lectures # week-3

Week 3: BigDecimal & Testing Equals (Friday, January 23, 2026)

Lecture Recording

Watch on Panopto

Related Assignment

This lecture covers concepts for Assignment A1c.

Lecture Preview [0:00]

Today's agenda:

1. **Quiz 1 preparation** — Topic list now available, student Q&A
2. **A1c demo** — Bulk pricing behavior walkthrough
3. **Bug fix from Wednesday** — Missing null check in `equals` method
4. **Testing equals** — When to use `assertEquals` vs `assertTrue`
5. **BigDecimal deep dive** — Creation, immutability, arithmetic, and rounding

---

Admin Notes [0:46]

Quiz 1 Placeholder: A "Test 1" assignment now appears in Canvas. This is solely a placeholder for grades—there's no Canvas quiz or submission. The actual test is in-person, one week from today.

Topic List Posted: A comprehensive list of Quiz 1 topics is now available. Key points:

- Just because something is on the list doesn't guarantee it's on the test
- The list provides high-level topics, not exact questions
- Covers three weeks of material—moving relatively quickly

Notes Sheet: One sheet of notes is allowed. Don't try to memorize—put anything you'd need to look up on your notes sheet.

Quiz 1 Q&A [2:05]

Student questions and instructor guidance for the upcoming quiz:

JUnit Testing:

- Expect questions asking you to write tests that identify bugs in provided methods
- Students historically struggle with "given this buggy method, write a test that catches the bug"

Memory Model:

- Understand object references, heap vs stack, how arguments pass by value (where values are memory addresses for objects)
- No drawing required, but understanding the underlying concepts is expected

Checkstyle Rules:

- Don't memorize exact rule names—that's what the tools are for
- Know the common rules you've encountered in A1a, A1b, A1c (e.g., instance variables start with `my`)
- Put specific rule syntax on your notes sheet if needed

Code Writing:

- Have examples ready on your notes sheet:
  - `equals` and `hashCode` implementations
  - Various JUnit assertions (`assertEquals`, `assertThrows`, `assertAll`)
  - BigDecimal arithmetic operations
- Be prepared to adapt examples to different requirements

Response Format:

- Short answer, not paragraphs
- Might be a sentence, might be a single term like "java.lang.Object"

A1c Demo: Bulk Pricing [12:51]

Quick demonstration of the bookstore application showing bulk pricing behavior:

Quantity	Membership	Price	Explanation
10	No	$9.50	$0.95 × 10 = $9.50
10	Yes	$5.00	Bulk price for exactly 10 items
11	No	$10.45	$0.95 × 11 = $10.45
11	Yes	$5.95	10 at bulk ($5.00) + 1 at regular ($0.95)

Key insight: Bulk pricing only applies to complete "buckets" of the bulk quantity. Remaining items use regular pricing. At certain quantities (like 16), buying 20 items at bulk pricing is actually cheaper than buying 16.

Single vs Multiple Return Statements [16:33]

A student asked about the convention of having a single exit point in methods versus the multiple `return` statements in our `equals` implementation.

The Traditional Rule:

- Methods should have one entry point and one exit point
- Research shows single-exit methods can lead to fewer bugs
- This is what you may have learned in TCSS 142/143

The Professional Reality:

Fast-fail returns improve readability:

```
// Fast-fail style (preferred for equals)
public boolean equals(Object obj) {
    if (this == obj) {
        return true;   // Fast exit: same reference
    }
    if (obj == null || obj.getClass() != getClass()) {
        return false;  // Fast exit: null or wrong type
    }
    // ... actual comparison logic
}
```

Single-return alternative (less readable):

```java
public boolean equals(Object obj) {
    final boolean result;
    if (this == obj) {
        result = true;
    } else if (obj == null || obj.getClass() != getClass()) {
        result = false;
    } else {
        // ... comparison logic sets result
    }
    return result;
}
```

Guidelines:

- Fast-fail exits at the top of a method: **use them**—they're clear and idiomatic Java
- Large methods with many returns scattered throughout: **refactor** into smaller methods first
- Checkstyle flags methods with 4+ returns—that's when to reconsider structure

> **Idiomatic Java**
>
> The fast-fail pattern with multiple returns is idiomatic Java for `equals` methods. What your TCSS 142 professor taught was appropriate for that level—as you grow as a developer, you'll learn when rules have nuanced exceptions.

