a1c # java-fundamentals # oop

Implementing equals, hashCode, and toString

TCSS 305 Programming Practicum

This guide teaches you HOW to implement `equals()`, `hashCode()`, and `toString()` correctly in Java. You'll learn step-by-step patterns, common pitfalls, and when to use helper methods like `Objects.equals()` and `Objects.hash()`.

> **Required Reading**
>
> This guide assumes you've read The equals and hashCode Contract, which explains the theory behind these methods and how to test them. Read that guide first if you haven't already.

Why Implementation Matters

You already know from The equals and hashCode Contract that these methods must follow specific rules. But knowing the contract and implementing it correctly are different skills. A broken implementation can cause:

- Objects that "disappear" from `HashSet` or `HashMap`
- Duplicate entries in collections that shouldn't allow them
- Confusing debug output that makes tracking bugs harder
- Subtle bugs that only surface in production

> **Important**
>
> Joshua Bloch's *Effective Java* dedicates three full chapters to these three methods (Items 10, 11, and 12). Getting them right is essential for any class that represents a value or will be used in collections.

The java.lang Package and Object Class

The java.lang Package

The `java.lang` package is special in Java—it's automatically imported into every Java program. You never need to write `import java.lang.*;` because the compiler does it for you. This package contains fundamental classes that are essential to the language itself:

- `Object` — the root of all class hierarchies
- `String` — text representation
- `System` — system-level operations (like `System.out`)
- `Math` — mathematical functions
- Wrapper classes (`Integer`, `Double`, `Boolean`, etc.)
- Exception classes (`NullPointerException`, `IllegalArgumentException`, etc.)

The Object Class: Root of Everything

Every class in Java extends `Object`, either directly or indirectly. When you write:

```
public class StoreItem { ... }
```

The compiler treats this as:

```
public class StoreItem extends Object { ... }
```

This means every object you create inherits methods from `Object`. Three of these methods are particularly important for defining object identity and representation:

Method	Default Behavior	Why Override?
`equals(Object)`	Returns `true` only if same reference (`==`)	Define logical equality based on field values
`hashCode()`	Returns an integer derived from object identity	Support hash-based collections (`HashMap`, `HashSet`)
`toString()`	Returns `ClassName@hexHashCode`	Provide useful debugging output

The default implementations rarely do what you want for **value-based classes**—classes where two instances with the same field values should be considered equal. A `StoreItem` with name "Pen" and price $1.99 should equal another `StoreItem` with the same name and price, even if they're different objects in memory. That's why we override these methods.

The Big Picture: Three Methods, One Purpose

These three methods work together to define object identity and representation:

Method	Purpose	When Called
`equals()`	Determines if two objects are logically equivalent	Collections, assertions, explicit comparison
`hashCode()`	Returns an integer for hash-based lookup	`HashMap`, `HashSet`, `Hashtable`
`toString()`	Returns a human-readable string for debugging	Logging, debugging, error messages

The Critical Rule

If you override one of `equals()` or `hashCode()`, you **MUST** override both. The `toString()` override is technically optional but strongly recommended for any class you'll debug.

Step-by-Step: Implementing equals()

Let's implement `equals()` for a `StoreItem` class with `name` (String) and `price` (BigDecimal) fields.

Step 1: Handle Identity Check

First, check if the argument is the same object (identical reference):

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;  // Same object in memory
    }
    // ... more checks
}
```

This is an optimization—if it's the same object, we don't need field comparisons.

The @Override Annotation

Always use the `@Override` annotation when overriding methods from a parent class or interface:

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

This annotation tells the compiler: "I intend to override a method from my superclass." The compiler then verifies that you're actually overriding something. If you make a mistake—like using the wrong parameter type—the compiler catches it:

```
// WRONG: This is overloading, not overriding!
// Without @Override, this compiles but doesn't work correctly
public boolean equals(StoreItem other) { ... }

// With @Override, the compiler catches the mistake:
@Override  // ERROR: Method does not override method from its
superclass
public boolean equals(StoreItem other) { ... }
```

The `@Override` annotation costs nothing and catches subtle bugs. Use it every time you override a method.

Step 2: Handle Null

Return `false` for null arguments (never throw an exception):

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null) {
        return false;  // null is never equal to anything
    }
    // ... more checks
}
```

Step 3: Check Type — getClass() vs instanceof

This is where things get interesting. You have two choices:

Option A: `getClass()` — Strict Type Matching

```
if (getClass() != obj.getClass()) {
    return false;  // Different types are never equal
}
```

Option B: `instanceof` — Allows Subclass Matching

```
if (!(obj instanceof StoreItem other)) {
    return false;  // Not a StoreItem or subclass
}
```

> **Which Should You Use?**
>
> **Use `getClass()` when:**
>
> - Different subclasses should NEVER be equal to each other
> - You want the strictest possible type checking
> - You're following the guidance in *Effective Java* for classes with value fields in subclasses
>
> **Use `instanceof` when:**
>
> - You're implementing a `final` class (no subclasses possible)
> - You want to allow subclass instances to be equal to parent instances (rare)
> - The specification explicitly states `instanceof` behavior

For Assignment 1C, the specification states that `StoreItem` and `StoreBulkItem` are **NEVER** equal, even if they share the same name and price. Use `getClass()` to enforce this:

```
// A StoreItem and StoreBulkItem are never equal
StoreItem item = new StoreItem("Pen", new BigDecimal("1.99"));
StoreBulkItem bulk = new StoreBulkItem("Pen", new BigDecimal("1.99"), 10, new
BigDecimal("15.00"));

item.equals(bulk);  // Must be false — different types
bulk.equals(item);  // Must be false — different types
```

Gen AI & Learning: Implementation Patterns

AI assistants can generate `equals()` and `hashCode()` implementations quickly. However, generated code may not match your assignment's specific requirements. For example:

- AI might use `instanceof` when your spec requires `getClass()`
- AI might include different fields than what's specified
- AI might use a different `toString()` format than required

Always compare generated code against your specification. For Assignment 1C, the spec explicitly states that `StoreItem` and `StoreBulkItem` are never equal—this requires `getClass()`, not `instanceof`. Use AI to understand the patterns, then adapt to match your requirements exactly.

Step 4: Cast and Compare Fields

After type checking, cast the object and compare all relevant fields:

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }

    StoreItem other = (StoreItem) obj;
    return Objects.equals(myName, other.myName)
        && Objects.equals(myPrice, other.myPrice);
}
```

Using Objects.equals()

Always use `Objects.equals(a, b)` instead of `a.equals(b)` for comparing fields. The `Objects.equals()` method handles null safely:

```
// DANGEROUS: throws NullPointerException if myName is null
myName.equals(other.myName)

// SAFE: handles null gracefully
Objects.equals(myName, other.myName)
```

`Objects.equals(null, null)` returns `true`. `Objects.equals(null, "something")` returns `false`. No exceptions.

Primitives vs Objects

Use `==` for primitive types (`int`, `double`, `boolean`, etc.) and `Objects.equals()` for object references:

```
// For primitives: use ==
myQuantity == other.myQuantity  // int comparison

// For objects: use Objects.equals()
Objects.equals(myName, other.myName)    // String comparison
```

While `Objects.equals()` technically works with primitives, it forces unnecessary **boxing**—converting the primitive to its wrapper class (`int` → `Integer`). This creates garbage objects and hurts performance. Stick with `==` for primitives.

Complete equals() Implementation

Here's the complete pattern:

```
@Override
public boolean equals(Object obj) {
    // Step 1: Identity check
    if (this == obj) {
        return true;
    }
    // Step 2 & 3: Null and type check
    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }
    // Step 4: Cast and compare fields
    StoreItem other = (StoreItem) obj;
    return Objects.equals(myName, other.myName)
        && Objects.equals(myPrice, other.myPrice);
}
```

Step-by-Step: Implementing hashCode()

Once you have `equals()`, you MUST implement `hashCode()` to maintain the contract: **equal objects must have equal hash codes.**

The Objects.hash() Shortcut

Java provides `Objects.hash()` to compute hash codes from multiple values:

```java
@Override
public int hashCode() {
    return Objects.hash(myName, myPrice);
}
```

This is clean, readable, and correct. Use the same fields that you use in `equals()`.

> **Include All Equality Fields**
>
> Every field used in `equals()` must be used in `hashCode()`. If you forget a field, equal objects might have different hash codes, breaking the contract.
>
> ```java
> // BAD: Missing myPrice - breaks the contract!
> @Override
> public int hashCode() {
>     return Objects.hash(myName);  // WRONG - equals uses both
> fields
> }
>
> // GOOD: Uses both fields
> @Override
> public int hashCode() {
>     return Objects.hash(myName, myPrice);
> }
> ```

Alternative: Manual Hash Code Calculation

For performance-critical code, you can compute hash codes manually:

```java
@Override
public int hashCode() {
    int result = 17;  // Start with a non-zero prime
    result = 31 * result + (myName == null ? 0 : myName.hashCode());
    result = 31 * result + (myPrice == null ? 0 : myPrice.hashCode());
    return result;
}
```

The number 31 is a standard choice because it's prime and `31 * x` can be optimized by the JVM to `(x << 5) - x`.

> **Note**
>
> For most applications, `Objects.hash()` is perfectly fine. The manual approach is only needed when profiling reveals hash code computation as a bottleneck—which is rare.

Understanding toString()

What is toString()?

The `toString()` method returns a string representation of an object. It's defined in `java.lang.Object`, which means every Java object has this method—even if you don't write one.

Why Does toString() Exist?

Java needs a way to convert any object to text. This happens constantly:

- When you concatenate an object with a string: `"Item: " + item`
- When you print an object: `System.out.println(item)`
- When you log an object: `logger.info("Processing: " + item)`
- When debugging in an IDE: the debugger calls `toString()` to display object values

Without `toString()`, Java wouldn't know how to represent your custom objects as text.

The Default toString() Behavior

If you don't override `toString()`, you inherit the default implementation from `Object`:

```
// Object.toString() returns something like:
"StoreItem@4e25154f"
```

This format is `ClassName@hexHashCode`—the class name followed by `@` and the hash code in hexadecimal.

> **Common Misconception**
>
> The default `toString()` does **NOT** return the memory address. Some instructors incorrectly teach this. It returns the **hash code** in hexadecimal format. While the default `hashCode()` implementation may be *based on* the memory address, the two are not the same thing. The hash code is an integer; the memory address is a pointer. After garbage collection moves objects in memory, the hash code stays the same but the memory address changes.

Why Override toString()?

The default output is useless for debugging. Compare:

```
// Default toString() – unhelpful
Cart contains: model.StoreItem@4e25154f

// Custom toString() – immediately useful
Cart contains: StoreItem[name='Computer Science Pen', price=2.00]
```

A well-implemented `toString()` lets you see an object's state at a glance, making debugging dramatically easier.

Step-by-Step: Implementing toString()

The `toString()` method serves debugging. A good implementation makes problems visible at a glance.

The Debugging Purpose

Consider this debug output when something goes wrong:

```
// BAD: Default toString()
Cart contains: model.StoreItem@4e25154f

// GOOD: Custom toString()
Cart contains: StoreItem[name='Computer Science Pen', price=2.00]
```

Which would you rather debug with?

Recommended Format

Use the "class name with bracketed properties" format:

```
@Override
public String toString() {
    return "StoreItem[name='" + myName + "', price=" + myPrice + "]";
}
```

Output: `StoreItem[name='Computer Science Pen', price=2.00]`

toString() vs getFormattedDescription()

In Assignment 1C, you implement both `toString()` and `getFormattedDescription()`. These serve different purposes:

Method	Audience	Purpose	Example Output
`toString()`	Developers	Debugging, logging, error messages	`StoreItem[name='Pen', price=2.00]`
`getFormattedDescription()`	Users	Display in GUI, receipts, reports	`Computer Science Pen, $2.00`

Don't Confuse These Methods

- `toString()` is for **debugging** — include the class name and all field values
- `getFormattedDescription()` is for **display** — format for end users with currency symbols, proper spacing, etc.

```
// toString() - for developers debugging
"StoreItem[name='Computer Science Pen', price=2.00]"

// getFormattedDescription() - for users seeing the GUI
"Computer Science Pen, $2.00"
```

Common Mistakes and How to Avoid Them

Mistake 1: Using the Wrong Method Signature

```
// WRONG: This is overloading, not overriding!
public boolean equals(StoreItem other) {
    // ...
}

// CORRECT: Parameter must be Object
@Override
public boolean equals(Object obj) {
    // ...
}
```

The `@Override` annotation catches this mistake at compile time. Always use it.

Mistake 2: Forgetting to Override hashCode()

```
// BROKEN: equals() overridden but hashCode() is not
@Override
public boolean equals(Object obj) {
```

