a1b # testing

Writing JUnit 5 Tests

TCSS 305 Programming Practicum

This guide teaches you how to write effective unit tests using JUnit 5, Java's standard testing framework. You'll learn test structure, assertions, exception testing, and best practices for writing tests that verify behavior from API specifications. This guide builds on concepts introduced in the Introduction to Unit Testing guide.

Why Write Tests?

Before diving into the mechanics of writing tests, let's understand why tests matter. This isn't just academic—it fundamentally changes how you approach software development.

Catching Bugs Early

The earlier you find a bug, the cheaper it is to fix. Consider this progression:

When Found	Cost to Fix	Context
While writing code	Minutes	You're already looking at the code
During code review	Hours	Someone else must understand your intent
During QA testing	Days	Bug must be reproduced, located, fixed, re-tested
After release	Weeks	Customer support, reputation damage, hotfixes

A bug caught while writing code might take 5 minutes to fix. That same bug discovered by a customer might require days of investigation and an emergency patch. **Tests are your first line of defense.**

Confidence in Your Code

How do you *know* your code works? "I ran it and it seemed fine" is not a reliable answer. When all tests pass, you have *documented proof* that specific behaviors work correctly.

This confidence extends to making changes:

- **Without tests:** "I'm afraid to change this—it might break something."
- **With tests:** "I'll change it and run the tests. If they pass, I didn't break anything."

Test-Driven Development (TDD)

TDD inverts the traditional development process: **write tests first, then write code to make them pass.**

```
Traditional: Write code → Hope it works → Maybe test later
TDD:         Write tests → Watch them fail → Write code → Watch them pass
```

Why write tests first?

1. **Forces you to think about behavior** before implementation details
2. **Defines "done"**—when tests pass, you're done
3. **Prevents over-engineering**—you write just enough code to pass tests
4. **Creates comprehensive test coverage** automatically

> **Important**
>
> In Assignment 1b, you practice TDD by writing tests against an API specification *before* you see any implementation. Your tests define what "correct behavior" means, then verify whether the provided library meets that definition.

> **Guide**
>
> Test-Driven Development — Deeper exploration of TDD philosophy and the red-green-refactor cycle.

JUnit 5 Test Class Structure

Every JUnit 5 test class follows a consistent structure. Understanding this structure helps you read existing tests and write your own.

Basic Structure

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class StoreItemTest {  // Note: class is NOT public

    @Test
    void testGetNameReturnsCorrectValue() {
        // Arrange - set up the test
        Item item = new StoreItem("Pen", new BigDecimal("1.99"));

        // Act - call the method being tested
        String name = item.getName();

        // Assert - verify the result
        assertEquals("Pen", name);
    }
}
```

Key Elements

Element	Purpose	Convention
`import static`	Allows `assertEquals()` instead of `Assertions.assertEquals()`	Always use static imports for assertions
Class name	Identifies what's being tested	`<ClassName>Test` (e.g., `StoreItemTest`)
Class visibility	Package-private (no `public`)	JUnit 5 doesn't require public test classes
`@Test` annotation	Marks a method as a test	Required on every test method
Test method names	Describe what's being tested	Descriptive names explaining the behavior
Method visibility	Package-private (no `public`)	JUnit 5 doesn't require public test methods

The Arrange-Act-Assert Pattern

Every test follows this universal structure:

Arrange: Set up the test data, create objects, establish preconditions.

Act: Call the method you're testing. This is usually a single line.

Assert: Verify the result matches expectations.

```
@Test
void testCalculateTotalWithQuantityOfTen() {
    // Arrange
    Item item = new StoreItem("Pen", new BigDecimal("2.50"));
    int quantity = 10;

    // Act
    BigDecimal total = item.calculateTotal(quantity, false);

    // Assert
    assertEquals(new BigDecimal("25.00"), total);
}
```

Tip

Keep each phase clearly separated. When tests fail, clear structure helps you quickly identify whether the problem is in setup, execution, or verification.

Assertions: The Heart of Testing

Assertions verify that your code behaves correctly. JUnit 5 provides many assertion methods, each designed for specific verification scenarios.

Essential Assertions

