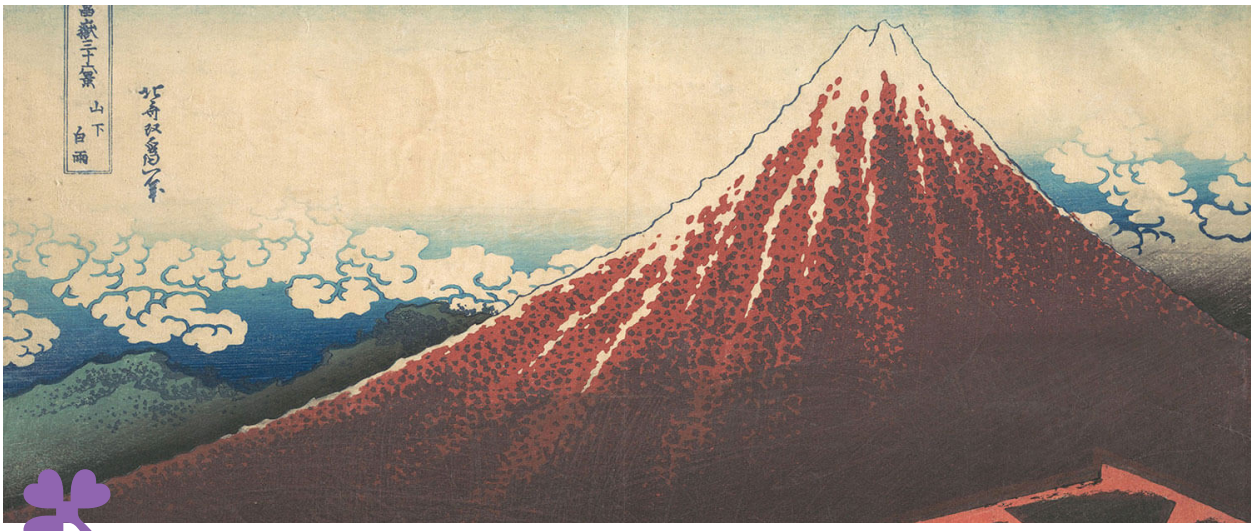

Annotation-Unit Testing

Annotations for Junit testing

The Junit 4.x framework is annotation based, so let's see the annotations that can be used while writing the test cases.

@Test annotation specifies that method is the test method.

@Test(timeout=1000) annotation specifies that method will be failed if it takes longer than 1000 milliseconds (1 second).

@BeforeClass annotation specifies that method will be invoked only once, before starting all the tests.

@Before annotation specifies that method will be invoked before each test.

@After annotation specifies that method will be invoked after each test.

@AfterClass annotation specifies that method will be invoked only once, after finishing all the tests.

@TestInstance() To change the behavior of test instances..
we have two option

@TestInstance(TestInstance.Lifecycle.PER_METHOD) → It will comes by default..

For every test method the instance will be created. Even if we don't specify it still it takes by default.

If we have 100 test method then for 100 instance will be created. If we don't want that then simply we can use PER_CLASS.

@TestInstance(TestInstance.Lifecycle.PER_CLASS)
Now in that scenario the instance will be created only once.

If we are using @TestInstance(TestInstance.Lifecycle.PER_CLASS)
then there is no need to use static method Infront of those methods before which we are using @BeforeAll.. Because if we are changing the behavior of the test methods then in case of @TestInstance(TestInstance.Lifecycle.PER_CLASS) it will create one instance will be created.

@DisplayName annotation provides a custom name for a test method or class. Here Test classes and test methods can declare custom display names — with spaces, special characters, and even emojis — that will be displayed by test runners and test reporting.

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

@DisplayName("A special test case")
class DisplayNameDemo {

    @Test
    @DisplayName("Custom test name containing spaces")
    void testWithDisplayNameContainingSpaces() {
    }

    @Test
    @DisplayName("╯°□°）╯")
    void testWithDisplayNameContainingSpecialCharacters() {
    }
```

@Tag annotation declares a tag for a test method or class. Tags can be used to filter which tests are run for a given test plan.

@ParameterizedTest

Parameterized tests make it possible to run a test multiple times with different arguments.They are declared just like regular *@Test* methods but use the @ParameterizedTest annotation instead. In addition, you must declare at least one source that will provide the arguments for each invocation and then consume the arguments in the test method. Example: The following example demonstrates a parameterized test that uses the @ValueSource annotation to specify a String array as the source of arguments.

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I ere I s
void palindromes(String candidate) {
    assertTrue(isPalindrome(candidate));
}
```

@RepeatedTest

JUnit Jupiter provides the ability to repeat a test a specified number of times simply by annotating a method with @RepeatedTest and specifying the total number of repetitions desired. Each invocation of a repeated test behaves like the execution of a regular *@Test* method with full support for the same lifecycle callbacks and extensions.

Example: The following example demonstrates how to declare a test named *repeatedTest()* that will be automatically repeated 10 times.

```
    @RepeatedTest(10)
    void repeatedTest() {
       // ...
    }
   @RepeatedTest(5)
    void repeatedTestWithRepetitionInfo(RepetitionInfo repetitio
        assertEquals(5, repetitionInfo.getTotalRepetitions());
```

```
    }

    @RepeatedTest(value = 1, name = "{displayName} {currentRepet
    @DisplayName("Repeat!")
    void customDisplayName(TestInfo testInfo) {
        assertEquals(testInfo.getDisplayName(), "Repeat! 1/1");
    }
```

@BeforeEach

Denotes that the annotated method should be executed before each @Test, @RepeatedTest, @ParameterizedTest, or @TestFactory method in the current class; analogous to JUnit 4’s @Before. Such methods are inherited unless they are overridden.

@AfterEach

Denotes that the annotated method should be executed after each @Test, @RepeatedTest, @ParameterizedTest, or @TestFactory method in the current class; analogous to JUnit 4’s @After. Such methods are inherited unless they are overridden.

```
class StandardTests {
    @BeforeEach
    void init() {
    }

    @Test
    void succeedingTest() {
    }

    @AfterEach
    void tearDown() {
```

```
    }
}
```

@ExtendWith

Used to register custom extensions. Such annotations are inherited. Example:For example, to register a custom RandomParametersExtension for a particular test method, you would annotate the test method as follows.

```
@ExtendWith(RandomParametersExtension.class)
@Test
void test(@Random int i) {
    // ...
}
```

Assert class

The org.junit.Assert class provides methods to assert the program logic.

Methods of Assert class

The common methods of Assert class are as follows:

void assertEquals(boolean expected,boolean actual): checks that two primitives/objects are equal. It is overloaded.
void
assertTrue(boolean condition): checks that a condition is true.
void
assertFalse(boolean condition): checks that a condition is false.
void
assertNull(Object obj): checks that object is null.
void
assertNotNull(Object obj): checks that object is not null.