Java JUnit

Table of contents

What is JUnit? 1
Writing tests 1
`@Before` and `@After` 3
`Assert` 4
Checking for exceptions 4
Black box tests in JUnit 5
White box tests in JUnit 5

Introduced in 2026-03-26-testing[1] (Lecture, Week 5). See java-testing[2] for the broader testing concepts JUnit is used to implement.

What is JUnit?

An open-source test automation framework for Java — one of many *xUnit* frameworks (JUnit, NUnit, CPPUnit, PyUnit…). This course uses **JUnit 4** (http://junit.org/junit4, and the cookbook at https://junit.org/junit4/cookbook.html). Depending on how software is structured, unit-testing frameworks like JUnit can also be used for other kinds of automated testing, but "unit testing" is sometimes *incorrectly* used to mean any automated test.

Writing tests

Tests are defined in classes — **one test class per real class**. `@Test` marks a method as a test JUnit should run; test methods **take no parameters and return `void`**, and should be named for what they check.

The full class and its test class:

[1] courses/csse2002/2026-03-26-testing.pdf
[2] courses/csse2002/java-testing.pdf

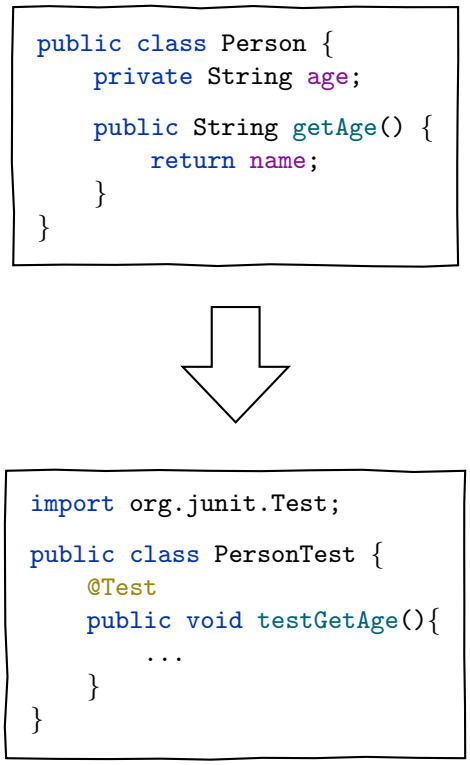

Figure 1: One test class per real class: `Person` produces `PersonTest`, which imports `org.junit.Test` and carries one `@Test`-annotated method per thing being checked. The hollow block arrow pointing down means produces/translates-to. The `Person` snippet does not compile as drawn: `getAge()` returns `name`, which the snippet never declares, and the field is typed `String age`. The full `Person` below types it `int` and is the version that compiles.

```java
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

```java
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class PersonTest {

    @Test
    public void TestGetAge() {
        Person p1 = new Person("Jack", 10);
        int age = p1.getAge();
        assertEquals(10, age);
    }

    @Test
    public void TestGetName() {
        Person p1 = new Person("Jack", 10);
        String name = p1.getName();
        assertEquals("Jack", name);
    }
}
```

The test class uses two imports: `org.junit.Test` for the annotation, and a **static** import of `org.junit.Assert.assertEquals` so the assertion can be called bare rather than as `Assert.assertEquals(...)`.

`@Before` and `@After`

Each test method should be independent, but you may want common things set up before each test (e.g. object creation):

```java
public class PersonTest {

    private Person person;

    @Before
    public void setUp() {
        person = new Person("Jack", 10);
    }

    @After
    public void tearDown() {
        person = null;
    }
    // ...
}
```

`@Before` will be run before **each** test; `@After` will be run after **each** test. Every test method gets a freshly built fixture rather than whatever the previous test left behind, which is how the tests are kept independent.

`Assert`

Some static methods in `Assert`:

- `assertEquals` — asserts that two objects are equal.
- `assertArrayEquals` — asserts that two object arrays are equal.
- `assertFalse` / `assertTrue` — asserts that a condition is false or true.
- `assertSame` / `assertNotSame` — asserts that two objects refer to the same object or not.

The difference between `assertSame` and `assertEquals` is reference identity vs `.equals`: two distinct `Person` objects with the same fields can be `assertEquals`-equal (if `equals` is overridden) but are never `assertSame`.

Checking for exceptions

Don't catch the exception — the test runner does. Tell the test runner which exception it should catch, via `@Test(expected = ...)`:

```java
@Test(expected = EOFException.class)
public void testExceptions() throws EOFException {
    callThatShouldThrowEOFException();
}
```

`.class` is required after the name of the class. The test **passes** if that exception is thrown and **fails** if it is not, or if a different one is. See java-exceptions[3].

Black box tests in JUnit

Write the tests purely from the specification/Javadoc, without looking at the implementation, then use smoke tests, boundary tests and equivalence classes (see java-testing[4]) to decide what to check. The hard constraint is that **tests must not violate preconditions**. A test that breaks `@requires` tells you nothing, because the method is entitled to do anything at all in that case.

White box tests in JUnit

Design tests based on the **internal logic**. You try to cover branches, loops and edge cases, and you may aim for a code coverage target (e.g. 100%).

The worked example is a discount calculator with two nested levels of branching:

```
public class DiscountCalculator {

    public double calculateDiscount(double amount, boolean isMember) {
        if (amount <= 0) {
            return 0;
        }

        if (isMember) {
            if (amount > 100) {
                return amount * 0.2;
            } else {
                return amount * 0.1;
            }
        } else {
            if (amount > 100) {
                return amount * 0.1;
            } else {
                return 0;
            }
        }
    }
}
```

The five tests, one per leaf:

[3] courses/csse2002/java-exceptions.pdf
[4] courses/csse2002/java-testing.pdf

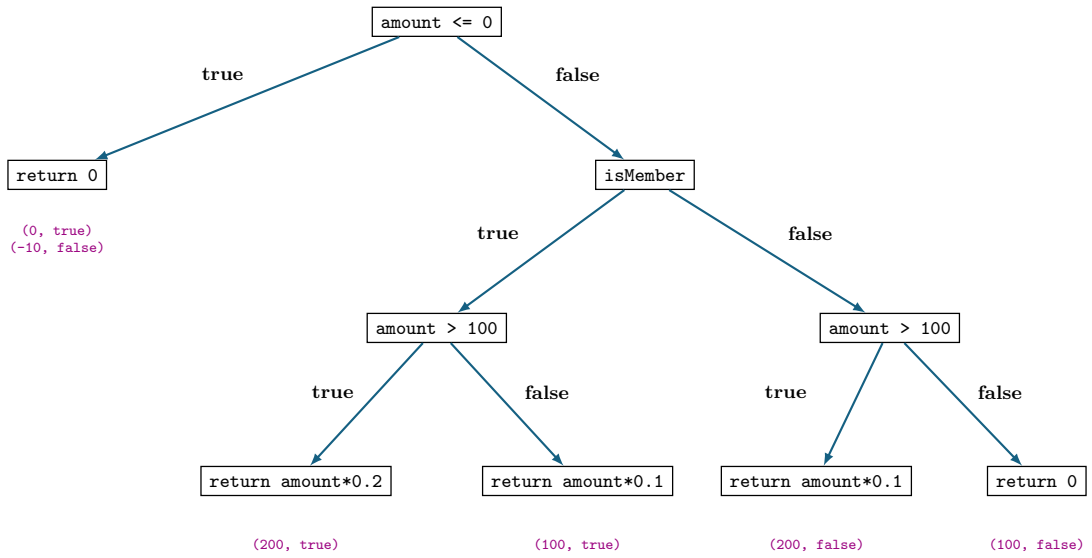

Figure 2: Why `DiscountCalculator` needs exactly five tests: the method as a decision tree, one leaf per test. Each box is a condition from the method or the `return` its branch reaches, so the expected values in the tests can be read off the leaves. The magenta `(amount, isMember)` pair under each leaf is the argument list of the test that reaches that leaf. The early return on `amount <= 0` is the one leaf covered by two assertions in a single test, which is why it carries two pairs.

```java
public class DiscountCalculatorTest {

    DiscountCalculator calc = new DiscountCalculator();

    @Test
    public void testAmountLessOrEqualZero() {
        assertEquals(0.0, calc.calculateDiscount(0, true));
        assertEquals(0.0, calc.calculateDiscount(-10, false));
    }

    @Test
    public void testMemberAbove100() {
        assertEquals(40.0, calc.calculateDiscount(200, true));
    }

    @Test
    public void testMemberBelowOrEqual100() {
        assertEquals(10.0, calc.calculateDiscount(100, true));
    }

    @Test
    public void testNonMemberAbove100() {
        assertEquals(20.0, calc.calculateDiscount(200, false));
    }

    @Test
    public void testNonMemberBelowOrEqual100() {
        assertEquals(0.0, calc.calculateDiscount(100, false));
    }
}
```

Two things worth reading off this example:

- The tests are **named for the branch they cover**, not for the method (`testMemberAbove100`, not `testCalculateDiscount3`), which is what naming a test for what it checks looks like in practice.
- `amount = 100` is the argument in two of the five tests. It is the boundary of `amount > 100`, so it is both a white-box branch probe and a black-box boundary test.

> **assertEquals on doubles**
>
> The tests above call `assertEquals(0.0, ...)` on `double` values. JUnit 4's two-argument `assertEquals(double, double)` is deprecated precisely because floating-point equality is unreliable; the three-argument `assertEquals(expected, actual, delta)` is the form that compiles without a deprecation warning. The code above will warn if you type it in.