```
import static org.junit.jupiter.api.Assertions.*;
```

Assertion	Use When	Example
`assertEquals(expected, actual)`	Comparing values	`assertEquals("Pen", item.getName())`
`assertNotEquals(a, b)`	Values must differ	`assertNotEquals(item1, item2)`

Assertion	Use When	Example
`assertTrue(condition)`	Condition must be true	`assertTrue(item.getPrice().compareTo(BigDecimal.ZERO) >= 0)`
`assertFalse(condition)`	Condition must be false	`assertFalse(item.equals(null))`
`assertNull(value)`	Value must be null	`assertNull(optionalItem.orElse(null))`
`assertNotNull(value)`	Value must not be null	`assertNotNull(item.getName())`
`assertThrows(exception, lambda)`	Code must throw exception	See section below
`assertSame(expected, actual)`	Must be same object (identity)	`assertSame(item, cart.getItem(0))`
`assertNotSame(a, b)`	Must be different objects	Rarely used in practice
`assertAll(executables...)`	Verify multiple related properties	`assertAll(() -> assertEquals(...), () -> assertEquals(...))`

Assertion Messages

All assertions accept an optional message as the last parameter. This message appears when the assertion fails, helping you understand what went wrong:

```
// Without message - failure shows generic info
assertEquals("Pen", item.getName());

// With message - failure explains the context
assertEquals("Pen", item.getName(),
    "getName() should return the name passed to constructor");
```

> **Note**
>
> Assertion messages are optional but helpful, especially for complex tests. Use them when the assertion's purpose isn't obvious from context.

> **TCSS 305 Requirement**
>
> For assignments in this course, **all assertions must include a descriptive message**. This is a grading requirement that builds good habits. In professional practice, trend toward including messages rather than omitting them—future you (and your teammates) will thank you when debugging a failing test at 2 AM.

assertEquals with Objects

`assertEquals` uses the `equals()` method to compare objects. For objects where `equals()` compares values (like `String`, `BigDecimal`, and properly-implemented domain objects), this works as expected:

```
// String comparison
assertEquals("Pen", item.getName());

// BigDecimal comparison
assertEquals(new BigDecimal("1.99"), item.getPrice());

// Custom object comparison (if equals() is properly implemented)
assertEquals(expectedItem, actualItem);
```

> **BigDecimal Equality**
>
> `BigDecimal` `equals()` considers scale: `new BigDecimal("1.99")` does NOT equal `new BigDecimal("1.990")`. See the BigDecimal section below for details.

Don't Use assertTrue/assertFalse for Equality

You might be tempted to write `assertTrue(item1.equals(item2))` since `equals()` returns a boolean. **Don't do this.** Use `assertEquals(item1, item2)` instead.

Why? When `assertEquals` fails, it shows you *both* values:

```
expected: <StoreItem[name=Pen, price=1.99]>
but was:  <StoreItem[name=Pen, price=2.99]>
```

When `assertTrue` fails, you only see:

```
expected: <true> but was: <false>
```

The first failure message helps you debug; the second tells you nothing useful.

Grouped Assertions with assertAll

When a test has multiple assertions, JUnit stops at the first failure. This means you might fix one failure only to discover another:

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

    // If getName() fails, we never learn about getPrice()
    assertEquals("Pen", item.getName(), "Name should match");
    assertEquals(new BigDecimal("1.99"), item.getPrice(), "Price should
match");
}
```

`assertAll` executes **all** assertions regardless of failures, giving you the complete picture:

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

    assertAll("StoreItem properties",
        () -> assertEquals("Pen", item.getName(), "Name should match"),
        () -> assertEquals(new BigDecimal("1.99"), item.getPrice(), "Price
should match")
    );
}
```

If both assertions fail, you see both failures in one test run.

Failure Output Comparison

Without assertAll — stops at first failure:

```
org.opentest4j.AssertionFailedError: Name should match
expected: <Pen>
but was: <Pencil>
```

You fix `getName()`, run again, and *then* discover:

```
org.opentest4j.AssertionFailedError: Price should match
expected: <1.99>
but was: <2.99>
```

With assertAll — reports all failures at once:

```
org.opentest4j.MultipleFailuresError: StoreItem properties (2 failures)
    org.opentest4j.AssertionFailedError: Name should match ==> expected: <Pen>
but was: <Pencil>
    org.opentest4j.AssertionFailedError: Price should match ==> expected:
<1.99> but was: <2.99>
```

One test run shows everything that's wrong.

When to Use assertAll

Good candidates for assertAll:

- Testing multiple properties of the same object after construction
- Verifying several related outputs from a single operation
- Testing equals/hashCode consistency together

When NOT to use assertAll:

- When assertions have dependencies (if assertion 1 fails, assertion 2 is meaningless)
- For unrelated behaviors (these should be separate tests)

> **Tip**
>
> The first parameter to `assertAll` is an optional heading that appears in failure messages. Use it to identify which group failed when you have multiple `assertAll` blocks.

Testing Exceptions

Testing that code *throws* exceptions is just as important as testing that it returns correct values. Why?

1. **Contract enforcement:** If a caller violates preconditions, they should get a clear error—not silent corruption or mysterious failures later.
2. **Documentation:** Exception tests document the boundaries of valid input.
3. **Defensive programming verification:** You verify that your guard clauses actually work.
4. **Regression prevention:** If someone accidentally removes a null check, your test catches it immediately.

Exception handling enforces the contract—without tests, you're just *hoping* your validation code runs.

Using assertThrows

`assertThrows` verifies that code throws the expected exception type:

```
@Test
void testConstructorRejectsNullName() {
    assertThrows(NullPointerException.class, () -> {
        new StoreItem(null, new BigDecimal("1.99"));
    });
}
```

The syntax breakdown:

```
assertThrows(
    NullPointerException.class,  // Expected exception type
    () -> {                       // Lambda containing code that should throw
        new StoreItem(null, new BigDecimal("1.99"));
    }
);
```

Why Use a Lambda?

The lambda (`() -> { ... }`) delays execution. Without it, the exception would be thrown *before* `assertThrows` could catch it:

```
// WRONG: Exception thrown before assertThrows can catch it
assertThrows(NullPointerException.class, new StoreItem(null, price));  //
Won't compile

// RIGHT: Lambda delays execution so assertThrows can catch the exception
assertThrows(NullPointerException.class, () -> new StoreItem(null, price));
```

Single-Line vs Block Lambdas

For simple expressions, use a single-line lambda:

```
assertThrows(NullPointerException.class, () -> new StoreItem(null, price));
```

For more complex setup, use a block lambda:

```
assertThrows(IllegalArgumentException.class, () -> {
    BigDecimal negativePrice = new BigDecimal("-5.00");
    new StoreItem("Pen", negativePrice);
});
```

TCSS 305 Linting Rules

Our Checkstyle configuration **does not allow multi-statement lambdas**. If you need setup code before the throwing statement, extract it outside the lambda:

```
// WRONG: Multi-statement lambda (Checkstyle violation)
assertThrows(IllegalArgumentException.class, () -> {
    BigDecimal negativePrice = new BigDecimal("-5.00");
    new StoreItem("Pen", negativePrice);
});

// RIGHT: Setup outside, single-expression lambda
BigDecimal negativePrice = new BigDecimal("-5.00");
assertThrows(IllegalArgumentException.class,
    () -> new StoreItem("Pen", negativePrice));
```

Verifying Exception Type Precisely

`assertThrows` checks for the *exact* exception type or any subclass. Be specific:

```
// Tests that NullPointerException is thrown (or any subclass)
assertThrows(NullPointerException.class, () -> new StoreItem(null, price));

// Tests that IllegalArgumentException is thrown (or any subclass)
assertThrows(IllegalArgumentException.class, () -> new StoreItem("", price));
```

Warning

Don't use `Exception.class` or `RuntimeException.class`—these are too broad. If the API specifies `NullPointerException`, test for exactly that type.

Capturing the Exception

Sometimes you need to verify exception details (like the message). `assertThrows` returns the exception:

```
@Test
void testNullPriceExceptionMessage() {
    NullPointerException exception = assertThrows(
        NullPointerException.class,
        () -> new StoreItem("Pen", null)
    );

    // Optionally verify the message
    assertTrue(exception.getMessage().contains("price"),
        "Exception message should mention 'price'");
}
```

Testing Multiple Invalid Inputs

Create separate test methods for each precondition. This provides clearer failure messages:

```
@Test
void testConstructorRejectsNullName() {
    assertThrows(NullPointerException.class,
        () -> new StoreItem(null, new BigDecimal("1.99")));
}

@Test
void testConstructorRejectsNullPrice() {
    assertThrows(NullPointerException.class,
        () -> new StoreItem("Pen", null));
}

@Test
void testConstructorRejectsEmptyName() {
    assertThrows(IllegalArgumentException.class,
        () -> new StoreItem("", new BigDecimal("1.99")));
}

@Test
void testConstructorRejectsNegativePrice() {
    assertThrows(IllegalArgumentException.class,
        () -> new StoreItem("Pen", new BigDecimal("-1.00")));
}
```

Test Naming Conventions

Test names are documentation. A good name tells you exactly what behavior is being verified without reading the code.

Naming Patterns

Several naming conventions are common. Pick one and be consistent:

Pattern 1: testMethodNameCondition

```
testGetNameReturnsCorrectValue()
testCalculateTotalWithQuantityZero()
testConstructorRejectsNullName()
```

Pattern 2: methodName_condition_expectedResult

```
getName_validItem_returnsCorrectName()
calculateTotal_negativeQuantity_throwsException()
constructor_nullName_throwsNullPointerException()
```

Pattern 3: shouldDoSomethingWhenCondition

```
shouldReturnNameWhenGetNameCalled()
shouldThrowExceptionWhenQuantityNegative()
shouldApplyBulkPricingWhenMembershipEnabled()
```

> **Tip**
>
> In TCSS 305, we use Pattern 1 (testMethodNameCondition) for consistency with provided examples. Follow the pattern used in your starter code.

Good vs Bad Names

Bad Name	Problem	Good Name
`test1()`	No information	`testGetNameReturnsCorrectValue()`
`testGetPrice()`	Doesn't say what about getPrice	`testGetPriceReturnsValueFromConstructor()`
`testException()`	Which exception? What causes it?	`testConstructorRejectsNegativePrice()`
`testStoreItem()`	Tests everything? Tests nothing specific	`testEqualsReturnsTrueForEqualItems()`

When Tests Fail

Good names immediately tell you what's broken:

```
testCalculateTotalWithMembershipAppliesBulkPricing  FAILED
testConstructorRejectsNegativePrice                  PASSED
testGetFormattedDescriptionIncludesBulkInfo          PASSED
```

From the failure, you know: "The bulk pricing calculation for members isn't working." No need to read the test code to understand what failed.

> **Gen AI & Learning: Test Generation vs. Test Thinking**
>
> AI tools can generate syntactically correct JUnit tests quickly—but that's precisely why you shouldn't use them for your assignments. Writing tests yourself develops *test thinking*: the ability to identify edge cases, reason about preconditions, and specify behavior from an API contract. This skill is essential for code review, debugging, and TDD in professional settings. Use AI to *explain* testing concepts (e.g., "Why test null separately from empty string?"), not to generate your tests. The cognitive work of test design is the learning objective.

Testing equals() and hashCode()

The `equals()` and `hashCode()` methods form a contract that's critical for collections like `HashSet` and `HashMap` to work correctly. Testing them properly requires understanding the contract.

The equals/hashCode Contract

The `equals()` and `hashCode()` contract is defined in the Java Language Specification and the `Object` class documentation. Joshua Bloch's *Effective Java* provides the clearest practical explanation of these requirements:

equals() must be:

- **Reflexive:** `x.equals(x)` returns true
- **Symmetric:** `x.equals(y)` returns true if and only if `y.equals(x)` returns true
- **Transitive:** If `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`
- **Consistent:** Multiple calls return the same result (if objects haven't changed)
- **Null-safe:** `x.equals(null)` returns false (never throws)

hashCode() must be:

- **Consistent with equals:** If `x.equals(y)`, then `x.hashCode() == y.hashCode()`
- **Consistent:** Multiple calls return the same value (if object hasn't changed)

Essential equals() Tests

```
@Test
void testEqualsReflexive() {
    Item item = new StoreItem("Pen", new BigDecimal("1.99"));
    assertEquals(item, item, "Item should equal itself");
}

