Mastering Test Driven Development using JUnit - TT3503

Explore TDD, Unit Testing, JUnit 5, Best Practices, Database Testing, Refactoring, Mocking, Advanced Features & More

Duration: 3 Days
Skill Level: Introductory
Available Format: Instructor-Led Online; Instructor-Led, Onsite In Person ; Blended; On Public Schedule

Test Driven Development (TDD) and Unit Testing Essentials is a three-day, comprehensive hands-on test-driven development / JUnit / TDD training course geared for developers who need to get up and running with essential Test-driven development programming skills using JUnit and various open-source testing frameworks.

What You'll Learn

Overview

This course provides a comprehensive, hands-on training experience for Java developers who want to build high-quality, maintainable software using modern **test-driven development (TDD)** and unit testing practices. The course is delivered using **JUnit 6** on a **Java 25** runtime, allowing participants to work with the latest testing tools and platform capabilities while focusing on proven testing principles rather than language-specific features.

Throughout the course, you will learn how to apply TDD effectively by writing tests first to drive design, clarify requirements, and reduce defects. You will explore the core features of JUnit, including test structure, annotations, assertions, parameterized tests, lifecycle management, and advanced testing capabilities. These concepts are reinforced through hands-on exercises that mirror real-world development scenarios.

Another key part of the course is **mocking and isolation using Mockito**. You will learn how and when to use test doubles—such as stubs and mocks, to isolate units under test, control dependencies, and verify interactions between components. The course covers Mockito fundamentals, best practices for mocking collaborators, argument matching, interaction verification, and common pitfalls, all within the context of TDD and clean test design.

The course also introduces the **use of AI-assisted techniques for generating test data**. You will learn how AI tools can be used to quickly create realistic, boundary, and edge-case test data, improve test coverage, and reduce the manual effort involved in setting up complex test scenarios. Emphasis is placed on using AI responsibly—reviewing, validating, and refining generated data so that tests remain accurate, meaningful, and maintainable.

Although the course uses **JUnit 6**, all concepts and techniques are fully applicable to **JUnit 5**. JUnit 6 builds directly on the JUnit 5 architecture, so developers working with JUnit 5 can immediately apply what they learn without adjustment.

The course does **not** require Java 25–specific knowledge. A solid understanding of **Java 11 or later** is sufficient. The emphasis is on testing strategy, framework usage, and design discipline rather than on advanced Java language features.

By the end of the course, you will be equipped to write clear and reliable unit tests, use Mockito effectively to isolate and verify behavior, leverage AI to enhance test data creation, practice disciplined test-driven development, and integrate automated testing into modern Java build and delivery pipelines with confidence.

Objectives

This skills-centric course is approximately **50% hands-on labs and 50% instructor-led discussion**, designed to build practical, job-ready expertise in **Test-Driven Development (TDD)** using **JUnit 6**, **Mockito**, and modern AI-assisted testing techniques. Participants work through guided exercises and labs that reinforce core testing concepts, with a strong focus on writing clean, maintainable, and well-tested Java code.

Working in a hands-on learning environment, guided by an expert instructor, attendees will learn to:

- Understand the principles, workflow, and benefits of **Test-Driven Development (TDD)**

- Explain the role of unit testing within modern Java development
- Understand the architecture and capabilities of **JUnit 6**, and how it relates to **JUnit 5**
- Write, organize, and execute unit tests using JUnit annotations, assertions, and lifecycle methods
- Use JUnit tests to **drive the implementation and design of Java code**
- Apply **best practices and patterns** for writing readable, maintainable, and reliable tests
- Understand how testing and debugging complement each other during development
- Create well-structured test suites, including parameterized and advanced test scenarios
- Use **Mockito** to isolate units under test by mocking dependencies
- Stub behavior and verify interactions using Mockito within a TDD workflow
- Generate realistic and edge-case **test data using AI-assisted techniques**
- Use AI tools to improve test coverage, reduce manual test data creation, and explore boundary conditions
- Evaluate when AI-generated test data is appropriate and how to validate its correctness
- Refactor production and test code safely with confidence, using tests as a safety net
- Discuss and compare different testing techniques and where each fits in the testing pyramid

Audience

This course is intended for **Java developers** who want to improve the quality, reliability, and maintainability of their code through disciplined testing, modern tooling, and TDD practices.

