TESTING

According to ANSI/IEEE 1059 standard, Testing in Software Engineering is a process of analyzing a software product or system to examining a software product or system to determine whether it satisfies or fails to satisfy established conditions (i.e., defects). The testing process evaluates the software products characteristics for requirements such as missing requirements, bugs, or errors in order to evaluate its reliability, security, and performance.

Why Do We need to Test

Answer is simple. We are not perfect. We all do mistake, because of lack of knowledge, not covering the whole scenario, fast coding, not being in good mood or day, not well managed merge etc. many other reason and excuse can be counted.

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.

Testing is necessary, Because

- 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.

So How Can We Test Our Application?

Application test can be done either `manually` or `automated`.

Manual Testing

Manual testing is done in person, by clicking through the application or interacting with the software and APIs with the appropriate tooling. This is very expensive since it requires someone to setup an environment and execute the tests themselves, and it can be prone to human error as the tester might make typos or omit steps in the test script.

Automated Testing

Automated tests, on the other hand, are performed by a machine that executes a test script that was written in advance. These tests can vary in complexity, from checking a single method in a class to making sure that performing a sequence of complex actions in the UI leads to the same results. It’s much more robust and reliable than manual tests – but the quality of your automated tests depends on how well your test scripts have been written.

The different types of tests

- Unit tests
- Integration tests
- Functional tests
- End-to-end tests
- Acceptance testing
- Performance testing
- Smoke testing

In this course we will talk about `Unit Testing.`

Unit tests

Unit tests are very low level and close to the source of an application. They consist in testing individual methods and functions of the classes, components, or modules used by your software. Unit tests are generally quite cheap to automate and can run very quickly by a continuous integration server.

Java Testing Frameworks

There are some popular test tool and framework for testing our Java application.

1. JUnit
2. Mockito
3. Selenium
4. Cucumber
5. TestNG

JUNIT

JUnit is an open-source foundation of testing frameworks on the Java Virtual Machine. The aim of the JUnit testing framework primarily focuses on efficiently creating and executing unit testing for an application. The JUnit testing framework establishes a Test Engine API that discovers and executes the unit tests on the testing framework platform.

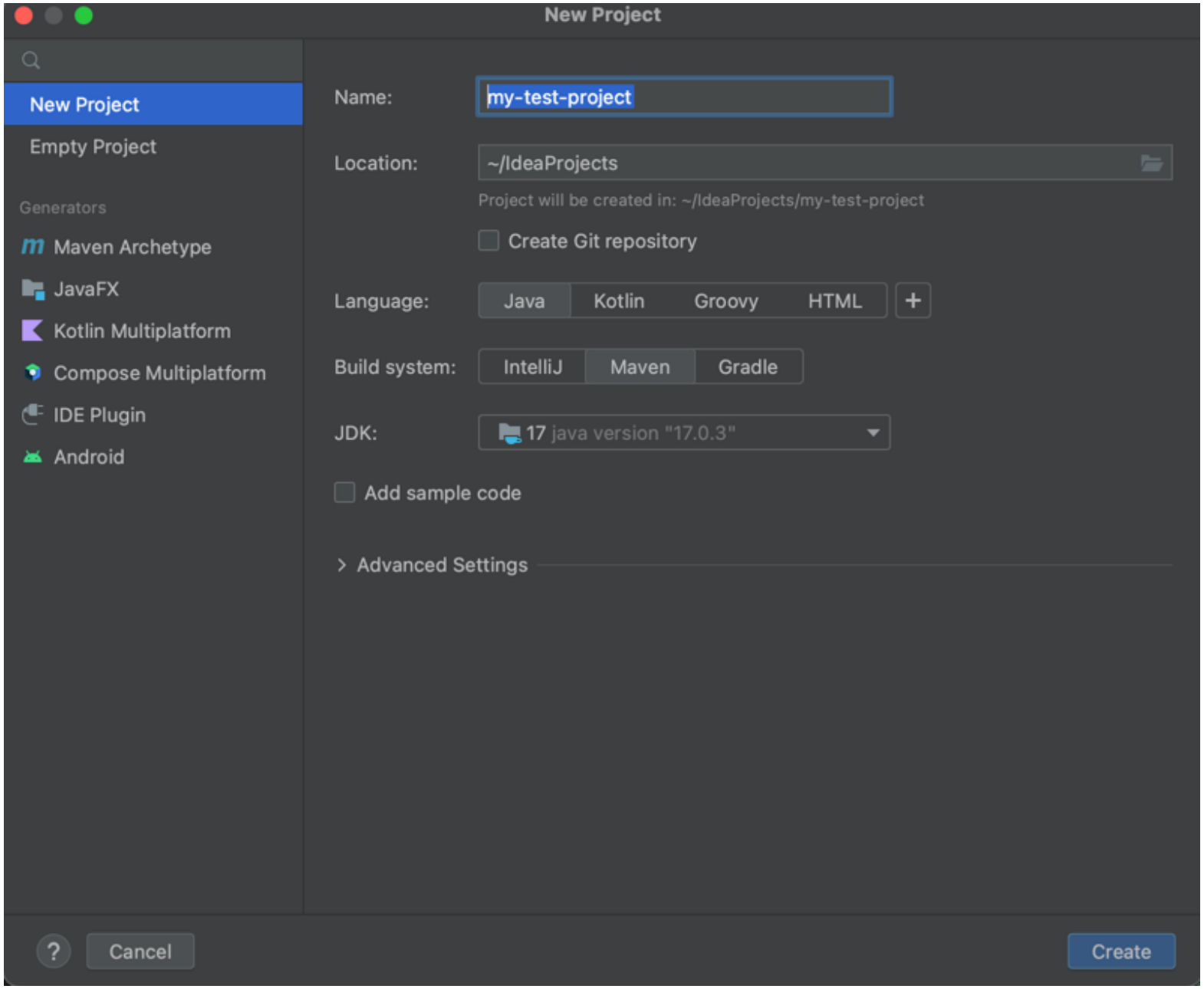
JUnit is linked as a JAR at compile-time. The latest version of the framework, JUnit 5, resides under package org.junit.jupiter.

