Java Tesing

TDD (Test-Driven Development) Principles

Definition:
TDD (Test-Driven Development) is a **software development approach** where **tests are written before writing the actual code**, and development is guided by these tests.

- Rule of thumb: **Test → Code → Refactor**

Purpose of TDD

- Ensure **correctness from the beginning**
- Reduce **bugs and rework**
- Improve **code design and confidence**
- Make code **maintainable and testable**

Core TDD Cycle (RED–GREEN–REFACTOR)

1. **RED**
   - Write a test that **fails** (feature not implemented yet)
2. **GREEN**
   - Write **minimum code** to make the test pass
3. **REFACTOR**
   - Improve code structure **without changing behavior**

Key TDD Principles

1. **Write Test First**
   - Tests define **what the code should do**
   - No production code without a failing test

---

2. **Write Minimal Code**
   - Only write code **required to pass the test**
   - Avoid over-engineering

---

3. **Refactor Continuously**
   - Clean and improve code after tests pass
   - Maintain **readability and design quality**

---

4. **Small, Incremental Steps**
   - Develop features in **small chunks**
   - Easier debugging and better control

---

5. **Tests as Documentation**
   - Tests act as **living documentation**
   - Anyone can understand system behavior by reading tests

Simple Example (Java + JUnit)
Test First (RED):

```
@Test
void addTest() {
    Calculator c = new Calculator();
    assertEquals(5, c.add(2, 3));
}
```

Code (GREEN):

```
class Calculator {
    int add(int a, int b) {
```

```
        return a + b;
    }
}
```

Refactor (REFACTOR):

- Improve naming, structure if needed

Real-Life Analogy

- TDD → Preparing **exam answers first**, then studying exactly what's needed
- Test → Question paper
- Code → Your answer
- Refactor → Writing the answer neatly

Advantages

- Early **bug detection**
- High **test coverage**
- Cleaner and **maintainable code**
- Safer refactoring
- Builds **developer confidence**

Limitations

- Initial learning curve for beginners
- Slower at the beginning
- Not ideal for **UI-heavy or prototype-based work**

Common Interview Questions (Cognizant Level)

Q1. What is TDD?
A. Writing tests before code and developing using the Red-Green-Refactor cycle.

Q2. What are the steps in TDD?
A. Red → Green → Refactor

Q3. Difference between TDD and traditional testing?

- TDD → Tests written **before** code
- Traditional → Tests written **after** code

Q4. Does TDD replace testing?
A. No, it complements other testing types like integration and system testing.

Q5. Which tools are used for TDD in Java?
A. JUnit, Mockito, TestNG

One-Line Summary (Quick Revision)
TDD is a development approach where tests are written before code to ensure correctness, clean design, and maintainability.

Red–Green–Refactor Cycle (TDD Core Cycle)

Definition:
The **Red–Green–Refactor cycle** is the **core workflow of Test-Driven Development (TDD)** where code is written in **small, controlled steps** guided by tests.

- Order is **mandatory**: **Red → Green → Refactor**

Purpose of the Cycle

- Ensure **correct behavior before implementation**
- Reduce **bugs and regressions**
- Improve **code design continuously**

- Build **confidence through tests**

RED Phase (Write a Failing Test)
What happens:
- Write a **test first**
- Test **fails** because feature is not implemented yet

Why RED is important:
- Confirms the test is **valid**
- Defines **expected behavior clearly**

Example:
```
@Test
void multiplyTest() {
    Calculator c = new Calculator();
    assertEquals(6, c.multiply(2, 3)); // Fails → RED
}
```

GREEN Phase (Make the Test Pass)
What happens:
- Write **minimum code** to pass the test
- No optimization, no extra features

Why GREEN is important:
- Focuses only on **correctness**
- Avoids over-engineering

Example:
```
class Calculator {
    int multiply(int a, int b) {
        return a * b; // Passes → GREEN
    }
}
```

REFACTOR Phase (Improve Code Quality)
What happens:
- Improve **design, readability, structure**
- Tests must **still pass**
- No change in functionality

Why REFACTOR is important:
- Keeps code **clean and maintainable**
- Prevents **technical debt**

Example:
- Rename variables
- Remove duplication
- Improve method structure

Key Rules to Remember
- No production code without a failing test
- No refactoring before GREEN
- Refactor only when tests pass
- Repeat cycle for **every small feature**

Real-Life Analogy
- **RED** → Teacher sets a **question you can't answer yet**
- **GREEN** → You write the **correct answer**
- **REFACTOR** → You **rewrite neatly and clearly**

Advantages

- Early **bug detection**
- Better **code design**
- High **test coverage**
- Safe and confident **refactoring**

Common Interview Questions (Cognizant Level)

Q1. What is Red–Green–Refactor?
A. Core TDD cycle: failing test → minimal code → improve design.
Q2. Why is RED important?
A. Ensures the test is valid and behavior is well-defined.
Q3. What should be done in GREEN phase?
A. Write only the minimum code to pass the test.
Q4. When should refactoring be done?
A. Only after all tests pass (GREEN).
Q5. Can functionality be changed during refactoring?
A. No, only structure and readability.

One-Line Summary (Quick Revision)

Red–Green–Refactor is a TDD cycle where you write a failing test, make it pass with minimal code, then improve the code safely.

Unit Testing

Definition:
Unit Testing is a **software testing technique** where **individual units or components (methods, functions, or classes)** are tested in isolation to verify they work correctly.

- A **unit** = smallest testable part of code (method/class).

Purpose of Unit Testing

- Detect **bugs early** in development
- Ensure each method works **as expected**
- Improve **code quality and reliability**
- Support **refactoring without fear**
- Reduce **cost of fixing errors later**

What is Tested in Unit Testing

- Methods
- Functions
- Classes
- Modules (small components)

Not tested: Database, UI, network (that is integration/system testing).

Key Characteristics of Unit Testing

1. **Automated** – Written using tools like JUnit
2. **Isolated** – Tests one unit without dependencies
3. **Repeatable** – Can run multiple times
4. **Fast** – Executes quickly
5. **Independent** – One test does not depend on another

Simple Java Example (JUnit)

Class to Test

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }
}
```

Unit Test

```
@Test
void addTest() {
    Calculator c = new Calculator();
    assertEquals(5, c.add(2, 3));
}
```

- This test checks only the **add() method** → one unit.

Real-Life Analogy

- Unit Testing → Checking **each part of a machine separately**
- Example: Test engine, brakes, lights individually before assembling a car.

Advantages

- Finds bugs **early**
- Improves **code design**
- Makes refactoring **safe**
- Saves **time and cost**
- Acts as **documentation**

Limitations

- Cannot test **full system behavior**
- Requires **extra time to write tests**
- Mocking dependencies can be complex

Common Interview Questions (Cognizant Level)

Q1. What is Unit Testing?

A. Testing individual units of code to verify correctness.

Q2. Who writes unit tests?

A. Developers.

Q3. Difference between Unit Testing and Integration Testing?

- Unit → Single component
- Integration → Multiple components together

Q4. Tools used for Unit Testing in Java?

A. JUnit, TestNG, Mockito

Q5. Is unit testing manual or automated?

A. Mostly automated.

One-Line Summary (Quick Revision)

Unit Testing verifies individual components of software to ensure they work correctly before integration.

Introduction to JUnit

Definition

JUnit is a **Java-based unit testing framework** used to **write and execute automated tests** to verify that individual units of Java code work correctly.

- It is the **most widely used testing framework** in Java.

Purpose of JUnit

- Automate **unit testing**
- Detect **bugs early**
- Support **TDD (Test-Driven Development)**
- Improve **code quality and confidence**
- Enable **safe refactoring**

Key Features of JUnit

- Simple and **annotation-based**
- Supports **assertions** to verify expected results
- Integrates with **IDE tools** (IntelliJ, Eclipse)
- Supports **test suites**
- Provides **test lifecycle management**

Versions of JUnit

Version	Description
JUnit 4	Annotation-based, widely used earlier
JUnit 5	Modern, modular, more powerful (**recommended**)

- JUnit 5 = **JUnit Platform + Jupiter + Vintage**

Basic Annotations in JUnit 5

Annotation	Purpose
@Test	Marks a test method
@BeforeEach	Runs before each test
@AfterEach	Runs after each test
@BeforeAll	Runs once before all tests
@AfterAll	Runs once after all tests
@Disabled	Skips a test

Simple JUnit Example

Class to Test

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }
}
```