The target audience includes:

- Java developers new to **Test-Driven Development**
- Developers with limited or informal experience using **JUnit or Mockito**
- Java developers transitioning from legacy testing approaches to modern JUnit (JUnit 5 / 6)
- Developers responsible for maintaining or refactoring existing Java codebases
- Team leads and senior developers who want to establish or reinforce testing best practices within their teams

Participants should have a working knowledge of **Java (Java 11 or later)**. While the course uses **JUnit 6** and a **Java 25 runtime**, no prior experience with Java 25–specific features or AI tools is required.

TT2100 Core Java 25 Programming Developer's Workshop

Pre-Requisites

Take Before: Students should have development skills at least equivalent to the following course(s) or should have attended as a pre-requisite:

- **TT2104** Core Java Programming for OO Experienced Developers – 4 days

TT2120	Basic Java Programming for Developers New to OO (C, COBOL, etc.)
TT2104	Fast Track to Core Java Programming for OO Experienced Developers
TT2000	Core Java Programming Developer's Workshop

Agenda

1) Test-Driven Development

Test-Driven Development (TDD) offers a focused and disciplined way to create software that is both reliable and easy to maintain. By starting with tests before any code is written, you clarify exactly what needs to be built and can confirm every change works as intended. This lesson explores why ad-hoc testing methods often fall short and how TDD provides a solution that ensures your code is always functional, well-documented, and ready for change. You will learn the step-by-step process behind TDD, see how continuous testing supports better design and fewer bugs, and discover how automation and coverage tools streamline the entire workflow. With these concepts in hand, you will be equipped to build software that stands up to real-world demands and keeps projects on track.

- Identify why software containing bugs is still shipped.
- Explain the rationale behind test-driven development (TDD).
- Describe the process and steps of TDD.
- Compare traditional development with TDD.
- Summarize the benefits and side effects of TDD.
- Recognize best practices for writing effective tests.
- Outline the importance of automation and code coverage in testing.

2) Unit Testing Fundamentals

Understanding unit testing is essential for building reliable software. You will learn how unit tests work at the smallest level, focusing on individual pieces of code to ensure they function as intended. By exploring the differences between unit, integration, and functional tests, you will see why unit testing is the most effective way to catch bugs early and improve code quality. This lesson will show you how to write clear, independent, and maintainable tests, as well as how to use frameworks and best practices to keep your test suite effective. By mastering unit testing, you will be able to find issues quickly, support refactoring efforts, and create documentation that benefits the entire development team.

- Define what unit testing is and its purpose.
- Distinguish between unit testing, integration testing, and functional testing.
- Identify characteristics of good unit tests.
- Recognize the importance of test isolation and managing dependencies.
- Describe best practices for writing readable and maintainable test code.
- Summarize the principles for reliable and effective tests.

3) Jumpstart: JUnit

JUnit is an essential tool for building reliable Java applications using automated testing. In this lesson, you will see how JUnit enables you to write and run tests that confirm your code behaves as intended. You will learn how to use core annotations to organize your tests, apply assertions to check results, and harness the power of modern IDEs to streamline your workflow. The lesson also covers how Apache Maven fits into the testing process, making it easier to manage dependencies and automate test execution. By understanding the structure and best practices of writing tests with JUnit, you gain confidence that your code changes are safe and your applications stay robust.

- Identify the main features and purpose of JUnit.
- Describe how JUnit tests are structured and executed.
- Explain the use of annotations and assertions in JUnit.
- Recognize the differences between JUnit 5 and JUnit 6.
- Demonstrate how to write and run unit tests in an IDE.
- Understand the role and configuration of Apache Maven for testing.
- Explain the purpose of the Maven Surefire plugin and reporting.
- Describe the use of test fixtures and lifecycle methods in JUnit.

