lectures # week-2

Week 2: JUnit Testing Syntax (Wednesday, January 15, 2026)

Lecture Recording

Watch on Panopto

Related Assignment

This lecture covers concepts for Assignment 1b.

Assignment 1b Clarifications [4:57]

Critical reminder: **You are NOT implementing any model code for A1b.** Don't create `StoreCart`, `StoreItem`, or `StoreBulkItem` classes. The implementation is provided in the JAR file inside `lib/`. Your only job is writing JUnit test cases.

When you clone the repo, you should be able to immediately run `ItemOrderTest`—it will execute tests against the provided JAR.

Common Mistake from Last Quarter

Students implemented their own model classes because they "needed something to test." The testable code is already provided. You're writing tests, not implementations.

API Documentation Update: Links to API specs were added to the assignment document. These present the same requirements from the blue callouts in a familiar Javadoc format. Using them is optional—you can complete the assignment without clicking those links.

Linting in Test Code [9:09]

Two perspectives on linting test code:

1. **Relaxed view:** Test code lives outside production code; it's just there to test. Don't worry about linter warnings.
2. **Professional view:** Test code is still production-ready code. When someone else on QA or dev needs to maintain it, they should see the same conventions.

For A1b specifically, since writing tests is the entire assignment, we want **zero lint warnings** in your test code. For future assignments, linting on test code will be more lenient.

Common lint warnings you'll encounter:

- Magic numbers (numeric literals)
- Duplicate string literals
- Missing message parameters on assertions

Magic Numbers and Constants [11:01]

A **magic number** is a numeric literal in your code with no contextual meaning. When you see `5` on line 40, what does it represent? Without context, you have no idea.

The fix: Replace literal values with named constants.

```
// Bad - what does 5 mean?
new ItemOrder(item, 5);

// Good - contextual meaning
private static final int TEST_ITEM_QUANTITY = 5;
new ItemOrder(item, TEST_ITEM_QUANTITY);
```

Key concepts: literal values, contextual meaning, constants, test fixtures

Real-World Example: Sales Tax [14:01]

Imagine 30,000 lines of code for Pierce County with `0.10` scattered across 40 different places. Some represent sales tax, others represent population percentages or school ratios.

When the city council raises sales tax to 11%, a global find-and-replace would break unrelated code. With a constant:

```
private static final BigDecimal SALES_TAX = new BigDecimal("0.10");
```

You change it once, and it updates everywhere it's actually used for sales tax.

Internationalization (i18n) [15:54]

Every user-facing string in applications like IntelliJ, Minecraft, or web browsers is defined somewhere in code. If those strings are hardcoded literals:

- Porting to German requires changing the entire codebase
- You must recompile and ship a new version
- Or ship 30 different versions for 30 languages

If strings are externalized to resource files:

- Translate the resource file to any language
- User switches language at runtime
- Single codebase supports all languages

> **Assignment 1c Preview**
>
> In the GUI code for A1c, every single literal value has been pulled into a constants class. This is the professional pattern you'll follow.

Java Annotations Review [21:06]

Annotations are metadata for tools that process your code. The `@` symbol followed by an identifier tells a specific tool how to handle the annotated element.

Annotation	Tool	Purpose
`@Override`	Java Compiler	Verify method actually overrides a superclass method
`@Test`	JUnit Runner	Mark method as a test case
`@author`	Javadoc	Document code author
`@version`	Javadoc	Document code version
`@throws`	Javadoc	Document exceptions a method can throw

> **Override vs Overload**
>
> Without `@Override`, if you accidentally overload instead of override, the compiler won't catch it. With the annotation, you get a compile error—exactly what you want.

Writing JUnit Tests [25:22]

Testing Constructors Through Accessors

When testing a constructor, you can't directly access private fields (encapsulation). Instead, use accessors to verify the object was set up correctly.

```
@Test
void testConstructorWithValidArguments() {
    SimpleFoodItem item = new SimpleFoodItem(ITEM_NAME, ITEM_CALORIES);

    assertEquals(ITEM_NAME, item.getName(), "Name should be set correctly");
    assertEquals(ITEM_CALORIES, item.getCaloriesPerServing(),
                 "Calories should be set correctly");
}
```

This is **black-box testing**: you don't know or care how it's implemented internally. You only verify that input produces expected output.

