Java Part 10: Testing with JUnit and Next Steps

A program that works today can break tomorrow, when someone changes one line. Automated tests are small programs that check our code and tell us at once when something breaks. Four steps build a reliable test: arrange, act, assert and automate. In this part, Maya, a quality lead in the billing team of a telecom company, uses them to protect an invoice calculation, and each step answers one practical question.

1. Arrange

A unit test checks one small piece of code, such as a method, with known inputs and an expected result. JUnit is a free testing framework for Java that finds and runs such tests. A test is an ordinary method marked with the annotation Test. Every test follows the same shape: arrange, act, assert. In the arrange part we prepare everything the test needs, such as an invoice with two lines whose total we can compute by hand. A method marked BeforeEach runs before every test and builds fresh objects, so tests never depend on each other. The question for this step is: which situation do we set up, and what do we already know about the correct answer?

Example: A bank tester checks the method that adds yearly interest to a savings account. She arranges the situation first: an account with a balance of one thousand dollars and a rate of five percent. Before the test runs, she computes the correct interest on paper and writes down fifty dollars. That is arrange: a known starting situation and a known correct answer.

At work: Maya starts with the simplest situation and gives the test a name that says what it checks: `@Test void emptyInvoiceHasZeroTotal()`. In the arrange part she creates an empty invoice. A second test arranges an invoice with two lines, and she calculates the expected total by hand first. A BeforeEach method gives every test a fresh invoice.

2. Act

In the act part we call the one thing we are testing, usually a single method call. Keeping it to one call makes a failure easy to locate. Good tests cover a normal case, the edges and the error cases: the value just below a limit, exactly at the limit and just above it, an empty input, and input that should be rejected. Each test checks one idea and has a name that describes the behaviour. Tests must be independent, fast and repeatable, which means they give the same answer every time and in any order. The question for this step is: which single action do we test, and which edge cases matter?

Example: A shipping team tests the method that calculates the delivery fee, and the fee rises when a parcel passes the weight limit. The act part is the same single call each time, with three different parcels: one just below the limit, one exactly at the limit and one just above it. Three small tests show exactly where the fee changes. A mistake at the boundary is caught the same day, and this shows why edge cases matter.

At work: Maya's act part is one line: it asks the invoice for its total. For the discount she writes three tests, with a total just below the loyalty threshold, one exactly at it and one just above it. Another test calls the method with a negative quantity, which should be rejected. Each test has a name such as discountStartsAtTheThreshold.

3. Assert

The assert part compares what happened with what we expected. The method assertEquals takes the expected value first and the actual value second, and assertTrue checks a condition. To check that a method refuses bad input, assertThrows names the expected exception and runs the code that should throw it. When an assertion fails, JUnit reports the expected value, the actual value and the test name, so we start with a

This document is a property of Asaseeyat Academy - © Copy Right Protected.

precise clue. For BigDecimal, equals treats an amount with one decimal and the same amount with two decimals as different, so we compare with compareTo or give both values the same scale. A test that cannot fail proves nothing, so we make sure each test fails when the code is wrong. The question for this step is: what exactly must be true when the code is correct?

Example: A payroll tester expects the net pay of an employee to be two thousand dollars, but the test reports an actual value of one thousand nine hundred. The report shows the expected value, the actual value and the name of the test. The developer opens that method and finds a deduction applied twice within minutes. This is the value of assert: a failure that comes with a precise clue.

At work: Maya asserts that the total of the empty invoice compares equal to zero, using compareTo because of the rule about decimals. For the negative quantity she uses assertThrows with IllegalArgumentException. She breaks the discount rule on purpose, and the threshold test fails and shows the expected and the actual total. She repairs the rule, and all tests pass again.

4. Automate

Tests help most when they run all the time. IDEs can run all tests with one click, and build tools such as Maven and Gradle run them as a standard step of every build. These tools also download libraries like JUnit, compile the application and package it. A team usually runs the tests automatically on every change, so a broken rule is found before release. A green result does not prove the absence of bugs, but a red result is a precise warning. To keep growing, we can learn more about collections and streams, databases through JDBC, and frameworks such as Spring for web services. The question for this step is: when do the tests run, and who sees the result?

Example: A software team uses a build tool that runs all the tests whenever a developer proposes a change. One afternoon a developer changes a tax rule, and one test turns red. The tool blocks the change and notifies its author, who fixes it before anyone else sees it. That is automation: the tests run every time, so broken code rarely reaches customers.

At work: Maya asks the team to run the billing tests with the build tool on every change. On the next Friday, a developer changes the discount rule by one line. The build turns red within minutes, and the change is fixed before release. Customers see the correct bills on Monday.

Conclusion

To recap, a good test arranges a known situation, acts with one call, asserts the expected result and runs automatically. JUnit finds the methods marked Test and reports clear failures, and BigDecimal amounts are compared with compareTo. Across this series we moved from a first program to a tested business application: values, decisions, methods, collections, classes, interfaces, exceptions, files and tests. Maya's invoice rule is now protected, and a bad change shows up as a red build within minutes. The next steps are practice on real problems, more of the standard library, and frameworks for web services. A good habit is to write the test first when a bug is reported, and to keep it forever.

Sources

- JUnit: JUnit User Guide (annotations, assertions, test structure) – https://junit.org/junit5/docs/current/user-guide/
- Oracle: The Java Tutorials, Annotations – https://docs.oracle.com/javase/tutorial/java/annotations/index.html
- Apache Maven Project: Introduction to the Build Lifecycle – https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html
- dev.java: Learn Java – https://dev.java/learn/
- JUnit: Writing Tests - Test Classes and Methods – https://docs.junit.org/current/writing-tests/test-classes-and-methods.html
- JUnit: Writing Tests - Assertions – https://docs.junit.org/current/writing-tests/assertions.html
- Oracle: BigDecimal (Java SE API) – https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/math/BigDecimal.html

- Java is a trademark of Oracle and/or its affiliates. JUnit, Maven and Gradle are projects of their respective owners. This video is independent training material and is not affiliated with Oracle.

This document is a property of Asaseeyat Academy - © Copy Right Protected.