JUnit Test

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class CalculatorTest {

    @Test
    void addTest() {
        Calculator c = new Calculator();
```

```
        assertEquals(5, c.add(2, 3));
    }
}
```

Common Assertions

Assertion	Purpose
assertEquals()	Compare expected and actual
assertTrue()	Check condition is true
assertFalse()	Check condition is false
assertNull()	Check object is null
assertNotNull()	Check object is not null
assertThrows()	Check exception is thrown

Real-Life Analogy

- JUnit → Quality checker
- Test case → Checklist item
- Assertion → Pass/Fail rule

Advantages

- Automates testing
- Saves time and effort
- Improves **code reliability**
- Encourages **TDD practices**
- Easy IDE and CI integration

Common Interview Questions (Cognizant Level)

Q1. What is JUnit?
A. Java unit testing framework.
Q2. Which JUnit version is recommended?
A. JUnit 5.
Q3. What is @Test annotation?
A. Marks a method as a test case.
Q4. What are assertions in JUnit?
A. Methods used to validate expected output.
Q5. Does JUnit support TDD?
A. Yes, JUnit is widely used for TDD.

One-Line Summary (Quick Revision)

JUnit is a Java framework used to write and run automated unit tests to ensure code correctness.

JUnit 5 Architecture

Definition

JUnit 5 Architecture defines the **modular structure** of JUnit that separates **test execution**, **test writing**, and **backward compatibility**.

- JUnit 5 is **not a single jar**; it is a **platform with multiple components**.

Why New Architecture in JUnit 5

- Support **multiple testing frameworks** on one platform

- Better **extensibility and flexibility**
- Clean separation between **engine and API**
- Backward compatibility with **JUnit 4**

JUnit 5 Architecture Overview
JUnit 5 consists of **three main modules**:

```
JUnit 5
 ├── JUnit Platform
 ├── JUnit Jupiter
 └── JUnit Vintage
```

JUnit Platform

Foundation layer
Purpose:
- Launches testing frameworks on JVM
- Provides **TestEngine API**
- Integrates with **IDEs, build tools (Maven/Gradle)**

Key Role:
Responsible for **discovering and executing tests**

JUnit Jupiter

Modern testing API
Purpose:
- Provides **annotations and assertions** for writing tests
- Supports **JUnit 5 test model**

Examples:
- Annotations: @Test, @BeforeEach, @AfterEach
- Assertions: assertEquals(), assertThrows()

This is what **developers mostly use**

JUnit Vintage

Backward compatibility module
Purpose:
- Allows running **JUnit 3 & JUnit 4 tests**
- Ensures **legacy test support**

Used when migrating old projects to JUnit 5

Architecture Flow (Execution)
1. Developer writes test using **JUnit Jupiter API**
2. IDE/Build Tool invokes **JUnit Platform**
3. Platform discovers tests
4. Appropriate **Test Engine** runs tests
5. Results are reported back

Real-Life Analogy
- **JUnit Platform** → Power supply & control unit
- **JUnit Jupiter** → New modern appliances
- **JUnit Vintage** → Old appliances still supported

Advantages of JUnit 5 Architecture
- Modular and extensible
- Supports multiple test engines

- Cleaner APIs
- Better IDE and CI integration
- Smooth migration from JUnit 4

Common Interview Questions (Cognizant Level)
Q1. What are the main components of JUnit 5?
A. Platform, Jupiter, Vintage.
Q2. Which module is used to write tests?
A. JUnit Jupiter.
Q3. Which module runs JUnit 4 tests?
A. JUnit Vintage.
Q4. What is the role of JUnit Platform?
A. Test discovery and execution.
Q5. Is JUnit 5 backward compatible?
A. Yes, using JUnit Vintage.

One-Line Summary (Quick Revision)
JUnit 5 architecture consists of Platform (execution), Jupiter (new tests), and Vintage (legacy tests support).

JUnit Annotations (JUnit 5)

Definition
JUnit Annotations are **metadata markers** used to **control test execution flow**, define **test lifecycle methods**, and manage **test behavior** in JUnit.

- They tell JUnit **what to run and when.**

Why Annotations are Needed

- Identify **test methods**
- Manage **setup and cleanup**
- Control **test execution**
- Improve **readability and structure**

Core JUnit 5 Annotations
@Test

- Marks a method as a **test case**

```
@Test
void additionTest() {
    assertEquals(5, 2 + 3);
}
```

@BeforeEach

- Runs **before each test method**
- Used for **test setup**

```
@BeforeEach
void setUp() {
    calculator = new Calculator();
}
```

@AfterEach

- Runs **after each test method**
- Used for **cleanup**

```
@AfterEach
void tearDown() {
    calculator = null;
}
```

@BeforeAll

- Runs **once before all tests**
- Must be **static**

```
@BeforeAll
static void initAll() {
    System.out.println("Start Tests");
}
```

@AfterAll

- Runs **once after all tests**
- Must be **static**

```
@AfterAll
static void cleanUpAll() {
    System.out.println("End Tests");
}
```

Test Control Annotations

@Disabled

- Skips a test method or class

```
@Disabled("Feature under development")
@Test
void skippedTest() {}
```

@DisplayName

- Custom name for test

```
@DisplayName("Addition Test Case")
@Test
void testAdd() {}
```

@RepeatedTest

- Runs a test multiple times

```
@RepeatedTest(3)
void repeatTest() {}
```

Parameterized Testing Annotations

@ParameterizedTest

- Runs same test with different inputs

```
@ParameterizedTest
@ValueSource(ints = {1, 2, 3})
void testPositive(int number) {
    assertTrue(number > 0);
}
```

Exception Testing Annotation

@Test + assertThrows

```
@Test
void exceptionTest() {
```

```
    assertThrows(ArithmeticException.class, () -> {
        int x = 10 / 0;
    });
}
```

Execution Order Summary

```
@BeforeAll
  @BeforeEach
    @Test
  @AfterEach
@AfterAll
```

Real-Life Analogy

- @BeforeAll → Office opens
- @BeforeEach → Desk setup
- @Test → Actual work
- @AfterEach → Desk cleanup
- @AfterAll → Office closes

Common Interview Questions (Cognizant Level)

Q1. Difference between @BeforeEach and @BeforeAll?
A. BeforeEach runs before every test; BeforeAll runs once.

Q2. Why @BeforeAll must be static?
A. It runs before object creation.

Q3. How to skip a test in JUnit 5?
A. Use @Disabled.

Q4. Which annotation supports multiple inputs?
A. @ParameterizedTest.

Q5. How to test exceptions?
A. assertThrows().

One-Line Summary (Quick Revision)

JUnit annotations define test methods, lifecycle events, execution control, and parameterized testing in JUnit 5.

JUnit Test Structure (JUnit 5)

Definition

JUnit Test Structure defines the **standard way a test class and test methods are organized** so tests are **readable, maintainable, and reliable.**

- It follows a **clear Arrange–Act–Assert (AAA)** pattern.

High-Level Test Structure

```
Test Class
 ├── Setup Methods
 ├── Test Methods
 └── Cleanup Methods
```

Standard JUnit Test Class Structure

Basic Template

```
class CalculatorTest {
```

```
    Calculator calculator;

    @BeforeEach
    void setUp() {
        calculator = new Calculator();
    }

    @Test
    void addTest() {
        int result = calculator.add(2, 3);
        assertEquals(5, result);
    }

    @AfterEach
    void tearDown() {
        calculator = null;
    }
}
```

AAA Pattern (Most Important for Interviews)

Arrange

- Prepare objects and input data

```
Calculator calculator = new Calculator();
```

Act

- Call the method under test

```
int result = calculator.add(2, 3);
```

Assert

- Verify the result

```
assertEquals(5, result);
```

Interview keyword: *JUnit follows AAA pattern*

Detailed Components of JUnit Test Structure

Test Class

- Normal Java class
- Usually named: ClassNameTest

```
class UserServiceTest {}
```

Setup Methods

Annotation	Purpose
@BeforeAll	Runs once before all tests
@BeforeEach	Runs before each test

Test Methods

- Marked with @Test
- Must be **void**
- No parameters (unless parameterized test)

```
@Test
void loginSuccessTest() {}
```

Cleanup Methods

Annotation	Purpose
@AfterEach	Cleanup after each test

Annotation	Purpose
@AfterAll	Cleanup after all tests

Execution Flow

