lectures # week-6

Week 6: Event-Driven Programming & ActionListeners (Monday, February 9, 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** — Running the solution, Q&A
2. **Event-Driven Programming** — Inversion of control concept
3. **Listeners in Java** — Terminology and types
4. **ActionListener Interface** — Handling button clicks
5. **Progressively Shorter Implementations** — External class → inner class → anonymous inner class → lambda → method reference

---

Admin Notes [0:43]

Assignment 3 Q&A:

- The undo button requires both adding an event handler AND calling the correct method on the back-end model
- Find the `myShapeCreator` reference in `SketcherGUI` to access the model
- API method names are intuitive: `undo()`, `clear()`, `setColor()`, `getColor()`, `setWidth()`, `getWidth()`

Reading Materials:

Starting this week, we're covering new material that most students haven't seen before. Spend extra time with the guides—everything from today's lecture is covered in more detail there.

Assignment Scope:

Don't add features beyond what's required. You can fork your own repository to experiment, but submit only what's asked for. Changing the API would require model changes—that's beyond the assignment scope.

Event-Driven Programming Concept [11:54]

The key insight: Event-driven programming is an **inversion of control** compared to console applications.

Console Applications	GUI Applications
Program controls flow	User controls flow
Program prompts user	Program waits for user
Sequential execution	Event-based execution

As a **user**, you've been working with event-driven applications your whole life. As a **programmer**, you've mostly written console-based programs where your code controls the flow.

The developer mindset shift:

- As a user: "I should be able to click this button and have something happen"

- As a user: "I should be able to stare at this screen for an hour and have nothing happen"
- As a developer: "Given this UI, what events do I want to handle?"

> **You Don't Handle Every Event**
>
> Events are constantly firing—mouse movement, key presses, etc. If you don't write a handler for an event, nothing happens. That's fine. Only write handlers for events you care about.

Listeners in Java [16:09]

Terminology:

Term	Scope
Event Handler	Language-agnostic term (works in Java, C#, C++, etc.)
Listener	Java-specific term

These terms are used interchangeably in this course. Both refer to the same concept: code that executes when an event occurs.

Types of Listeners [18:05]

Java provides different listener types for different events:

Listener Type	Events
MouseListener	Mouse pressed, released, clicked, drag
ActionListener	Button clicks, menu selections
KeyListener	Key pressed, released, typed

Key pressed vs. key typed:

- **Key pressed:** Fires continuously while key is held (what you want for WASD game controls)
- **Key typed:** Fires once per press-and-release cycle (not useful for continuous movement)

> **Video Game Movement**
>
> If you used `keyTyped` for movement, you'd have to mash the W key repeatedly to move forward. `keyPressed` fires continuously while the key is held, which is what games need.

ActionListener for Buttons [21:03]

You might think clicking a button with a mouse requires a `MouseListener`. **Wrong.**

Buttons can be activated by:

- Mouse clicks
- Pressing the space bar when focused
- Keyboard navigation (important for accessibility/ADA compliance)

Because buttons respond to multiple input methods, they use a more general `ActionListener` rather than a mouse-specific listener.

The ActionListener interface:

```
public interface ActionListener {
    void actionPerformed(ActionEvent e);
}
```

That's it—just one method. This is significant (we'll see why soon).

How it works:

1. You create a class that `implements ActionListener`
2. You instantiate an object of that class
3. You pass that object to the button via `addActionListener()`
4. When the button is clicked, the button calls `actionPerformed()` on your object

Student Question

Q: Why does it need to be an interface if it's only one method?

A: The interface creates a **contract**. Without it, you could pass any object to the button—a `Car`, a `StoreItem`—but the button wouldn't know what to do with it. The interface guarantees that whatever object you pass has an `actionPerformed()` method the button can call. The number of methods doesn't matter; the contract does.

The ActionEvent parameter:

The `ActionEvent` object contains information about the event:

- `getSource()` — Which component fired the event
- `getWhen()` — Timestamp of the event

For button clicks, you typically don't need this information. For mouse events, the event object becomes essential (coordinates, which button, click count, etc.).

Java Swing Basics [33:32]

Why Swing?

Swing is old (late 1990s—possibly older than you). We're not teaching Swing; we're teaching **event-driven programming** using Swing as a tool.

Modern Swing applications can look modern with third-party look-and-feel libraries (like FlatLaf). You'll use these in your assignments.

About JavaFX:

JavaFX is newer (late 2000s) and arguably better for new desktop applications. However:

- Oracle removed it from the standard library
- It's now third-party (OpenJavaFX)
- It adds layers of complexity we don't need for teaching event-driven concepts

Swing component naming:

All Swing components start with `J`:

- `JButton` — Button

- `JLabel` — Text label
- `JPanel` — Container panel
- `JFrame` — Window

Swing is highly configurable:

Any visual property you've ever seen in an application probably has a setter:

- `setForeground()` — Text color
- `setBackground()` — Background color
- `setText()` — Label text
- `setFont()` — Font

> **Finding Methods**
>
> Ask ChatGPT: "I have a JLabel. How do I change the font?" That's fine.
>
> Don't ask: "Here's my assignment. Write the code." You're not learning anything.
>
> Swing is old, so there's tons of training data. The models know it well.