4) Annotations

In this lesson, you will explore advanced features of JUnit that help manage and organize test execution efficiently. You will learn how to skip tests when necessary, set custom names for better clarity in test reports, and enforce strict execution time limits to catch slow tests early. You will also see how to control the order in which tests run and apply conditions so that tests execute only in suitable environments, based on factors like operating system or Java version. Tags provide another layer of control, allowing you to include or exclude specific tests in different contexts, both from build tools and within IDEs. Composing custom annotations and automating resource cleanup further supports effective and maintainable test code. These tools equip you to create robust and adaptable test suites for a wide range of Java projects.

- Explain how to use the @Disabled annotation for skipping tests.
- Describe how to set custom display names for tests with a DisplayNameGenerator.
- Apply timeouts to test methods using the @Timeout annotation.
- Use assertTimeout and assertTimeoutPreemptively for fine-grained timeout control.
- Control test execution order with @TestMethodOrder and MethodOrderer options.
- Configure conditional test execution based on OS, Java version, properties, and environment variables.
- Tag tests with @Tag and filter tests using Maven Surefire, command line, or IDE.
- Compose custom meta-annotations for reusable test annotations.
- Automatically manage resource cleanup with the @AutoClose annotation.

5) Fluent Assertions with AssertJ

AssertJ gives you a powerful way to write clear and maintainable assertions in your Java tests. With AssertJ, your checks read almost like natural language, making it easier to understand what your tests are doing and why they fail. You will see how to chain assertions, work with a variety of data types, and apply advanced techniques like property extraction and recursive comparison. By integrating AssertJ into your projects, you gain access to expressive APIs for collections, objects, and more, all while improving test readability and reliability. This lesson covers everything from adding AssertJ to a Maven project to handling tricky cases like nulls and reporting multiple failures at once.

- Demonstrate how to write basic assertions with AssertJ.
- Identify the reasons AssertJ is preferred in modern Java testing.
- Add AssertJ as a dependency in a Maven project.
- Use type-specific assertions for various data types.
- Apply collection assertions to check size, order, and content.
- Extract and assert properties from objects in collections.
- Filter collections before making assertions.

- Handle null and empty collections with AssertJ. -
- Utilize soft assertions to validate multiple properties in one test.

6) Parameterized Tests

Parameterized tests let you write a single test method that runs automatically with different input values. This approach cuts down on duplicated code and makes your test coverage easier to expand. You will see how JUnit provides a variety of ways to supply input data, from simple values listed directly in the code to complex data loaded from files or generated at runtime. You will explore built-in sources like @ValueSource, @CsvSource, and @EnumSource, as well as techniques for handling null and empty values. You will also learn how to use factory methods and custom providers for maximum flexibility. Along the way, you will discover how to set up your test methods, match data types, and customize the display names for each test run. With these tools, you can build tests that are both thorough and maintainable.

- Explain the purpose and benefits of parameterized tests in JUnit.
- Use @ValueSource to provide simple input values to a test.
- Apply @CsvSource and @CsvFileSource for supplying multiple arguments.
- Demonstrate the use of @EnumSource for testing with enum constants.
- Use @MethodSource for custom and complex data sets
- Implement null and empty argument sources in parameterized tests.
- Customize test invocation display names using placeholders.
- Create and use custom ArgumentsProvider classes with @ArgumentsSource.

7) Advanced Features

Understanding advanced JUnit features boosts the flexibility and maintainability of your test suites. With nested unit tests, you can organize related test cases into logical groups, making complex scenarios easier to manage. Dynamic test creation, enabled by @TestFactory, lets you generate tests at runtime using data from files, databases, or other sources. This approach is especially useful when the number of tests is unknown in advance or when test inputs change frequently. Hierarchical reporting provides clear insight into test execution, adapting automatically as your data evolves. You'll also see how to validate exception scenarios, repeat tests for reliability, and keep your tests isolated with temporary directories. Extensions offer powerful ways to customize test behavior, while grouped assertions help ensure thorough validation in a single run. These techniques provide practical solutions for building robust, data-driven, and maintainable test automation with JUnit.

- Identify how nested unit tests organize test cases in JUnit.

