# lectures # week-5

# Week 5: Comparable, Comparator & Event-Driven Programming (Friday, February 6, 2026)

**Lecture Recording**

Watch on Panopto

**Related Assignments**

This lecture wraps up Assignment 2 Q&A and introduces Assignment 3.

## Lecture Preview [0:00]

Today's agenda:

1. **Admin updates** — Quiz 1 grades, GitHub grading issues
2. **Assignment 2 Q&A** — Testing coverage, fields, common bugs
3. **Assignment 3 introduction** — SketchPad GUI application
4. **Comparable interface** — Natural ordering for objects
5. **Comparator interface** — Custom external ordering
6. **Event-driven programming** — Paradigm shift from console applications

---

## Admin Notes [0:00]

**Quiz 1 Grades:** Class average is approximately 60%. A curve will be applied to bring the average to the low-to-mid 80s. Tests are graded but waiting on one student situation before returning.

**GitHub Grading Issues:** If you received a zero due to GitHub push issues on A1a-c, email the instructor (not Discord) to have your assignment regraded.

**Assignment 2:** Due Sunday night.

**Assignment 3:** Will be published tomorrow morning with six new guides covering GUI and event-driven programming.

---

# Assignment 2 Q&A [5:27]

## Fields in Abstract Classes [6:06]

**Student Question**

Q: Is it okay to have a lot of fields in the abstract class? Where do I find them?

A: Yes, having many fields is appropriate when representing complex state. They belong in the abstract class when shared across all subclasses.

Consider what a vehicle needs to track:

| State to Track | Fields Needed |
|---|---|
| Current position | X, Y coordinates (2 fields) |
| Starting position (for reset) | Starting X, Starting Y (2 fields) |
| Current direction | 1 field |
| Starting direction | 1 field |
| Death time (how long disabled) | 1 field |
| Current disabled count | 1 field |
| Alive/dead status | 1 field |

That's already 9+ fields just for basic vehicle state. Some values are shared across all instances of a class (use `static`), some are constant (use `static final`), but many are instance-specific.

**Student Question**

Q: Should some of those fields be static, like the vehicle's mass or death time?

A: Yes. Values that are common across all objects of a class should use the `static` modifier. Values that never change should also be `final`. Use the right modifier for each field's behavior.

## Testing Coverage Requirements [10:54]

**Student Question**

Q: Do we need 100% coverage? What counts toward coverage?

A: Yes, 100% coverage of each vehicle class. When testing Car, you're looking for 100% coverage of the Car class specifically—not Truck, not Taxi, just Car. But because you've refactored common behavior into AbstractVehicle, your Truck tests will likely cover the abstract class methods too.

**Coverage targets:**

- 100% coverage of all six vehicle classes (Car, Truck, ATV, Taxi, etc.)
- 100% coverage of AbstractVehicle (achieved through testing concrete subclasses)
- Branch coverage matters—test both true and false cases for each condition

## Branch Coverage [13:02]

**Student Question**

Q: What about branch coverage? I'm seeing zeros in the gutter for some branches.

A: If the gutter shows zero for a branch, you never put the vehicle in a situation that reached that branch. You need to test both the true and false cases for each condition.

For a method like `chooseDirection()` with if-else chains:

```
if (canGoStraight) {
    return Direction.STRAIGHT;
```

```
} else if (canGoLeft) {
    return Direction.LEFT;
} else if (canGoRight) {
    return Direction.RIGHT;
} else {
    return Direction.REVERSE;
}
```

Each branch needs both true AND false cases tested:

| Branch | True Case | False Case |
|---|---|---|
| Straight | Can go straight → goes straight | Can't go straight → doesn't |
| Left | Can go left → goes left | Can't go left → doesn't |
| Right | Can go right → goes right | Can't go right → doesn't |
| Reverse | Falls through to reverse | N/A (default case) |

**Thorough testing would mean:** For absolute completeness, you'd test every terrain type for every direction check. Can't go forward on grass? Test it. Can't go forward on trails? Test it. Can't go forward on walls? Test it. Then repeat for streets, lights, crosswalks...

For this assignment, you don't need to be that thorough. But if you decompose `isValidTerrain` into a helper method, testing it completely in one place covers it for all direction checks.

**Parameterized Tests**

JUnit 5 supports parameterized tests for testing multiple inputs with the same test logic. See the "Testing Complex Scenarios" guide, section 5.26.2 for `@ParameterizedTest` examples.

## Common Bug: Static Fields [22:08]

**Student Question**

Q: I found a funny bug—I made my taxi stop cycles static, so all taxis shared the same counter!

A: That's a classic bug. Be careful with `static` modifiers on instance state.

**Static Field Bug**

Making `taxiStopCycles` static causes ALL taxis to share the same stop counter. One taxi at a crosswalk would affect all other taxis. Remember the memory model: static variables are shared across all instances. Instance state should not be static.

---

## Assignment 3 Introduction [22:54]

Assignment 3 is **SketchPad**—a drawing application where you draw on a canvas with your mouse.

**Student Question**

Q: This looks more complex than Assignment 2...

