**Unit Testing:-** Unit testing is a software testing technique where individual units or components of a software application are tested in isolation to ensure that they function correctly

**JUnit5 Framework:**

It is a simple framework to write repeated tests.

**Project SetUp:**

* Add JUnit5 dependency in our Java project
* For Maven project add as follows:

**<dependency>**

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

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

**<version>5.8.1</version>**

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

**</dependency>**

The class or method we want to test is termed as **subject under test.**

* To create a java test class, we create a java test class and use the JUnit api to make some assumptions towards the behaviour of the subject under test.
* Then JUnit verifies whether the assumptions are correct or not, by passing or failing the test.

**JUnit Architecture:**

3rd Party Testing Frameworks

Junit Vintage

Junit Jupiter

Console

Build Tools

IDEs

JUnit Platforms

JUnit Jupiter:

JUnit Jupiter is the next generation programming and extension model for writing tests and extensions in JUnit 5. It introduces new features like parameterized tests, nested tests, dynamic tests, and extension model improvements.

Annotations such as **@Test**, **@BeforeEach**, **@AfterEach**, **@BeforeAll**, **@AfterAll**, etc., are part of JUnit Jupiter.

We need to add dependencies in our maven project in pom.xml to use Junit5 as follows:Info maven needs to download the dependencies from the internet

<dependencies>

<dependency>

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

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

<version>5.9.1</version>

<scope>test</scope>

</dependency>

</dependencies>

1. @Test annotation: In JUnit, the **@Test** annotation is used to mark a method as a test method, indicating that it should be executed by the JUnit framework during testing. When you write test classes in JUnit, you annotate the methods you want to be considered as tests with **@Test**. Here's a brief overview of how **@Test** annotations work in Junit

**import** org.junit.jupiter.api.Assertions;

**import** org.junit.jupiter.api.Test;

**public** **class** ContactManagerTest {

@Test

**public** **void** shouldCreateContact() {

ContactManager contactManager = **new** ContactManager();

contactManager.addContact("Pushpan", "Bhaumik", "0123");

Assertions.*assertTrue*(contactManager.getAllContacts()>0);

}

}

**In class ContactManager.java:**

Map<String, Contact> contactList = **new** ConcurrentHashMap<String, Contact>();

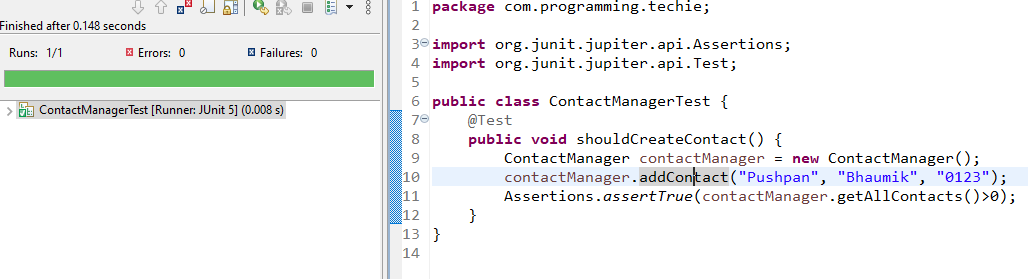
**public** **void** addContact(String firstName, String lastName, String phoneNumber) {

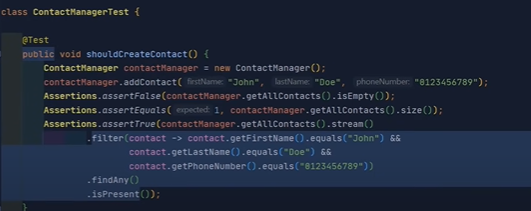
Contact contact = **new** Contact(firstName, lastName, phoneNumber);

contactList.put(generateKey(contact), contact);

}

JUnit provides a variety of assertion methods to validate different conditions in your test cases





We can also use Java8 in Junit5 Testing but this is from youtube and here getAllContacts() method is returning all the contacts added

The getAllContacts() method returns the size of the HashMap, and in the test method we are using assertTrue(contactMgr.getAllContacts()>0),i.e. true so the test case passes.