Implementation 1: External Class [30:28]

The most verbose approach—a separate file for the listener:

```
package edu.uw.tcss.view.first.actionlisteners;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JLabel;

public class HelloActionListener implements ActionListener {

    private final JLabel myLabel;

    public HelloActionListener(final JLabel label) {
        myLabel = label;
    }

    @Override
    public void actionPerformed(final ActionEvent event) {
        myLabel.setText("Hello World!");
    }
}
```

The problem: This external class doesn't have access to the private fields of the class using it. We have to pass the label through the constructor.

Attaching the listener:

```
private void addListeners() {
    final ActionListener listener = new HelloActionListener(myMessageLabel);
    myHelloButton.addActionListener(listener);
}
```

Note: The variable type is `ActionListener` (the interface), not `HelloActionListener` (the concrete class). This is good practice—program to the interface.

`addActionListener` vs `setActionListener`

Java Swing uses `addActionListener()`. Android uses `setOnClickListener()`.

- **add** = You can attach multiple listeners to one button
- **set** = Only one listener; setting replaces any existing listener

This is an API design choice. In this course, we'll typically only add one listener per button.

Implementation 2: Inner Class [47:03]

An **inner class** is a class defined inside another class:

```
public class HelloGoodByeEmpty extends JPanel {

    private JLabel myMessageLabel;
    // ... other fields ...

    private void addListeners() {
        myGoodbyeButton.addActionListener(new GoodByeActionListener());
    }

    // Inner class - defined inside the containing class
    class GoodByeActionListener implements ActionListener {
        @Override
        public void actionPerformed(final ActionEvent e) {
            myMessageLabel.setText("Goodbye!");
        }
    }
}
```

Benefits of inner classes:

1. **Organization:** Don't need 20 separate files for 20 small listener classes
2. **Access:** Inner classes have access to the private fields of the containing class

> **Student Question**
>
> Q: Does the inner class have access to ALL of the outer class's fields?
>
> A: Yes. There's no syntax to hide fields from an inner class. If you write an inner class, it has access to all private fields of the containing class. This is technically a "break" of encapsulation, but we're already inside the class we're encapsulating, so it's acceptable.

Anonymous instantiation:

Rather than storing the listener in a variable:

```
// Named instantiation
GoodByeActionListener listener = new GoodByeActionListener();
myGoodbyeButton.addActionListener(listener);

// Anonymous instantiation (more idiomatic)
myGoodbyeButton.addActionListener(new GoodByeActionListener());
```

If you're not going to reference the listener again, anonymous instantiation is cleaner.

Implementation 3: Anonymous Inner Class [55:42]

Combine the inner class concept with anonymous instantiation—define the class at the point of instantiation:

```
myWaitButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(final ActionEvent event) {
        myMessageLabel.setText("Wait! Don't go yet, there's more...");
    }
});
```

What's happening here:

- We're creating an object (`new`)
- Of an anonymous class that implements `ActionListener`
- The class definition is inline (between the braces)
- The object is immediately passed to `addActionListener()`

Pre-Java 8, this was the standard approach. It compresses the external class pattern significantly, but it's still verbose for what amounts to a single line of actual behavior.

Implementation 4: Lambda Expression [59:58]

Java 8 introduced **lambda expressions**—a way to represent behavior without all the boilerplate:

```
myLambdaButton.addActionListener(event -> myMessageLabel.setText("Java 8 is fantastic!"));
```

The syntax:

```
(parameters) -> expression
```

- `event` — The parameter (the `ActionEvent`)
- `->` — The arrow operator (yes, it's an actual Java operator)
- `myMessageLabel.setText(...)` — The body

This single line replaces the entire anonymous inner class. All that structure was just formality to execute one line of code.

Implementation 5: Method Reference [1:03:38]

If you already have a method that matches the signature of `actionPerformed`, you can pass a **reference** to it:

```
myMethodReferenceButton.addActionListener(this::methodReference);

// The referenced method
private void methodReference(final ActionEvent theEvent) {
    myMessageLabel.setText("Now that's strange looking syntax.");
}
```

The `::` operator:

This is the method reference operator. It creates a reference to an existing method.

The method must match the signature:

- Same return type (void)
- Same parameter types (ActionEvent)
- Name doesn't matter

Why Lambdas and Method References Work [1:05:06]

Remember: `ActionListener` has **exactly one abstract method**.

This makes it a **functional interface**.

```
@FunctionalInterface
public interface ActionListener {
    void actionPerformed(ActionEvent e);
}
```

Functional interface requirements:

- Exactly one abstract method
- Can have multiple default methods
- Can have multiple static methods
- Only the one abstract method counts

Why this matters:

A lambda expression represents the implementation of ONE method. If the interface had three abstract methods, how would the lambda know which one to implement?

The lambda/method reference replaces the single abstract method's implementation. The JVM creates an object implementing the interface behind the scenes.

