lectures # week-4

Week 4: A2 Introduction & Abstract Classes (Monday, January 26, 2026)

Lecture Recording

Watch on Panopto

Related Assignment

This lecture covers concepts for Assignment A2.

Lecture Preview [0:00]

Today's agenda:

1. **Quiz 1 preparation** — Test details and Q&A
2. **Assignment 2 introduction** — Demo of Road Rage simulation, architecture overview
3. **Inheritance syntax review** — Interfaces vs abstract classes, what can go inside each
4. **Constructor patterns** — How constructors chain through inheritance hierarchies

---

Admin Notes — Quiz 1 [0:51]

Test Building Status: Questions are being compiled; a solid draft will be ready by Wednesday.

DRS Students: Test approvals have been processed. Schedule your exam through DRS now.

Test Format:

- Handwritten, on paper (bring a pencil, no paper needed)
- Approximately 6–8 pages, front and back
- One handwritten note sheet allowed (typed sheets will be confiscated)
- Designed for the average student to finish with 5–10 minutes to spare

Question Types:

- Coding questions are scoped to individual methods—you won't get a blank page and "write this class"
- Class stubs may be provided where you implement multiple methods as parts of one question
- Expect up to ~50% code writing, with the balance being read-code-and-give-output, short answer, and matching
- Questions will be short answer, not paragraph-length responses

Topics Covered:

- Everything from guides: Java packages, linters/Checkstyle, interface contracts, logging, unit testing/TDD, writing JUnit 5 tests, equals/hashCode contracts, inheritance hierarchies, sealed types and records (lightly), implementing equals/hashCode/toString, BigDecimal (no BigInteger), defensive programming
- All lecture content from Weeks 1–3
- **Not covered:** Environment setup, IDE basics, Git, Checkstyle rules reference, IntelliJ inspections reference (setup/tooling references are not testable, but concepts like *what linters do* and *why we use them* are fair game)

> **Non-Native English Readers**
>
> If any question wording is unclear during the test, come up and ask for clarification. Don't assume—ask.

Assignment 2 Introduction [10:07]

Timeline: Two weeks starting now. This is intentional—A2 requires significantly more work than previous assignments.

Read the Document First: The assignment document is long. Read it today, let it sit, then re-read tomorrow before touching code.

> **This Assignment Is Different**
>
> Previous assignments could be knocked out in an afternoon. A2 cannot. You need to balance Quiz 1 prep, other coursework, *and* A2 work this week. Don't wait until after the test to start.

Testing Emphasis: This assignment requires writing tests for every class. The testing requirements are heavier than past offerings of the course. This is deliberate—QA and human-written testing skills are increasingly valuable in the industry, arguably more so than junior-level code writing.

Step-by-Step Structure:

1. **Steps 1–3:** Implement three vehicle classes from provided stubs (Truck, Bicycle, Car) with provided partial tests. Write additional tests for each.
2. **Step 4:** Refactor—identify duplicate code across the three classes and extract it into an `AbstractVehicle` class that sits between the `Vehicle` interface and the concrete classes.
3. **Steps 5+:** Implement three more vehicle classes from scratch (Taxi, ATV, Human) with full test coverage.

Testing After Refactoring: Once duplicate behavior is moved into `AbstractVehicle`, you don't need to re-test that behavior in the later vehicle classes. The existing tests from Steps 1–3 already cover it. This is one of the great benefits of having automated tests *before* refactoring—if tests that previously passed start failing after the refactor, you know you broke something.

A2 Demo: Road Rage Simulation [13:33]

The simulation is a GUI showing vehicles moving around a grid with streets, crosswalks, traffic lights, grass, trails, and walls. Each vehicle type follows its own movement rules.

Collision Modes (selectable at runtime):

- **Mass-based:** Vehicles with greater mass win collisions (default)
- **Less mass / Underdog:** Vehicles with *less* mass win—a human can disable a truck
- **Chaos:** Coin flip decides every collision outcome

Architecture: The GUI and simulation logic are provided. Students implement six model classes, all implementing the `Vehicle` interface. The interface defines the contract between the front-end/logic code and the models.

