JUnit5 Presentation & Documentation (Short Version)

- Written By Payam Aghaei
- For usage of SabaPardazesh Co. developers

Why Testing ?

Human errors can cause a ***defect*** or ***failure*** at any stage of the software development life cycle . The results are classified as trivial or catastrophic, depending on the consequences of the error.

The testing is important since **it discovers defects/bugs before the delivery to the client, which guarantees the quality of the software**. It makes the software more reliable and easy to use. Thoroughly tested software ensures reliable and high-performance software operation.

What is Unit Testing

- A unit test is a way of testing a unit - the smallest piece of code that can be logically isolated in a system. A unit can be almost anything you want it to be -- a line of code, a method, or a class . Generally though, smaller is better . Smaller tests give you a much more granular view of how your code is performing. There is also the practical aspect that when you test very small units, your tests can be run fast; like a thousand tests in a second fast.

What is Junit?

- JUnit is an open source Unit Testing Framework for Java. It is useful for Java Developers to write and run repeatable tests. Erich Gamma and Kent Beck initially develop it. It is an instance of XUnit architecture. As the name implies, it is used for Unit Testing of a small chunk of code.
- Once you are done with code, you should execute all tests, and it should pass. Every time any code is added, you need to re-execute all test cases and makes sure nothing is broken.
- Developers who are following test-driven methodology must write and execute unit test first before any code.

Test-driven development

- **Test-driven development** (**TDD**) is a software development process relying on software requirements being converted to test cases before software is fully developed, and tracking all software development by repeatedly testing the software against all test cases. This is as opposed to software being developed first and test cases created later.

What is the benefits of JUnit ?

- It finds bugs early in the code, which makes our code more reliable.
- Unit testing forces a developer to read code more than writing.
- You develop more readable, reliable and bug-free code which builds confidence during development.
- Unit Testing is used to identify defects early in software development cycle.

Simple Java Calculation Class

```java
public class Calculation {

    public static int findMax(int arr[]){
        int max=0;
        for(int i=1;i<arr.length;i++){
            if(max<arr[i])
                max=arr[i];
        }
        return max;
    }
}
```

Test Class For Calculation Class

```
public class ClaculationTest {

    @Test
    public void testFindMax(){
      assertEquals(4,Calculation.findMax(new int[]{1,3,4,2}));
      assertEquals(-1,Calculation.findMax(new int[]{-12,-1,-3,-4,-2}));
    }
}
```

Assertions

- JUnit provides overloaded assertion methods for all primitive types and Objects and arrays (of primitives or Objects). The parameter order is expected value followed by actual value. Optionally the first parameter can be a String message that is output on failure. There is a slightly different assertion, `assertThat` that has parameters of the optional failure message, the actual value, and a `Matcher` object. Note that expected and actual are reversed compared to the other assert methods.

Usefull Assertions

- **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.

Simple Example For Assertions

```
 @Test
public void testAssertEquals() {
  assertEquals( "text", "text");
}

@Test
public void testAssertFalse() {
  assertFalse("failure - should be false", false);
}

@Test
public void testAssertNotNull() {
  assertNotNull("should not be null", new Object());
}
```

Exception Handling

assertThrows() Assert that execution of the supplied executable throws an exception of the expectedType and return the exception. if no exception is thrown, or if an exception of a different type is thrown, this method will fail. If you do not want to perform additional checks on the exception instance, ignore the return value.

```
@Test
void exception(){
    String str = null;
    assertThrows(NullPointerException.class,()->
str.length());
}
```

Assumptions

- Ideally, the developer writing a test has control of all of the forces that might cause a test to fail. If this isn't immediately possible, making dependencies explicit can often improve a design. For example, if a test fails when run in a different locale than the developer intended, it can be fixed by explicitly passing a locale to the domain code.
- However, sometimes this is not desirable or possible. It's good to be able to run a test against the code as it is currently written, implicit assumptions and all, or to write a test that exposes a known bug. For these situations, JUnit now includes the ability to express assumptions

Usefull Assumptions

- assumeTrue(**boolean** assumption)

The `assumeTrue()` method validates the given assumption to be *true* and if the assumption is *true* – the test proceed, otherwise, test execution is aborted.

