**JUnit5 Interview Questions**

Contents

[Architecture 1](#_Toc180525189)

[1. Explain Junit 5 architecture. 1](#_Toc180525190)

[2. Explain the most commonly used annotations in JUnit 5. 2](#_Toc180525191)

[3. What are the main differences between JUnit 4 and JUnit 5, and why was JUnit 5 introduced? 4](#_Toc180525192)

[4. How does the @ExtendWith annotation work in JUnit 5, and can you give an example of its usage? 5](#_Toc180525193)

[5. Explain the purpose of the @Nested annotation in JUnit 5. How does it help in organizing tests? 7](#_Toc180525194)

[6. Describe how you would use parameterized tests in JUnit 5. What are the advantages of using parameterized tests? What is @ValueSource in this context? 7](#_Toc180525195)

[7. Explain @Nested annotation. 8](#_Toc180525196)

[8. Explain Test Interfaces and Default Methods. 10](#_Toc180525197)

[9. Explain Argument Sources. 11](#_Toc180525198)

[10. Explain Argument Conversion. 14](#_Toc180525199)

[11. What are @TestInstance annotations? 16](#_Toc180525200)

[12. Explain assumptions in JUnit 5. 17](#_Toc180525201)

# Architecture

### Explain Junit 5 architecture.

JUnit 5 has a modular architecture composed of three main sub-projects1. Here’s a breakdown:

1. **JUnit Platform**: This is the foundation for launching testing frameworks on the JVM1. It defines the TestEngine API for developing testing frameworks that run on the platform1. The platform also provides a Console Launcher for launching tests from the command line and integrates with popular IDEs and build tools like IntelliJ IDEA, Eclipse, NetBeans, Visual Studio Code, Gradle, Maven, and Ant1.
2. **JUnit Jupiter**: This sub-project includes the programming model and extension model for writing tests and extensions in JUnit 51. It provides a TestEngine for running Jupiter-based tests on the platform1. Jupiter introduces new annotations and features like @DisplayName, @Nested, @Tag, @ExtendWith, and more, which enhance the capabilities of writing and organizing tests2.
3. **JUnit Vintage**: This sub-project provides a TestEngine for running JUnit 3 and JUnit 4 based tests on the platform1. It allows you to run tests written for older versions of JUnit within the JUnit 5 framework1.

Together, these components make JUnit 5 a powerful and flexible testing framework that supports a wide range of testing needs and integrates seamlessly with modern development tools

### Explain the most commonly used annotations in JUnit 5.

 JUnit 5 introduces several powerful and flexible annotations that make writing tests more intuitive and expressive. Here are some of the most commonly used ones:

1. **@Test:** This marks a method as a test method.

@Test

void myTest() {

// test logic here

}

**2. @BeforeEach:** This is used to denote a method that should be executed before each test. Ideal for setting up test data.

@BeforeEach

void setUp() {

// setup code here

}

**3. @AfterEach:** This is used to denote a method that should be executed after each test. Useful for cleaning up after tests.

@AfterEach

void tearDown() {

// cleanup code here

}

**4. @BeforeAll:** Marks a method to be executed once before all tests in the current class. Methods annotated with this must be static.

@BeforeAll

static void initAll() {

// initialization code here

}

**5. @AfterAll:** Marks a method to be executed once after all tests in the current class. Methods annotated with this must be static.

@AfterAll

static void tearDownAll() {

// cleanup code here

}

**6. @DisplayName:** This annotation lets you specify a custom name for the test method.

@Test

@DisplayName("Custom test name")

void testWithCustomName() {

// test logic here

}

**7. @Disabled:** Use this to disable a test method or an entire class of tests.

@Test

@Disabled("Not implemented yet")

void testNotImplemented() {

// this test won't run

}

**8. @ParameterizedTest:** Enables parameterized tests, which run the same test multiple times with different inputs.

@ParameterizedTest

@ValueSource(strings = {"Hello", "JUnit"})

void testWithParams(String input) {

assertNotNull(input);

}

**9. @ExtendWith:** This annotation is used to register extensions to test classes or methods.

@ExtendWith(MyCustomExtension.class)

class MyTests {

// test logic here

}

These annotations help make your tests more readable, maintainable, and expressive. They provide you with powerful capabilities to set up test contexts, clean up resources, and customize test execution.