```
@BeforeAll
   @BeforeEach
      @Test
   @AfterEach
@AfterAll
```

Real-Life Analogy

- **Arrange** → Gather ingredients
- **Act** → Cook food
- **Assert** → Taste & verify
- **Cleanup** → Wash dishes

Best Practices (Cognizant Focus)

- One assertion per test (preferred)
- Test method name should describe behavior
- Avoid test dependency
- Keep tests independent

Common Interview Questions

Q1. What is AAA pattern in JUnit?
A. Arrange, Act, Assert.
Q2. Can we have multiple @Test methods?
A. Yes.
Q3. Why test class name ends with Test?
A. Naming convention for clarity and tools.
Q4. Can test methods return value?
A. No, must be void.

One-Line Summary

JUnit test structure follows AAA pattern using setup, test, and cleanup methods for clean and reliable testing.

JUnit Assertions (JUnit 5)

Definition

Assertions are **methods used to verify expected vs actual results** in a test.
If the assertion fails → **test fails.**

- *Assertion = decision maker of a test*

Why Assertions Are Needed

- Validate **business logic**
- Ensure **correct output**
- Detect **bugs early**
- Decide **pass / fail**

Assertion Pattern (AAA Reminder)

```
// Arrange
int a = 5, b = 3;

// Act
int result = a + b;

// Assert
assertEquals(8, result);
```

Most Common JUnit Assertions

assertEquals(expected, actual)
Checks equality

```
assertEquals(10, calculator.add(5,5));
```

---

assertNotEquals()
Ensures values are different

```
assertNotEquals(0, result);
```

---

assertTrue() / assertFalse()
Validates conditions

```
assertTrue(age >= 18);
assertFalse(isBlocked);
```

---

assertNull() / assertNotNull()
Checks object reference

```
assertNotNull(user);
assertNull(errorMessage);
```

---

assertThrows()
Verifies exception

```
assertThrows(ArithmeticException.class, () -> {
    int x = 10 / 0;
});
```

---

assertDoesNotThrow()
Ensures no exception occurs

```
assertDoesNotThrow(() -> service.process());
```

---

assertSame() / assertNotSame()
Checks **object reference**, not value

```
assertSame(obj1, obj2);
```

Grouped Assertions
assertAll()
Executes multiple assertions together

```
assertAll(
    () -> assertEquals("John", user.getName()),
    () -> assertEquals(25, user.getAge())
);
```

All assertions are checked even if one fails.

Custom Failure Messages

```
assertEquals(100, balance, "Balance mismatch!");
```

Helps in debugging during interviews & projects

Real-Life Analogy

- Expected result → **Answer key**
- Actual result → **Student answer**
- Assertion → **Teacher checking**

Best Practices (Interview Focus)

- One main assertion per test
- Use meaningful messages
- Prefer assertThrows for exceptions
- Avoid logic inside assertions

Common Interview Questions

Q1. What happens if an assertion fails?
A. Test case fails immediately.
Q2. Difference between assertEquals and assertSame?
A. Equals → value, Same → reference.
Q3. How to test exceptions?
A. assertThrows().
Q4. Can we write multiple assertions in one test?
A. Yes, using assertAll().

Quick Revision Table

Assertion	Purpose
assertEquals	Value check
assertTrue	Condition
assertNull	Null check
assertThrows	Exception
assertAll	Group assertions

One-Line Summary
Assertions validate expected behavior and decide whether a JUnit test passes or fails.

_____

JUnit Test Lifecycle (JUnit 5)

Definition
Test Lifecycle describes the **order in which JUnit executes setup, test, and cleanup methods** for a test class.

- It explains **what runs before, during, and after tests.**

Why Test Lifecycle Is Important

- Proper **resource initialization**
- Avoid **test interference**
- Ensure **clean & reliable tests**
- Control **execution flow**

JUnit 5 Lifecycle Annotations

Annotation	When it Runs
@BeforeAll	Once before all tests

Annotation	When it Runs
@BeforeEach	Before each test
@Test	Actual test
@AfterEach	After each test
@AfterAll	Once after all tests

Execution Flow (Most Asked in Interviews)

```
@BeforeAll
   @BeforeEach
      @Test
   @AfterEach
@AfterAll
```

Simple Lifecycle Example

```
class LifeCycleTest {

    @BeforeAll
    static void initAll() {
        System.out.println("Before All");
    }

    @BeforeEach
    void init() {
        System.out.println("Before Each");
    }

    @Test
    void testOne() {
        System.out.println("Test One");
    }

    @AfterEach
    void cleanUp() {
        System.out.println("After Each");
    }

    @AfterAll
    static void tearDownAll() {
        System.out.println("After All");
    }
}
```

Output Order

```
Before All
Before Each
Test One
After Each
After All
```

Key Rules (Important for Freshers)

- @BeforeAll & @AfterAll must be **static**
- @BeforeEach runs **before every test**
- New test instance is created **for each test**
- Tests should be **independent**

Real-Life Analogy

- @BeforeAll → Office opens
- @BeforeEach → Desk setup
- @Test → Work
- @AfterEach → Desk cleanup
- @AfterAll → Office closes

Lifecycle vs Test Instance
Default Behavior

- One **new object per test**
- Prevents **shared state bugs**

Common Interview Questions
Q1. Why @BeforeAll is static?
A. Runs before object creation.
Q2. How many times @BeforeEach runs?
A. Once for each test method.
Q3. Can @AfterEach fail the test?
A. Yes, if exception occurs.
Q4. What happens if @BeforeEach fails?
A. Test is skipped.

One-Line Summary
JUnit Test Lifecycle controls the execution order of setup, test, and cleanup methods to ensure reliable testing.

Parameterized Tests (JUnit 5)

Definition
Parameterized Tests allow you to **run the same test logic multiple times with different input values.**

- *One test method + many data sets*

Why Parameterized Tests Are Needed

- Avoid **duplicate test code**
- Improve **coverage**
- Test **multiple scenarios easily**
- Clean & maintainable tests

Basic Structure (Pattern)

```
@ParameterizedTest
@DataSource
Test Method(with parameters)
Assertions
```

Required Annotation
@ParameterizedTest
Marks a test method that receives parameters.

```
@ParameterizedTest
void testMethod(int value) {}
```

Common Parameter Sources

@ValueSource

Single parameter

```
@ParameterizedTest
@ValueSource(ints = {2, 4, 6})
void testEvenNumbers(int number) {
    assertTrue(number % 2 == 0);
}
```

@CsvSource

Multiple parameters in one line

```
@ParameterizedTest
@CsvSource({
    "2,3,5",
    "5,5,10"
})
void testAddition(int a, int b, int sum) {
    assertEquals(sum, a + b);
}
```

@CsvFileSource

Read data from CSV file

```
@ParameterizedTest
@CsvFileSource(resources = "/data.csv", numLinesToSkip = 1)
void testFromFile(int a, int b, int sum) {}
```

@MethodSource

Dynamic data using method

```
@ParameterizedTest
@MethodSource("numberProvider")
void testNumbers(int num) {
    assertTrue(num > 0);
}

static Stream<Integer> numberProvider() {
    return Stream.of(1, 2, 3);
}
```

@EnumSource

Enum values

```
@ParameterizedTest
@EnumSource(DayOfWeek.class)
void testEnum(DayOfWeek day) {
    assertNotNull(day);
}
```

Execution Flow

For each input:

```
@BeforeEach
  Parameterized Test
@AfterEach
```

Real-Life Analogy

- Same exam paper
- Different students

- Same evaluation logic

Best Practices (Interview Focus)

- Keep test logic simple
- Use meaningful parameter names
- Prefer @MethodSource for complex data
- Avoid hardcoding large data inside test

Common Interview Questions

Q1. Difference between @Test and @ParameterizedTest?
A. @Test runs once, parameterized runs multiple times.
Q2. Can parameterized tests use lifecycle methods?
A. Yes (@BeforeEach, @AfterEach).
Q3. Which annotation supports multiple parameters?
A. @CsvSource.
Q4. Which is best for dynamic data?
A. @MethodSource.

Quick Comparison Table

Source	Use Case
@ValueSource	Single input
@CsvSource	Multiple inputs
@MethodSource	Complex/dynamic data
@EnumSource	Enum testing

One-Line Summary
Parameterized tests execute the same test logic multiple times using different input values, improving coverage and reducing code duplication.

@ValueSource & @CsvSource (JUnit 5)

Definition
@ValueSource and @CsvSource are **parameter providers** used with @ParameterizedTest to supply **test data.**

- They allow **same test logic** to run with **different inputs.**

When to Use Which (Quick Idea)

- **@ValueSource → Single parameter**
- **@CsvSource → Multiple parameters**

@ValueSource
Purpose

- Supplies **one value per test run**
- Used for **simple inputs**

Syntax

