lectures # week-3

Week 3: A1c Introduction & Object Methods (Wednesday, January 22, 2026)

Lecture Recording

Watch on Panopto

Related Assignment

This lecture covers concepts for Assignment A1c.

Lecture Preview [0:00]

Today's agenda:

1. **Admin** — GitHub Classroom invite workaround, Test 1 details
2. **A1c Introduction** — Demo of working solution
3. **equals() Implementation** — Two approaches: `getClass()` vs `instanceof`
4. **hashCode() Implementation** — Using `Objects.hash()`
5. **toString()** — Purpose and common misconceptions

BigDecimal Coverage

BigDecimal will be covered briefly on Friday, but this is **by design**. Read the guides and work through examples on your own—learning to work with unfamiliar APIs independently is an essential professional skill.

Admin Notes [2:03]

GitHub Classroom Invite Glitch: Some students aren't being automatically added to their A1c repositories. Workarounds:

- Check Discord announcements for detailed steps
- Look for an email invitation to accept
- DM the instructor on Discord if neither works—they can manually send the invite link

Assignment 2 Timeline: A2 will be published within the next few days (by Sunday at latest). Important notes:

- A2 has a **two-week window** for a reason—it requires significant thinking time
- Don't wait until after the test to start; balance test prep with A2 work
- **Day 1:** Read the entire assignment document multiple times, then let it sit
- **Day 2:** Read again, then start working
- You'll spend more time thinking and designing than coding

Test 1 Information [4:54]

When: One week from Friday (in-class, full 80 minutes)

Format:

- Handwritten on paper
- One sheet of notes allowed (double-sided, **must be handwritten**)
- Typed notes will be confiscated

> **Sheet of Notes Strategy**
>
> Create a draft sheet first, then a final version. By the time you finish this process, you'll have studied the material thoroughly. The sheet of notes is partly a "trick" to get you to study.

Question Types:

- Read code and explain what it does / give output
- Write code (focus on **content and semantics**, not minor syntax like missing parentheses)
- Multiple choice, fill-in-the-blank, definition matching
- Short answer (1-2 sentences)
- No true/false questions

> ⚠️ **Syntax vs Semantics**
>
> Missing a parenthesis won't cost you points. But writing `testEquals(actual, expected, "message")` instead of `assertEquals(expected, actual, "message")` **will**—that's a semantics error showing you don't know the API.

Content Coverage:

- Everything from lectures (compressed notes—like what you're reading now!—plus Panopto recordings)
- Assignment topics from A1a, A1b, A1c
- All guides associated with those assignments
- **Nothing from Week 4+** (A2 material won't be on Test 1)

What to Study:

- Lecture notes are primary—anything covered in lecture is fair game
- Guides go deeper than lectures; expect a few questions testing guide material
- Reference sections (Checkstyle reference, etc.) won't be tested in detail, but know the major rules you've corrected for
- Import statements won't be required—if you need a class, it will be provided

---

A1c Demo [22:46]

The working solution demonstrates:

- **Three campus locations** (switching clears the cart)
- **Items loaded from text file** (not hardcoded)
- **Cart functionality** — `getFormattedDescription()` displays items
- **Membership discount** — bulk pricing applies when toggled

Key Implementation Notes:

- The starter code won't compile until you create the required classes
- Stubbed classes returning `null` will cause `NullPointerException`
- **Copy your A1b test cases** into A1c and use red-to-green TDD flow
- Tests that failed due to implementation bugs in A1b should pass in A1c (your implementation shouldn't have those bugs)

> **Defensive Programming**
>
> The GUI validates input (rejects decimals, negative numbers, etc.), but you **must still code defensively** in your backend classes. Why? The frontend might change, have bugs, or be replaced entirely (e.g., with a web frontend).

The equals() Method [31:46]

Where does it come from? `java.lang.Object` — every class inherits it.

Purpose: Determine if two objects are **equal** (not just aliases).

- Default implementation (from `Object`): same as `==`, compares memory addresses
- For data classes: override to compare **state**

What state to compare? It's up to you, the developer. Usually all state, but not always. If you exclude state, have a very good reason.

Critical Rule: The equals method should **never throw an exception**.

- `null` passed in → return `false` (not `NullPointerException`)
- Wrong type passed in → return `false` (not `ClassCastException`)

Equivalence Relation Rules:

Rule	Meaning
Reflexive	`x.equals(x)` always returns `true`
Symmetric	`a.equals(b)` ↔ `b.equals(a)`
Transitive	`a.equals(b)` and `b.equals(c)` → `a.equals(c)`

> **Inheritance Breaks These Rules**
>
> Using `instanceof` in equals can break symmetry/transitivity when child classes are involved. This is why A1c uses a **sealed hierarchy** with **final classes**.

The @Override Annotation [41:09]

```
@Override
public boolean equals(Object obj) { ... }
```

Purpose: Tells the **compiler** to verify you're actually overriding a parent method.

