a1b # java-fundamentals # testing

The equals and hashCode Contract

TCSS 305 Programming Practicum

This guide explains the equals/hashCode contract in Java and how to write tests that verify an implementation follows it correctly. Understanding this contract is essential for testing classes that override these methods, such as `StoreItem` and `StoreBulkItem` in Assignment 1B.

Why equals and hashCode Matter

Before we dive into the contract details, let's understand why these methods are so important.

Object Identity vs Object Equality

Java distinguishes between two concepts:

Object Identity: Are these the *same* object in memory?

```
Item item1 = new StoreItem("Pen", new BigDecimal("1.99"));
Item item2 = item1;  // Same object
System.out.println(item1 == item2);  // true - same memory location
```

Object Equality: Do these objects represent the *same thing*?

```
Item item1 = new StoreItem("Pen", new BigDecimal("1.99"));
Item item2 = new StoreItem("Pen", new BigDecimal("1.99"));
System.out.println(item1 == item2);     // false - different objects
System.out.println(item1.equals(item2)); // true - same logical value
```

The `==` operator checks identity. The `equals()` method checks equality. Without overriding `equals()`, Java uses identity as the default comparison, which is rarely what you want for data objects.

Why Collections Depend on These Methods

Here's where things get critical. Java collections like `HashSet` and `HashMap` depend entirely on `equals()` and `hashCode()` working correctly together:

```
Set<Item> cart = new HashSet<>();
Item pen1 = new StoreItem("Pen", new BigDecimal("1.99"));
Item pen2 = new StoreItem("Pen", new BigDecimal("1.99"));

cart.add(pen1);
cart.add(pen2);

// With proper equals/hashCode: size is 1 (duplicates rejected)
// With broken contract: size might be 2 (treats equal objects as different!)
System.out.println(cart.size());
```

A `HashSet` uses both `hashCode()` and `equals()` to determine if an object is already in the set:

1. First, it calls `hashCode()` to find the right "bucket"
2. Then, it calls `equals()` to check if the object matches any in that bucket

If these methods don't work together correctly, you get bizarre bugs: duplicate entries, objects that can't be found, inconsistent behavior.

> **Important**
>
> The equals/hashCode contract isn't just a Java convention—it's essential for collections to work correctly. Breaking this contract can cause subtle, hard-to-diagnose bugs in any code that uses `HashMap`, `HashSet`, or other hash-based collections.

The equals Contract

The `equals()` method must follow five specific rules, as defined in the `Object.equals()` Javadoc. These rules ensure that equality behaves logically and predictably.

1. Reflexive

An object must equal itself.

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

This seems obvious, but it's the foundation of the contract. If an object doesn't equal itself, nothing else makes sense.

2. Symmetric

If x equals y, then y must equal x.

```
x.equals(y) == y.equals(x)  // must always be true
```

Equality must work the same in both directions. If a `StoreItem` equals another `StoreItem`, that relationship must be mutual.

3. Transitive

If x equals y and y equals z, then x must equal z.

```
if (x.equals(y) && y.equals(z)) {
    x.equals(z);  // must return true
}
```

Equality must be consistent across chains of equal objects. You can't have A = B and B = C but A != C.

Inheritance Can Break Symmetry

The most common way to violate symmetry is through inheritance:

```
// Parent class checks only shared fields
class Point {
    public boolean equals(Object obj) {
        if (obj instanceof Point p) {
            return x == p.x && y == p.y;
        }
        return false;
    }
}

// Subclass adds a field and checks it
class ColorPoint extends Point {
    public boolean equals(Object obj) {
        if (obj instanceof ColorPoint cp) {
            return super.equals(cp) && color.equals(cp.color);
        }
        return false;  // Not a ColorPoint? Not equal.
    }
}

Point p = new Point(1, 2);
ColorPoint cp = new ColorPoint(1, 2, Color.RED);

p.equals(cp);   // true  - Point only checks x, y
cp.equals(p);   // false - ColorPoint requires a ColorPoint
// Symmetry violated!
```

This is why Effective Java recommends: "There is no way to extend an instantiable class and add a value component while preserving the equals contract." Prefer composition over inheritance when adding fields.

4. Consistent

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

```
x.equals(y);  // returns true
x.equals(y);  // must still return true
x.equals(y);  // must still return true
```

The result can only change if one of the objects is modified.