II)@DisplayName annotation:

JUnit [**@DisplayName**](https://www.digitalocean.com/community/users/displayname) annotation is used to provide a custom name for the test class and test methods. We can use spaces, special characters, and even emojis in the display name

@Test

@DisplayName("Should Not Create Contact when first name is null")

**public** **void** shouldThrowRunTimeExceptionFirstNameNull() {

ContactManager con = **new** ContactManager();

Assertions.*assertThrows*(RuntimeException.**class**, ()->{

con.addContact(**null**, "Roy", "0123456789");

});

}

**Understanding Junit5 Test Lifecycle:-**

Each Junit Test, when created, creates a new instance of the Test class and it follows different phases as part of the execution

Each phase is represented using an annotation in Junit5

**@BeforeAll**, **@BeforeEach**, **@AfterAll**, **@AfterEach** annotation

**@BeforeAll** and **@AfterAll** annotations are used to set up and tear down the database connection. These methods run once before and after all the tests in the test class, respectively.Used for initialization tasks

// Close the database connection

// Initialize database connection (e.g., connect to a test database)

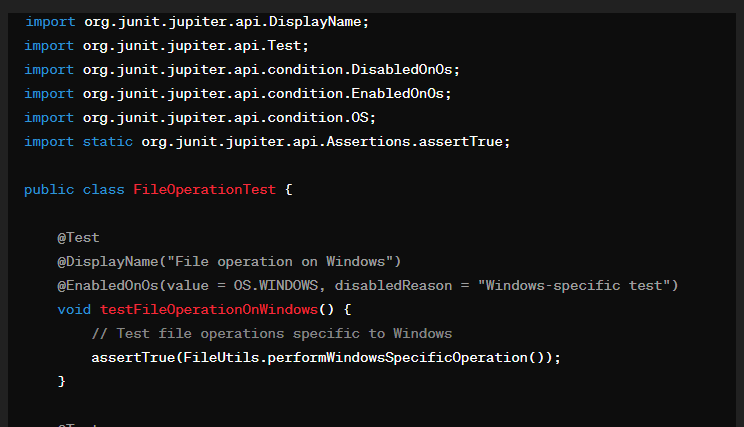
**@BeforeEach** and **@AfterEach** annotations are used to set up and tear down the **UserService** object before and after each test method runs, respectively.Used for cleanup tasks for testing

// Clear any test-specific data or reset states

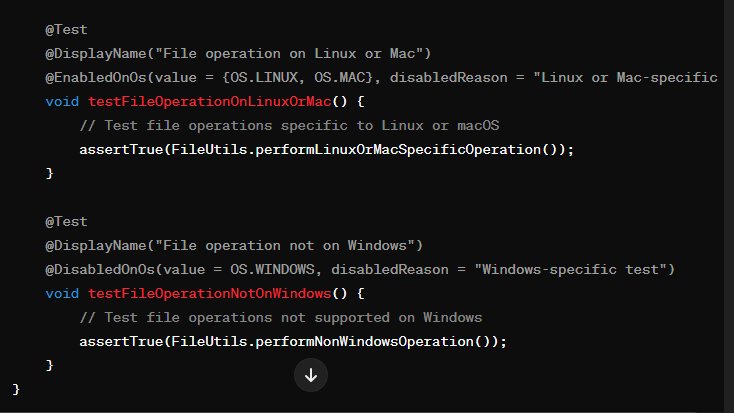
// Initialize UserService with the database connection

**Conditional Executions:-**

These includes annotations such as **@EnabledOnOS,@DisabledOnOS**



It conditionally enables tests based on the operating system



It conditionally enables or tests based on the operating system

**Assumptions:-**

In JUnit 5, assumptions are a way to conditionally skip the execution of tests based on certain conditions. They allow you to specify conditions under which a test should be executed or skipped, making it easier to manage test suites and handle scenarios where tests may not be applicable.