Magic Numbers and Constants [24:52]

Q: Should we ignore magic number warnings when the number seems obvious?

No. Move toward eliminating literal values in code. Use IntelliJ's refactoring:

1. Right-click the literal → Refactor → Extract → Constant
2. IntelliJ creates the constant (but may need cleanup—often makes it `public` and puts it in the wrong place)

Naming constants: Don't name a constant `ZERO` or `FIVE`—give it contextual meaning like `NO_ITEMS` or `SINGULAR_ITEM`.

Exceptions: Checkstyle doesn't flag `0` or `1`, but consider making them constants anyway for clarity.

Protected Constructors in Abstract Classes [29:44]

Q: Why is the constructor `protected` in `AbstractFood`?

```
public abstract sealed class AbstractFood implements Food
        permits SimpleFood, ComboFood {

    protected AbstractFood(final String name, final int caloriesPerServing) {
        // ...
    }
}
```

Key points:

- Abstract classes cannot be instantiated (`new AbstractFood(...)` is a compiler error)
- Child classes must call the parent constructor—so a constructor must exist
- Making it `protected` reinforces that this class is designed for inheritance only
- Could be `public` (you still can't instantiate it), but `protected` is more intentional
- Cannot be `private`—child classes wouldn't be able to call it

Bug Fix: Null Check in equals [32:45]

The `ComboFood.equals` method from Wednesday's lecture had a bug—it didn't handle `null`:

```
// BUGGY VERSION - throws NullPointerException when obj is null
public boolean equals(final Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj.getClass() != getClass()) {  // NPE if obj is null!
        return false;
    }
    // ...
}
```

The fix—add explicit null check:

```
// FIXED VERSION
public boolean equals(final Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null || obj.getClass() != getClass()) {  // null check first!
        return false;
    }
    // ...
}
```

Why does order matter?

The `||` operator is a short-circuit operator:

- If the left side is `true`, the right side is never evaluated
- `obj == null` evaluates to `true` when null → we return `false` without calling `obj.getClass()`

> **IntelliJ Catches This**
>
> If you reverse the order (`obj.getClass() != getClass() || obj == null`), IntelliJ's static analysis flags it—the null check would be "always false" because we'd already have thrown an NPE.

Why doesn't `SimpleFood` have this bug?

```
// SimpleFood uses instanceof
if (!(obj instanceof SimpleFood other)) {
    return false;
}
```

The `instanceof` operator returns `false` for `null`—no explicit null check needed. This is safe here because `SimpleFood` is `final` (no subclasses to worry about for symmetry/transitivity).

Testing equals: assertEquals vs assertTrue [42:47]

When testing the `equals` method, which assertion should you use?

```
@Test
void testEqualsSymmetric() {
    final SimpleFood food1 = new SimpleFood("Apple", 100);
    final SimpleFood food2 = new SimpleFood("Apple", 100);

    // Option 1: assertTrue (works, but not ideal)
    assertTrue(food1.equals(food2), "Equal food objects should be equal");

    // Option 2: assertEquals (preferred)
    assertEquals(food1, food2, "Equal food objects should be equal");
}
```

Use `assertEquals` for equality testing. Two reasons:

1. **Semantic correctness:** `assertEquals` is designed for testing equality
2. **Better failure diagnostics:**
3. `assertTrue` failure shows: `Expected: true, Actual: false`

4. `assertEquals` failure shows: `Expected: [object details], Actual: [object details]` —you can see what's different

> **When to use assertTrue/assertFalse**
>
> Use these for methods that return boolean but aren't testing object equality:
>
> ```
> assertTrue(vehicle.isEnabled(), "Vehicle should be enabled");
> assertFalse(account.isLocked(), "Account should not be locked");
> ```

BigDecimal Deep Dive [49:33]

The Floating Point Problem

```
System.out.println(0.1);           // 0.1
System.out.println(0.2);           // 0.2
System.out.println(0.3);           // 0.3
System.out.println(0.1 + 0.2);     // 0.30000000000000004
System.out.println(0.1 + 0.2 == 0.3);  // false!
```

Why does this happen?