- Explain the purpose and use of the @TestFactory annotation.
- Describe how dynamic tests are generated and executed.
- Demonstrate how hierarchical test reporting works with dynamic containers.
- Recognize when to use @TestFactory for runtime-driven test creation.
- Use assertThrows to test for expected exceptions.
- Apply @RepeatedTest for repeated executions of test methods.
- Summarize the role of JUnit extensions and how to register them.
- Illustrate the use of @TempDir for managing temporary directories in tests.
- Group multiple assertions with assertAll for comprehensive validation.

8) JUnit Best Practices

Unit testing is a cornerstone of robust software development, ensuring your code behaves as intended now and in the future. In this lesson, you will see what separates a good test from a weak one and learn why test code deserves as much care as production code. By focusing on readability, structure, and reliability, you can create tests that serve as both documentation and safety nets for ongoing development. You will also explore common pitfalls in unit testing, practical strategies to manage dependencies, and techniques to keep your test suite lean and meaningful. Understanding the difference between unit and integration tests, as well as the value of automation and coverage analysis, will help you build more maintainable and trustworthy software systems.

- Define the qualities of a good unit test.
- Identify factors that affect test readability and maintainability.
- Distinguish between reliable and unreliable tests.
- Evaluate the quality of test assertions.
- Detect common unit test code smells and apply solutions.
- Identify effective test data management strategies.
- Apply coding practices that improve test quality.
- Understand challenges of testing legacy code.
- Prepare a suitable environment for unit testing.
- Recognize the role of automation and coverage in testing.

9) Mocking of Components

Unit testing relies on creating a controlled environment where your code can be checked independently of outside influences. By using test dummies such as stubs, fakes, spies, and mocks, you can replace real systems like databases and services with predictable stand-ins. This approach makes your tests faster, more reliable, and easier

to interpret, since failures are not caused by external factors. You will see how different types of test dummies fit specific testing needs, from simply returning fixed data to verifying how your code interacts with its dependencies. Through practical examples, you will discover how to design tests that are both clear and robust, supporting not only quality but also the pace of development.

- Define the purpose and advantages of using test dummies in unit testing.
- Identify how isolation improves the focus and reliability of unit tests.
- Explain how test dummies contribute to faster and more stable tests.
- Distinguish between different types of test dummies, including stubs, fakes, spies, and mocks.
- Demonstrate how to use stubs to provide fixed responses in tests.
- Describe the typical structure of unit tests that use mocks.

10) Introduction to Mockito

Mockito is a powerful tool for writing effective unit tests in Java, especially when following Test Driven Development practices. By using mocks, you can focus your tests on a single unit of code, replacing its collaborators with controlled test doubles. This ensures that tests run quickly and consistently, with no reliance on real databases, networks, or external systems. Understanding the differences between mocks and stubs, and how to use each effectively, will help you design clear and robust tests. With Mockito, you have fine-grained control over how dependencies behave during testing, and you can verify that your code interacts with its collaborators exactly as intended. This lesson will guide you through setting up Mockito, creating and configuring mocks, using argument matchers, and applying verification techniques to ensure your tests capture the right behavior.

- Explain the role of Mockito in unit testing with TDD.
- Set up Mockito as a test dependency in a Maven project.
- Create and configure manual mocks using Mockito.
- Stub mock behavior to control method responses.
- Use argument matchers and custom matchers in tests.
- Verify interactions, arguments, call counts, and call order with Mockito.
- Identify and avoid common pitfalls in using matchers and verification.

11) Working with Mockito

Mockito annotations offer a powerful way to simplify your unit tests by reducing the amount of setup code you need to write. By learning how to enable and use these annotations with JUnit 6, you can create cleaner, more maintainable tests for your Java

code. You will explore how to use @Mock and @InjectMocks to automate mock creation and injection, making your test classes easier to read and modify. You will also see how Mockito handles exceptions in mocks, allowing you to test error handling paths without relying on real failures or external systems. Finally, you will discover what spies are, how they differ from mocks, and when it makes sense to use them in your testing strategy.