Common Mistake — Overloading Instead of Overriding:

```
// WRONG - This overloads, doesn't override!
public boolean equals(ComboFood other) { ... }

// CORRECT - Must use Object parameter
public boolean equals(Object obj) { ... }
```

Without `@Override`, the compiler won't catch this mistake. You'll have an overloaded method that never gets called when collections use `equals()`.

Key Distinction:

- **Overriding:** Same method signature as parent (can widen visibility)
- **Overloading:** Same name, different parameters

---

equals() Implementation with getClass() [40:45]

```
@Override
public boolean equals(Object obj) {
    if (this == obj) return true;                          // Same reference
    if (obj == null || getClass() != obj.getClass())       // Type check
        return false;

    ComboFood other = (ComboFood) obj;                     // Cast
    return Objects.equals(getName(), other.getName())      // Compare state
        && getCaloriesPerServing() == other.getCaloriesPerServing()
        && myFamilyServings == other.myFamilyServings
        && myFamilyCalories == other.myFamilyCalories;
}
```

Line-by-line breakdown:

1. `this == obj` — If same reference, they're equal (reflexive property)
2. `getClass() != obj.getClass()` — Strict type check; rejects subclasses
3. **Cast** — Safe after type check; creates variable with correct type
4. **State comparison:**
5. Strings: use `Objects.equals()` (handles null safely)

6. Primitives (`int`): use `==`
7. Objects: use `Objects.equals()` or their `.equals()`

> **Accessing Private Fields**
>
> You can access `other.myFamilyServings` directly (private field) because you're **inside the class**. Private means "accessible within this class," not "only on this instance."

---

equals() Implementation with instanceof [58:43]

```
@Override
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (!(obj instanceof SimpleFood other)) return false;  // Pattern matching

    return Objects.equals(getName(), other.getName())
        && getCaloriesPerServing() == other.getCaloriesPerServing();
}
```

Pattern Matching (New Java): The `instanceof SimpleFood other` syntax:

- Checks if `obj` is a `SimpleFood`
- If true, automatically casts to variable `other`
- No separate cast line needed

When is instanceof Safe?

- When the class is `final` (no subclasses can break symmetry)
- In a **sealed hierarchy** where all permitted classes are final

---

Sealed Hierarchies and equals() [1:01:49]

Why Seal a Hierarchy?

1. **Compiler knowledge** — Compiler knows exactly which classes implement the interface
2. **Security** — Prevents unintended inheritance of library classes
3. **Safe equals()** — Can use `instanceof` without breaking equivalence relation

A1c Structure:

```
sealed interface Item permits AbstractItem
    sealed abstract class AbstractItem permits StoreItem, StoreBulkItem
        final class StoreItem
        final class StoreBulkItem
```

Rules for Sealed Hierarchies:

- Classes in a sealed hierarchy must be: `final`, `sealed` (and permit others), or `non-sealed` (reopens hierarchy)
- A1c classes are all `final` — no subclasses possible
- This guarantees `instanceof` won't break equals symmetry/transitivity

---

hashCode() Implementation [1:08:16]

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

The Rule: Include exactly the fields used in `equals()`.

- If it's in `equals()` → must be in `hashCode()`
- If it's not in `equals()` → must NOT be in `hashCode()`

> ⚠ **Common Mistake**
>
> Don't use `Objects.hashCode()` (singular) — that just calls `hashCode()` on the Objects class. Use `Objects.hash()` (no "Code").

Why use Objects.hash()?

- Already tested and proven
- Faster than hand-rolled implementations (can use lower-level optimizations)
- Consistent and correct

Don't write custom hashing algorithms— `Objects.hash()` handles it.

---

toString() Method [1:11:39]

Purpose: Return a string representation for **debugging purposes**.

> ⚠ **Not for UI**
>
> Don't use `toString()` for user interfaces (console or GUI). Create a separate method like `getDisplayString()` for UI output.

Default Implementation (if not overridden):

```
ClassName@hexHashCode
```

Example: `ComboFood@1a2b3c4d`

Common Misconception: "toString() shows the memory address"

This is **incorrect**. The hex value is the **hashCode**, not the memory address. The confusion arises because:

- The default `hashCode()` implementation uses the memory address as a **seed** for hashing
- But it doesn't return the memory address directly
- And even if it did, Java doesn't let you do anything with memory addresses

For data classes: Override `toString()` to show meaningful state for debugging.

Key Takeaways

1. **equals()** — Compare state, never throw exceptions, maintain equivalence relation
2. **@Override** — Always use it; catches overload-vs-override mistakes
3. **getClass() vs instanceof** — Both work in sealed/final hierarchies
4. **hashCode()** — Use `Objects.hash()` with the same fields as `equals()`
5. **toString()** — For debugging only, not UI display
6. **Sealed hierarchies** — Enable safe use of `instanceof` in equals

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