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

    assertEquals(item1, item2, "Equal items should be equal");
    assertEquals(item2, item1, "Equality should be symmetric");
}

@Test
void testEqualsWithNull() {
    Item item = new StoreItem("Pen", new BigDecimal("1.99"));
    assertFalse(item.equals(null), "Item should not equal null");
}

@Test
void testEqualsWithDifferentType() {
    Item item = new StoreItem("Pen", new BigDecimal("1.99"));
    assertFalse(item.equals("Not an Item"),
        "Item should not equal object of different type");
}

@Test
void testEqualsWithDifferentName() {
    Item item1 = new StoreItem("Pen", new BigDecimal("1.99"));
    Item item2 = new StoreItem("Pencil", new BigDecimal("1.99"));
    assertNotEquals(item1, item2, "Items with 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, "Items with different prices should not be
equal");
}
```

Essential hashCode() Tests

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

    assertEquals(item1, item2, "Items should be equal");
    assertEquals(item1.hashCode(), item2.hashCode(),
        "Equal items must have equal hash codes");
}

@Test
void testHashCodeConsistent() {
    Item item = new StoreItem("Pen", new BigDecimal("1.99"));
    int hashCode1 = item.hashCode();
    int hashCode2 = item.hashCode();

    assertEquals(hashCode1, hashCode2,
        "hashCode() should return consistent values");
}
```

Guide

The equals/hashCode Contract — Deeper dive into the contract requirements and testing strategies.

Testing with BigDecimal

`BigDecimal` is used for precise decimal arithmetic, especially for money. Testing with `BigDecimal` requires understanding its quirks.

Scale Matters for equals()

`BigDecimal`'s `equals()` method considers **scale** (number of decimal places):

```java
BigDecimal a = new BigDecimal("1.99");   // Scale 2
BigDecimal b = new BigDecimal("1.990");  // Scale 3

// These are mathematically equal but not equals() equal!
assertFalse(a.equals(b));                // Different scales
assertTrue(a.compareTo(b) == 0);         // Same mathematical value
```

Testing BigDecimal Equality

When scale is specified in the API: Use `assertEquals` directly:

```
// API specifies scale of 2 for prices
assertEquals(new BigDecimal("25.00"), cart.calculateTotal());
```

When scale might vary: Use `compareTo`:

```
BigDecimal expected = new BigDecimal("25.00");
BigDecimal actual = item.calculateTotal(10, false);

assertEquals(0, expected.compareTo(actual),
    "Totals should be mathematically equal");
```

Creating BigDecimal Test Values

Always use the `String` constructor to avoid floating-point imprecision:

```
// GOOD: Exact representation
BigDecimal price = new BigDecimal("1.99");
```

Never Use the double Constructor

```
// DANGEROUS: Floating-point imprecision
BigDecimal price = new BigDecimal(1.99);  // Actually
1.98999999999999991...
```

This creates a `BigDecimal` with the *exact* value of the `double` literal—which is NOT `1.99` due to IEEE 754 representation. Your tests will fail with confusing precision errors.

What if you have a `double` variable?

If you're working with an existing `double` value (perhaps from user input or external data), use `BigDecimal.valueOf()`:

```
double userInput = 1.99;

// WRONG: Captures floating-point imprecision
BigDecimal bad = new BigDecimal(userInput);

// RIGHT: Converts via String representation, avoiding imprecision
BigDecimal good = BigDecimal.valueOf(userInput);
```

`BigDecimal.valueOf(double)` internally converts the double to a String first, giving you the expected value.

Coming in Assignment 1c

Assignment 1c includes a dedicated guide on `BigDecimal` that covers arithmetic operations, rounding modes, scale management, and common pitfalls in depth.