How To Set up Our Test Environment

1 Create a Java Maven Project

For this course we are going to use Intellij Idea.

Create a new Project and select Maven on list


New Project
New Project
Empty Project
Generators
Maven Archetype
JavaFX
Kotlin Multiplatform
Compose Multiplatform
IDE Plugin
Android
Name:
my-test-project
Location:
~/IdeaProjects
Project will be created in: ~/IdeaProjects/my-test-project
Create Git repository
Language:
Java
Kotlin
Groovy
HTML
Build system:
IntelliJ
Maven
Gradle
JDK:
17 java version "17.0.3"
Add sample code
Advanced Settings
Cancel
Create

2 Put Junit Dependency Library

open pom.xml file and put junit dependency

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>my-test-project</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter</artifactId>
            <version>5.8.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

3 Write First Test Code

Open project and under src package you will see test package.

Create a class under src > test > java.

write your first test method as on example.

Example:

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

public class TestApp {

    @Test
    void firstTest(){
        int total=2+2;

        assertEquals(4,total);
    }
}
```

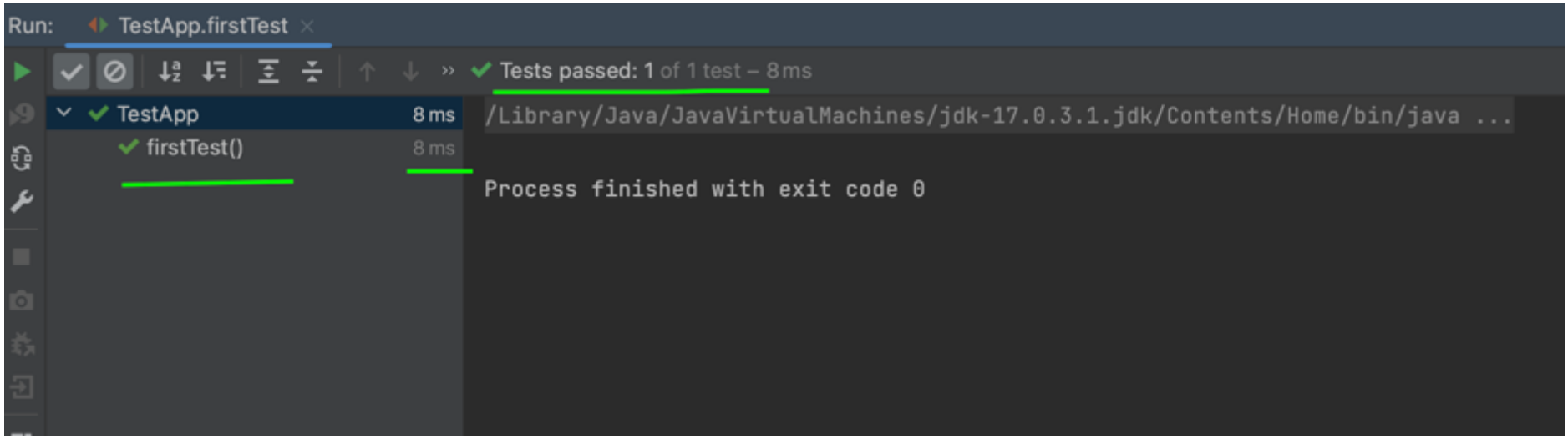
4 Run Your Test Code

- On the left side of the method name you will see a green button. You can explicitly test your function by clicking that icon.

- Or you can right-click on method name and choose run

Test Success

If your application passed the test you will see the result on console and how much time passed to execute this test method.



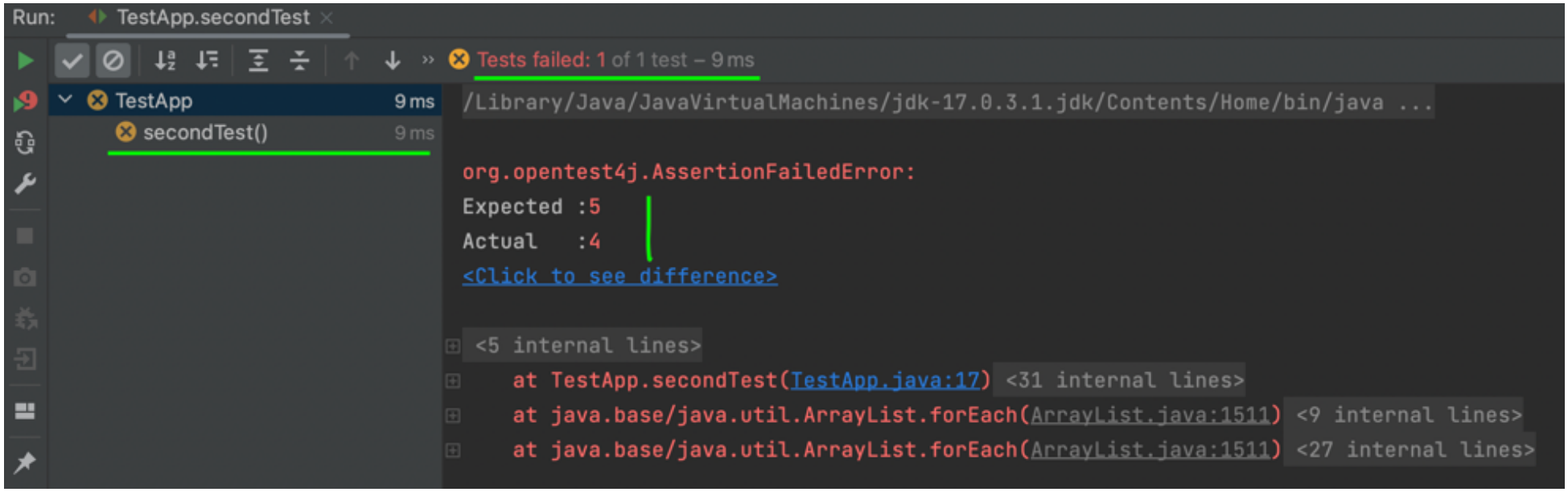
Test Fail

If your application fails you can see the result on console, and detailed description. As What was the actual and what was the expectation.

Example: Fail test