### What are the main differences between JUnit 4 and JUnit 5, and why was JUnit 5 introduced?

This question tests your understanding of the new features and improvements in JUnit 5 compared to JUnit 4.

JUnit 5 brought significant changes and improvements over JUnit 4. Here are the main differences and the motivations behind them:

**1. Modular Architecture**:

**JUnit 4**: Monolithic framework with a single jar file.

**JUnit 5**: Split into three main modules—JUnit Platform, JUnit Jupiter, and JUnit Vintage. This allows for better separation of concerns and more flexibility.

**2. New Annotations**:

**JUnit 4**: Limited to @Test, @Before, @After, @BeforeClass, and @AfterClass.

**JUnit 5**: Introduces new annotations like @BeforeEach, @AfterEach, @BeforeAll, @AfterAll, @DisplayName, @Nested, @Tag, @Disabled, @ExtendWith, and more, providing more granular control and expressiveness.

**3. Parameterized Tests**:

**JUnit 4**: Parameterized tests were cumbersome to implement and less flexible.

**JUnit 5**: Makes parameterized tests more straightforward and powerful with annotations like @ParameterizedTest, @ValueSource, @EnumSource, @CsvSource, etc.

**4. Extensions and Test Engines**:

**JUnit 4**: Extensions were limited and less flexible.

**JUnit 5**: Introduces a new extension model via @ExtendWith, allowing for more powerful and flexible extensions. It also supports multiple test engines, enabling integration with other testing frameworks.

**5. Improved Assertions and Assumptions**:

**JUnit 4**: Basic assertions and assumptions.

**JUnit 5**: More expressive and powerful assertions (e.g., assertThrows, assertAll, etc.) and assumptions (assumeTrue, assumeFalse, etc.).

**6. Improved IDE and Tooling Support**:

**JUnit 5**: Better integration with IDEs and build tools like Maven and Gradle, providing a smoother development experience.

**Motivation Behind JUnit 5**: JUnit 5 was introduced to address the limitations of JUnit 4 and to modernize the framework to better support contemporary testing needs. It aims to provide a more modular, extensible, and flexible testing framework that can grow and adapt with the needs of developers.

### How does the @ExtendWith annotation work in JUnit 5, and can you give an example of its usage?

* This assesses your knowledge of the extension model in JUnit 5 and your ability to integrate custom extensions.

The @ExtendWith annotation in JUnit 5 is a powerful way to integrate extensions into your test classes. Extensions provide additional functionality, such as lifecycle callbacks, parameter injection, and more.

When you annotate a test class or test method with @ExtendWith, you specify one or more extension classes that implement the Extension interface. These extension classes can then intercept various phases of the test execution lifecycle.

Here’s an example that demonstrates how to use @ExtendWith to create a simple custom extension:

**1. Create a Custom Extension:**

import org.junit.jupiter.api.extension.BeforeEachCallback;

import org.junit.jupiter.api.extension.ExtensionContext;

import org.junit.jupiter.api.extension.Extension;

public class MyCustomExtension implements BeforeEachCallback {

@Override

public void beforeEach(ExtensionContext context) throws Exception {

System.out.println("Executing before each test method");

// You can add setup code here

}

}

In this example, MyCustomExtension implements the BeforeEachCallback interface, which means it will execute some logic before each test method.

**2. Use the Custom Extension in a Test Class:**

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.extension.ExtendWith;

@ExtendWith(MyCustomExtension.class)

public class CalculatorTest {

@Test

void testAdd() {

System.out.println("Running testAdd()");

// Test logic here

}

@Test

void testSubtract() {

System.out.println("Running testSubtract()");

// Test logic here

}

}

In this test class:

* The @ExtendWith annotation is used to register MyCustomExtension.
* MyCustomExtension's beforeEach method runs before each test method, printing a message to the console.

**What Happens:**

* When testAdd and testSubtract are executed, you’ll see the custom extension’s message printed before each test method.

Extensions in JUnit 5 provide a lot of flexibility for adding custom behavior to your tests, making it a powerful feature for complex testing needs.