```
@ParameterizedTest
@ValueSource(ints = {1, 2, 3})
void testPositive(int number) {
    assertTrue(number > 0);
```

```
}
```

Supported Types

- ints
- longs
- doubles
- strings
- booleans
- classes

Does NOT support null

Real-Life Example

Checking multiple user ages:

```
@ParameterizedTest
@ValueSource(ints = {18, 21, 25})
void testValidAge(int age) {
    assertTrue(age >= 18);
}
```

Interview Point

Best for **one input, one assertion**

@CsvSource

Purpose

- Supplies **multiple values per test run**
- Used when **input & expected output** are required

Syntax

```
@ParameterizedTest
@CsvSource({
    "2,3,5",
    "5,5,10"
})
void testAddition(int a, int b, int sum) {
    assertEquals(sum, a + b);
}
```

Handling Strings

```
@CsvSource({
    "'John', 25",
    "'Alice', 30"
})
void testUser(String name, int age) {
    assertNotNull(name);
}
```

Null Values

```
@CsvSource({
    "null, 0"
})
void testNull(String input, int value) {
    assertNull(input);
}
```

Interview Point

Best for **input + expected output**

Execution Flow
For each row/value:
@BeforeEach
Parameterized Test
@AfterEach

Comparison Table (Very Important)

Feature	@ValueSource	@CsvSource
Parameters	Single	Multiple
Complexity	Simple	Medium
Null Support	No	Yes
Use Case	Validation	Calculation

Common Interview Questions
Q1. Can @ValueSource pass multiple parameters?
A. No.
Q2. Which annotation supports null values?
A. @CsvSource.
Q3. Can we use both together?
A. No, only one source per test.
Q4. When to avoid @ValueSource?
A. When expected output is needed.

Best Practices

- Use @ValueSource for validation checks
- Use @CsvSource for business logic
- Keep test data readable
- Avoid complex logic inside test

One-Line Summary
@ValueSource is for single-parameter testing, while @CsvSource is for multi-parameter and expected-result testing.

Test Suites (JUnit 5)

Definition:
A **Test Suite** is a **collection of multiple test classes** that are executed **together as a single unit**.

- *Run many tests with one click*

Why Test Suites Are Needed

- Run **related tests together**
- Organize tests **module-wise**
- Save **execution time**
- Useful for **regression testing**

Test Suite Pattern (Structure)

```
Test Suite Class
 ├── TestClass1
```

```
 ├── TestClass2
 └── TestClass3
```

JUnit 5 Test Suite Annotations
Required Annotations

```
@Suite
@SelectClasses({TestClass1.class, TestClass2.class})
```

Simple Test Suite Example

```
import org.junit.platform.suite.api.SelectClasses;
import org.junit.platform.suite.api.Suite;

@Suite
@SelectClasses({
    CalculatorTest.class,
    UserServiceTest.class
})
class ApplicationTestSuite {
}
```

Running ApplicationTestSuite runs **all listed tests.**

Selecting Tests in Different Ways
@SelectPackages
Run all tests in a package

```
@Suite
@SelectPackages("com.app.service")
class ServiceTestSuite {}
```

@IncludeTags / @ExcludeTags
Run tests by tag

```
@Suite
@IncludeTags("fast")
class FastTestSuite {}
```

Tagging Tests

```
@Test
@Tag("fast")
void quickTest() {}
```

Execution Flow

- Test suite **does not have lifecycle methods**
- Each test class follows its **own lifecycle**

Real-Life Analogy

- Playlist → Test Suite
- Songs → Test Classes
- Play button → Run Suite

Best Practices (Interview Focus)

- Group tests logically
- Use tags for flexible execution
- Keep suite class empty (no logic)
- Use meaningful suite names

Common Interview Questions

Q1. Can a test suite contain methods?
A. No, only configuration.

Q2. Can we run specific tests using tags?
A. Yes, using @IncludeTags.

Q3. Does test suite control execution order?
A. No (JUnit does not guarantee order).

Q4. Difference between @SelectClasses and @SelectPackages?
A. Class-based vs package-based selection.

Quick Comparison

Feature	Test Class	Test Suite
Purpose	Test logic	Group tests
Contains tests	Yes	No
Lifecycle	Yes	No

One-Line Summary

Test suites group and execute multiple test classes together for organized and efficient testing.

Nested Tests (JUnit 5)

Definition:

Nested Tests allow you to **group related test cases inside an outer test class** using inner classes.

- Used to represent **hierarchical / scenario-based testing**

Why Nested Tests Are Needed

- Improve **readability**
- Organize tests **by feature or condition**
- Mimic **real-world scenarios**
- Cleaner test structure

Basic Pattern (Structure)

```
Outer Test Class
 ├── @Test
 └── @Nested Inner Class
       ├── @Test
       └── @Test
```

Required Annotation

@Nested

- Used on **inner non-static classes**
- Inner class **represents a test context**

Simple Example

```
class CalculatorTest {

    Calculator calculator = new Calculator();

    @Test
```

```
    void generalTest() {
        assertNotNull(calculator);
    }

    @Nested
    class AdditionTests {

        @Test
        void addPositiveNumbers() {
            assertEquals(5, calculator.add(2, 3));
        }

        @Test
        void addNegativeNumbers() {
            assertEquals(-5, calculator.add(-2, -3));
        }
    }
}
```

Lifecycle Behavior (Important)

Annotation	Scope
@BeforeEach (outer)	Runs before nested tests
@BeforeEach (inner)	Runs before inner tests
@BeforeAll	Not allowed inside nested

- Each nested class has its **own lifecycle**

Execution Flow

Outer @BeforeEach
Inner @BeforeEach
@Test
Inner @AfterEach
Outer @AfterEach

Real-Life Analogy

- **Restaurant**
  - Menu → Test Class
  - Category (Starters) → Nested Class
  - Dish → Test Method

Best Practices (Interview Focus)

- Use nested tests for **context-based scenarios**
- Keep nesting **1–2 levels only**
- Name inner classes clearly
- Avoid complex logic inside tests

Common Interview Questions

Q1. Why use @Nested?
A. To group related test cases.
Q2. Can @BeforeAll be used in nested tests?
A. No.
Q3. Are nested test classes static?
A. No, must be non-static.
Q4. Does nested testing affect test isolation?
A. No, each test is still isolated.

Quick Comparison

Feature	Normal Test	Nested Test
Structure	Flat	Hierarchical
Readability	Medium	High
Use case	Simple tests	Scenario-based tests

One-Line Summary
Nested tests organize related test cases into hierarchical structures, improving clarity and maintainability.

Dynamic Tests (JUnit 5)

Definition
Dynamic Tests are tests **generated at runtime**, rather than being statically defined with @Test.

- Useful when **test cases are not known at compile time.**

Why Dynamic Tests Are Needed

- Handle **data-driven scenarios**
- Generate **tests programmatically**
- Useful for **collections, loops, or external data sources**
- Reduce **boilerplate code**

Key Features

- Created using **@TestFactory**
- Returns **DynamicTest instances** or **Collection/Stream**
- Can **iterate over multiple inputs dynamically**

Required Annotations

Annotation	Purpose
@TestFactory	Marks method as factory for dynamic tests
DynamicTest.dynamicTest	Creates individual dynamic test

Simple Example

```
import static org.junit.jupiter.api.DynamicTest.dynamicTest;
import org.junit.jupiter.api.*;

import java.util.Arrays;
import java.util.Collection;

class DynamicTestExample {

    @TestFactory
    Collection<DynamicTest> dynamicTests() {
        return Arrays.asList(
            dynamicTest("1 + 1 = 2", () -> Assertions.assertEquals(2, 1 + 1)),
            dynamicTest("2 * 2 = 4", () -> Assertions.assertEquals(4, 2 * 2))
        );
    }
```

```
}
```

Using Streams (Recommended)

```
@TestFactory
Stream<DynamicTest> streamTests() {
    return Stream.of(2, 4, 6)
        .map(n -> dynamicTest("Check even: " + n,
                () -> Assertions.assertTrue(n % 2 == 0)));
}
```

Execution Flow