Key concepts: black-box testing, test fixtures, accessors

Test Fixtures [29:05]

A **test fixture** is a constant object of the class under test with known, fixed values. Pull common test values into constants:

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

What Makes a Complete Test? [30:38]

Testing just the "happy path" isn't enough. A complete constructor test needs:

1. **Positive flow:** Does it create the object correctly with valid input?
2. **Edge cases:** Zero calories (boundary), very large numbers
3. **Exception cases:** Null values, negative numbers, empty strings

> **Why Test Zero?**
>
> Zero is critical because it's the boundary between valid (>=0) and invalid (<0). It's easy to accidentally include zero in the negative check. If zero calories is allowed (like seltzer water), test it explicitly.

Assert Methods [33:33]

assertEquals with Messages

Always use the overload with a message parameter:

```
assertEquals(expected, actual, "Description of what's being tested");
```

When tests fail, you'll see:

- The expected value
- The actual value
- Your descriptive message

Without the message, you only get the test method name—much harder to debug.

One Behavior Per Test Method [35:05]

Rule of thumb: Each test method should test ONE behavior.

Problem with multiple asserts: If the first assert fails, subsequent asserts never run. You lose visibility into other potential failures.

```
// If first assert fails, second never runs
assertEquals(expected1, actual1);
assertEquals(expected2, actual2);  // Never reached if line above fails
```

assertAll for Grouped Assertions [37:09]

JUnit 5's `assertAll` lets multiple assertions run even if some fail:

```
@Test
void testConstructorWithValidArguments() {
    SimpleFoodItem item = new SimpleFoodItem(ITEM_NAME, ITEM_CALORIES);

    assertAll("Constructor should initialize all fields",
        () -> assertEquals(ITEM_NAME, item.getName(),
                           "Name should be set correctly"),
        () -> assertEquals(ITEM_CALORIES, item.getCaloriesPerServing(),
                           "Calories should be set correctly")
    );
}
```

The lambda syntax (`() ->`) will be explained later in the quarter. For now, follow this pattern: parentheses, arrow, assertion.

When to use assertAll: Group assertions that test the same behavior. Don't use it to cram an entire class's tests into one method.

Testing Exceptions with assertThrows [47:30]

Use `assertThrows` to verify code throws the expected exception:

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

Key concepts: lambda expressions, exception class

> ⚠ **Be Specific with Exception Types**
>
> Don't use `RuntimeException.class` —be specific! If you expect `NullPointerException` but the code throws `IllegalArgumentException`, using the parent class would incorrectly pass. Check the API documentation for exactly which exception should be thrown.

Separate Tests for Each Exception Case [53:08]

For `StoreBulkItem` which throws `NullPointerException` if name, price, OR bulk price is null, you need **three separate tests**:

```
@Test
void testConstructorThrowsNPEForNullName() {
    assertThrows(NullPointerException.class,
        () -> new StoreBulkItem(null, VALID_PRICE, VALID_BULK_QTY,
VALID_BULK_PRICE));
}

@Test
void testConstructorThrowsNPEForNullPrice() {
    assertThrows(NullPointerException.class,
        () -> new StoreBulkItem(VALID_NAME, null, VALID_BULK_QTY,
VALID_BULK_PRICE));
}

@Test
void testConstructorThrowsNPEForNullBulkPrice() {
    assertThrows(NullPointerException.class,
        () -> new StoreBulkItem(VALID_NAME, VALID_PRICE, VALID_BULK_QTY,
null));
}
```

You could group these in an `assertAll`, but it's more idiomatic to write individual exception tests.

Test-Driven Development Flow [57:25]

In true TDD, you write tests before implementation:

1. Write tests → they all fail (red)
2. Implement constructor → constructor tests go green
3. Implement next method → those tests go green
4. Continue until all tests pass

For A1b, we're "cheating"—the implementation is already provided. But this is the real-world TDD workflow.

Memory Model Introduction [1:00:20]

Stack vs Heap

When a method executes, its local variables (including parameters) live in a **stack frame** on the call stack.

```
void example(int x, int y) {
    Point p1 = new Point(22, 22);
    Point p2 = new Point(x, y);
    Point p3 = p1;
    boolean eq;
}
```

Stack frame contains: `x`, `y`, `p1`, `p2`, `p3`, `eq` (6 local variables)

Object Instantiation [1:05:02]

When you call `new`:

1. The `new` operator finds space in the **heap** for the object
2. The constructor initializes the object's state
3. The memory address of that heap location is returned
4. That address is stored in the stack variable

```
Point p1 = new Point(22, 22);
// p1 doesn't contain the Point object
// p1 contains the memory address (e.g., 0xFACE) where the Point lives
```

Reference Variables [1:09:08]

When you assign one reference variable to another:

```
Point p3 = p1;
```

You're copying the **memory address**, not the object. Now `p1` and `p3` both contain `0xFACE` — they point to the **same object** in memory.