Testing Price Calculations

```
@Test
void testCalculateTotalSimple() {
    Item item = new StoreItem("Pen", new BigDecimal("2.50"));
    BigDecimal total = item.calculateTotal(4, false);

    assertEquals(new BigDecimal("10.00"), total);
}

@Test
void testCalculateTotalWithDecimalPrice() {
    Item item = new StoreItem("Widget", new BigDecimal("0.33"));
    BigDecimal total = item.calculateTotal(3, false);

    // 0.33 * 3 = 0.99
    assertEquals(new BigDecimal("0.99"), total);
}
```

Verifying Scale and Rounding

If the API specifies rounding behavior, test it:

```
@Test
void testCalculateTotalUsesHalfEvenRounding() {
    // Set up a calculation that requires rounding
    Item item = new StoreItem("Widget", new BigDecimal("0.333"));
    BigDecimal total = item.calculateTotal(1, false);

    // Verify scale of 2 with HALF_EVEN rounding
    assertEquals(2, total.scale(), "Result should have scale of 2");
    assertEquals(new BigDecimal("0.33"), total);
}
```

Testing Records

Java **records** are a modern feature that automatically generates `equals()`, `hashCode()`, and `toString()` methods. In Assignment 1b, `ItemOrder` is a record.

About Records

Records are immutable data classes that automatically generate accessor methods, `equals()`, `hashCode()`, and `toString()`. You'll learn more about records in Assignment 1c. For now, just know that record-generated methods follow the contract correctly by definition—but the constructor can still have validation that needs testing.

What to Test in Records

Even though records auto-generate methods, you should still test:

1. **Constructor validation** - Records can include validation in compact constructors
2. **Accessor methods** - Verify they return correct values
3. **equals() behavior** - The auto-generated implementation should work, but verify it

```
@Test
void testItemOrderConstructorRejectsNull() {
    assertThrows(NullPointerException.class,
        () -> new ItemOrder(null, 5));
}

@Test
void testItemOrderConstructorRejectsNegativeQuantity() {
    Item item = new StoreItem("Pen", new BigDecimal("1.99"));
    assertThrows(IllegalArgumentException.class,
        () -> new ItemOrder(item, -1));
}

@Test
void testItemOrderAccessors() {
    Item item = new StoreItem("Pen", new BigDecimal("1.99"));
    ItemOrder order = new ItemOrder(item, 5);

    assertSame(item, order.item());
    assertEquals(5, order.quantity());
}

@Test
void testItemOrderEquals() {
    Item item = new StoreItem("Pen", new BigDecimal("1.99"));
    ItemOrder order1 = new ItemOrder(item, 5);
    ItemOrder order2 = new ItemOrder(item, 5);

    assertEquals(order1, order2);
}
```

Common Pitfalls

Pitfall 1: Testing Implementation Instead of Behavior

Problem: Tests that depend on *how* code works internally, not *what* it produces.

```
// BAD: Testing implementation details
@Test
void testItemStoresNameInField() {
    StoreItem item = new StoreItem("Pen", new BigDecimal("1.99"));
    // Accessing private field via reflection - DON'T DO THIS
    Field nameField = StoreItem.class.getDeclaredField("myName");
    nameField.setAccessible(true);
    assertEquals("Pen", nameField.get(item));
}

// GOOD: Testing behavior through public API
@Test
void testGetNameReturnsConstructorValue() {
    Item item = new StoreItem("Pen", new BigDecimal("1.99"));
    assertEquals("Pen", item.getName());
}
```

Why it matters: Implementation can change while behavior stays the same. Tests coupled to implementation break unnecessarily.

Example 2: Copying a Bug into Your Test

Consider this `Circle` class with a bug in `getArea()`:

```
public class Circle {
    private final BigDecimal radius;

    public Circle(BigDecimal radius) {
        this.radius = radius;
    }

    public BigDecimal getArea() {
        // BUG: Using 3.14 instead of PI, and wrong formula!
        return radius.multiply(new BigDecimal("3.14"));  // Should be π × r²
    }
}
```

A student writes this test:

```
@Test
void testGetArea() {
    Circle circle = new Circle(new BigDecimal("2.0"));
    // Student calculated: 2.0 × 3.14 = 6.28
    assertEquals(new BigDecimal("6.28"), circle.getArea(),
        "Area should be calculated correctly");
}
```

Can you spot the problem? The test passes, but the implementation is wrong! The area of a circle with radius 2 should be $\pi \times 2^2 \approx 12.57$, not 6.28.

The student tested *what the code does* rather than *what the code should do*. Always derive expected values from the specification, not from running the code.

