ActionListener`
- `ActionListener` has one abstract method → functional interface
- Therefore, you **can** put a lambda expression here

Now look at `addMouseListener(...)`:

- The parameter type is `MouseListener`
- `MouseListener` has **five** abstract methods → **not** a functional interface
- Therefore, you **cannot** put a lambda expression here

> ⚠ **This Is the Key Question**
>
> "How do I know when to use a lambda?" → Check the expected type. Is it a functional interface (exactly one abstract method)? If yes, lambda works. If no, you need a class.

Examples of Functional Interfaces [43:55]

`Runnable` — One abstract method: `void run()` with no parameters

```
() -> expression
```

Empty parentheses because `run()` takes no parameters. You've been using this in JUnit with `assertThrows` and `assertAll`.

`BiFunction<T, U, R>` — One abstract method: `R apply(T t, U u)` with two parameters (it also has a default method `andThen`, but default methods don't count)

```
(x, y) -> expression
```

Two comma-separated parameters.

Lambda Expression Syntax [37:57]

A lambda expression has three parts: **parameters**, **arrow operator**, and **body**.

Parameters [39:33]

The parameters must match the abstract method's parameter list from the functional interface.

Type inference: The compiler can almost always infer the parameter types from context. By **hard convention**, omit the types:

```
// Not conventional — explicit type
(ActionEvent e) -> expression

// Conventional — inferred type
e -> expression
```

Parentheses rules:

Parameters	Syntax	Example
None	`()` required	`() -> expression`
One	Parentheses optional (convention: omit)	`e -> expression`
Multiple	`()` required, comma-separated	`(x, y) -> expression`

Arrow Operator [47:09]

```
->
```

A single dash followed by a greater-than sign. Java uses `->`. JavaScript uses `=>`. If you teach both languages, you **will** mix them up.

Body: Expression vs Statements [48:25]

Single expression — no braces, no return, no semicolon:

```
e -> myLabel.setText("Hello")
```

An expression evaluates to a value. Variable references (`num`), arithmetic (`num + 5`), and method calls (`name.toUpperCase()`) are all expressions. The lambda itself is also an expression — it doesn't require a semicolon any more than `num` requires a semicolon when passed as an argument.

Multiple statements — braces required, full Java syntax (return, semicolons):

```
e -> {
    if (flag) {
        return xyz;
    } else {
        return abc;
    }
}
```

As soon as you add curly braces, you're in full statement mode — returns and semicolons are required.

Both Syntaxes Are Valid

You *can* write a single expression with curly braces, but then you need the return and semicolon:

```
// Expression syntax (preferred)
e -> myLabel.setText("Hello")

// Statement syntax (valid but verbose)
e -> { return myLabel.setText("Hello"); }
```

Convention: if it fits in a single expression, use expression syntax.

Best Practice: Keep Lambdas Short [57:55]

Rule of thumb: Make your lambda expressions a single expression.

If you need more logic, extract it into a **helper method** with a descriptive name, then call that method from the lambda:

```
// Bad: complex lambda with poor readability
myButton.addActionListener(e -> {
    // ten lines of code stuffed into a lambda...
});

// Good: delegate to a well-named helper method
myButton.addActionListener(e -> handleColorChange());
```

This keeps your lambdas readable and your event-wiring code clean.

For A3: The clear and undo buttons can each be a single expression. The color and line-width buttons require more logic (opening `JColorChooser` / `JOptionPane`) — put that in helper methods and call them from the lambda.

Mouse Events Demo [1:01:36]

The demo application shows different mouse events firing in real time:

Event	When It Fires
Mouse Moved	Mouse moves within a component

Event	When It Fires
Mouse Entered	Mouse enters a component's bounds
Mouse Exited	Mouse leaves a component's bounds
Mouse Pressed	Mouse button pushed down
Mouse Released	Mouse button let go
Mouse Clicked	Press + release **without moving**
Mouse Dragged	Press + move (while button held)

Click vs Drag [1:04:22]

A **click** is press → release without moving the mouse. A **drag** is press → move → release.

If you press the mouse button and move even slightly before releasing, you get **pressed** and **released** events but **no click event**. This matters for users with motor difficulties (e.g., Parkinson's) who struggle to keep the mouse still during a click.

Click Count [1:03:48]

The `MouseEvent` object includes a click count. Mouse clicked event 16 means the user has clicked 16 times. This is how you detect double-clicks — check if `getClickCount() == 2`.

Which Event Do You Use? [1:05:29]

It depends on what you're building:

Application	Events You Care About
Drawing (A3 sketch pad)	Pressed, Dragged, Released
Tetris piece rotation on click	Clicked
Button activation	ActionListener (not mouse events at all)

MouseListener & MouseMotionListener [1:06:52]

Mouse events are split across **two** interfaces:

`MouseListener` — 5 abstract methods:

- `mouseClicked(MouseEvent e)`
- `mouseEntered(MouseEvent e)`
- `mouseExited(MouseEvent e)`
- `mousePressed(MouseEvent e)`
- `mouseReleased(MouseEvent e)`

`MouseMotionListener` — 2 abstract methods:

- `mouseMoved(MouseEvent e)`
- `mouseDragged(MouseEvent e)`

Neither is a functional interface, so **you cannot use lambda expressions** for either one.

If you implement `MouseListener`, you must implement all five methods — even if you only care about one. The unused methods sit there empty.

> **API Easter Egg**
>
> The `MouseListener` API documentation describes it as handling **"interesting"** mouse events (in quotes). `MouseMotionListener` has no such qualifier. Somebody at Sun Microsystems in 1996 had opinions about which mouse events were interesting.

The Adapter Pattern: MouseAdapter [1:11:02]

The problem: Implementing `MouseListener` forces you to write five methods when you might only need one or two. Same with `MouseMotionListener`.

The solution: `MouseAdapter` is a class that implements **both** `MouseListener` and `MouseMotionListener` (plus `MouseWheelListener`) with empty method bodies.

Instead of implementing the interfaces directly, **extend** `MouseAdapter` and override only the methods you need:

```
// Bad: must implement all 5 methods
class MyListener implements MouseListener {
    @Override public void mouseClicked(MouseEvent e) { /* what I care about */
}
    @Override public void mousePressed(MouseEvent e) { }   // empty
    @Override public void mouseReleased(MouseEvent e) { }   // empty
    @Override public void mouseEntered(MouseEvent e) { }    // empty
    @Override public void mouseExited(MouseEvent e) { }     // empty
}

// Good: only override what you need
class MyListener extends MouseAdapter {
    @Override
    public void mouseClicked(MouseEvent e) {
        // what I care about
    }
}
```

Since `MouseAdapter` implements both interfaces, a single class extending it can override methods from **either** interface. For the A3 sketch pad, you'll need `mousePressed`, `mouseDragged`, and `mouseReleased` — all three available through `MouseAdapter`.

Student Question

Q: Can a class implement multiple interfaces?

A: Yes. Classes can implement as many interfaces as they want, comma-separated: `public class Foo implements InterfaceA, InterfaceB, InterfaceC`. Just make sure you implement all abstract methods from every interface (or declare the class abstract). `MouseAdapter` implements four interfaces this way.

Looking Ahead [1:14:33]

Friday:

- Test 1 returned and reviewed
- Test 2 details (Friday, February 20)
- Group project introduction and team assignments

Coming Next Week

The design pattern for communicating state changes from model to GUI — the problem of keeping frontend and backend in sync when they're loosely coupled.

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