Three variables, but only two objects in the heap.

> ⚠ **The P-Word**
>
> In Java, we call these **reference variables**, not pointers. "Pointer" is a four-letter word in Java circles. They store memory references to where objects live in the heap.

Coming Friday

We'll explore what `p1 == p2`, `p1 == p3`, and `p1.equals(p2)` actually mean in terms of this memory model.

Lecture Demo Code

Food.java

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

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 {

    /**
     * Returns the name of this food item.
     *
     * @return the food name
```

```
     */
    String getName();

    /**
     * Returns the calories per serving for this food item.
     *
     * @return calories per serving
     */
    int getCaloriesPerServing();

    /**
     * Calculates the total calories for the given number of servings.
     *
     * @param servings the number of servings
     * @param familyStyle whether to use family-style portions (affects
ComboFood only)
     * @return the total calories
     * @throws IllegalArgumentException if servings is negative
     */
    int calculateTotalCalories(int servings, boolean familyStyle);

    /**
     * Returns a formatted description suitable for display.
     *
     * @return formatted description string
     */
    String getFormattedDescription();
}
```

AbstractFood.java

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

package edu.uw.tcss.model;

import java.util.Objects;

/**
 * Abstract base class for food items.
 *
 * <p>This sealed abstract class provides common implementation for
 * {@link SimpleFood} and {@link ComboFood}.
 *
 * @author TCSS 305 Instructors
 * @version Winter 2026
 */
public abstract sealed class AbstractFood implements Food
        permits SimpleFood, ComboFood {

    /** The name of this food item. */
    private final String myName;

    /** The calories per serving. */
```

```
    private final int myCaloriesPerServing;

    /**
     * Constructs an AbstractFood 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
     */
    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
 *
 * 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 {

    /**
     * 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());
    }
}
```

ComboFood.java

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

package edu.uw.tcss.model;

import java.util.Objects;

/**
 * Represents a combo food item with family-style portion options.
 *
 * <p>ComboFood offers reduced calories when ordering family-style portions.
 * When {@code familyStyle} is true, complete sets of {@code familyServings}
 * use {@code familyCalories} per set, with remaining servings using standard
calories.
 *
 * <p>Example: A food with 200 cal/serving, family size of 4 for 600 cal
total.
 * Ordering 10 servings family-style: (2 × 600) + (2 × 200) = 1600 cal.
 *
 * @author TCSS 305 Instructors
 * @version Winter 2026
 */
public final class ComboFood extends AbstractFood {

    /** The number of servings in a family portion. */
    private final int myFamilyServings;

    /** The total calories for one family portion. */
    private final int myFamilyCalories;

    /**
     * Constructs a ComboFood with the given parameters.
     *
     * @param name the food name
     * @param caloriesPerServing calories per individual serving
     * @param familyServings number of servings in a family portion
     * @param familyCalories total calories for one family portion
     * @throws NullPointerException if name is null
     * @throws IllegalArgumentException if name is empty, or any numeric value
is negative
     */
    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) {
            // Standard calculation: calories × servings
            totalCalories = getCaloriesPerServing() * servings;
        } else {
            // Family-style: apply family pricing to complete sets
            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;
        }
        if (!(obj instanceof ComboFood other)) {
            return false;
        }
        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

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

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

public class SimpleFoodTest {

    private static final String ITEM_NAME = "Apple";
    private static final int ITEM_CALORIES = 100;
    private static final SimpleFood FOOD = new SimpleFood(ITEM_NAME,
ITEM_CALORIES);

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

        final String actualName = food.getName();
        final int actualCalories = food.getCaloriesPerServing();

        assertAll("Test the SimpleFood constructor",
                () -> assertEquals(ITEM_NAME,
                                actualName,
                                "Name should be set correctly"),
                () -> assertEquals(ITEM_CALORIES,
                        actualCalories,
                        "Calories should be set correctly")
        );

    }

    @Test
    void testConstructorWithValidArgumentsEdgeCase() {
        final SimpleFood food = new SimpleFood("A", 0);

        final String actualName = food.getName();
        final int actualCalories = food.getCaloriesPerServing();

        assertAll("Test the SimpleFood constructor edge cases",
                () -> assertEquals("A",
                        actualName,
                        "Name should be set correctly with a single char
string"),
                () -> assertEquals(0,
                        actualCalories,
                        "Calories should be set correctly with 0")
        );
```

```
    }

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

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

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