A: It's actually simpler. The entire back-end model and GUI layout are provided. You're only adding event handlers—small snippets of code that respond to user actions.

**What's provided:**

- Complete back-end model
- Complete GUI layout (buttons, toolbar, canvas)
- All visual components positioned and styled

**What you implement:**

- Event handlers for buttons (Clear, Undo)
- Mouse listeners for drawing on the canvas
- Color chooser integration
- Line width slider integration

**Code Volume**

The amount of code is very small. Requirements 1 and 2 are essentially single lines each. If you have GUI/Swing experience, this could take 15-30 minutes. If event-driven programming is new to you, expect to spend time understanding the concepts.

### Key Terminology [25:50]

| Term | Meaning |
|---|---|
| Event handler | Language-agnostic term for code that responds to events |
| Listener | Java-specific term for event handlers |

The terms are interchangeable in practice.

### Swing Framework [27:33]

**Student Question**

Q: What's Swing?

A: Swing is the Java package/framework/library containing all the classes you need to build a GUI application. This SketchPad is written entirely in Java using Swing—no external libraries.

---

## The Comparable Interface [28:26]

Not all classes need to implement `Comparable`. A GUI panel doesn't need ordering. But **data classes**—like Student, Course, or SimpleFood—often benefit from having a natural ordering.

### Why We Need Comparable [28:26]

Primitive types have natural ordering via relational operators:

```
int num1 = 5;
int num2 = 10;
if (num1 < num2) { ... }  // Works fine
```

But objects cannot use relational operators:

```
Student obj1 = new Student("Alice");
Student obj2 = new Student("Bob");
if (obj1 < obj2) { ... }  // COMPILER ERROR!
```

The `Comparable` interface provides a contract for establishing **natural ordering** among objects of a class.

## Natural Ordering Examples [30:13]

| Class | Natural Ordering |
|---|---|
| String | Alphabetical (A before B before C) |
| BigDecimal | Numeric (1.1 before 10.1) |
| Student | Depends on requirements (ID? Last name? GPA?) |

**Natural Ordering Isn't Always Obvious**

For a Student class, what's the natural ordering? When asked in class, students suggested: GPA, last name, first name, student ID. Five different answers! Sometimes you need to consult requirements or discuss with stakeholders to determine the most common use case.

## The Comparable Interface [36:08]

```
public interface Comparable<T> {
    int compareTo(T other);
}
```

**Student Question**

Q: What's that `T` doing in the interface definition?

A: That's Java generics. The `T` is a type parameter—a placeholder for the actual class type. When you implement `Comparable<SimpleFood>`, every `T` becomes `SimpleFood`. This means your `compareTo` method receives a `SimpleFood` parameter, not `Object`, so you don't need to cast.

The generic type `T` lets you specify what type you're comparing, avoiding casts:

```
public class SimpleFood implements Comparable<SimpleFood> {
    @Override
    public int compareTo(SimpleFood other) {
        // Parameter is already SimpleFood, no casting needed
```

```
    }
}
```

If you leave off the generic type, you're using the old pre-generics `Comparable` interface where the parameter is `Object`—and you're back to casting.

## Understanding compareTo Return Values [39:04]

For `a.compareTo(b)`:

| Return Value | Meaning |
|---|---|
| `> 0` (positive) | `a` comes **before** `b` in a sorted list |
| `< 0` (negative) | `b` comes **before** `a` in a sorted list |
| `= 0` (zero) | `a` and `b` are in the **same position** |

> ⚠ **Don't Think Greater/Less Than**
>
> Don't think "a is greater than b." Think "a comes before b in the list." A lower student ID doesn't mean the student is "less than" another—it just means they come first when sorted by ID.

**What about zero?** If `compareTo` returns zero, it means the two objects should be in the same position. That's not `compareTo`'s problem to solve—the sorting algorithm decides how to handle ties.

## Implementation Example [43:38]

```
public class SimpleFood extends AbstractFood implements Comparable<SimpleFood>
{

    @Override
    public int compareTo(SimpleFood other) {
        // Delegate to String's natural ordering (alphabetical)
        return this.getName().compareTo(other.getName());
    }
}
```

Since `String` already implements `Comparable`, we can delegate to its `compareTo` method for alphabetical ordering by name.

---

# The Comparator Interface [49:52]

## When Comparable Isn't Enough [49:52]

Scenarios where you need `Comparator`:

1. **Multiple sort orders** — A food app sorted alphabetically by default, but user clicks "Calories" column header
2. **Class doesn't implement Comparable** — Developers decided no natural ordering exists
3. **External sorting** — You can't modify the class but need to sort its objects

## The Comparator Interface [50:07]

```
public interface Comparator<T> {
    int compare(T a, T b);  // Note: compare, not compareTo
}
```

Key difference from `Comparable`:

| Comparable | Comparator |
|---|---|
| `a.compareTo(b)` | `comparator.compare(a, b)` |
| Implemented inside the class | Implemented externally |
| Defines natural ordering | Defines custom ordering |

## Implementation Example [51:48]

```
public class SimpleFoodComparatorByCal implements Comparator<SimpleFood> {

    @Override
    public int compare(SimpleFood a, SimpleFood b) {
        return Integer.compare(a.getCaloriesPerServing(),
b.getCaloriesPerServing());
    }
}
```

## Using Comparators with Collections.sort() [53:50]

```
// Sort by natural ordering (Comparable)
Collections.sort(foodList);

// Sort by custom ordering (Comparator)
Collections.sort(foodList, new SimpleFoodComparatorByCal());
```

**Assignment 2 Connection**

The collision resolution modes in A2 use comparators to determine which vehicle "wins" a collision. The mass-based collision mode is essentially a comparator comparing vehicle masses.