@Test

@DisplayName("Test Contact Creation on Developer Machine")

This value is usually given in VM Arguments while Run As Configurations

**public** **void** shouldTestContactCreationOnDEV() {

Assumptions.*assumeTrue*("DEV".equals(System.*getProperty*("ENV")));

contactManager.addContact("John", "Doe", "0123456789");

*assertFalse*(contactManager.getAllContacts().isEmpty());

*assertEquals*(1, contactManager.getAllContacts().size());

}

The test will get executed only if the environment is DEV OR env parameter is set to DEV

@Test

void testWithAssumingThat() {

String environment = System.getProperty("env");

Assumptions.assumingThat("development".equals(environment),

() -> {

// Test logic specific to development environment

});

// Common test logic

}

**Repeated Tests:-** In JUnit 5, repeated tests allow you to execute a test method multiple times with different parameters or configurations. @RepeatedTest annotation is used in place of @Test annotation

@DisplayName("Repeat Contact Creation Test 5 Times")

@RepeatedTest(value = 5,name = "Repeating Contact Creation Test {currentRepetition} of {totalRepetitions}")

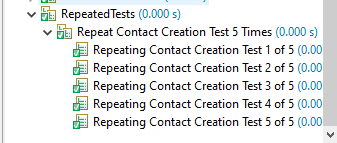
**public** **void** shouldTestContactCreationRepeatedly() {

contactManager.addContact("John", "Doe", "0123456789");

*assertFalse*(contactManager.getAllContacts().isEmpty());

*assertEquals*(1, contactManager.getAllContacts().size());

}

****

**Parameterized Tests:-**

Parameterized tests in JUnit allow you to run the same test method with different sets of parameters. This is particularly useful when you want to test the behavior of a method under various input values or conditions.

Data is provided using @ValueSource, @CsvSource, @CsvFileSource annotation@ParameterizedTest annotation is used in place of @Test annotation@ValueSource : Provides a single value or array of values as input parameters.

@DisplayName("Phone Number should match the required Format")

@ParameterizedTest

@ValueSource(strings = {"0123456789", "0123456798", "0123456897"})

**public** **void** shouldTestPhoneNumberFormatUsingValueSource(String phoneNumber){

contactManager.addContact("John", "Doe", phoneNumber);

*assertFalse*(contactManager.getAllContacts().isEmpty());

*assertEquals*(1, contactManager.getAllContacts().size());

}

@CsvSource: Allows us to specify comma-separated values as input parameters

@DisplayName("CSV Source Case - Phone Number should match the required Format")

@ParameterizedTest

@CsvSource({"0123456789", "0123456798", "0123456897"})

**public** **void** shouldTestPhoneNumberFormatUsingCSVSource(String phoneNumber){

contactManager.addContact("John", "Doe", phoneNumber);

*assertFalse*(contactManager.getAllContacts().isEmpty());

*assertEquals*(1, contactManager.getAllContacts().size());

}

@CsvFileSource: It allows you to provide input parameters for parameterized tests from a CSV (Comma-Separated Values) file.

@DisplayName("CSV File Source Case - Phone Number should match the required Format")

@ParameterizedTest

@CsvFileSource(resources = "/data.csv")

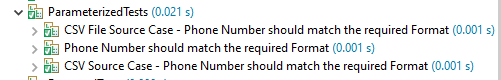
**public** **void** shouldTestPhoneNumberFormatUsingCSVFileSource(String phoneNumber) {

contactManager.addContact("John", "Doe", phoneNumber);

*assertFalse*(contactManager.getAllContacts().isEmpty());

*assertEquals*(1, contactManager.getAllContacts().size());

}

****

**@MethodSource –** It is an annotation in JUnit 5 that allows you to specify a method as the source of parameters for a parameterized test

@DisplayName("Method Source Case - Phone Number should match the required Format")

Will take the parameters from the method phoneNumberList

@ParameterizedTest

@MethodSource("phoneNumberList")

**public** **void** shouldTestPhoneNumberFormatUsingMethodSource(String phoneNumber) {

contactManager.addContact("John", "Doe", phoneNumber);

*assertFalse*(contactManager.getAllContacts().isEmpty());

*assertEquals*(1, contactManager.getAllContacts().size());

}

**private** **static** List<String> phoneNumberList() {

**return** Arrays.*asList*("0123456789", "0123456798", "0123456897");

}

**Nested Tests:-**

Nested tests in JUnit 5 allow you to group related tests within a test class, providing a more structured and readable way to organize your test code.

Suppose all the Repeated Tests are under the same umbrella . Similarly all the parameterized tests also.

How?