5. Null Handling

Comparing any non-null object to null must return false (never throw an exception).

```
x.equals(null)  // must return false, never throw
```

> **Warning**
>
> Notice that `equals(null)` returns `false` rather than throwing a `NullPointerException`. This is a rare case where we handle null gracefully instead of failing fast—because it's explicitly part of the contract.

6. Different Types

Comparing objects of unrelated types must return false:

```
storeItem.equals(someString)  // must return false
storeItem.equals(someInteger) // must return false
```

> **Note**
>
> This includes subtypes in some cases. In Assignment 1B, the specification states that a `StoreItem` and `StoreBulkItem` are **never** equal, even if they share the same name and price. This is a design decision—the types represent fundamentally different pricing models.

> **equals Must Never Throw**
>
> A properly implemented `equals()` method must **never** throw an exception. It must handle:
>
> - `null` arguments (return `false`)
> - Arguments of unexpected types (return `false`)
> - Any internal state gracefully
>
> If `equals()` throws, hash-based collections like `HashMap` and `HashSet` can fail unpredictably.

The hashCode Contract

The `hashCode()` method has a critical relationship with `equals()`. Its contract has three main rules:

1. Equal Objects Must Have Equal Hash Codes

This is the most critical rule:

```
if (x.equals(y)) {
    x.hashCode() == y.hashCode()  // MUST be true
}
```

If two objects are equal according to `equals()`, they **must** return the same hash code. Violating this breaks hash-based collections completely.

2. Unequal Objects Should Have Different Hash Codes

This is a "should" rather than a "must":

```
if (!x.equals(y)) {
    x.hashCode() != y.hashCode()  // SHOULD be true (but not required)
}
```

For performance, unequal objects *should* have different hash codes. If they don't, hash-based collections still work, but they become slow (everything ends up in the same bucket).

> **Why This Can't Be a 'Must'**
>
> Mathematically, it's impossible to guarantee different hash codes for unequal objects. A hash code is a 32-bit integer, meaning there are only about 4 billion possible values. But there are infinitely many possible objects (consider all possible `String` values). By the **pigeonhole principle**, some unequal objects *must* share a hash code.
>
> This is called a **hash collision**, and it's expected and handled by hash-based collections. However, *good* hash functions minimize collisions to maintain performance.

3. Consistency

Multiple calls to `hashCode()` on the same object must return the same value (assuming the object hasn't been modified):

```
x.hashCode();  // returns 42
x.hashCode();  // must still return 42
x.hashCode();  // must still return 42
```

The Critical Link: Override Both or Neither

Here's the fundamental rule that ties everything together:

> **The Cardinal Rule**
>
> If you override `equals()`, you **MUST** override `hashCode()`. If you override only one, you break the contract.

Why Both Must Be Overridden

Consider what happens with only `equals()` overridden:

```
public class BrokenItem {
    private String name;

    @Override
    public boolean equals(Object obj) {
        // WARNING: This is an incomplete equals() for illustration only.
        // A proper implementation should also handle null 'name' field
        // and verify the exact class type. See Effective Java Item 10.
        if (obj instanceof BrokenItem other) {
            return name.equals(other.name);
        }
        return false;
    }

    // hashCode NOT overridden - inherits Object.hashCode()
    // THIS IS THE BUG being demonstrated
}

// Now watch things break:
Set<BrokenItem> set = new HashSet<>();
BrokenItem item1 = new BrokenItem("Pen");
BrokenItem item2 = new BrokenItem("Pen");

set.add(item1);
System.out.println(set.contains(item2));  // false! Even though
item1.equals(item2) is true!
```

The `HashSet` uses the default `hashCode()` (based on memory address), which gives different values for different objects. It looks in the wrong bucket and never finds the equal object.

Testing equals: Verifying the Contract

Now we arrive at the practical focus of Assignment 1B: how do you write tests that verify an `equals()` implementation follows the contract?