Vehicle Behavior Rules [21:29]

Each vehicle has specific rules for movement that must be translated from English requirements into code logic. Two examples discussed:

Truck:

- Can travel on streets, traffic lights, and crosswalks only (grass, trails, walls are impassable)
- When multiple directions are viable, **randomly selects** between straight, left, or right
- If all three are impassable, reverses direction
- Drives through all traffic lights without stopping
- Stops for red crosswalk lights; drives through yellow and green

Car:

- Can travel on streets, traffic lights, and crosswalks only
- **Deterministic priority:** prefers straight → left → right → reverse
- No randomness involved

> **Unique Rules Per Vehicle**
>
> Every vehicle type has its own rule set. These behavior rules are specific to each concrete class—they will *not* go in the abstract class. The abstract class holds only shared behavior.

Testing Random Behavior [23:48]

Testing the Truck's random direction selection presents a challenge: how do you assert correctness when the output is non-deterministic?

Unlike testing `calculateTotal()` where you know the exact expected result, a truck that "chose to go straight" doesn't tell you whether the random selection is working correctly. It could go straight 100 times in a row by pure chance.

Practical Approach: Run the test ~50 times. If all three viable directions (straight, left, right) are chosen at least once across those iterations, call it good. False negatives are possible but unlikely. The A2 testing guide covers this pattern in detail.

> **A2 Testing Guide**
>
> See the A2 Testing Guide for patterns on testing random behavior, setting up specific scenarios, and avoiding false positives/negatives. Mocking frameworks are *not* required for this assignment.

A2 Starter Project Notes [30:02]

When you first get the starter project from GitHub Classroom:

- **It compiles and runs**—a GUI will appear
- **It will crash almost immediately** because the stub methods return `null` or throw exceptions
- **Tests will show false positives:** Some provided tests pass on the stubs because the stub returns `false` and the test happens to expect `false` (e.g., "can I pass on a red crosswalk?" → expected: `false`, stub: `return false`)
- **Vehicle activation:** As you complete each vehicle, uncomment its line in the configuration file to add it to the simulation. This uses reflection to load classes dynamically.

Interfaces: What's Inside [32:21]

Review and expansion of what Java interfaces can contain:

Classic (Pre-Java 8):

- **Public abstract methods** — Method signatures with no implementation. The `public abstract` modifiers are implicit and redundant to write explicitly. Implementing classes must either provide implementations or be declared `abstract` themselves.
- **Static fields** — Class-level variables (not instance variables). Implicitly `public static final`. There is only one copy, shared across all implementations, accessed via the class/interface name.

Modern (Post-Java 8):

- **Static methods** — Fully implemented methods that belong to the interface itself. Like all static methods, they cannot access instance fields—there's no object context to reference.
- **Default methods** — Fully implemented methods that *are* attached to objects. They cannot directly access instance fields (the interface doesn't define any), but they *can* call the interface's abstract methods, which are guaranteed to exist on any implementing class. This enables shared behavior at the interface level.

```
public interface Point {
    int getX();  // abstract — implementing classes must define
    int getY();  // abstract — implementing classes must define

    // Default method: uses guaranteed abstract methods
    default double distanceTo(Point other) {
```

```
        int dx = this.getX() - other.getX();
        int dy = this.getY() - other.getY();
        return Math.sqrt(dx * dx + dy * dy);
    }
}
```

> **Where You'll See Default Methods**
>
> You won't implement default methods in A2, but they appear throughout the Java API—especially in functional interfaces and the Streams API. Understanding that they exist helps when reading documentation.

Abstract Classes: Filling the Gap [44:22]

Abstract classes sit in the "sweet spot" of the inheritance hierarchy—between the interface (which defines the contract) and the concrete classes (which provide full implementations).

The Problem They Solve: Consider the food example. Both `SimpleFood` and `ComboFood` have a `name` field and a `getName()` method. If we have five food types, that's duplicate code in all five classes. We'd like to "bubble up" that shared state and behavior, but the interface can't hold instance fields or implemented instance methods.

What Abstract Classes Provide:

Feature	Interface	Abstract Class	Concrete Class
Instance fields	No	**Yes**	Yes
Implemented methods	Default only	**Yes**	Yes
Abstract methods	Yes	**Yes**	No (must implement all)
Can be instantiated	No	**No**	Yes

The abstract class implements the shared parts (like `name` field and `getName()`) while leaving the differing behavior (like `calculateTotalCalories()`) abstract for concrete classes to define.