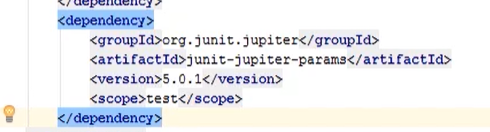
### Explain the purpose of the @Nested annotation in JUnit 5. How does it help in organizing tests?

* This question is about understanding nested test classes and their advantages for structuring complex test scenarios.

### Describe how you would use parameterized tests in JUnit 5. What are the advantages of using parameterized tests? What is @ValueSource in this context?

* This question checks your grasp of running the same test with different parameters and its benefits for reducing redundancy in test code.

**Parameterized tests** in JUnit 5 allow you to run the same test method multiple times with different inputs. This reduces code duplication and makes tests more maintainable.



**Advantages:**

* **Efficiency:** Instead of writing multiple test methods for different inputs, you write one test method and provide a range of inputs.
* **Clarity:** Your test logic stays concise and organized.
* **Maintainability:** Easier to add new test cases without duplicating code.

**Using Parameterized Tests:** Here’s how you’d implement a parameterized test using @ParameterizedTest and @ValueSource:

import org.junit.jupiter.params.ParameterizedTest;

import org.junit.jupiter.params.provider.ValueSource;

import static org.junit.jupiter.api.Assertions.assertTrue;

public class ParameterizedTestExample {

// Parameterized test method that checks if the input string is not empty

@ParameterizedTest

@ValueSource(strings = {"hello", "JUnit", "5"})

void testWithStringParameter(String input) {

assertTrue(input.length() > 0);

}

}

**Explanation:**

* **@ParameterizedTest:** Marks the method as a parameterized test.
* **@ValueSource:** Provides a simple source of values. In this case, strings.

You can use different sources for values, such as @CsvSource, @EnumSource, and more. This is super handy when you want to test various scenarios without cluttering your test class with repetitive code.

### Explain @Nested annotation.

The @Nested annotation in JUnit 5 is used to create nested test classes. These classes are like mini-test classes within your main test class, allowing you to group related tests together logically. It helps in organizing complex test scenarios, making your tests more readable and maintainable.

Here's an example to illustrate:

import org.junit.jupiter.api.Nested;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

class CalculatorTest {

private final Calculator calculator = new Calculator();

@Nested

class AdditionTests {

@Test

void whenAddingTwoAndThree\_thenResultShouldBeFive() {

assertEquals(5, calculator.add(2, 3));

}

@Test

void whenAddingNegativeNumbers\_thenResultShouldBeCorrect() {

assertEquals(-5, calculator.add(-2, -3));

}

}

@Nested

class SubtractionTests {

@Test

void whenSubtractingThreeFromFive\_thenResultShouldBeTwo() {

assertEquals(2, calculator.subtract(5, 3));

}

@Test

void whenSubtractingWithNegativeNumbers\_thenResultShouldBeCorrect() {

assertEquals(1, calculator.subtract(-2, -3));

}

}

}

In this example:

* AdditionTests and SubtractionTests are nested test classes within the CalculatorTest class.
* Each nested class groups related tests, making it clear which tests belong together.

Using @Nested helps improve the structure and readability of your tests, especially when dealing with complex testing scenarios.

### Explain Test Interfaces and Default Methods.

In JUnit 5, you can use test interfaces with default methods to share common test code across multiple test classes. This allows for more modular and reusable tests. Let's break this down with an example:

**Test Interface**

Create a test interface with default methods for shared test logic:

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertTrue;

public interface StringTests {

String createString();

@Test

default void stringShouldNotBeEmpty() {

String str = createString();

assertTrue(str.length() > 0, "String should not be empty");

}

@Test

default void stringShouldHaveValidLength() {

String str = createString();

assertTrue(str.length() < 100, "String length should be less than 100");

}

}

**Implementing the Test Interface**

Now, implement this interface in your test classes:

import org.junit.jupiter.api.DisplayName;

@DisplayName("String Tests for Hello String")

public class HelloStringTest implements StringTests {

@Override

public String createString() {

return "Hello, World!";

}

}

@DisplayName("String Tests for Empty String")

public class EmptyStringTest implements StringTests {

@Override

public String createString() {

return "";

}

}

In this example:

* **StringTests** interface defines common test methods with default implementations.
* **HelloStringTest** and **EmptyStringTest** implement the interface and provide specific string values by overriding the createString() method.