```
    // ... proper implementation
}

// Missing hashCode() - objects will "disappear" from HashMaps!
```

If you override `equals()`, you MUST override `hashCode()`.

Mistake 3: Using Incorrect Field Comparison for BigDecimal

```
// DANGEROUS with BigDecimal: 2.0 and 2.00 have different scales
myPrice.equals(other.myPrice)

// This works correctly because you're comparing the values as stored
Objects.equals(myPrice, other.myPrice)
```

BigDecimal Equality Quirk

`BigDecimal` considers scale when comparing with `equals()`: `new BigDecimal("2.0").equals(new BigDecimal("2.00"))` returns `false` because the scales differ (1 vs 2).

For value comparison ignoring scale, use `compareTo()`: `a.compareTo(b) == 0`. However, for `equals()` and `hashCode()` in our context, we want exact field equality, so `Objects.equals()` is appropriate.

Mistake 4: Including Mutable Fields That Change

```
// DANGEROUS: If quantityInCart changes after adding to a HashSet,
// the object becomes "lost" - its hash code changed!
@Override
public int hashCode() {
    return Objects.hash(myName, myPrice, quantityInCart);  // BAD
}
```

Only include fields that define the object's identity—not mutable state like cart quantities.

Mistake 5: Allowing Cross-Type Equality When It Shouldn't

```
// WRONG for Assignment 1C: instanceof allows StoreBulkItem to match
if (!(obj instanceof StoreItem other)) {
    return false;
}

// CORRECT: getClass() ensures exact type match
if (obj == null || getClass() != obj.getClass()) {
```

```
        return false;
    }
```

The specification states `StoreItem` and `StoreBulkItem` are never equal—use `getClass()` to enforce this.

Mistake 6: Not Using @Override Annotation

```
// No annotation — typos go unnoticed!
public boolean equals(Object o) {
    // ...
}

// With annotation — compiler catches typos
@Override
public boolean equals(Object obj) {
    // ...
}
```

Always use `@Override`. It costs nothing and catches mistakes.

The Objects Utility Class

The `java.util.Objects` class (note the plural—not `Object`) is a utility class introduced in Java 7. It provides static helper methods for common operations on objects, particularly null-safe operations.

Unlike `java.lang.Object` (which every class extends), `java.util.Objects` is a helper class you explicitly use. It cannot be instantiated—all its methods are static.

Why does this class exist? Before Java 7, developers wrote the same null-checking boilerplate over and over. The `Objects` class standardizes these patterns:

Method	Purpose	Use Case
`Objects.equals(a, b)`	Null-safe equality check	Field comparison in `equals()`
`Objects.hash(values...)`	Compute hash code from multiple values	`hashCode()` implementation
`Objects.requireNonNull(obj)`	Throw NPE if null	Constructor validation

Method	Purpose	Use Case
`Objects.toString(obj, default)`	Null-safe string conversion	Defensive `toString()`

Import it at the top of your class:

```
import java.util.Objects;
```

Summary

Concept	Key Point
equals() pattern	Identity check -> null check -> type check -> field comparison
getClass() vs instanceof	Use `getClass()` when different subclasses should never be equal
Objects.equals()	Null-safe field comparison—always use it
Objects.hash()	Simple way to compute hash codes from multiple fields
hashCode() rule	Use exactly the same fields as equals()
toString() purpose	Debugging—show class name and all relevant field values
getFormattedDescription()	User display—format for human readability
StoreItem vs StoreBulkItem	Never equal to each other—different types by design

---

Further Reading

External Resources

- Oracle JDK 25: Objects Class - The Objects utility class with equals(), hash(), and requireNonNull()
- Oracle JDK 25: Object.equals() - The equals() contract specification
- Oracle JDK 25: Object.hashCode() - The hashCode() contract specification
- Baeldung: Java equals() and hashCode() Contracts - Practical tutorial with examples

---

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; Item 12: Always override toString.
- Horstmann, C. S. (2022). *Core Java, Volume I: Fundamentals* (12th ed.). Oracle Press. Chapter 5: Inheritance — Sections on Object: The Cosmic Superclass.

Language Documentation:

- Oracle JDK 25: Object — The root class defining equals(), hashCode(), and toString()
- Oracle JDK 25: Objects — Utility methods for object operations

Related Guide:

- The equals and hashCode Contract — Understanding the theory and testing these implementations

---

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