Pitfall 2: Not Testing Edge Cases

Problem: Only testing "happy path" scenarios.

```
// Only testing positive cases - incomplete!
@Test
void testCalculateTotal() {
    Item item = new StoreItem("Pen", new BigDecimal("2.00"));
    assertEquals(new BigDecimal("10.00"), item.calculateTotal(5, false));
}
```

Solution: Test boundaries and edge cases:

```
@Test
void testCalculateTotalWithZeroQuantity() {
    Item item = new StoreItem("Pen", new BigDecimal("2.00"));
    assertEquals(new BigDecimal("0.00"), item.calculateTotal(0, false));
}

@Test
void testCalculateTotalWithQuantityOne() {
    Item item = new StoreItem("Pen", new BigDecimal("2.00"));
    assertEquals(new BigDecimal("2.00"), item.calculateTotal(1, false));
}

@Test
void testCalculateTotalWithLargeQuantity() {
    Item item = new StoreItem("Pen", new BigDecimal("2.00"));
    assertEquals(new BigDecimal("2000.00"), item.calculateTotal(1000, false));
}

@Test
void testCalculateTotalRejectsNegativeQuantity() {
    Item item = new StoreItem("Pen", new BigDecimal("2.00"));
    assertThrows(IllegalArgumentException.class,
        () -> item.calculateTotal(-1, false));
}
```

Pitfall 3: Multiple Assertions Testing Unrelated Things

Problem: One test method verifying multiple unrelated behaviors.

```
// BAD: Testing too many things in one test
@Test
void testStoreItem() {
```

```
    Item item = new StoreItem("Pen", new BigDecimal("1.99"));
    assertEquals("Pen", item.getName());
    assertEquals(new BigDecimal("1.99"), item.getPrice());
    assertEquals("Pen, $1.99", item.getFormattedDescription());
    assertThrows(IllegalArgumentException.class,
        () -> item.calculateTotal(-1, false));
}
```

Problem: If this test fails, which behavior is broken? You have to investigate to find out.

Solution: One test per behavior:

```
@Test
void testGetName() {
    Item item = new StoreItem("Pen", new BigDecimal("1.99"));
    assertEquals("Pen", item.getName());
}

@Test
void testGetPrice() {
    Item item = new StoreItem("Pen", new BigDecimal("1.99"));
    assertEquals(new BigDecimal("1.99"), item.getPrice());
}

@Test
void testGetFormattedDescription() {
    Item item = new StoreItem("Pen", new BigDecimal("1.99"));
    assertEquals("Pen, $1.99", item.getFormattedDescription());
}
```

Pitfall 4: Tests That Always Pass

Problem: Tests that don't actually verify anything meaningful.

```
// BAD: This test always passes regardless of getName() behavior
@Test
void testGetNameNotNull() {
    Item item = new StoreItem("Pen", new BigDecimal("1.99"));
    String name = item.getName();  // Result is never checked!
}

// GOOD: Actually verify the result
@Test
void testGetNameNotNull() {
    Item item = new StoreItem("Pen", new BigDecimal("1.99"));
    assertNotNull(item.getName());
}
```

Pitfall 5: Forgetting to Test Both Boolean States

Problem: Only testing one value of a boolean parameter.

```
// INCOMPLETE: Only tests membership = false
@Test
void testCalculateTotal() {
    Item item = new StoreBulkItem("Pen", new BigDecimal("2.00"), 6, new
BigDecimal("10.00"));
    assertEquals(new BigDecimal("20.00"), item.calculateTotal(10, false));
}
```

Solution: Test both true and false:

```
@Test
void testCalculateTotalWithoutMembership() {
    Item item = new StoreBulkItem("Pen", new BigDecimal("2.00"), 6, new
BigDecimal("10.00"));
    // Without membership: 10 * $2.00 = $20.00
    assertEquals(new BigDecimal("20.00"), item.calculateTotal(10, false));
}

@Test
void testCalculateTotalWithMembership() {
    Item item = new StoreBulkItem("Pen", new BigDecimal("2.00"), 6, new
BigDecimal("10.00"));
    // With membership: 1 bulk set ($10.00) + 4 * $2.00 = $18.00
    assertEquals(new BigDecimal("18.00"), item.calculateTotal(10, true));
}
```