This approach allows you to write reusable test logic in the interface and implement it in multiple test classes. It keeps your tests DRY (Don't Repeat Yourself) and makes maintenance easier.

### Explain Argument Sources.

Argument sources are annotations that provide input values for parameterized tests. They define where the parameters come from. Here are some commonly used argument sources:

**1. @ValueSource:** Provides a fixed set of values.

import org.junit.jupiter.params.ParameterizedTest;

import org.junit.jupiter.params.provider.ValueSource;

import static org.junit.jupiter.api.Assertions.assertTrue;

public class ValueSourceTest {

@ParameterizedTest

@ValueSource(strings = {"hello", "world"})

void testWithStringValueSource(String input) {

assertTrue(input.length() > 0);

}

}

**2. @EnumSource:** Provides values from an enumeration.

import org.junit.jupiter.params.ParameterizedTest;

import org.junit.jupiter.params.provider.EnumSource;

public class EnumSourceTest {

enum Day {

MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY

}

@ParameterizedTest

@EnumSource(Day.class)

void testWithEnumSource(Day day) {

assertNotNull(day);

}

}

**3. @CsvSource:** Provides values from comma-separated values.

import org.junit.jupiter.params.ParameterizedTest;

import org.junit.jupiter.params.provider.CsvSource;

public class CsvSourceTest {

@ParameterizedTest

@CsvSource({

"test, 1",

"test2, 2"

})

void testWithCsvSource(String str, int num) {

assertTrue(str.length() > 0);

assertTrue(num > 0);

}

}

**4. @MethodSource:** Provides values from a static method within the test class.

import org.junit.jupiter.params.ParameterizedTest;

import org.junit.jupiter.params.provider.MethodSource;

import java.util.stream.Stream;

public class MethodSourceTest {

@ParameterizedTest

@MethodSource("stringProvider")

void testWithMethodSource(String argument) {

assertNotNull(argument);

}

static Stream<String> stringProvider() {

return Stream.of("hello", "world");

}

}

**5. @ArgumentsSource:** Provides values from a custom argument provider.

import org.junit.jupiter.params.ParameterizedTest;

import org.junit.jupiter.params.provider.ArgumentsSource;

import java.util.stream.Stream;

public class CustomArgumentSourceTest {

@ParameterizedTest

@ArgumentsSource(CustomArgumentProvider.class)

void testWithCustomArgumentProvider(String argument) {

assertNotNull(argument);

}

static class CustomArgumentProvider implements ArgumentsProvider {

@Override

public Stream<? extends Arguments> provideArguments(ExtensionContext context) {

return Stream.of(Arguments.of("hello"), Arguments.of("world"));

}

}

}

These argument sources offer great flexibility in running parameterized tests with a variety of inputs, making your test cases more robust and comprehensive.

### Explain Argument Conversion.

Argument conversion in JUnit 5 is the process of automatically converting arguments provided by parameterized tests into the required types. JUnit 5 includes built-in converters and allows custom converters as well.

**Built-in Converters**

JUnit 5 automatically converts simple types (like int, long, String, etc.) when using argument sources like @ValueSource and @CsvSource.

Here's an example using @ValueSource to convert String arguments:

import org.junit.jupiter.params.ParameterizedTest;

import org.junit.jupiter.params.provider.ValueSource;

import static org.junit.jupiter.api.Assertions.assertNotNull;

public class ValueSourceTest {

@ParameterizedTest

@ValueSource(strings = {"hello", "world"})

void testWithStringValueSource(String input) {

assertNotNull(input);

}

}

**Custom Converters**

For more complex conversions, you can implement custom argument converters using @ConvertWith and ArgumentConverter.