- assumeFalse(**boolean** assumption)

The `assumeFalse()` method validates the given assumption to *false* and if the assumption is *false* – test proceed, otherwise, test execution is aborted.

- assumingThat (**boolean** assumption, Executable executable)

This method executes the supplied `Executable`, but only if the supplied assumption is valid.

Unlike the other assumption methods, this method will not abort the test.

If the assumption is invalid, this method does nothing.

If the assumption is valid and the `executable` throws an exception, it will be treated like a regular test *failure*. The thrown exception will be rethrown *as is* but `masked` as an unchecked exception.

AssumeTrue & AssumeFalse

```
@Test
 void testOnDev() {
 System.setProperty("ENV", "App");
 Assumptions.assumeTrue("App".equals(System.getProperty("ENV")));
 //remainder of test will proceed }

@Test
 void testOnDev() {
 System.setProperty("ENV", "App");
 Assumptions.assumeFalse("App".equals(System.getProperty("ENV")),
 AppTest::message);
 //remainder of test will be aborted }
```

AssumingThat

```
@Test
 void testInAllEnvironments() {
 assumingThat("DEV".equals(System.getenv("ENV")),
() -> {
 // perform these assertions only on the DEV server
assertEquals(2, calculator.divide(4, 2)); });
 // perform these assertions in all environments
 assertEquals(42, calculator.multiply(6, 7)); }
```

@JUnit5 Annotations

You will notice that in Junit 5, one of the most obvious changes is that test classes and methods do not have to be public anymore.

Now, let’s go through the list of most common JUnit 5 Annotations.

@Test

This annotation denotes that a method is a test method. Note this annotation does not take any attributes.

@DisplayName

- Test classes and test methods can declare custom display names that will be displayed by test runners and test reports.
- `@DisplayName("DisplayName Demo")`
- `class JUnit5Test {`
- `    @Test`
- `    @DisplayName("Custom test name")`
- `    void testWithDisplayName() {`
- `    }`

@BeforeEach

- The @BeforeEach annotation denotes that the annotated method should be executed before each test method, analogous to JUnit 4’s @Before.

```
@BeforeEach
    void init(TestInfo testInfo) {
        String callingTest
= testInfo.getTestMethod().get().getName();
        System.out.println(callingTest);
    }
```

@AfterEach

- This annotation denotes that the annotated method should be executed after each test method, analogous to JUnit 4's `@After`. For example, if the tests need to reset a property after each test, we can annotate a method with `@AfterEach` for that task.

```
@AfterEach
    void after(TestInfo testInfo) {
        String callingTest =
testInfo.getTestMethod().get().getName();
        System.out.println(callingTest);
    }
```

@BeforeAll

- This annotation executes a method before all tests. This is analogous to JUnit 4's `@BeforeClass`. The `@BeforeAll` annotation is typically used to initialize various things for the tests.

- `@BeforeAll`
- `    static void init() {`
- `        System.out.println("Only run once before all tests");`
- `    }`

@AfterAll

- The `@AfterAll` annotation is used to execute the annotated method, only after all tests have been executed. This is analogous to JUnit 4's `@AfterClass`. We use this annotation to tear down or terminate all processes at the end of all tests.

```
@AfterAll

    static void after() {

        System.out.println("Only run once after all tests");
    }
```

@Disabled

- The @Disabled annotation is used to disable or skip tests at class or method level. This is analogous to JUnit 4’s @Ignore. When declared at class level, all @test methods are skipped. When we use @Disabled at the method level, only the annotated method is disabled.

```
• @Disabled
• class DisabledClassDemo {
•
•     @Test
     @Disabled
     void testWillBeSkipped() {
     }
}
```

@Tag

- We can use this annotation to declare tags for filtering tests, either at the class or method level. The @Tag annotation is useful when we want to create a test pack with selected tests.

```
@Test
@Tag("login")

  void validLoginTest() {

    }
```

@RepeatedTest

JUnit 5 has 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.