Test Reflexivity

Every object must equal itself:

```
@Test
void testEqualsReflexive() {
    Item item = new StoreItem("Pen", new BigDecimal("1.99"));

    assertEquals(item, item, "An object must equal itself");
}
```

Test Symmetry

If x equals y, then y must equal x:

```
@Test
void testEqualsSymmetric() {
    Item item1 = new StoreItem("Pen", new BigDecimal("1.99"));
    Item item2 = new StoreItem("Pen", new BigDecimal("1.99"));

    // Both directions must agree
    assertEquals(item1, item2, "item1 should equal item2");
    assertEquals(item2, item1, "item2 should equal item1 (symmetry)");
}
```

Test Transitivity

If x equals y and y equals z, then x must equal z:

```
@Test
void testEqualsTransitive() {
    Item item1 = new StoreItem("Pen", new BigDecimal("1.99"));
    Item item2 = new StoreItem("Pen", new BigDecimal("1.99"));
    Item item3 = new StoreItem("Pen", new BigDecimal("1.99"));

    // Verify the chain
    assertEquals(item1, item2, "item1 should equal item2");
    assertEquals(item2, item3, "item2 should equal item3");
    assertEquals(item1, item3, "item1 should equal item3 (transitivity)");
}
```

Test Consistency

Multiple calls should return the same result:

```
@Test
void testEqualsConsistent() {
    Item item1 = new StoreItem("Pen", new BigDecimal("1.99"));
    Item item2 = new StoreItem("Pen", new BigDecimal("1.99"));
```

```
    // Call equals multiple times - must be consistent
    boolean firstCall = item1.equals(item2);
    boolean secondCall = item1.equals(item2);
    boolean thirdCall = item1.equals(item2);

    assertEquals(firstCall, secondCall, "equals must be consistent");
    assertEquals(secondCall, thirdCall, "equals must be consistent");
    assertTrue(firstCall, "Equal objects should return true");
}
```

Test Null Handling

Comparison to null must return false, not throw:

```
@Test
void testEqualsNull() {
    Item item = new StoreItem("Pen", new BigDecimal("1.99"));

    // Must return false, not throw NullPointerException
    assertFalse(item.equals(null), "equals(null) must return false");
}
```

Test with Different Types

Comparing to a completely different type should return false:

```
@Test
void testEqualsDifferentType() {
    Item item = new StoreItem("Pen", new BigDecimal("1.99"));
    String notAnItem = "Pen";

    assertFalse(item.equals(notAnItem), "Different types should not be
equal");
}
```

Test with Equal Objects

Objects with the same field values should be equal:

```
@Test
void testEqualsWithEqualObjects() {
    Item item1 = new StoreItem("Pen", new BigDecimal("1.99"));
    Item item2 = new StoreItem("Pen", new BigDecimal("1.99"));

    assertEquals(item1, item2, "Objects with same values should be equal");
}
```

Test with Unequal Objects

Objects with different field values should not be equal:

```
@Test
void testEqualsWithDifferentName() {
    Item item1 = new StoreItem("Pen", new BigDecimal("1.99"));
    Item item2 = new StoreItem("Pencil", new BigDecimal("1.99"));

    assertNotEquals(item1, item2, "Different names should not be equal");
}

@Test
void testEqualsWithDifferentPrice() {
    Item item1 = new StoreItem("Pen", new BigDecimal("1.99"));
    Item item2 = new StoreItem("Pen", new BigDecimal("2.99"));

    assertNotEquals(item1, item2, "Different prices should not be equal");
}
```

Tip

When testing for inequality, test each field independently. If a class has three fields, write separate tests for each field being different. **Keep the fields NOT under test identical** while varying only the field you're testing. This isolation helps pinpoint exactly which field comparison is broken if a test fails.

For example, when testing that different prices make items unequal, use the *same* name for both items—only the price should differ. Why do you think this must be the case?

Testing hashCode: Verifying Consistency

Testing `hashCode()` focuses on its relationship with `equals()`.

Equal Objects Must Have Equal Hash Codes

This is the most important hashCode test:

```
@Test
void testHashCodeEqualObjects() {
    Item item1 = new StoreItem("Pen", new BigDecimal("1.99"));
    Item item2 = new StoreItem("Pen", new BigDecimal("1.99"));

    // First verify they're equal
    assertEquals(item1, item2, "Precondition: objects should be equal");

    // Then verify hash codes match
```

```
    assertEquals(item1.hashCode(), item2.hashCode(),
        "Equal objects must have equal hash codes");
}
```

Hash Code Consistency

Multiple calls must return the same value:

```
@Test
void testHashCodeConsistent() {
    Item item = new StoreItem("Pen", new BigDecimal("1.99"));

    int hash1 = item.hashCode();
    int hash2 = item.hashCode();
    int hash3 = item.hashCode();

    assertEquals(hash1, hash2, "hashCode must be consistent");
    assertEquals(hash2, hash3, "hashCode must be consistent");
}
```

Note

We don't typically test that unequal objects have different hash codes because the contract only says they "should" differ, not "must." Having the same hash code for unequal objects is legal (just suboptimal for performance).