**1. Create a Custom Converter:**

import org.junit.jupiter.params.converter.ArgumentConversionException;

import org.junit.jupiter.params.converter.SimpleArgumentConverter;

public class StringToIntegerConverter extends SimpleArgumentConverter {

@Override

protected Object convert(Object source, Class<?> targetType) throws ArgumentConversionException {

if (source instanceof String && targetType == Integer.class) {

return Integer.parseInt((String) source);

}

throw new IllegalArgumentException("Conversion from " + source.getClass() + " to " + targetType + " not supported.");

}

}

**2. Use the Custom Converter in a Test:**

import org.junit.jupiter.params.ParameterizedTest;

import org.junit.jupiter.params.converter.ConvertWith;

import org.junit.jupiter.params.provider.ValueSource;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class CustomConverterTest {

@ParameterizedTest

@ValueSource(strings = {"42", "43"})

void testWithCustomConverter(@ConvertWith(StringToIntegerConverter.class) Integer number) {

assertEquals(42, number + 0); // Example check, convert "42" to 42, "43" to 43

}

}

**Explanation:**

* **StringToIntegerConverter**: Custom converter that converts String to Integer.
* **@ConvertWith**: Annotation to specify the custom converter to use.

Custom converters give you the flexibility to handle complex conversion scenarios, making your parameterized tests even more powerful.

### What are @TestInstance annotations?

The @TestInstance annotation in JUnit 5 determines the lifecycle of test instances. It specifies whether a new instance of the test class should be created for each test method or if a single instance should be reused for all test methods. This can have a significant impact on how state is managed in your tests.

**1. Test Instance Lifecycle Modes:**

* **PER\_METHOD** (default): A new test instance is created for each test method. This ensures that each test runs in isolation, with no shared state between tests.
* **PER\_CLASS**: A single test instance is used for all test methods in the class. This allows for shared state between test methods, which can be useful for expensive setup operations.

**Example:** Here's a simple example demonstrating the use of @TestInstance:

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.TestInstance;

import static org.junit.jupiter.api.Assertions.assertEquals;

@TestInstance(TestInstance.Lifecycle.PER\_CLASS)

public class TestInstanceExample {

private int counter = 0;

@Test

void testIncrementCounter1() {

counter++;

assertEquals(1, counter);

}

@Test

void testIncrementCounter2() {

counter++;

assertEquals(2, counter);

}

}

In this example:

* By setting the lifecycle to PER\_CLASS, the same instance of TestInstanceExample is used for both test methods.
* The counter field is shared between the test methods, so it retains its state across method executions.

If you used the default PER\_METHOD lifecycle:

* A new instance of TestInstanceExample would be created for each test method.
* The counter would be reset to 0 for each test, and the tests would fail as written.

So, @TestInstance gives you control over how test instances are managed, which can be particularly useful for managing expensive setup operations or stateful tests.

### Explain assumptions in JUnit 5.

In JUnit 5, **assumptions** are a way to express preconditions for a test. If an assumption fails, the test is **skipped**, not failed. This is useful when tests only make sense under certain conditions. Here's how assumptions work and why they're useful:

**Common Assumption Methods**

1. **assumeTrue(boolean) / assumeFalse(boolean)**: Skip the test if the given condition is false/true.

java

Copy

import org.junit.jupiter.api.Test;

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

class AssumptionsTest {

@Test

void testOnlyOnCertainCondition() {

// Assume a certain condition

assumeTrue(System.getenv("ENV") != null, "ENV variable should be set");

// Test logic here

System.out.println("This runs only if ENV variable is set");

}

}

1. **assumingThat(boolean, Executable)**: Conditional execution of code within a test based on the assumption.

java

Copy

import org.junit.jupiter.api.Test;

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

import static org.junit.jupiter.api.Assertions.assertEquals;

class ConditionalTest {

@Test

void testConditionalExecution() {

String env = System.getenv("ENV");

assumingThat(env != null && env.equals("TEST"),

() -> {

// Execute only if the assumption is true

System.out.println("Special test execution");

assertEquals(2, 1 + 1);

});

// Normal test logic here

assertEquals(2, 1 + 1);

}

}

**Why Use Assumptions?**

1. **Environment-Specific Tests**: Skip tests that depend on a specific environment configuration.
2. **Expensive Setup**: Avoid running tests that would require an expensive setup if certain preconditions are not met.
3. **Conditional Logic**: Execute certain parts of a test conditionally, based on assumptions.

By using assumptions, you ensure your tests run only when the necessary conditions are satisfied, leading to more robust and meaningful test results.