- Identify core Mockito annotations and their purposes.
- Enable Mockito annotations using JUnit 6 integration.
- Apply @Mock and @InjectMocks to streamline test setup.
- Configure mock objects to throw exceptions for both void and non-void methods.
- Describe the concept and utility of spies in Mockito.
- Safely stub spies to prevent unintended side effects.
- Recognize scenarios where spies are appropriate in testing.

12) More Mockito

You will examine how Mockito's capabilities have evolved to remove many early limitations and support advanced mocking scenarios. You will see how different mock makers work and why inline mocking is now the default for modern Java projects. You will also learn the technical configurations required to enable dynamic agent loading, as well as how to mock static methods, final classes, constructors, and even default interface methods. Throughout, you will explore best practices and common pitfalls, so you can recognize when advanced mocking is appropriate and when it is better to rethink your test or design. By mastering these techniques, you can confidently handle complex testing challenges in modern Java applications.

- List the historical limitations of Mockito and how they have changed.
- Describe the difference between subclass, proxy, and inline mock makers.
- Identify when and how to use static, final, and constructor mocking in Mockito.
- Explain the risks and best practices for mocking private and static methods.
- Demonstrate how to configure dynamic agent loading for inline mocking in Java.
- Show how to mock default methods and understand default return behaviors in Mockito

13) Generating Test Data with AI

Great testing depends on the quality of your test data. You'll see how using realistic, varied, and purpose-driven data can reveal issues that might go undetected with generic or repetitive inputs. This lesson highlights the characteristics that make test data effective and demonstrates why "good enough" is often not enough for real-world

software. You'll explore the pitfalls of poor data, such as missed bugs, broken integrations, and security vulnerabilities.

Modern tools and techniques can simplify the creation of valuable test data. With AI-powered solutions and libraries like Faker, you can quickly generate hundreds or thousands of records that mimic actual user behavior and edge cases. Mockaroo offers a no-code way to produce complex datasets, while ChatGPT enables prompt-based data generation, even for tricky scenarios. You'll also learn how to check the data your tools create to make sure it truly supports your testing goals. By the end of this lesson, you'll have a toolkit of strategies to create, validate, and use high-quality test data for more reliable and efficient testing.

- Explain why realistic and varied test data is essential.
- Use AI models to generate mock data and input structures.
- Apply tools like Mockaroo or DataFaker to generate structured datasets.
- Generate data for edge cases, normal cases, and randomized scenarios.
- Validate the quality and usefulness of AI-generated test data.
- Ensure data anonymization while maintaining data relationships.
- Match tools and techniques to specific data generation goals.

14) Improving Code Quality Through Refactoring

Refactoring is a key technique for keeping codebases healthy, maintainable, and adaptable to change. In this lesson, You will explore the principles behind refactoring, discover why it is an essential part of professional development, and see how to apply it to Java code using practical examples. You will learn to use both manual and automated refactoring tools, understand how tests keep your changes safe, and see how modern Java features can further simplify and clarify your code. By the end, you will be able to identify opportunities for refactoring and confidently improve code quality without changing its behavior.

- Define refactoring and its purpose in software development.
- Explain how refactoring maintains external behavior while improving internal structure
- Identify the role of tests as a safety net during refactoring.
- Recognize common refactoring techniques in Java.
- Apply polymorphism to replace conditionals in code.
- Update code examples using modern Java features such as switch expressions and streams.
- Assess the impact of refactoring on code quality, speed, and risk.

Related Courses

TTAI2140	AI in Software Testing

Setup Made Simple! Learning Experience Platform (LXP)

All applicable course software, digital courseware files or course notes, labs, data sets and solutions, live coaching support channels and rich extended learning and post training resources are provided for you in our “easy access, no install required” online Learning Experience Platform (LXP), remote lab and content environment. Access periods vary by course. We’ll collaborate with you to ensure your team is set up and ready to go well in advance of the class. Please inquire about set up details and options for your specific course of interest.

For More Information

Please contact us or call 844-475-4559 toll free for more information about our training services (instructor-led, self-paced or blended), coaching and mentoring services, public course enrollment or questions, partner programs, courseware licensing options and more.