Why Not Just Use an Abstract Class? You *could* skip the interface entirely, but idiomatic Java separates the contract (interface) from partial implementation (abstract class). Client code

programs to the interface—the contract. Implementation code programs to the contract. This is cleaner design.

Sealed vs Non-Sealed Hierarchies [47:49]

The decision to seal or not seal an inheritance hierarchy is independent of whether you use abstract classes.

When to Seal:

- You control the entire hierarchy and don't anticipate external code adding subclasses
- You want compile-time exhaustiveness checking (the compiler knows all possible subtypes)

When Not to Seal:

- You're designing a hierarchy meant to be extended by code outside your control
- The compiler can't do exhaustive type checking anyway if it doesn't know all subtypes

> **Default to Sealed**
>
> Just as the recommendation is to make classes `final` by default, the recommendation is to **seal hierarchies by default**. Only unseal after deliberate design consideration about whether external extension is needed.

Constructors in Inheritance [51:56]

Three rules that apply to every Java class:

1. **Every class has a constructor.** Even if you don't write one, the compiler generates a no-argument default constructor.
2. **Every constructor calls `super()`.** Even if you don't write the call, the compiler inserts `super()` (no-arg) as the first statement.
3. **Every class has a parent.** If no `extends` clause is written, the class implicitly extends `java.lang.Object`.

Protected Constructors on Abstract Classes: Since abstract classes can't be instantiated, their constructors only exist for subclasses to call. Marking them `protected` makes this intent explicit—"this constructor exists for inheritance, period."

Common Compiler Error: If you rely on the default no-arg constructor in a subclass, but the parent class *only* has a parameterized constructor (no no-arg constructor), compilation fails. The compiler-inserted `super()` can't find a matching constructor in the parent.

```
// Parent only has this constructor:
protected AbstractFood(String theName, int theCalories) { ... }

// This child class won't compile — no matching super()
public class SimpleFood extends AbstractFood {
    // Compiler inserts: public SimpleFood() { super(); }
    // ERROR: no AbstractFood() constructor exists
}
```

Java 25: Flexible Constructor Bodies [53:54]

Prior to Java 25, `super()` was required to be the **first statement** in every constructor—no exceptions. As of Java 25, you can now place statements *before* the `super()` call.

Restriction: Pre-super statements cannot reference the child class's instance fields. You can only work with the constructor parameters.

Use Cases:

- Validate or transform constructor arguments before passing them to the parent
- Throw more specific exceptions than the parent constructor would
- Prepend prefixes or apply formatting to values before sending them up

```
public ComboFood(String theName, int theCalories,
                 int theFamilyServings, int theFamilyCalories) {
    // Java 25: statements before super() are now allowed
    // Can work with parameters, but NOT with this.myFamilyServings
    super(theName, theCalories);
    myFamilyServings = theFamilyServings;
    myFamilyCalories = theFamilyCalories;
}
```

> ⚠ **LLM Limitations with Java 25**
>
> Most AI coding assistants were trained before Java 25's release (September 2025). They will often tell you that pre-super statements are not allowed in Java. This is no longer true—but it's a good reminder to verify AI-generated answers against current documentation.

Constructor Argument Patterns [58:01]

Child constructors don't need to match the parent constructor's parameter count or types. Two common patterns:

Pattern 1 — Child has more arguments than parent:

The child class needs additional state beyond what the parent stores.

```
// Parent stores name and calories
protected AbstractFood(String theName, int theCalories) { ... }

// Child stores name, calories, PLUS family servings and family calories
public ComboFood(String theName, int theCalories,
                 int theFamilyServings, int theFamilyCalories) {
    super(theName, theCalories);  // Send shared state up
    myFamilyServings = theFamilyServings;  // Keep child-specific state
    myFamilyCalories = theFamilyCalories;
}
```

Pattern 2 — Parent has more arguments than child:

The child knows a default value that the parent needs to store but clients shouldn't need to specify.

```
// Parent stores name, calories, AND category
protected AbstractFood(String theName, int theCalories, String theCategory) {
... }

// Child doesn't expose category to clients — it's always "Simple"
public class SimpleFood extends AbstractFood {
    private static final String CATEGORY = "Simple";

    public SimpleFood(String theName, int theCalories) {
        super(theName, theCalories, CATEGORY);  // Default value sent up
    }
}
```

A2 Application: Vehicle Mass

Every vehicle has a mass (Truck: 100, Car: 50, Bicycle: 20, Taxi: 50, ATV: 40). Mass should be stored in `AbstractVehicle`, but each concrete class knows its own mass as a constant. The concrete constructor doesn't need a mass parameter from clients—it passes its constant up to the parent via `super()`.

Testing equals() Thoroughly [1:05:14]

Brief discussion on achieving thorough test coverage for `equals()` methods. As the number of fields increases, the number of test cases grows.

Minimum test cases for a two-field class (e.g., `SimpleFood` with `name` and `calories`):

Test	Expected	What It Checks
`x.equals(x)`	`true`	Reflexive property
`x.equals(null)`	`false`	Null safety
`x.equals("string")`	`false`	Different type
`x.equals(y)` where same state	`true`	Structural equality
`y.equals(x)` where same state	`true`	Symmetric property
`x.equals(a)` where different name	`false`	Field-specific check (name)
`x.equals(b)` where different calories	`false`	Field-specific check (calories)

Key Insight: You need test objects that differ in exactly *one* field to verify that each field is actually being checked by the `equals()` implementation. For classes with more fields (like `ComboFood` with four), you need more test objects to isolate each field comparison.

Lecture Demo Code

The following classes were used to demonstrate concepts in this lecture.

Food.java

```
/*
 * TCSS 305 - Lecture Demo
 */
package edu.uw.tcss.model;

/**
 * Represents a food item with calorie information.
 *
 * <p>This is a sealed interface - only {@link AbstractFood} can implement it,
 * and only {@link SimpleFood} and {@link ComboFood} can extend AbstractFood.
 *
 * @author TCSS 305 Instructors
```

```
 * @version Winter 2026
 */
public sealed interface Food permits AbstractFood {

    String getName();

    int getCaloriesPerServing();

    int calculateTotalCalories(int servings, boolean familyStyle);

    String getFormattedDescription();
}
```

AbstractFood.java

```
/*
 * TCSS 305 - Lecture Demo
 */
package edu.uw.tcss.model;

import java.util.Objects;

/**
 * Abstract base class for food items.
 *
 * @author TCSS 305 Instructors
 * @version Winter 2026
 */
public abstract sealed class AbstractFood implements Food
        permits SimpleFood, ComboFood {

    private final String myName;
    private final int myCaloriesPerServing;

    /**
     * Constructs an AbstractFood with the given name and calories.
     * Note: Constructor is protected - this class is for inheritance only.
     */
    protected AbstractFood(final String name, final int caloriesPerServing) {
        myName = Objects.requireNonNull(name, "Name cannot be null");

        if (name.isEmpty()) {
            throw new IllegalArgumentException("Name cannot be empty");
        }
        if (caloriesPerServing < 0) {
            throw new IllegalArgumentException("Calories cannot be negative");
        }

        myCaloriesPerServing = caloriesPerServing;
    }

    @Override
    public String getName() {
        return myName;
    }

    @Override
```

```
    public int getCaloriesPerServing() {
        return myCaloriesPerServing;
    }

    @Override
    public String toString() {
        return getClass().getSimpleName() + "[name=" + myName
                + ", calories=" + myCaloriesPerServing + "]";
    }
}
```

SimpleFood.java

```
/*
 * TCSS 305 - Lecture Demo
 */
package edu.uw.tcss.model;

import java.util.Objects;

/**
 * Simple food item with standard calorie calculation.
 * Uses instanceof in equals - safe because this class is final.
 *
 * @author TCSS 305 Instructors
 * @version Winter 2026
 */
public final class SimpleFood extends AbstractFood {

    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");
        }
        return getCaloriesPerServing() * servings;
    }

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

    @Override
    public boolean equals(final Object obj) {
        if (this == obj) {
            return true;
        }
        // instanceof handles null check automatically
        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());
    }
}
```

