**JUNIT:**

JUnit is a unit testing framework for Java that helps developers write and run tests to verify that individual parts of their code (usually methods or classes) work as expected. It plays a key role in test-driven development (TDD), where tests are written before the actual code to guide design and ensure correctness.

The latest version, JUnit 5, introduces a modular architecture with three main components:

* JUnit Platform: Launches testing frameworks on the JVM.
* JUnit Jupiter: Provides the new programming model and extension model for writing tests.
* JUnit Vintage: Supports running older JUnit 3 and 4 tests on the JUnit 5 platform.

With annotations like @Test, @BeforeEach, and @AfterAll, JUnit makes it easy to structure and automate your tests. It also integrates well with build tools like Maven and Gradle, and IDEs like IntelliJ and Eclipse.

**MAVEN:**

Maven is a **build automation and project management tool** primarily used for Java projects. It simplifies the process of building, testing, and managing dependencies by using a central configuration file called the **POM (Project Object Model)** file, typically named pom.xml.

Here’s what makes Maven so useful:

* **Dependency Management**: Automatically downloads and manages libraries your project needs.
* **Standardized Project Structure**: Encourages a consistent layout across projects.
* **Build Lifecycle**: Automates tasks like compiling code, running tests, packaging, and deploying.
* **Plugins**: Offers a wide range of plugins for tasks like code analysis, documentation, and deployment.
* **Integration**: Works seamlessly with IDEs like IntelliJ IDEA and Eclipse, and with CI/CD tools like Jenkins.

**IS MAVEN MUST?**  
You *don’t* strictly need Maven to use JUnit—but Maven makes life a whole lot easier.

Without Maven, you’d have to manually download JUnit and all its dependencies (like hamcrest) and add them to your project’s classpath. That can get pretty tedious and error-prone, especially if you're juggling multiple libraries or versions.

With Maven, all you need is a simple snippet in your pom.xml like:

**<dependency>**

**<groupId>org.junit.jupiter</groupId>**

**<artifactId>junit-jupiter</artifactId>**

**<version>5.10.0</version>**

**<scope>test</scope>**

**</dependency>**

Now, JUnit is ready to go. Maven grabs it and handles everything behind the scenes.

**POM.XML FILE:**

The pom.xml file is purely a configuration file written in XML, not Java. It tells Maven *what* to do—like which dependencies your project needs (e.g., JUnit), what the project's packaging type is, what plugins to use, and how to handle the build lifecycle.

Think of it like this: Java files contain what your app does, while the pom.xml defines how your app is built and managed.

That said, you *can* influence Java behavior indirectly—like telling Maven to compile with Java 17 or to run tests a certain way using plugins.

For Junit5:

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.junit</groupId>

<artifactId>junit-bom</artifactId>

<version>5.11.0</version> <type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependencies>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-api</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-params</artifactId> <scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-engine</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>3.3.0</version> </plugin>

</plugins>

</build>

When u copy pom.xml also make sure to change group id, artifact id and name

**JUNIT DEPENDENCIES:**

JUnit dependencies are the libraries you include in your project to use JUnit’s testing features. The exact dependencies depend on which version of JUnit you're using:

For **JUnit 5** (a.k.a. JUnit Jupiter)

You typically include:

* **junit-jupiter-api** – the core API for writing tests (annotations like @Test, assertions, etc.)
* **junit-jupiter-engine** – the runtime engine that executes the tests
* **junit-jupiter-params** – for parameterized tests
* **junit-platform-launcher** – if you need to launch tests programmatically or integrate with IDEs and build tools

**MAVEN TEST vs SINGLE METHOD TEST (Junit) vs TEST CLASS TEST (Junit):**

In Eclipse, **maven test** is a way to run your project's unit tests using Maven's build lifecycle—without leaving the IDE.

Here’s how it works:

🛠️ What it Does

When you run maven test in Eclipse:

* It compiles your test code (src/test/java)
* It runs all test classes using the **Surefire plugin**
* It reports results in the console and generates test reports in target/surefire-reports

🧭 How to Run It in Eclipse

1. **Right-click** your Maven project in the Project Explorer.
2. Go to **Run As → Maven test**
3. Eclipse will execute the test phase of the Maven lifecycle.

If you don’t see the Maven options, make sure the **M2E plugin** (Maven integration for Eclipse) is installed.

**Run a Specific Test**

You can also run a specific test class or method:

* Right-click the test file placing cursor on method signature→ **Run As → JUnit Test**

**Run test class (Junit)**

* Right-click the test file → **Run As → JUnit Test**

**CREATING MAVEN PROJECT:**

* 1. **Archetypes:**

 Archetypes are templates. For a basic Java project, select:

* + - **maven-archetype-quickstart (apache is recommended version 1.5)**
      * **version 1.5 includes junit 5**
      * **if any other version is used, update pom.xml with junit5 dependencies tag**, **also change the by default test.java to junit5 syntaxes to avoid errors.**
  1. **Project Details:**
     + **Group Id**: This uniquely identifies your organization or project group. It typically follows a reverse domain name pattern (e.g., com.mycompany.projectname, org.apache.maven). It's a conceptual identifier for your group of related projects.
     + **Artifact Id**: This uniquely identifies the *specific project* or module within your groupId. It's usually the name of the project (e.g., my-application, data-processing-lib).
     + **Version**: leave default or customize
  2. **What are App.java and AppTest.java**

When you create a new Maven project, especially using archetypes like the maven-archetype-quickstart, it often generates two default files:

**App.java (in src/main/java):**

* + - **Purpose:** This file represents a very simple **"application" or "production code"** class. It's meant to be the code that your actual program would use.

**AppTest.java (in src/test/java):**

* **Purpose:** This file is the **unit test class** for your App.java code. It contains methods specifically designed to verify that the code in App.java (or any other production code) behaves as expected.

Use these as files through which we can access others… main file

So create other files as test and app in respective folders…. Follows modularity and all as well.

**Catalog:**

In Maven, a catalog—specifically an *archetype catalog*—is an XML file that lists available project templates (called *archetypes*) that you can use to generate new Maven projects.

When you create a Maven project using an archetype (like maven-archetype-quickstart), Maven checks catalogs to find and list available archetypes.

**CONTENTS OF POM.XML in version 1.5:**

📦 junit-jupiter-api

This is the **core API** for writing JUnit 5 tests. It provides:

* Annotations like @Test, @BeforeEach, @AfterAll
* Assertions like assertEquals(), assertTrue(), assertThrows()

Basically, without this, you can't *write* JUnit tests.

🔄 junit-jupiter-params

This adds support for **parameterized tests**—where a test method runs multiple times with different arguments.

With this, you can use:

* @ParameterizedTest
* @ValueSource, @CsvSource, @MethodSource, etc.

Great for testing functions with multiple inputs efficiently

🚀 junit-jupiter-engine

This is the **runtime engine** that actually executes your JUnit 5 tests. Think of it as the test runner.

Without it, Maven (via Surefire) won’t know how to *run* your annotated test classes—even if you've written them perfectly.

**BOM** stands for **Bill of Materials**. It’s a clever Maven feature that lets you manage all JUnit 5 dependency versions *in one place*.

Now you don’t need to write in every JUnit dependency—Maven pulls them all from the BOM, keeping things tidy and consistent.

It manages dependency versions in one centralized way