```
@TestFactory method runs
  Generates DynamicTest instances
Each DynamicTest executes with its own lifecycle
```

Real-Life Analogy

- Restaurant → Daily specials menu changes
- Chef generates dishes dynamically → Dynamic Tests generate test cases at runtime

---

Best Practices (Interview Focus)

- Use for **runtime-generated scenarios**
- Keep assertions **simple**
- Avoid complex logic inside test factory
- Use streams for **readability**

Common Interview Questions

Q1. Difference between @Test and @TestFactory?
A. @Test → static test, @TestFactory → dynamic test at runtime.
Q2. Can dynamic tests be parameterized?
A. Yes, using collections or streams.
Q3. How many dynamic tests can a factory return?
A. Any number, runtime determined.
Q4. Do dynamic tests have annotations like @BeforeEach?
A. Lifecycle annotations apply per dynamic test instance.

Quick Comparison

Feature	@Test	@TestFactory
Known at compile	Yes	No (runtime)
Repetition	Single	Multiple
Data source	Fixed	Collection / Stream

One-Line Summary

Dynamic Tests are runtime-generated test cases using @TestFactory, allowing flexible and programmatic testing.

Assumptions in JUnit 5

Definition:
Assumptions are **conditions checked at runtime** to decide whether a test should **run or be skipped.**

- If an assumption **fails,** the test is **ignored,** not failed.

Why Assumptions Are Needed

- Handle **environment-dependent tests**
- Skip tests if **preconditions are not met**
- Avoid **false failures** in tests
- Useful for **OS, DB, or configuration specific tests**

Key Methods for Assumptions (org.junit.jupiter.api.Assumptions)

Method	Purpose
assumeTrue(condition)	Runs test only if true
assumeFalse(condition)	Runs test only if false
assumingThat(condition, executable)	Executes block only if condition is true

---

Simple Example

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assumptions.assumeTrue;

class AssumptionExample {

    @Test
    void testOnlyOnDev() {
        String env = System.getProperty("env"); // "dev" or "prod"
        assumeTrue("dev".equals(env));
        System.out.println("Running test only in dev environment");
    }
}
```

- If env != "dev", test is **skipped.**

Using assumingThat

```
import static org.junit.jupiter.api.Assumptions.assumingThat;

@Test
void testConditionalLogic() {
    boolean isFeatureEnabled = false;
    assumingThat(isFeatureEnabled, () -> {
        System.out.println("Feature-specific test runs");
    });
    System.out.println("Other test logic always runs");
}
```

Execution Flow

1. Assumption checked
2. If **true → test runs**
3. If **false → test skipped, not failed**

Real-Life Analogy

- Exam → Only students with ID allowed
- ID check → Assumption
- Fail check → Test skipped, not marked wrong

Best Practices (Cognizant Focus)

- Use for **environment-dependent or conditional tests**
- Avoid assumptions in **critical unit tests**
- Keep **assumption logic simple**

Common Interview Questions

Q1. What happens if an assumption fails?

A. Test is skipped, not failed.

Q2. Difference between assumeTrue and assertTrue?

- assumeTrue → skips test if false
- assertTrue → fails test if false

Q3. Can assumptions be used with dynamic tests?

A. Yes, per test execution.

Q4. When to prefer assumptions over assertions?

A. When preconditions determine whether the test should run.

One-Line Summary

Assumptions let JUnit tests run only if preconditions are met, skipping the test otherwise instead of failing it.

Timeouts and Exceptions in JUnit 5

Definition

- **Timeouts** → Ensure a test **completes within a specified time**. Test fails if it **exceeds the limit**.
- **Exceptions** → Verify that a test **throws an expected exception**.

Both are crucial for **robust, reliable unit tests**.

Why Timeouts and Exception Testing Are Needed

- **Timeouts**: Detect **infinite loops** or **slow performance**.
- **Exception Testing**: Ensure **error handling** works correctly.
- Helps in **TDD and defensive coding**.

Timeout Testing

Using assertTimeout()

- Fails if test exceeds specified duration.

```
import static org.junit.jupiter.api.Assertions.*;
import java.time.Duration;
import org.junit.jupiter.api.Test;

class TimeoutExample {

    @Test
    void testWithTimeout() {
        assertTimeout(Duration.ofSeconds(2), () -> {
            Thread.sleep(1000); // completes within time
        });
    }
}
```

Using assertTimeoutPreemptively()

- Stops the test **immediately** if timeout exceeded.

```
@Test
void testPreemptiveTimeout() {
    assertTimeoutPreemptively(Duration.ofSeconds(1), () -> {
        Thread.sleep(2000); // test fails immediately
    });
```

```
}
```

Exception Testing
Using assertThrows()

- Ensures the **expected exception is thrown**

```
@Test
void testDivideByZero() {
    assertThrows(ArithmeticException.class, () -> {
        int x = 10 / 0;
    });
}
```

Capturing the Exception

- To check **exception message**

```
@Test
void testExceptionMessage() {
    ArithmeticException ex = assertThrows(ArithmeticException.class, () -> {
        int x = 10 / 0;
    });
    assertEquals("/ by zero", ex.getMessage());
}
```

Real-Life Analogy

- **Timeout** → Traffic light timer, stop if it exceeds
- **Exception** → Safety alarm, ensures errors are handled

Key Rules

- Use **Duration.ofSeconds()/Millis()** for clarity
- Prefer **assertTimeout** over Thread.sleep in real tests
- assertThrows is better than try-catch in tests

Common Interview Questions

Q1. Difference between assertTimeout and assertTimeoutPreemptively?
A. Preemptively stops execution immediately, assertTimeout waits until execution finishes.

Q2. Can we check exception message?
A. Yes, capture exception with assertThrows.

Q3. Can timeout be used with dynamic/parameterized tests?
A. Yes, applied per test invocation.

Q4. What happens if test exceeds timeout?
A. Test fails.

Quick Summary Table

Feature	Purpose	Example
Timeout	Fail if test too long	assertTimeout(Duration.ofSeconds(2), ...)
Exception	Validate expected errors	assertThrows(ArithmeticException.class, ...)

One-Line Summary

Timeouts ensure tests complete within time, and exceptions validate proper error handling in JUnit 5.

Mocking in Unit Testing (Mockito / JUnit 5)

Definition

Mocking is the practice of creating **fake objects** that **simulate the behavior of real dependencies** in unit tests.

- It isolates the class under test, allowing you to **focus only on its logic**.

Why Mocking Is Needed

- Avoid **actual database, network, or API calls**
- Ensure **tests run fast and reliably**
- Test **specific behavior of a class** without external dependencies
- Useful in **TDD and unit testing**

Real-Life Analogy

- Testing a car → Don't need a **real engine**, use a **mock engine** to check how car behaves.
- Mock = **dummy dependency** that acts like the real one.

Key Concepts

Concept	Explanation
Mock Object	Fake object that simulates a dependency
Stub	Provides predefined outputs for method calls
Spy	Partial mock, real object but some behavior is overridden
Verify	Check if method was called on mock

Mockito Example

Step 1: Add Mockito dependency

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>5.5.0</version>
    <scope>test</scope>
</dependency>
```

Step 2: Create a Mock

```
import static org.mockito.Mockito.*;
import org.junit.jupiter.api.Test;

class UserServiceTest {

    @Test
    void testUserRepository() {
        // Mock dependency
        UserRepository repo = mock(UserRepository.class);

        // Define behavior
        when(repo.getUserName(1)).thenReturn("John");

        // Call method under test
        UserService service = new UserService(repo);
        String name = service.getUserName(1);

        // Assertion
        assertEquals("John", name);
```

```
        // Verify interaction
        verify(repo).getUserName(1);
    }
}
```

Mock vs Stub vs Spy

Type	Definition	Use Case
Mock	Fake object with behavior defined	Unit testing dependencies
Stub	Provides fixed output	Simplify data return
Spy	Partial mock	Override some methods, keep real behavior

Common Mockito Methods

- mock(Class.class) → Create a mock
- when(...).thenReturn(...) → Stub a method
- verify(mock).method() → Check method call
- doThrow().when(mock).method() → Simulate exception

Best Practices (Cognizant Focus)

- Only mock **dependencies, not the class under test**
- Avoid over-mocking → makes tests hard to read
- Use **verify** to ensure proper interaction
- Keep mocks **simple and relevant**

Common Interview Questions

Q1. What is mocking?
A. Creating fake objects to simulate dependencies in unit tests.
Q2. Difference between mock and spy?
A. Mock = fully fake, Spy = partially real.
Q3. Can you mock final classes?
A. Yes, using Mockito inline extension.
Q4. Why use mocking in unit testing?
A. To isolate class behavior and avoid external dependencies.

One-Line Summary

Mocking simulates real dependencies in unit tests, allowing isolated and fast testing of a class's behavior.

Mock vs Stub vs Spy (Mockito / Unit Testing)

Definition

Term	Definition
Mock	Fake object used to **verify behavior and interactions** of the class under test.
Stub	Fake object used to **provide predetermined outputs** for method calls.
Spy	Partially real object where **real methods run**, but some methods can be **overridden or mocked**.

Purpose / Usage

Feature	Mock	Stub	Spy
Focus	Behavior verification	Return data	Partial behavior override
Real code execution	No	No	Yes (unless overridden)
Interaction verification	Yes	No	Yes
Example scenario	Check if repository method is called	Return test data without real DB	Real object but override some methods

---

Simple Example in Mockito