### Will This Be Tested? [56:23]

**Student Question**

Q: Is Comparable/Comparator testable material?

A: Yes, but probably not full implementation. You should know the terminology—natural ordering, `compareTo`, comparator objects, `compare(a, b)`—and understand the differences between the two interfaces.

---

## Introduction to Event-Driven Programming [58:05]

### Console vs GUI: A Paradigm Shift [58:51]

**Console-based applications (142/143 style):**

```
Program → prompts user → User responds → Program processes → Program prompts again
```

The **program controls the flow**. You wait for it to tell you what to do. Think of the number guessing game: the program picks a number, prompts you to guess, tells you high or low, prompts again. You can only respond to what it asks.

**GUI applications:**

```
User interacts → Program responds → User interacts → Program responds
```

The **user controls the flow**. The program waits for you to do something.

> **The Inversion**
>
> This is a complete inversion of control. In console apps, you wait for the program. In GUI apps, the program waits for you. As a developer, this means you don't know what the user will do next—so you write code to handle whatever they might do.

## Events Are Happening Constantly [1:03:08]

Every mouse movement generates thousands of events:

1. Hardware registers mouse movement
2. Operating system receives events
3. JVM receives events from OS
4. Application receives events from JVM

Most events are ignored. Components only respond to events they **care about**.

## Event Handlers and Listeners [1:03:51]

When a component cares about an event:

1. You write an **event handler** (small snippet of code)
2. You **attach** (register) that handler to the component
3. When the event occurs, your code executes

**Example:** Mouse cursor changing when entering a button

- Event: Mouse entered component
- Component: The button
- Handler: Code that changes cursor appearance

Watch your mouse move over UI elements—cursor changes, highlighting appears. Those are all events being handled. But moving the mouse over most of the window? Nothing happens. Those components haven't registered handlers for mouse movement events.

## What You'll Do in Assignment 3 [1:05:34]

1. Identify which components need to respond to which events

2. Write event handlers for those specific cases
3. Register the handlers with the components

Most events will be ignored—you only write handlers for the interactions that matter for your application.

> **Coming Monday**
>
> Lambda expressions and more GUI development. All event handlers in A3 will use lambda syntax.

---

## Lecture Demo Code

**SimpleFood.java**

```
/*
 * TCSS 305 - Lecture Demo
 *
 * Demo concrete class for TDD lecture demonstration.
 */

package edu.uw.tcss.model;

import java.util.Objects;

/**
 * Represents a simple food item with standard calorie calculation.
 *
 * <p>SimpleFood calculates total calories as: {@code caloriesPerServing ×
servings}.
 * The {@code familyStyle} parameter is ignored for SimpleFood.
 *
 * @author TCSS 305 Instructors
 * @version Winter 2026
 */
public final class SimpleFood extends AbstractFood implements
Comparable<SimpleFood> {

    /**
     * Constructs a SimpleFood with the given name and calories.
     *
     * @param name the food name
     * @param caloriesPerServing calories per serving
     * @throws NullPointerException if name is null
     * @throws IllegalArgumentException if name is empty or caloriesPerServing
is negative
     */
    public SimpleFood(final String name, final int caloriesPerServing) {
```

```
        super(name, caloriesPerServing);
    }

    @Override
    public int calculateTotalCalories(final int servings, final boolean
familyStyle) {
        if (servings < 0) {
            throw new IllegalArgumentException("Servings cannot be negative");
        }
        // familyStyle is ignored for SimpleFood
        return getCaloriesPerServing() * servings;
    }

    @Override
    public String getFormattedDescription() {
        return getName() + ", " + getCaloriesPerServing() + " cal";
    }

    @Override
    public boolean equals(final Object obj) {
        if (this == obj) {
            return true;
        }
        if (!(obj instanceof SimpleFood other)) {
            return false;
        }
        return Objects.equals(getName(), other.getName())
                && getCaloriesPerServing() == other.getCaloriesPerServing();
    }

    @Override
    public int hashCode() {
        return Objects.hash(getName(), getCaloriesPerServing());
    }

    @Override
    public int compareTo(final SimpleFood other) {
        return getName().compareTo(other.getName());
    }
}
```

**SimpleFoodComparatorByCal.java**

```
package edu.uw.tcss.model;

import java.util.Comparator;

public class SimpleFoodComparatorByCal implements Comparator<SimpleFood> {

    @Override
    public int compare(final SimpleFood o1, final SimpleFood o2) {
        return Integer.compare(o1.getCaloriesPerServing(),
o2.getCaloriesPerServing());
    }
}
```

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