A1b # lectures # testing # week-2

Week 2: Object Equality (Friday, January 16, 2026)

Lecture Recording

Watch on Panopto

Related Assignment

This lecture covers concepts for Assignment A1b — testing the `equals()` and `hashCode()` contracts.

Lecture Preview [0:00]

We're one-fifth through the quarter already—two weeks down, eight to go. Today's agenda:

1. **Object equality on the whiteboard** — Finish the memory model discussion with reference vs object equality
2. **The `java.lang.Object` class** — Look at the Javadoc for `equals()` and `hashCode()`
3. **The contracts** — What these methods *mean* and what rules they must follow
4. **Testing the contracts** — How to write tests that verify correct implementation

Implementation Next Week

Today we focus on what `equals()` and `hashCode()` should *do* according to their contracts. Next week we'll look at how to *implement* them.

Admin Notes [3:01]

A1a Grading: Grades will be posted by end of weekend. The vast majority of submissions look good—most are scoring 15-16 out of 16. Feedback will appear as a plain text comment in Canvas showing test results and specific feedback.

A1c GitHub Classroom: If you had issues accepting A1c, the link has been fixed. *(Note: This was a transient issue resolved after lecture.)*

The java.lang.Object Class [10:44]

At the very top of the Java inheritance hierarchy sits `java.lang.Object`. Every single Java class ever created inherits from this class, which means every object is guaranteed to have certain methods available.

Key concepts: inheritance hierarchy, java.lang package, auto-import

The `java.lang` package is special—all classes in it are automatically imported into every Java project. You never need to write `import java.lang.String` or `import java.lang.Object`.

Because `Object` is the parent of all classes, we're guaranteed that any object in Java has:

- `toString()` — convert the object to a String representation
- `equals(Object obj)` — check if another object is "equal" to this one
- `hashCode()` — return an integer hash code for the object

> **Related Guide**
>
> Implementing equals, hashCode, and toString — How to override these methods correctly.

Reference Equality: The `==` Operator [14:02]

Before understanding `.equals()`, we need to understand what `==` actually does with reference types.

Primitives: Value Comparison

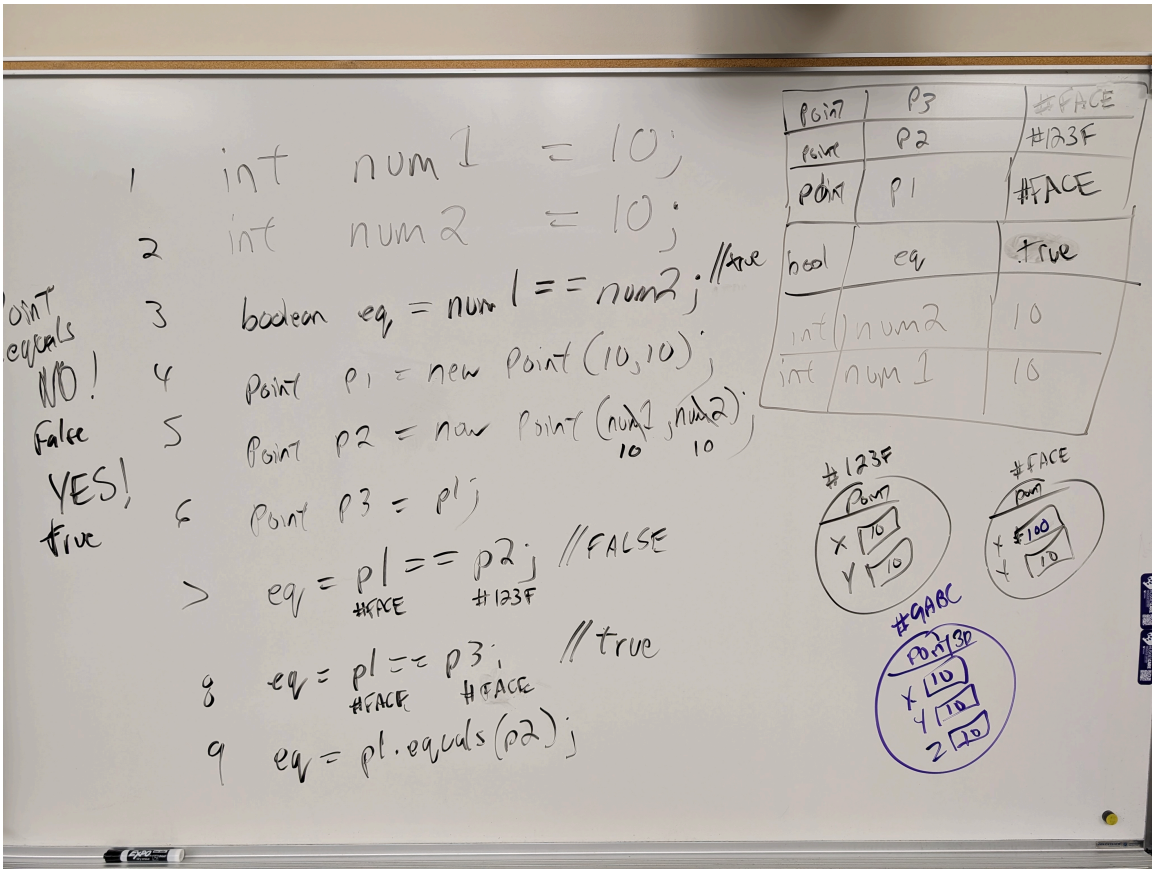
```
int num1 = 10;
int num2 = 10;
boolean eq = (num1 == num2);  // true
```