ComboFood.java

```
/*
 * TCSS 305 - Lecture Demo
 */
package edu.uw.tcss.model;

import java.util.Objects;

/**
 * Combo food item with family-style portion options.
 * Uses getClass() in equals - required because class is not final.
 *
 * @author TCSS 305 Instructors
 * @version Winter 2026
 */
public final class ComboFood extends AbstractFood {

    private final int myFamilyServings;
    private final int myFamilyCalories;

    public ComboFood(final String name, final int caloriesPerServing,
                     final int familyServings, final int familyCalories) {
        super(name, caloriesPerServing);

        if (familyServings < 0) {
            throw new IllegalArgumentException("Family servings cannot be
negative");
        }
        if (familyCalories < 0) {
            throw new IllegalArgumentException("Family calories cannot be
negative");
        }

        myFamilyServings = familyServings;
        myFamilyCalories = familyCalories;
    }

    @Override
    public int calculateTotalCalories(final int servings, final boolean
familyStyle) {
        if (servings < 0) {
            throw new IllegalArgumentException("Servings cannot be negative");
        }

        final int totalCalories;

        if (!familyStyle || myFamilyServings == 0) {
```

```
            totalCalories = getCaloriesPerServing() * servings;
        } else {
            final int completeFamilySets = servings / myFamilyServings;
            final int remainingServings = servings % myFamilyServings;

            totalCalories = (completeFamilySets * myFamilyCalories)
                    + (remainingServings * getCaloriesPerServing());
        }

        return totalCalories;
    }

    @Override
    public String getFormattedDescription() {
        final String description;

        if (myFamilyServings == 0) {
            description = getName() + ", " + getCaloriesPerServing() + " cal";
        } else {
            description = getName() + ", " + getCaloriesPerServing() + " cal
("
                    + myFamilyServings + " for " + myFamilyCalories + " cal)";
        }

        return description;
    }

    @Override
    public boolean equals(final Object obj) {
        if (this == obj) {
            return true;
        }
        // Explicit null check required when using getClass()
        if (obj == null || obj.getClass() != getClass()) {
            return false;
        }

        final ComboFood other = (ComboFood) obj;
        return Objects.equals(getName(), other.getName())
                && getCaloriesPerServing() == other.getCaloriesPerServing()
                && myFamilyServings == other.myFamilyServings
                && myFamilyCalories == other.myFamilyCalories;
    }

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

SimpleFoodTest.java

```
/*
 * TCSS 305 - Lecture Demo
 */
package edu.uw.tcss.model;
```

```java
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

/**
 * Example JUnit tests demonstrating assertion patterns.
 *
 * @author TCSS 305 Instructors
 * @version Winter 2026
 */
public class SimpleFoodTest {

    private static final String ITEM_NAME = "Apple";
    private static final int ITEM_CALORIES = 100;

    @Test
    void testConstructorWithValidArguments() {
        final SimpleFood food = new SimpleFood(ITEM_NAME, ITEM_CALORIES);

        assertAll("Test the SimpleFood constructor",
                () -> assertEquals(ITEM_NAME, food.getName(),
                        "Name should be set correctly"),
                () -> assertEquals(ITEM_CALORIES,
food.getCaloriesPerServing(),
                        "Calories should be set correctly")
        );
    }

    @Test
    void testConstructorWithNullName() {
        assertThrows(NullPointerException.class,
                () -> new SimpleFood(null, ITEM_CALORIES),
                "Constructor should throw NullPointerException for null
name");
    }

    @Test
    void testConstructorWithNegativeCalories() {
        assertThrows(IllegalArgumentException.class,
                () -> new SimpleFood(ITEM_NAME, -1),
                "Constructor should throw IllegalArgumentException for
negative calories");
    }

    @Test
    void testEqualsSymmetric() {
        final SimpleFood food1 = new SimpleFood(ITEM_NAME, ITEM_CALORIES);
        final SimpleFood food2 = new SimpleFood(ITEM_NAME, ITEM_CALORIES);

        // Use assertEquals for equality testing - better diagnostics on
failure
        assertEquals(food1, food2, "Equal food objects should be equal");
    }
}
```

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