In decimal (base 10), some fractions can't be represented exactly:

- 1/2 = 0.5 ✓
- 1/4 = 0.25 ✓
- 1/3 = 0.333... (repeating)

In binary (base 2), different fractions repeat—including 1/10:

- 0.1 in decimal is a **repeating number in binary**
- Computers approximate it as closely as possible
- Arithmetic on approximations produces slightly wrong results

This isn't a Java problem—it's how IEEE floating point works in virtually all languages.

Creating BigDecimal Objects

From a literal value (use String):

```
// CORRECT - exact value
BigDecimal price = new BigDecimal("99.10");  // 99.10
```

```
// WRONG - inherits floating point imprecision
BigDecimal price = new BigDecimal(99.10);    // 99.099999999...
```

From a double variable (use valueOf):

```
double amount = 99.10;

// CORRECT - human-expected representation
BigDecimal price = BigDecimal.valueOf(amount);  // 99.1

// Note: valueOf loses trailing zeros (99.1 not 99.10)
// Use setScale if you need specific decimal places
```

From an integer (constructor is safe):

```
int quantity = 5;
BigDecimal qty = new BigDecimal(quantity);  // Safe - integers are exact
```

BigDecimal is Immutable

Common Mistake

```
BigDecimal price = new BigDecimal("99.10");
price.setScale(2, RoundingMode.HALF_EVEN);  // Returns new
object!
System.out.println(price);  // Still 99.10, not rounded!
```

Correct approach—reassign the result:

```
BigDecimal price = new BigDecimal("99.10");
price = price.setScale(2, RoundingMode.HALF_EVEN);  // Capture the new object
```

All "mutating" methods (`setScale` , `add` , `multiply` , `divide` , `subtract`) return **new** BigDecimal objects.

Method Chaining

Because methods return new objects, you can chain operations:

```
BigDecimal price = BigDecimal.valueOf(999)
        .multiply(new BigDecimal("5"))
        .setScale(2, RoundingMode.HALF_EVEN);
```

Rounding Modes

When setting scale or dividing, you must specify how to round:

```
price = price.setScale(2, RoundingMode.HALF_EVEN);
```

Mode	Description
`HALF_UP`	What you learned in elementary school (5 rounds up)
`HALF_EVEN`	Banker's rounding (5 rounds to nearest even number)

For A1c, use the rounding mode specified in the assignment (typically `HALF_EVEN`).

Why banker's rounding? Over many transactions, `HALF_UP` slightly favors one party. `HALF_EVEN` (9.5 → 10, 8.5 → 8) balances out over time.

Set Scale at the End

```
// CORRECT - full precision during calculation, round at end
BigDecimal result = price
        .multiply(quantity)
        .add(tax)
        .setScale(2, RoundingMode.HALF_EVEN);

// AVOID - rounding intermediate results loses precision
BigDecimal temp = price.multiply(quantity).setScale(2, ...);
BigDecimal result = temp.add(tax).setScale(2, ...);
```

Deprecated Constants

IntelliJ may suggest `BigDecimal.ROUND_HALF_EVEN`—note the strikethrough. These integer constants are deprecated.

Use `RoundingMode` enum instead:

```
// OLD (deprecated)
price.setScale(2, BigDecimal.ROUND_HALF_EVEN);

// NEW (correct)
price.setScale(2, RoundingMode.HALF_EVEN);
```

Key Takeaways

1. **Quiz prep:** Have code examples on your notes sheet— `equals` / `hashCode`, JUnit assertions, BigDecimal operations
2. **Fast-fail returns:** Idiomatic and readable for guard clauses; refactor if you have many scattered returns
3. **Null in equals:** Use explicit check with `getClass()`, or rely on `instanceof` (which handles null)
4. **Test equality with `assertEquals`:** Better semantics and failure diagnostics
5. **BigDecimal creation:** String constructor for literals, `valueOf` for doubles
6. **BigDecimal immutability:** Always capture the return value
7. **Rounding:** Use `RoundingMode` enum, set scale only at the end

Lecture Demo Code

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

Food.java

```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

```java
/*
 * TCSS 305 - Lecture Demo
```

```java
 */
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

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

```java
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

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

import java.util.Objects;
```

```java
/**
 * 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;

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.