This is particularly useful in UI testing with Selenium.

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

@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.

For example, the following example demonstrates a parameterized test that uses the `@ValueSource` annotation to specify a String array as the source of arguments.

Example for Parameterized Test

```
class JUnit5Test {

    @ParameterizedTest
    @ValueSource(strings = { "cali", "bali", "dani" 
})
    void endsWithI(String str) {
        assertTrue(str.endsWith("i"));
    }
```

@ParameterizedTest with more than one arguments

- ParameterizedTest With more than one Arguments
- With name attribute we can put a name for every tests inside method

```
@ParameterizedTest(name = "{0} length is {1}")
@CsvSource(value = {"as , 2" , "bahs , 4" , "a , 1"})
void csv(String str , int length){
    Assertions.assertEquals(length , str.length());
}
```

@Nested

@Nested tests give the test writer more capabilities to express the relationship among several groups of tests. Such nested tests make use of Java's nested classes and facilitate hierarchical thinking about the test structure.

```
class OrderedNestedTestClassesDemo {

    @Nested
    class PrimaryTests {

        @Test
        void test1() {
        }
    }
}
```

@Order

@Order is an annotation that is used to configure the order in which the annotated element (i.e., field, method, or class) should be evaluated or executed relative to other elements of the same category.

```
@Test
@Order(value = 1)
void order() {
}
```

@Timeout

`@Timeout` is used to define a timeout for a method or all testable methods within one class and its `@Nested` classes.

This annotation may also be used on lifecycle methods annotated with `@BeforeAll` , `@BeforeEach` , `@AfterEach` , or `@AfterAll` .

Applying this annotation to a test class has the same effect as applying it to all testable methods, i.e. all methods annotated or meta-annotated with `@Test` , `@TestFactory` , or `@TestTemplate` , but not to its lifecycle methods.

```
@Test
@Timeout(22)
void timeout() {
}
```

Lifecycle Of Test Classes

In order to allow individual test methods to be executed in isolatin and to avoid unexpected side effects due to mutable test instance state, JUnit creates a new instance of each test class before executing each test method . This "per-method" test instance lifecycle is the default behavior in JUnit Jupiter and is analogous to all previous versions of JUnit .

@TestInstance

```
@TestInstance(
value = TestInstance.Lifecycle.PER_CLASS)
class OrderedNestedTestClassesDemo {

        @Test
        void test1() {
        }
    }
```

@TestFactory

- @TestFactory method is not itself a test case but rather a factory for test cases. Thus, a dynamic test is the product of a factory. Technically speaking, a @TestFactory method must return a single DynamicNode or a Stream, Collection, Iterable, Iterator, or array of DynamicNode instances.
- allows us to declare and run test cases generated at run-time. Contrary to Static Tests, which define a fixed number of test cases at the compile time, Dynamic Tests allow us to define the test cases dynamically in the runtime.

```
@TestFactory
List dynamicTestsWithInvalidReturnType() {
    return (List) new ArrayList();
}
```

Dependency Injection

- In all prior JUnit versions, test constructors or methods weren’t allowed to have parameters. As one of the major changes in JUnit 5, both test constructors and methods are now permitted to have parameters. This allows for greater flexibility and enables dependency injection for constructors and methods.

- `ParameterResolver` defines the API for test extensions that wish to dynamically resolve parameters at runtime. If a test class constructor, a test method, or a lifecycle method accepts a parameter, the parameter must be resolved at runtime by a registered `ParameterResolver`. You can inject as many parameters as you want in any order you want them to be.

Test Class Example for Dependency Injection

```
@ExtendWith(BankAccountParameterResolver.class)
class BankAccountTest {


@Test
void deposit(BankAccount bankAccount) {
    int amount = 100;
    Assumptions.assumeTrue(bankAccount.getBalance() +
amount <= bankAccount.getMaxBalance() &&
bankAccount.isActive());
} }
```