Summary: The Evolution [1:06:57]

From most verbose to most concise:

Approach	Lines of Code	Use Case
External class	~20	Reusable across multiple classes

Approach	Lines of Code	Use Case
Inner class	~8	Multiple uses within one class
Anonymous inner class	~5	Single use, pre-Java 8
Lambda expression	1	Single use, Java 8+ (preferred)
Method reference	1	Method already exists with matching signature

Assignment 3 Requirement

For A3, you **must** use lambda expressions or method references for button listeners. Don't write anonymous inner classes, inner classes, or external classes when a lambda will do.

You'll see anonymous inner classes in older codebases. Know how to read them. But for new code, use lambdas.

Lecture Demo Code

HelloGoodByeEmpty.java

```
package edu.uw.tcss.view.first.actionlisteners;

import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.Serial;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

/**
 * Demonstrating ActionListeners.
 *
 * @author Charles Bryan
 * @version Autumn 2023
 */
public class HelloGoodByeEmpty extends JPanel {

    @Serial
    private static final long serialVersionUID = -1155574959121886543L;
```

```
    /** A button to say hello. */
    private JButton myHelloButton;

    /** A button to say goodbye. */
    private JButton myGoodbyeButton;

    /** A button to say Wait, come back. */
    private JButton myWaitButton;

    /** A button to say Lambda. */
    private JButton myLambdaButton;

    /** A button to say Method References. */
    private JButton myMethodReferenceButton;

    /** A label to display the message. */
    private JLabel myMessageLabel;

    /**
     * Initializes all of the fields.
     */
    public HelloGoodByeEmpty() {
        super();
        buildComponents();
        layoutComponents();
        addListeners();
    }

    /**
     * Instantiate the graphical components.
     */
    private void buildComponents() {
        myMessageLabel = new JLabel("Message");
        myMessageLabel.setOpaque(true);

        myHelloButton = new JButton("Say Hello");
        myGoodbyeButton = new JButton("Say Goodbye");
        myWaitButton = new JButton("Wait...!");
        myLambdaButton = new JButton("Lambda Style");
        myMethodReferenceButton = new JButton("Method Reference");
    }

    /**
     * Add Listeners to the components.
     */
    private void addListeners() {
        // Implementation 1: External class
        final ActionListener listener = new
HelloActionListener(myMessageLabel);
        myHelloButton.addActionListener(listener);

        // Implementation 2: Inner class (anonymous instantiation)
        myGoodbyeButton.addActionListener(new GoodByeActionListener());

        // Implementation 3: Anonymous inner class
        myWaitButton.addActionListener(new ActionListener() {
```

```
            @Override
            public void actionPerformed(final ActionEvent event) {
                myMessageLabel.setText("Wait! Don't go yet, there's more...");
            }
        });

        // Implementation 4: Lambda expression
        myLambdaButton.addActionListener(
            event -> myMessageLabel.setText("Java 8 is fantastic!")
        );

        // Implementation 5: Method reference
        myMethodReferenceButton.addActionListener(this::methodReference);
    }

    private void methodReference(final ActionEvent theEvent) {
        myMessageLabel.setText("Now that's strange looking syntax.");
    }

    /**
     * Lay out the components.
     */
    private void layoutComponents() {
        setLayout(new BorderLayout());

        final JPanel labelPanel = new JPanel();
        labelPanel.add(myMessageLabel);
        add(labelPanel, BorderLayout.SOUTH);

        final JPanel buttonPanel = new JPanel();
        buttonPanel.add(myHelloButton);
        buttonPanel.add(myGoodbyeButton);
        buttonPanel.add(myWaitButton);
        buttonPanel.add(myLambdaButton);
        buttonPanel.add(myMethodReferenceButton);
        add(buttonPanel, BorderLayout.CENTER);
    }

    /**
     * Creates a JFrame to demonstrate this panel.
     *
     * @param theArgs Command line arguments, ignored.
     */
    public static void main(final String[] theArgs) {

javax.swing.SwingUtilities.invokeLater(HelloGoodByeEmpty::createAndShowGui);
    }

    /**
     * Create the GUI and show it.
     */
    public static void createAndShowGui() {
        final HelloGoodByeEmpty mainPanel = new HelloGoodByeEmpty();

        final JFrame window = new JFrame("A Message");
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setContentPane(mainPanel);
```

```
        window.pack();
        window.setVisible(true);
    }

    /**
     * Inner class implementation of ActionListener.
     */
    class GoodByeActionListener implements ActionListener {
        @Override
        public void actionPerformed(final ActionEvent e) {
            myMessageLabel.setText("Goodbye!");
        }
    }
}
```

HelloActionListener.java

```
package edu.uw.tcss.view.first.actionlisteners;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JLabel;

/**
 * External ActionListener class.
 * Demonstrates the most verbose approach to event handling.
 */
public class HelloActionListener implements ActionListener {

    private final JLabel myLabel;

    public HelloActionListener(final JLabel label) {
        myLabel = label;
    }

    @Override
    public void actionPerformed(final ActionEvent event) {
        myLabel.setText("Hello World!");
    }
}
```

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