```
import static org.mockito.Mockito.*;
import org.junit.jupiter.api.Test;

class ExampleTest {

    @Test
    void mockStubSpyExample() {
        // MOCK
        List<String> mockList = mock(List.class);
        mockList.add("Hello");
        verify(mockList).add("Hello"); // verify behavior

        // STUB
        List<String> stubList = mock(List.class);
        when(stubList.get(0)).thenReturn("Mocked"); // predefined return
        assertEquals("Mocked", stubList.get(0));

        // SPY
        List<String> spyList = spy(new ArrayList<>());
        spyList.add("Real");
        assertEquals("Real", spyList.get(0)); // real method executes
        doReturn("Stubbed").when(spyList).get(0);
        assertEquals("Stubbed", spyList.get(0)); // stubbed method
    }
}
```

Real-Life Analogy

Concept	Analogy
Mock	Fake car engine to test dashboard reaction
Stub	Fake ATM returning fixed money without real calculation
Spy	Real car engine, but override speedometer reading

Key Rules / Best Practices

- **Mock** → Verify method calls, focus on behavior
- **Stub** → Provide controlled data, focus on output
- **Spy** → Use when you want **partial real behavior**
- Don’t overuse spies → can hide design issues

Common Interview Questions

Q1. Difference between Mock and Stub?

A. Mock verifies behavior; Stub only returns predefined data.

Q2. Can we verify interactions on a stub?
A. No, only on mocks or spies.
Q3. Can spies call real methods?
A. Yes, unless overridden.
Q4. When to prefer spy over mock?
A. When testing a **real object but need partial override.**

Quick Summary Table

Feature	Mock	Stub	Spy
Real methods executed	No	No	Yes
Focus	Interaction	Output	Partial behavior
Verify calls	No	No	Yes
Example	mock(repo)	when(repo.get()).thenReturn()	spy(realObject)

One-Line Summary
Mock = behavior check, Stub = data return, Spy = partial real object with overrides.

Creating Mocks in Mockito (JUnit 5)

Definition
Creating a mock means generating a **fake instance of a class or interface** that can be used to **simulate its behavior** in unit tests.

- Mocks allow **isolated testing** of a class without depending on real implementations.

Why Creating Mocks Is Needed

- Avoid **real database/API calls**
- Focus on **class under test**
- Control **behavior of dependencies**
- Test **edge cases or exceptions** easily

Ways to Create Mocks in Mockito
Using mock() Method

- Directly create a mock instance

```
import static org.mockito.Mockito.*;
import org.junit.jupiter.api.Test;

class MockExample {

    @Test
    void createMockTest() {
        // Create mock
        UserRepository repo = mock(UserRepository.class);

        // Define behavior
        when(repo.getUserName(1)).thenReturn("John");

        // Test class using mock
        UserService service = new UserService(repo);
        assertEquals("John", service.getUserName(1));
```

```
        // Verify interaction
        verify(repo).getUserName(1);
    }
}
```

Using @Mock Annotation

- Automatically create mock objects

```
import static org.mockito.Mockito.*;
import org.junit.jupiter.api.Test;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;
import org.junit.jupiter.api.extension.ExtendWith;

@ExtendWith(MockitoExtension.class)
class MockAnnotationExample {

    @Mock
    UserRepository repo;

    @Test
    void testMockAnnotation() {
        when(repo.getUserName(2)).thenReturn("Alice");
        UserService service = new UserService(repo);
        assertEquals("Alice", service.getUserName(2));
        verify(repo).getUserName(2);
    }
}
```

Using @Spy Annotation (Optional)

- Partially real object with some mocked behavior

```
@Spy
List<String> listSpy = new ArrayList<>();

doReturn("Mocked").when(listSpy).get(0);
```

Key Rules

- mock() → programmatic creation
- @Mock → annotation-based, requires @ExtendWith(MockitoExtension.class)
- Mocks **do not execute real methods** unless explicitly stubbed
- Verify **interactions with mocks** using verify()

Real-Life Analogy

- Mock = **fake vending machine** for testing coins and buttons, without delivering real drinks

---

Common Interview Questions

Q1. Difference between mock() and @Mock?
A. mock() → create manually, @Mock → create via annotation and extension.
Q2. Can we mock a final class?
A. Yes, using Mockito inline.
Q3. How do we verify mock interactions?
A. Using verify(mock).method().
Q4. Can we combine @Mock with @InjectMocks?
A. Yes, for automatic dependency injection in tests.

Quick Summary Table

Method	Description	Example
mock()	Programmatically create a mock	UserRepository repo = mock(UserRepository.class);
@Mock	Annotation-based mock creation	@Mock UserRepository repo;
@Spy	Partial real object	@Spy List<String> list = new ArrayList<>();

One-Line Summary
Mocks are fake objects created to simulate dependencies, enabling isolated and controlled unit testing.

Stubbing Methods in Mockito

Definition:
Stubbing is the process of **telling a mock object how to behave** when a method is called.

- *"When this method is called, return this value"*

Why Stubbing Is Needed

- Control **mock behavior**
- Simulate **real-world scenarios** without actual dependencies
- Test **edge cases, exceptions, or specific outputs**
- Essential for **unit testing in isolation**

Syntax / Pattern

```
when(mock.method(args)).thenReturn(value);
or for exceptions:
when(mock.method(args)).thenThrow(Exception.class);
```

Common Examples
Simple Stubbing

```
import static org.mockito.Mockito.*;
import org.junit.jupiter.api.Test;

class StubbingExample {

    @Test
    void simpleStub() {
        UserRepository repo = mock(UserRepository.class);

        // Stub method
        when(repo.getUserName(1)).thenReturn("John");

        assertEquals("John", repo.getUserName(1));
    }
}
```

Stubbing with Exceptions

```
@Test
void stubException() {
```

```
    UserRepository repo = mock(UserRepository.class);

    // Stub method to throw exception
    when(repo.getUserName(2)).thenThrow(new RuntimeException("User not found"));

    assertThrows(RuntimeException.class, () -> repo.getUserName(2));
}
```

Stubbing Void Methods (doReturn / doThrow)

```
List<String> mockList = mock(List.class);

// Void method stub
doThrow(new RuntimeException("Cannot add null"))
    .when(mockList).add(null);

// This will throw exception
assertThrows(RuntimeException.class, () -> mockList.add(null));
```

Chained Stubbing (Multiple Calls)

```
when(repo.getUserName(1))
    .thenReturn("John")
    .thenReturn("Alice");

assertEquals("John", repo.getUserName(1));
assertEquals("Alice", repo.getUserName(1));
```

Real-Life Analogy

- Stubbing = **program a vending machine**
- Press button → predefined output without making a real drink

---

Best Practices

- Stub **only necessary methods**
- Avoid over-stubbing → makes tests brittle
- Use **doReturn / doThrow** for **void methods**
- Keep stubs **simple and readable**

Common Interview Questions

Q1. Difference between thenReturn and doReturn?

- thenReturn → normal methods
- doReturn → void methods or spies

Q2. Can we stub multiple return values?

- Yes, chain thenReturn(val1).thenReturn(val2)

Q3. How to stub exceptions?

- thenThrow(Exception.class) or doThrow(Exception.class)

Q4. Difference between stub and mock?

- Stub → defines behavior
- Mock → verifies interaction

Quick Summary Table

Feature	Syntax	Example
Return value	when(...).thenReturn(...)	when(repo.getUserName(1)).thenReturn("John");
Exception	when(...).thenThrow(...)	when(repo.getUserName(2)).thenThrow(RuntimeException.class);

Feature	Syntax	Example
Void methods	doThrow(...).when(...)	doThrow(...).when(list).add(null);

One-Line Summary
Stubbing tells mocks how to behave when a method is called, including return values or exceptions.

---

when().thenReturn() in Mockito

Definition
when().thenReturn() is a **stubbing mechanism in Mockito** that defines the **output of a mock method** when it is called with specific arguments.

- *"When this method is called with these parameters, return this value."*

Why when().thenReturn() Is Needed

- Control **mock behavior**
- Avoid calling **real dependencies** (DB, API, etc.)
- Test **specific logic in isolation**
- Simulate **expected responses** for unit tests

Syntax / Pattern

