# J-UNIT:

# How do you mock dependencies in JUnit?

Use **Mockito** with @Mock, @InjectMocks, and Mockito.when().

1. **Which JUnit versions have you used?**

Commonly used versions are **JUnit 4** and **JUnit 5 (JUnit Jupiter)**.

**👉 One line:** JUnit 5 provides more powerful and flexible testing features than JUnit 4.

1. **What is the difference between JUnit 4 and JUnit 5?**

JUnit 5 has better support for **Java 8+ features**, supports **dynamic tests**, and has **modular architecture**.

**👉 One line:** JUnit 5 is more extensible and supports modern Java programming styles.

1. **What are the main JUnit annotations?**

* @Test – marks a test method
* @BeforeEach / @Before – runs before each test
* @AfterEach / @After – runs after each test
* @BeforeAll, @AfterAll – run once before/after all tests
* @Disabled – disables a test

1. **How do you test exceptions in JUnit?**

Use assertThrows() in JUnit 5 or expected in JUnit 4.

1. **How do you test private methods in JUnit?**

Private methods are tested indirectly through **public methods**; if needed, use **reflection**.

**👉 One line:** Private methods are tested by validating the public methods that use them.

1. **What is the difference between @Mock and @InjectMocks?**

* @Mock creates a mock instance.
* @InjectMocks injects the mocks into the class being tested.

1. **What is parameterized testing in JUnit?**

It allows you to run the **same test** multiple times with different input values.

1. **Can you test Spring components with JUnit?**

Yes, by using **Spring Boot Test** annotations like @SpringBootTest, @WebMvcTest, etc.

1. **What is the purpose of @SpringBootTest?**

It loads the **entire Spring application context** for integration testing.

1. **What is the difference between Unit Test and Integration Test?**

* Unit tests check **individual methods** in isolation.
* Integration tests check **multiple components** working together.

1. **What is the purpose of assertEquals() in JUnit?**

It checks whether the **expected and actual** values are equal.

1. **What is the difference between assertEquals() and assertSame()?**

* assertEquals() checks value equality
* assertSame() checks reference equality

1. **What is @Nested in JUnit 5?**

It allows grouping related test cases inside inner classes.  
**👉 One line:** @Nested helps organize related tests with shared setup code.

1. **How do you test void methods?**

Use **Mockito's verify()** to ensure the method was called with the right arguments.  
**👉 One line:** Verify behaviour or side effects instead of return values.

1. **How do you handle database testing in JUnit?**

Use @DataJpaTest for JPA-related testing with in-memory databases like H2.

1. **What is test coverage, and how do you measure it?**

Test coverage shows how much of your code is exercised by tests, measured using tools like **JaCoCo**.

1. **Can we use JUnit for integration testing?**

Yes, by using annotations like @SpringBootTest, @DataJpaTest, @WebMvcTest.

1. **What is the difference between Mockito and JUnit?**

* **JUnit** is a testing framework.
* **Mockito** is a mocking framework.

1. **Can we mock static methods in JUnit?**

Yes, using **Mockito 3.4+** with MockedStatic.

1. **What’s the difference between @BeforeEach and @BeforeAll?**

* @BeforeEach runs **before every test method**.
* @BeforeAll runs **once before the class starts**.

1. **How to write parameterized tests using JUnit 5?**

Use @ParameterizedTest with sources like @ValueSource, @CsvSource, or @MethodSource.

1. **What is @DisplayName in JUnit 5?**

Used to provide a custom name for a test method or class.

1. **What is @ExtendWith in JUnit 5?**

It allows you to **plug in custom extensions** (e.g., Mockito, timing, DB cleanup).  
**👉 One line:** @ExtendWith replaces @RunWith from JUnit 4 for extension support.

# Difference between unit testing and integration testing.

**Unit Testing**

* Tests a single component (method/class) in isolation
* Uses mocks/stubs for dependencies
* Very fast
* Validates individual logic correctness
* Example: Testing a Service method with a mocked Repository

**Integration Testing**

* Tests how multiple components/modules work together
* Uses real dependencies (DB, API, service, etc.)
* Slower than unit tests
* Validates interactions and data flow between modules
* Example: Testing a REST API flow (Controller → Service → Repository → DB)
* **Unit Testing** → Best for quickly validating **individual components** (fast feedback, easy debugging).
* **Integration Testing** → Best for ensuring **different modules/services work correctly together**.
* **Automation Testing** → Best for **reducing manual effort** and running tests (unit, integration, regression, UI) automatically for faster delivery.

# What is Functional testing:

* Testing a login feature → enter valid credentials, expect success; enter invalid credentials, expect error.
* Testing a payment module → ensure correct amount is deducted and receipt is generated.

# How test static method?

* **Directly test** static methods if they contain simple logic.
* **Use mocking** (Mockito / PowerMockito) when static methods depend on external systems (DB, APIs, Utils).
* @RunWith(PowerMockRunner.class) → required to enable PowerMockito.
* @PrepareForTest(Utils.class) → tells PowerMockito which class contains the static methods you want to mock.
* Use PowerMockito.mockStatic() to mock and when(...).thenReturn(...) to define behavior.

# 1. Public Methods

* **Definition:** Part of class API, accessible from anywhere.
* **How to Test:** Call directly in JUnit and assert expected results.
* **Example:**
* Calculator calc = new Calculator();
* assertEquals(5, calc.add(2, 3));
* **Key Point:** Standard practice; focus on behavior, not implementation.

# 2. Private Methods

* **Definition:** Internal implementation; not accessible from outside.
* **Testing Approaches:**

1. **Indirect via Public Method (Recommended):**assertEquals(18, calc.multiplyViaPublic(3, 6));
2. **Direct via Reflection (Not Recommended):**

* Method m = Calculator.class.getDeclaredMethod("multiply", int.class, int.class);
* m.setAccessible(true);
* int result = (int) m.invoke(calc, 4, 5);
* **Key Point:** Prefer indirect testing; reflection breaks encapsulation.

# 3. Static Methods

* **Definition:** Belong to the class, not an instance.
* **Testing Approaches:**

1. **Direct Call:** assertEquals(16, Calculator.square(4));
2. **Mocking (Optional, for complex cases):**

* try (MockedStatic<Calculator> mocked = mockStatic(Calculator.class)) {
* mocked.when(() -> Calculator.square(2)).thenReturn(100);
* assertEquals(100, Calculator.square(2));}
* **Key Point:** Easy to test if pure; mock if dependent on external resources.

|  |  |  |  |
| --- | --- | --- | --- |
| **Method Type** | **Access** | **How to Test** | **Notes** |
| Public | Anywhere | Direct call + assert  @Test, @BeforeEach/@AfterEach | Standard practice |
| Private | Class only | Indirect via public OR Reflection  @Test (via public method) | Prefer indirect; reflection = hack |
| Static | Class only | Direct call or mock with Mockito  @Test, Mockito annotations | Mock if external dependency exists |