With primitives, `==` compares the **values** stored in each variable. Both contain `10`, so the result is `true`.

References: Memory Address Comparison

```
Point p1 = new Point(10, 10);
Point p2 = new Point(num1, num2);  // Also (10, 10)
Point p3 = p1;
```

What's actually stored in reference variables? The **memory address** of the object on the heap—not the object itself.



Memory model: p1 and p3 reference the same object; p2 references a different object with identical state.

In this diagram:

- `p1` stores a memory address (e.g., `#FACE`) pointing to a Point object
- `p2` stores a different memory address (e.g., `#123F`) pointing to a *different* Point object
- `p3` stores the *same* memory address as `p1` (`#FACE`)—they reference the same object

Testing Equality with `==`

```
boolean eq = (p1 == p2);  // false – different memory addresses
boolean eq = (p1 == p3);  // true – same memory address
```

The `==` operator asks: **Do these two variables reference the exact same object in memory?**

It does *not* ask whether the objects have the same state. Even though `p1` and `p2` both have `x=10` and `y=10`, they are different objects in memory, so `==` returns `false`.

> ⚠ **Your 142 Professor Was Right**
>
> You learned to never use `==` with Strings. That advice was correct, but now you understand *why*: `==` checks if two variables point to the same String object in memory, not whether the strings contain the same characters.

Object Equality: The `.equals()` Method [22:53]

If we want to compare objects by their **state** (the values of their fields), we use the `.equals()` method.

```
boolean eq = p1.equals(p2);  // true or false?
```

The answer depends on whether the Point class has overridden `equals()`.

Default Behavior (Not Overridden)

If a class doesn't override `equals()`, it inherits the implementation from `Object`, which does the exact same thing as `==`:

```
// Object's default equals implementation (simplified)
public boolean equals(Object obj) {
    return (this == obj);
}
```

So if `Point` hasn't overridden `equals()`, then `p1.equals(p2)` returns `false`—same as `==`.

Overridden Behavior (State Comparison)

If `Point` *has* overridden `equals()` to compare state, then `p1.equals(p2)` returns `true` because both points have the same `x` and `y` values.