```
when(mock.method(arguments)).thenReturn(value);
```

- mock → mocked object
- method(arguments) → method to stub
- value → return value when method is called

Simple Example

```
import static org.mockito.Mockito.*;
import org.junit.jupiter.api.Test;

class WhenThenReturnExample {

    @Test
    void testStub() {
        // Create mock
        UserRepository repo = mock(UserRepository.class);

        // Stub method
        when(repo.getUserName(1)).thenReturn("John");

        // Test
        assertEquals("John", repo.getUserName(1));
    }
}
```

Output: "John" returned without calling real DB

Stubbing Multiple Calls

```
when(repo.getUserName(1))
    .thenReturn("John")
    .thenReturn("Alice");
```

```
assertEquals("John", repo.getUserName(1));
assertEquals("Alice", repo.getUserName(1)); // second call
```

Real-Life Analogy

- Press a button on a **mock vending machine** → predefined drink is delivered
- No real machine, but predictable output

Key Rules / Best Practices

- Use **only for mocked objects**
- Do **not use on real objects** (unless using spy())
- Avoid over-stubbing → reduces test clarity
- Combine with **verify()** to check interactions

Common Interview Questions

Q1. What does when().thenReturn() do?
A. Stubs a mock method to return a specific value when called.
Q2. Can when().thenReturn() be chained?
A. Yes, to return different values on consecutive calls.
Q3. Can it be used for void methods?
A. No, use doReturn() or doThrow() instead.
Q4. Difference between thenReturn() and doReturn()?

- thenReturn() → normal mock methods
- doReturn() → void methods or spies

Quick Summary Table

Feature	Usage	Example
Basic stubbing	Return a value	when(repo.getUserName(1)).thenReturn("John")
Multiple returns	Chain thenReturn	thenReturn("John").thenReturn("Alice")
Void methods	Use doReturn / doThrow	doReturn("X").when(spy).method()

One-Line Summary
when().thenReturn() stubs mock methods to return predefined values, allowing isolated and predictable unit testing.

Stubbing and Verification in Mockito

Definition

- **Stubbing** → Telling a **mock object how to behave** when a method is called.
- **Verification** → Checking if a **mock method was actually called**, how many times, and with what arguments.

Together, they **control behavior** and **ensure correct interactions** in unit tests.

Why Needed

- **Stubbing:** Simulate data / responses without calling real dependencies (DB, API).
- **Verification:** Ensure the class under test **interacts correctly** with dependencies.
- Helps implement **TDD and isolated unit testing.**

Stubbing (How Mock Behaves)
Syntax

```
when(mock.method(args)).thenReturn(value);
```

Example

```
UserRepository repo = mock(UserRepository.class);
when(repo.getUserName(1)).thenReturn("John");

UserService service = new UserService(repo);
assertEquals("John", service.getUserName(1));
```

Output: Returns "John" without accessing the real repository.

Stubbing Exceptions

```
when(repo.getUserName(2)).thenThrow(new RuntimeException("User not found"));
assertThrows(RuntimeException.class, () -> repo.getUserName(2));
```

Stubbing Void Methods

```
List<String> list = mock(List.class);
doThrow(new RuntimeException("Cannot add null")).when(list).add(null);
assertThrows(RuntimeException.class, () -> list.add(null));
```

Verification (Checking Interactions)

Syntax

```
verify(mock).method(args);
```

Example

```
UserRepository repo = mock(UserRepository.class);
UserService service = new UserService(repo);

service.getUserName(1);

// Verify the method was called
verify(repo).getUserName(1);
```

Verify Call Count

```
verify(repo, times(1)).getUserName(1);
verify(repo, never()).deleteUser(anyInt());
verify(repo, atLeast(1)).getUserName(1);
```

Verify Argument Matchers

```
verify(repo).getUserName(eq(1));
verify(repo).getUserName(anyInt());
```

Real-Life Analogy

- **Stubbing:** Program a vending machine to return Coke when button A is pressed
- **Verification:** Check if the button was actually pressed during the test

---

Best Practices

- Stub **only necessary methods**
- Verify **important interactions**
- Avoid over-verifying → keeps tests maintainable
- Combine **stubbing + verification** for TDD style tests

Common Interview Questions

Q1. Difference between stubbing and verification?

- Stubbing → defines behavior
- Verification → checks method call

Q2. Can you verify a stubbed method?

- Yes, stubbing does not replace verification

Q3. How to verify number of calls?

- Use times(n), never(), atLeast(n)

Q4. Can you stub and verify a void method?

- Yes, use doReturn(), doThrow() and verify()

Quick Summary Table

Feature	Purpose	Syntax
Stubbing	Control mock behavior	when(mock.method()).thenReturn(value)
Verification	Check interactions	verify(mock).method()
Void Methods	Stub / Exception	doThrow().when(mock).voidMethod()
Call Counts	Ensure correct usage	verify(mock, times(2)).method()

One-Line Summary

Stubbing defines how mocks behave, and verification ensures they are used correctly, enabling reliable and isolated unit tests.

Argument Matchers in Mockito

Definition

Argument Matchers are used to **flexibly verify or stub mock method calls** without relying on exact argument values.

- Useful when **input values vary** but behavior/output is consistent.

Why Argument Matchers Are Needed

- Avoid **hardcoding exact values** in stubbing or verification
- Support **dynamic or unknown inputs**
- Combine with **stubbing and verification** for cleaner tests

---

Common Argument Matchers (Mockito)

Matcher	Purpose
any()	Matches **any object of any type**
anyInt(), anyString(), etc.	Matches specific primitive type
eq(value)	Matches exact value
isNull()	Matches null
notNull()	Matches non-null
argThat(predicate)	Matches based on **custom condition**

Syntax / Pattern

Stubbing with Matchers

```
when(mock.method(anyInt())).thenReturn("Mocked Value");
```

Verifying with Matchers

```
verify(mock).method(anyString());
```

Note: **All arguments must use matchers or all literal values** – cannot mix.

Simple Example

```
import static org.mockito.Mockito.*;
import static org.mockito.ArgumentMatchers.*;
import org.junit.jupiter.api.Test;

class ArgumentMatchersExample {
```

```
    @Test
    void testArgumentMatchers() {
        UserRepository repo = mock(UserRepository.class);

        // Stubbing with matcher
        when(repo.getUserName(anyInt())).thenReturn("John");

        assertEquals("John", repo.getUserName(1));
        assertEquals("John", repo.getUserName(99));

        // Verification with matcher
        repo.getUserName(5);
        verify(repo).getUserName(anyInt());
    }
}
```

Advanced Example Using argThat

```
verify(repo).getUserName(argThat(id -> id > 0 && id < 10));
```

- Verifies method is called with **id between 1 and 9**

Real-Life Analogy

- Vending machine → Press any button in a certain row → always gives soda
- any() → any button
- argThat(condition) → only buttons in row 1–3

Best Practices

- Prefer **specific matchers** for clarity
- Avoid mixing **literal values and matchers** in same method call
- Use argThat for **complex conditions**
- Combine with **verify()** for robust tests

Common Interview Questions

Q1. What is an argument matcher?

A. A way to flexibly stub or verify mock method calls without exact arguments.

Q2. Can you mix matchers and real values?

A. No, either use **all matchers** or all **literal values.**

Q3. Difference between any() and eq(value)?

- any() → matches any argument
- eq(value) → matches exactly that value

Q4. When to use argThat()?

A. For **custom condition-based matching.**

Quick Summary Table

Matcher	Example	Purpose
anyInt()	anyInt()	Matches any integer
anyString()	anyString()	Matches any string
eq(value)	eq(10)	Matches exact value
isNull()	isNull()	Matches null
argThat()	argThat(x -> x>0)	Matches custom condition

One-Line Summary

Argument Matchers allow flexible stubbing and verification by matching method arguments dynamically in Mockito.

---

verify() and Verification Modes in Mockito

Definition

- **verify()** → Used to **check if a mock method was called** during a test.
- **Verification Modes** → Specify **how many times** or **under what conditions** the method should have been called.

Helps ensure the **class under test interacts correctly** with its dependencies.

Why Verification Is Needed

- Confirms **expected interactions** with dependencies
- Ensures **methods are called correct number of times**
- Detects **unexpected or missing calls**
- Essential for **TDD and behavior-driven testing**

Basic Syntax