**Create a Test Class:** Define your test class and annotate it with **@Nested** to indicate that it contains nested test classes or methods.

@Nested

**class** RepeatedTests {

@DisplayName("Repeat Contact Creation Test 5 Times")

@RepeatedTest(value = 5,name = "Repeating Contact Creation Test {currentRepetition} of {totalRepetitions}")

**public** **void** shouldTestContactCreationRepeatedly() {

contactManager.addContact("John", "Doe", "0123456789");

*assertFalse*(contactManager.getAllContacts().isEmpty());

*assertEquals*(1, contactManager.getAllContacts().size());

}

}

@Nested

In JUnit 5, methods inside a nested test class are typically not static because they are intended to be instance methods

**class** ParameterizedTests {

@DisplayName("Phone Number should match the required Format")

@ParameterizedTest

@ValueSource(strings = {"0123456789", "0123456798", "0123456897"})

**public** **void** shouldTestPhoneNumberFormatUsingValueSource(String phoneNumber) {

contactManager.addContact("John", "Doe", phoneNumber);

*assertFalse*(contactManager.getAllContacts().isEmpty());

*assertEquals*(1, contactManager.getAllContacts().size());

}

@DisplayName("CSV Source Case - Phone Number should match the required Format")

@ParameterizedTest

@CsvSource({"0123456789", "0123456798", "0123456897"})

**public** **void** shouldTestPhoneNumberFormatUsingCSVSource(String phoneNumber) {

contactManager.addContact("John", "Doe", phoneNumber);

*assertFalse*(contactManager.getAllContacts().isEmpty());

*assertEquals*(1, contactManager.getAllContacts().size());

}

@DisplayName("CSV File Source Case - Phone Number should match the required Format")

@ParameterizedTest

@CsvFileSource(resources = "/data.csv")

**public** **void** shouldTestPhoneNumberFormatUsingCSVFileSource(String phoneNumber) {

contactManager.addContact("John", "Doe", phoneNumber);

*assertFalse*(contactManager.getAllContacts().isEmpty());

*assertEquals*(1, contactManager.getAllContacts().size());

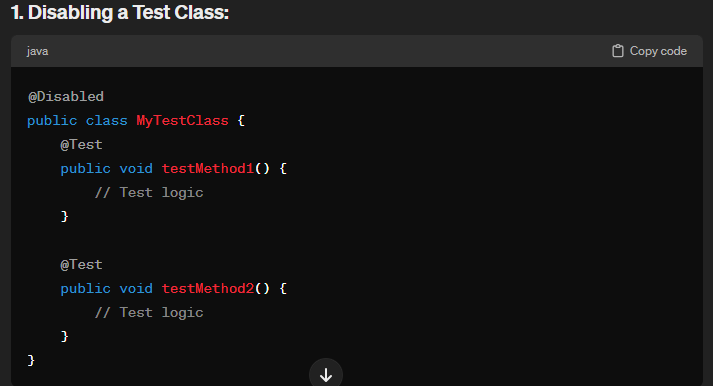
}

By making test methods instance methods, they can access instance variables and other non-static resources of the enclosing class.

}

**Disabled Tests:-**

**@Disabled** annotation is used to disable a test class or test method from execution. This annotation can be applied at both the class level and the method level.

****

@Test

@DisplayName("Test Should Be Disabled")

@Disabled

**public** **void** shouldBeDisabled() {

**throw** **new** RuntimeException("Test Should Not be executed");

}

Disabled annotation skips the method named shouldBeDisabled() annotated with **@Test** annotation

**Diff between JUnit4 & JUnit5:-**

**Improved Architecture:**

* JUnit 5 is built using Java 8 and takes advantage of its features like lambdas and streams.
* The modular architecture allows for better separation of concerns and easier integration with third-party libraries and tools.

**Annotations:**

* JUnit 5 introduces new annotations like **@Test**, **@BeforeAll**, **@AfterAll**, **@BeforeEach**, and **@AfterEach**, which provide more control over test lifecycle methods.
* Additional annotations like **@DisplayName**, **@Disabled**, **@ParameterizedTest**, and **@RepeatedTest** are introduced for better test organization and readability.

**Exception Handling in Junit Testing:-**