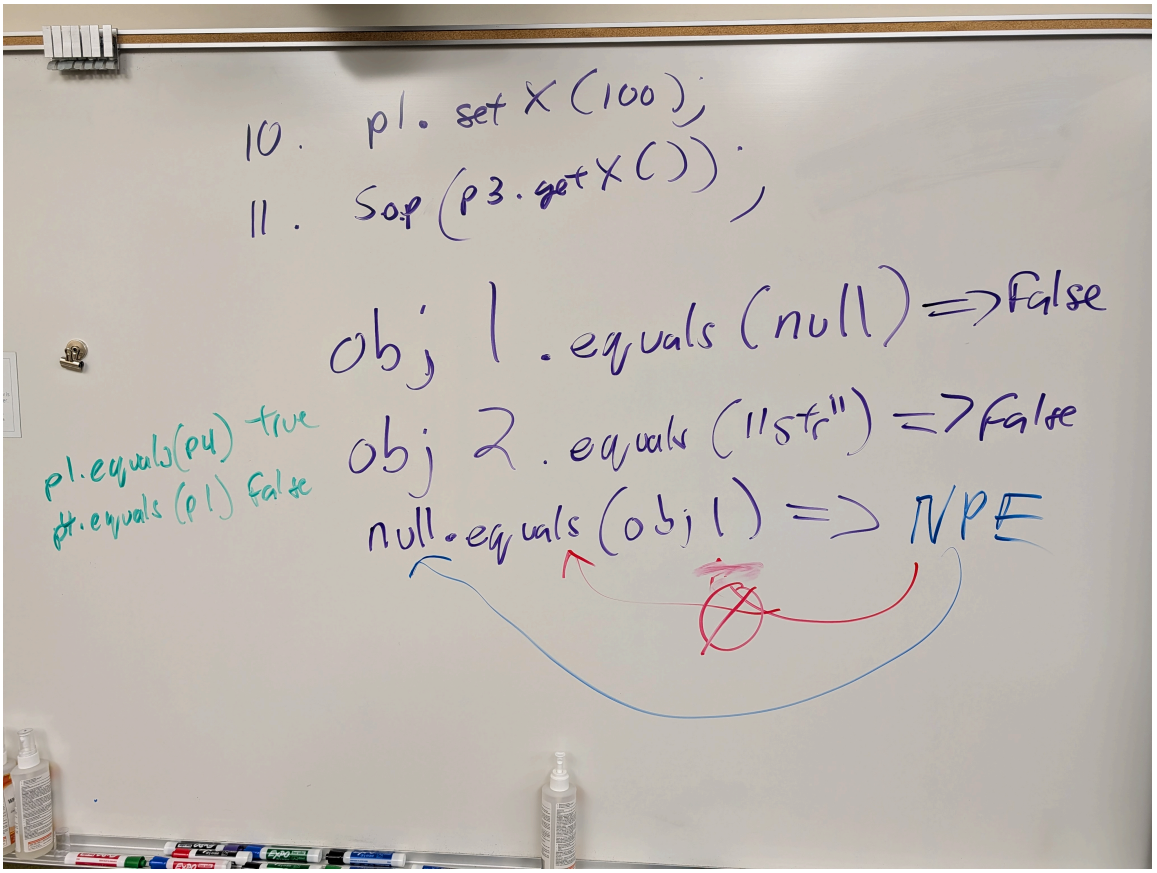
When to Override equals()

If you're writing a **data class**—a class whose objects exist to store data—you should probably implement `equals()` to compare state.

If you're writing a logic/controller class (like game logic), you probably don't need custom equality.

Aliases and Mutation [33:21]

When two variables reference the same object, they are called **aliases**.



After mutating through p1, the change is visible through p3 (same object).

```
p1.setX(100);
System.out.println(p3.getX());  // Prints 100!
```

Since `p1` and `p3` reference the same object, changes made through one variable are visible through the other.

This is a feature, not a bug—sometimes you *want* to pass an object to a method so that method can modify it. But it can also be **the source of many bugs** when mutation happens unexpectedly.

> ⚠ **Real-World Bug Story**
>
> A student's solitaire game had a bug where dragging a card made it drift further and further from the mouse cursor. After two hours of debugging, the cause was found: a mutable Point object was aliased by two different threads, both updating the position simultaneously.

Key takeaway: Prefer immutability when possible. Immutable objects eliminate an entire class of aliasing bugs.

> 📖 **Related Guide**
>
> Defensive Programming — Strategies for preventing mutation bugs.

The `equals()` Contract [41:07]

When you override `equals()`, you must follow a specific contract defined in the Object.equals() Javadoc. Breaking this contract can cause collections (like `Set` and `HashMap`) to malfunction.

No Exceptions

The `equals()` method should **never throw an exception**:

```
obj1.equals(null);        // Must return false, not throw NPE
obj2.equals("a string");  // Must return false, not throw ClassCastException
```

If passed `null` or an incompatible type, just return `false`.