Common Testing Patterns

Here are assertion patterns you'll use frequently when testing equals/hashCode:

Using assertEquals and assertNotEquals

```
// For equal objects
assertEquals(expected, actual);
assertEquals(expected, actual, "Custom failure message");

// For unequal objects
assertNotEquals(unexpected, actual);
assertNotEquals(unexpected, actual, "Custom failure message");
```

Using assertTrue and assertFalse

```
// Direct boolean checks
assertTrue(item1.equals(item2), "Should be equal");
```

```
assertFalse(item1.equals(null), "equals(null) must be false");
```

Testing Multiple Scenarios in One Test Class

Organize tests logically:

```
class StoreItemEqualsHashCodeTest {

    // equals contract tests
    @Test void testEqualsReflexive() { /* ... */ }
    @Test void testEqualsSymmetric() { /* ... */ }
    @Test void testEqualsTransitive() { /* ... */ }
    @Test void testEqualsConsistent() { /* ... */ }
    @Test void testEqualsNull() { /* ... */ }
    @Test void testEqualsDifferentType() { /* ... */ }

    // equals behavior tests
    @Test void testEqualsWithEqualObjects() { /* ... */ }
    @Test void testEqualsWithDifferentName() { /* ... */ }
    @Test void testEqualsWithDifferentPrice() { /* ... */ }

    // hashCode tests
    @Test void testHashCodeEqualObjects() { /* ... */ }
    @Test void testHashCodeConsistent() { /* ... */ }
}
```

A Note About Records

Coming in Assignment 1C

Java **records** automatically generate `equals()` and `hashCode()` implementations based on all their components. For example:

```
public record Point(int x, int y) { }
// Automatically generates equals() and hashCode() using x and y
```

Records guarantee the equals/hashCode contract is satisfied. **You typically don't need to test the contract itself** (reflexivity, symmetry, transitivity, etc.) for records—that would be testing the Java compiler, not your code. However, you may still write tests that document expected equality behavior (e.g., "two Points with the same x and y are equal") as specification tests.

You'll learn more about records in Assignment 1C.

Summary

Concept	Key Point
equals contract	Reflexive, symmetric, transitive, consistent, null-safe
hashCode contract	Equal objects must have equal hash codes
The critical link	Override both equals and hashCode, or neither
Why it matters	Hash-based collections break if the contract is violated
Testing reflexivity	`assertEquals(item, item)`
Testing symmetry	`assertEquals(a, b)` AND `assertEquals(b, a)`
Testing null	`assertFalse(item.equals(null))`
Testing hashCode	If `a.equals(b)`, then `a.hashCode() == b.hashCode()`

Understanding and testing the equals/hashCode contract is fundamental to working with Java objects. These methods enable objects to work correctly in collections and comparisons throughout your code.

Further Reading

External Resources

- Oracle JDK: Object.equals() Javadoc - Official specification of the equals contract
- Oracle JDK: Object.hashCode() Javadoc - Official specification of the hashCode contract
- Baeldung: equals and hashCode in Java - Practical tutorial with examples
- JUnit 5 Assertions Documentation - Complete list of assertion methods

---

References

Primary Texts:

- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley. Item 10: Obey the general contract when overriding equals; Item 11: Always override hashCode when you override equals.
- Horstmann, C. S. (2022). *Core Java, Volume I: Fundamentals* (12th ed.). Oracle Press. Chapter 5: Inheritance — Sections on equals() and hashCode().

Language Documentation:

- Oracle JDK 25: Object.equals() — Official equals() specification
- Oracle JDK 25: Object.hashCode() — Official hashCode() specification
- Oracle JDK 25: Record Classes — Auto-generated equals/hashCode in records

Testing:

- JUnit 5 User Guide: Assertions — Standard assertions for testing
- JUnit 5 API: Assertions — Complete assertion method reference

---

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