

Practical Java Testing with JUnit Jupiter

Assertions, parameterised tests, mocking and a Linux workflow

FA-0643 · 1 day

Software Development

Write and debug selected Jupiter tests, use parameterised inputs and mocks, and inspect reports from a prepared Java project.

Introduction

This one-day workshop is for Java developers who are new to automated unit testing. It teaches Jupiter test structure, assertions, parameterised inputs, dependency isolation and debugging using one prepared project and an authorised Linux sandbox.

The source's JUnit 5/Jupiter concepts are retained and aligned with a compatible current JUnit release. The environment is prepared in advance, one build tool is used hands-on and advanced mocking, parallelism and CI systems are demonstrations or follow-up discussions rather than mastery targets.

Learning outcomes

The course teaches participants to:

- Run tests in a prepared Java/build-tool environment.
- Use Jupiter lifecycle annotations, test fixtures and meaningful assertions.
- Exercise selected edge cases with parameterised inputs.
- Isolate one dependency with Mockito and distinguish stubbing from verification.
- Debug a failing test and interpret its stack trace.
- Inspect available test and coverage reports without equating coverage with correctness.
- Identify CI, isolation and interface-testing considerations for further work.

Prerequisites

- Java programming skills.
- Fundamental experience with POSIX CLI
- Familiarity with build tools like Maven or Gradle.
- Exposure to an Integrated Development Environment (IDE) such as IntelliJ IDEA, Eclipse, or VSCode.

No previous testing or JUnit experience is required.

A prepared compatible JDK, Jupiter, Mockito and Maven or Gradle stack. Current JUnit 6 requires Java 17 or later at runtime; older project stacks need their matching JUnit version.

SSH access to the supplied disposable Linux sandbox, or an equivalent supplied local Linux environment.

Training outline

1. Testing Foundations and Project Context

- Testing premise and background.
- Relate test examples to practical behaviour and risks rather than a professional-versus-academic divide.
- Where tests fit in a development/build pipeline and which environment assumptions matter.
- POSIX and virtualized systems for testing.

Java Project Boundaries

- Fundamentals of Java structure.
- Framework boundaries relevant to the prepared Java project.
- Middleware/integration boundaries: distinguish unit tests from integration tests.

2. Prepared Linux Environment

- Inspect a supplied disposable Linux sandbox and its authorised access.
- Check the prepared JDK and one selected build tool; review Maven/Gradle alternatives briefly.
- Inspect the supplied pom.xml or build.gradle configuration.
- Initial verification tests to ensure correct setup.

3. Jupiter Tests and Lifecycle

- JUnit Platform versus the Jupiter programming/extension model; relevant JUnit 5-to-current-release compatibility.

Lifecycle annotations:

- `@BeforeAll` and `@AfterAll`: Class-level setups.
 - `@BeforeEach` and `@AfterEach`: Method-level fixtures.
 - Writing basic tests using `@Test`.
- Creating efficient test fixtures and managing resources effectively.

4. Assertions and Optional Matchers

- Practise selected equality, truth, null, exception and timeout assertions; review rather than exhaust every assertion.

Optional Hamcrest integration for selected expressive assertions; matchers are not all part of Jupiter itself.

- Matchers (`containsString`, `greaterThan`, `is`, `not`, and `allOf`).
- Practical scenarios demonstrating assertion best practices.

5. Parameterised Inputs

- Use selected `@ParameterizedTest` inputs and compare `@ValueSource`, `@CsvSource`, `@EnumSource` and `@MethodSource`.
- Choose a small set of representative boundaries and edge cases.
- Best practices for efficient parameter management and reuse.

6. Dependency Isolation with Mockito

- Integrating Mockito into Maven/Gradle project environments.

Practical mocking and dependency injection:

- Annotating mocks (`@Mock`, `@InjectMocks`).
 - Stubbing methods (`when(...)/thenReturn(...)`).
 - Verification of mock interactions (`verify(...)` methods).
- Demonstrate argument captors/spies where useful and discuss mock-reset/state pitfalls; not advanced Mockito mastery.

7. Debugging

- Practical debugging workflow in Linux terminals and IDEs.

- Leveraging IDE debuggers (breakpoints, step-in/step-out, inspecting variables).
- Reading and interpreting stack traces for faster problem resolution.
- Common pitfalls and real-world examples of test failures.

8. Build Reports and CI Overview

- Run tests through the compatible Maven Surefire or Gradle configuration.
- Inspect standard HTML/XML reports where provided by the selected tooling; JSON requires an appropriate exporter and is not a universal default.
- Inspect one prepared CI example and compare Jenkins, GitHub Actions and GitLab CI/CD roles without building three pipelines.
- Review a prepared JaCoCo report; coverage measures exercised code, not test adequacy or correctness.

9. Maintainable Test Design

- Naming conventions and test structure: Arrange-Act-Assert paradigm.
- Managing state effectively in unit tests: isolation vs. shared fixtures.
- Designing maintainable and readable test cases.
- Balancing integration and unit testing strategies.
- Parallel execution trade-offs, isolation and shared-resource failures; measure before changing concurrency.

10. Testing Modern System Boundaries

- Microservice boundaries: unit tests versus network/integration checks.
- Queue/stream dependencies: isolate a prepared interface and discuss asynchronous behaviour.
- Event-driven interfaces and deterministic test inputs.
- AI integration boundaries: controlled fixtures and limits of deterministic assertions for variable outputs; overview only.

11. Guided Mini-Project and Review

- Write a small selected test set for the prepared Java service, not a complete industrial application suite.
- Troubleshoot and refine tests on the disposable sandbox.
- Combine a parameterised case, one mock and a prepared build/CI example, using fictional transaction-service data.
- Review readability, meaningful assertions, failures and remaining coverage gaps.