> ⚠ **Watch Out for This**
>
> ```
> null.equals(obj1);  // NullPointerException!
> ```
>
> This throws NPE, but the exception comes from trying to call a method on `null`—not from the `equals()` method itself.

Reflexive

For any non-null reference `x`:

```
x.equals(x)  // Must return true
```

An object must be equal to itself.

Symmetric

For any non-null references `x` and `y`:

```
if (x.equals(y) == true) {
    y.equals(x)  // Must also return true
}
```

If A equals B, then B must equal A.

Transitive

For any non-null references `x`, `y`, and `z`:

```
if (x.equals(y) == true && y.equals(z) == true) {
    x.equals(z)  // Must also return true
}
```

If A equals B and B equals C, then A must equal C.

Consistent

Multiple calls to `equals()` with the same objects must return the same result (assuming neither object has been modified).

> ⚠️ **Mutable Objects in Collections**
>
> If you put a mutable object in a `Set`, then mutate it, the Set may now contain "duplicates" or fail to find the object. This is why keys in collections should ideally be immutable.

Inheritance Breaks Symmetry [48:20]

Here's where it gets tricky. Consider:

```
class Point3D extends Point {
    private int z;
    // ...
}
```

```
Point p1 = new Point(10, 10);
Point3D p4 = new Point3D(10, 10, 50);

p1.equals(p4);  // true? (Point only checks x and y)
p4.equals(p1);  // false! (Point3D checks x, y, AND z—p1 has no z)
```

This breaks **symmetry**! The fix is that to maintain the equivalence relation, we must break a fundamental OOP principle: a `Point3D` cannot be considered equal to a `Point`, even though a Point3D "is-a" Point.

> **Assignment Requirement**
>
> In A1b/A1c, the API explicitly states that `StoreItem` objects are **never** equal to `StoreBulkItem` objects, and vice versa—even though they're both Items. This is how we maintain symmetry.

The `hashCode()` Contract [59:13]

If you override `equals()`, you **must** also override `hashCode()`. This is required for objects to work correctly in hash-based collections like `HashMap` and `HashSet`.

What Is a Hash Code?

A hash code is a single `int` derived from an object's state. Hash collections use this integer to determine where to store and look up objects.

```
public int hashCode() {
    // Returns an int based on the object's "important state"
}
```

Important state = the same fields you use in `equals()`. If you only compare `x` and `y` in `equals()`, only use `x` and `y` to compute `hashCode()`.

The Contract

1. **Consistency**: Calling `hashCode()` multiple times on the same (unmodified) object must return the same integer.

2. **Equal objects must have equal hash codes**: If `x.equals(y)` returns `true`, then `x.hashCode()` must equal `y.hashCode()`.
3. **Unequal objects *may* have equal hash codes**: Two different objects can have the same hash code (this is called a *collision*).

> **Why Allow Collisions?**
>
> `hashCode()` returns an `int`, which has 2^{32} possible values. That's a lot, but it's finite. With just a `for` loop, you can create $2^{32} + 1$ unique Point objects—mathematically, at least two must share a hash code.

What's Testable?

Scenario	Testable?
`p1.equals(p2)` is true	**Yes** — hash codes must be equal
`p1.equals(p2)` is false	**No** — hash codes might or might not be equal
`p1.hashCode() == p2.hashCode()`	**No** — tells us nothing about equals
`p1.hashCode() != p2.hashCode()`	**Yes** — objects must NOT be equal

For A1b, the testable cases are:

1. If two objects are equal by `equals()`, their hash codes must match
2. Calling `hashCode()` twice on the same object returns the same value

---

Summary: What to Test in A1b

For the `equals()` and `hashCode()` methods:

Test	What to Verify
Reflexive	`x.equals(x)` returns `true`
Symmetric	If `x.equals(y)`, then `y.equals(x)`

Test	What to Verify
Null safety	`x.equals(null)` returns `false`
Type safety	`x.equals(differentType)` returns `false`
State comparison	Objects with same state are equal
Different state	Objects with different state are not equal
hashCode consistency	Same object returns same hash code
hashCode contract	Equal objects have equal hash codes

Related Guide

The equals and hashCode Contract — Detailed testing strategies for these methods.

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