Putting It All Together: Test Class Structure

Here's the *structure* of a well-organized test class (method bodies omitted—that's your job!):

```
import org.junit.jupiter.api.Test;
import java.math.BigDecimal;
import static org.junit.jupiter.api.Assertions.*;

class StoreItemTest {

    // === Constructor Tests ===
    // Group all constructor validation tests together

    @Test
    void testConstructorWithValidArguments() {
        // Verify object is created correctly with valid inputs
    }

    @Test
    void testConstructorRejectsNullName() {
        // Use assertThrows to verify NullPointerException
    }

    @Test
    void testConstructorRejectsNullPrice() { /* ... */ }
```

```
    @Test
    void testConstructorRejectsEmptyName() { /* ... */ }

    @Test
    void testConstructorRejectsNegativePrice() { /* ... */ }

    // === Accessor Tests ===
    // Test each getter returns the expected value

    @Test
    void testGetNameReturnsCorrectValue() { /* ... */ }

    @Test
    void testGetPriceReturnsCorrectValue() { /* ... */ }

    // === calculateTotal Tests ===
    // Test normal cases, edge cases, and error conditions

    @Test
    void testCalculateTotalWithZeroQuantity() { /* ... */ }

    @Test
    void testCalculateTotalWithQuantityOne() { /* ... */ }

    @Test
    void testCalculateTotalWithMultipleQuantity() { /* ... */ }

    @Test
    void testCalculateTotalRejectsNegativeQuantity() { /* ... */ }

    // === getFormattedDescription Tests ===

    @Test
    void testGetFormattedDescription() { /* ... */ }

    // === equals/hashCode Tests ===
    // Test the contract: reflexive, symmetric, null-safe, consistent with
hashCode

    @Test
    void testEqualsReflexive() { /* ... */ }

    @Test
    void testEqualsSymmetric() { /* ... */ }

    @Test
    void testEqualsWithNull() { /* ... */ }

    @Test
    void testHashCodeConsistentWithEquals() { /* ... */ }
}
```

Key organizational principles:

1. **Group related tests** with comment headers (`// === Section ===`)

2. **Constructor tests first** — if the object can't be created correctly, nothing else matters
3. **One behavior per test** — each `@Test` method tests exactly one thing
4. **Descriptive names** — the test name tells you what broke when it fails

---

Summary

Concept	Key Point
Why test	Catch bugs early, gain confidence, enable safe refactoring
Test structure	Arrange-Act-Assert pattern for every test
Assertions	Use specific assertions: `assertEquals`, `assertThrows`, etc.
assertAll	Groups related assertions; reports all failures at once
Exception testing	Use `assertThrows` with lambdas to verify exception types
Test naming	Names should describe behavior being tested
equals/hashCode	Test reflexive, symmetric, consistent, null-safe properties
BigDecimal	Use String constructor; beware of scale in `equals()`
Edge cases	Test boundaries: 0, 1, negative, large values
One behavior per test	When a test fails, you should know exactly what broke

---

Further Reading

External Resources

- JUnit 5 User Guide - Official JUnit 5 documentation
- JUnit 5 Assertions Javadoc - Complete list of assertion methods
- Baeldung: JUnit 5 Tutorial - Practical tutorials and examples
- Oracle: Java Records - Official records documentation
- Martin Fowler: Given When Then - Alternative naming for Arrange-Act-Assert

---

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; Chapter 10: Exceptions.
- Beck, K. (2002). *Test Driven Development: By Example*. Addison-Wesley. — Foundational TDD methodology text.
- Horstmann, C. S. (2022). *Core Java, Volume I: Fundamentals* (12th ed.). Oracle Press. Chapter 5: Inheritance (equals/hashCode), Chapter 7: Exceptions.

Testing Frameworks:

- JUnit 5 User Guide — Official JUnit 5 documentation
- JUnit 5 API Documentation — Javadoc for JUnit 5 assertions and annotations

Language Documentation:

- Oracle JDK 25 Documentation — Official Java SE reference
- Oracle JDK 25: BigDecimal — BigDecimal class documentation
- Oracle JDK 25: Object.equals — equals() contract specification

Testing Methodology:

- Fowler, M. (2014). Test Driven Development — Overview of TDD methodology
- Arrange-Act-Assert Pattern — Test structure pattern documentation

---

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