```
verify(mock).method(args);
verify(mock, times(n)).method(args);
verify(mock, never()).method(args);
verify(mock, atLeast(n)).method(args);
verify(mock, atMost(n)).method(args);
```

Common Verification Modes

Mode	Meaning	Example
times(n)	Called exactly n times	verify(repo, times(2)).getUserName(1);
never()	Never called	verify(repo, never()).deleteUser(anyInt());
atLeast(n)	Called n or more times	verify(repo, atLeast(1)).getUserName(1);
atMost(n)	Called n or fewer times	verify(repo, atMost(3)).getUserName(1);
only()	Called **only once** and no other interactions	verify(repo, only()).getUserName(1);

Simple Example

```
import static org.mockito.Mockito.*;
import org.junit.jupiter.api.Test;

class VerifyExample {

    @Test
    void testVerification() {
        UserRepository repo = mock(UserRepository.class);
        UserService service = new UserService(repo);

        service.getUserName(1);
        service.getUserName(1);

        // Verify exact calls
        verify(repo, times(2)).getUserName(1);

        // Verify never called
        verify(repo, never()).deleteUser(anyInt());
    }
}
```

Verification with Argument Matchers

```
verify(repo).getUserName(anyInt());
verify(repo, atLeast(1)).getUserName(eq(1));
```

Verification of Void Methods

```
List<String> list = mock(List.class);

list.add("Hello");
verify(list).add("Hello");

list.clear();
verify(list, times(1)).clear();
```

Real-Life Analogy

- Factory → Check if **worker pressed the correct buttons**
- verify() = Inspector confirming tasks were done as expected
- times(n) / never() = Inspector checking frequency

Best Practices

- Verify **important interactions only**
- Use **times(), never(), atLeast()** for clear expectations
- Combine **argument matchers** with verification for flexibility
- Avoid verifying **everything** → reduces test clarity

Common Interview Questions

Q1. What is verify() in Mockito?
A. It checks whether a mock method was called as expected.

Q2. Difference between times() and atLeast()?

- times(n) → exactly n times
- atLeast(n) → n or more times

Q3. Can verify work with stubs?
A. Yes, stubbing defines behavior, verification checks interaction.

Q4. What does only() do?
A. Ensures **no other method was called** except the verified one.

Quick Summary Table

Mode	Meaning	Example
times(n)	Exactly n calls	verify(mock, times(2)).method();
never()	Not called	verify(mock, never()).method();
atLeast(n)	n or more calls	verify(mock, atLeast(1)).method();
atMost(n)	n or fewer calls	verify(mock, atMost(3)).method();
only()	Only this call, no others	verify(mock, only()).method();

One-Line Summary

verify() checks mock interactions, and verification modes control how often and under what conditions methods are expected to be called.

Spies (@Spy) in Mockito

Definition

A **Spy** in Mockito is a **partial mock**: it wraps a **real object** but allows **overriding specific methods**.

- Useful when you want to **call real methods** but still **control some behavior** in tests.

Why Spies Are Needed

- Test **real object behavior** while controlling **specific methods**
- Useful for **legacy code** or objects with **complex logic**
- Combine **real execution + mocking** in the same test

How Spies Work

- Calls **real methods by default**
- Can **stub specific methods** using doReturn() / doThrow()
- Can **verify interactions** like a normal mock

Creating a Spy

Using spy() Method

```
import static org.mockito.Mockito.*;
import org.junit.jupiter.api.Test;
import java.util.ArrayList;
import java.util.List;

class SpyExample {

    @Test
    void spyUsingMethod() {
        List<String> list = new ArrayList<>();
        List<String> spyList = spy(list);

        // Real method executes
        spyList.add("Hello");
        assertEquals(1, spyList.size());

        // Stub a method
        doReturn(100).when(spyList).size();
        assertEquals(100, spyList.size());
    }
}
```

Using @Spy Annotation

```
import org.mockito.Spy;
import org.mockito.junit.jupiter.MockitoExtension;
import org.junit.jupiter.api.extension.ExtendWith;

@ExtendWith(MockitoExtension.class)
class SpyAnnotationExample {

    @Spy
    List<String> spyList = new ArrayList<>();

    @Test
    void testSpy() {
        spyList.add("Hello");
        assertEquals(1, spyList.size());

        doReturn(50).when(spyList).size();
        assertEquals(50, spyList.size());
    }
}
```

Key Rules

- Real methods are called **by default**
- Use **doReturn() / doThrow()** to override methods
- Can verify **method calls** like mocks: verify(spyList).add("Hello")
- Avoid overusing spies → can hide design issues

Real-Life Analogy

- Spy = **real car** with a mechanic override
- Most parts work normally, but some functions can be **controlled or faked** for testing

---

Common Interview Questions

Q1. Difference between Mock and Spy?

- Mock → fully fake object, real methods never called
- Spy → real object, can override some methods

Q2. How do you stub a spy method?

- Use doReturn(value).when(spy).method()

Q3. Can you verify a spy method call?

- Yes, same as mocks: verify(spy).method()

Q4. When to prefer spy over mock?

- When testing **partial real behavior** or **legacy code**

---

8. Quick Summary Table

Feature	Mock	Spy
Real method execution	No	Yes, by default
Can stub methods	Yes	Yes(using doReturn/doThrow)
Verify interactions	Yes	Yes
Use case	Isolate class behavior	Partial real behavior with control

---

One-Line Summary

Spy is a partial mock that calls real methods by default but allows selective stubbing and verification in Mockito.

Argument Captors in Mockito

Definition

ArgumentCaptor is used to **capture arguments** passed to mock methods for further **assertions or verification.**

- Helps **verify what data was sent** to dependencies during tests.

Why Argument Captors Are Needed

- Test **actual values passed** to a method
- Useful for **complex objects** or **dynamically generated data**
- Ensures **correct data flow** in unit tests

How to Create an Argument Captor

Using @Captor Annotation

```
import org.mockito.Captor;
import org.mockito.junit.jupiter.MockitoExtension;
import org.junit.jupiter.api.extension.ExtendWith;
```

```
@ExtendWith(MockitoExtension.class)
class CaptorExample {

    @Captor
    ArgumentCaptor<String> captor;

}
```

Using ArgumentCaptor.forClass()

```
ArgumentCaptor<String> captor = ArgumentCaptor.forClass(String.class);
```

Capturing Arguments

```
import static org.mockito.Mockito.*;
import org.mockito.ArgumentCaptor;
import org.junit.jupiter.api.Test;

class ArgumentCaptorDemo {

    @Test
    void captureArgumentTest() {
        List<String> mockList = mock(List.class);

        mockList.add("Hello");
        mockList.add("World");

        // Create captor
        ArgumentCaptor<String> captor = ArgumentCaptor.forClass(String.class);

        // Capture arguments
        verify(mockList, times(2)).add(captor.capture());

        // Verify captured values
        List<String> capturedValues = captor.getAllValues();
        assertEquals("Hello", capturedValues.get(0));
        assertEquals("World", capturedValues.get(1));
    }
}
```

Capturing Complex Objects

```
ArgumentCaptor<User> captor = ArgumentCaptor.forClass(User.class);
verify(repo).save(captor.capture());
User capturedUser = captor.getValue();
assertEquals("John", capturedUser.getName());
```

Real-Life Analogy

- Inspector at a factory **records exact items** sent to packaging
- ArgumentCaptor = captures **what values were passed** for validation

---

Best Practices

- Use **when you need to validate actual arguments**
- Combine with **verify()** for robust testing
- Avoid unnecessary captors → reduces readability

Common Interview Questions

Q1. What is an ArgumentCaptor?

A. Captures method arguments passed to a mock for assertion.

Q2. Difference between getValue() and getAllValues()?

- getValue() → single call argument
- getAllValues() → all captured arguments

Q3. Can ArgumentCaptor capture primitive types?

- Yes, use boxed types (Integer, Double)

Q4. When to prefer ArgumentCaptor over matchers?

- When you need **exact runtime values** for verification.

Quick Summary Table

Feature	Example	Purpose
Capture single argument	captor.getValue()	Verify one method call
Capture multiple arguments	captor.getAllValues()	Verify multiple calls
Complex objects	ArgumentCaptor.forClass(User.class)	Inspect object fields

---

One-Line Summary

ArgumentCaptor captures arguments passed to mock methods, enabling detailed verification of data in unit tests.