```
    @Test
     void secondTest(){
        int total =2+2;
        assertEquals(5,total);
```

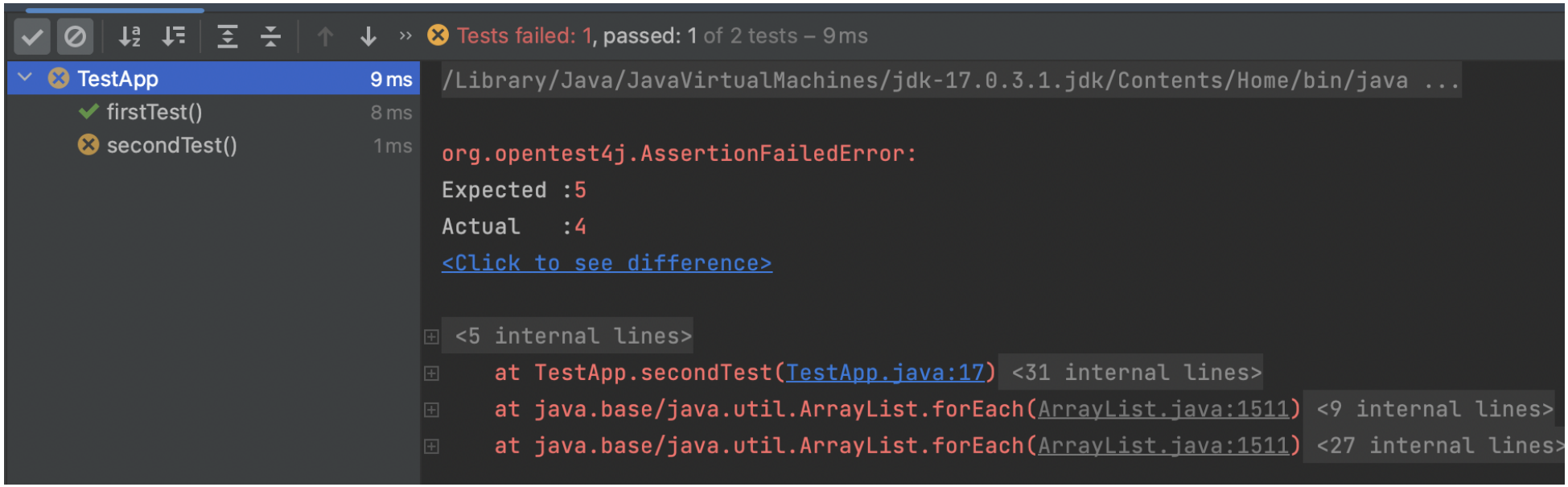
```
    }
```



Test All Methods

- You run all your test methods in once either clicking run button on the left of the class declaration
- Or you can right-click on class file or class name and choose run button.

When your test codes run you will see the result of all test methods on the console.



JUnit Assertions Class

Assertions is a JUnit API or library of functions through which you can verify if a particular logic or condition returns true or false after execution of the test. If it returns false, then an AssertionError is thrown.Only failed assertions are recorded.

Some important methods of Assert class are as follows

Method	Description
void assertEquals(boolean expected, boolean actual)	Checks that two primitives/objects are equal.
void assertFalse(boolean condition)	Checks that a condition is false.
void assertTrue(boolean condition)	Checks that a condition is true.
void assertNotNull(Object object)	Checks that an object isn’t null.

Method	Description
void assertNull(Object object)	Checks that an object is null.

Example

```
public class DummyClass {

    public long cubeFunction(int val){
        return val*val*val;
    }
}
```

```
    @Test
    void testCubeFunction(){

        DummyClass dummyClass=new DummyClass();

        long cube=dummyClass.cubeFunction(2);

        Assertions. assertEquals(8,cube);
    }
```

Assignment01 : Anagram

Exception Testing

There are two ways of exception testing in JUnit 5, both of which we can implement using the assertThrows() method:

```
@Test
void shouldThrowException() {
    Throwable exception = assertThrows(UnsupportedOperationException.class, () -> {
      throw new UnsupportedOperationException("Not supported");
    });
    assertEquals("Not supported", exception.getMessage());
}

@Test
void assertThrowsException() {
    String str = null;
    assertThrows(IllegalArgumentException.class, () -> {
      Integer.valueOf(str);
    });
}
```

The first example verifies the details of the thrown exception, and the second one validates the type of exception.

JUnit Annotations

JUnit Annotations is a special form of syntactic meta-data that can be added to Java source code for better code readability and structure. Variables, parameters, packages, methods and classes can be annotated.

Some of the JUnit annotations:

- `@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).
- `@BeforeEach` – denotes that the annotated method will be executed before each test method

- `@BeforeAll` – denotes that the annotated method will be executed before all test methods in the current class
- `@AfterEach` – denotes that the annotated method will be executed after each test method
- `@AfterAll` – denotes that the annotated method will be executed after all test methods in the current class
- `@Disable` – disables a test class or method
- `@DisplayName` – defines a custom display name for a test class or a test method

@BeforeAll and @BeforeEach

```
@BeforeAll
static void setup() {
    log.info("@BeforeAll - executes once before all test methods in this class");
}

@BeforeEach
void init() {
    log.info("@BeforeEach - executes before each test method in this class");
}
```

WARNING | It's important to note that the method with the @BeforeAll annotation needs to be static, otherwise the code won't compile

@DisplayName and @Disabled

```
@DisplayName("Single test successful")
@Test
void testSingleSuccessTest() {
    log.info("Success");
}
```

```
@Test
@Disabled("Not implemented yet")
void testShowSomething() {
}
```

NOTE | As we can see, we can change the display name or disable the method with a comment, using new annotations.

@AfterEach and @AfterAll

```
@AfterEach
void tearDown() {
    log.info("@AfterEach - executed after each test method.");
}

@AfterAll
static void done() {
    log.info("@AfterAll - executed after all test methods.");
}
```

Assignment02:

Parameterized Test

JUnit `@ParameterizedTest` annotation enables us to execute a single test method multiple times with different parameters.

To test method with different parameters we must provide a source to method with `@ValueSource` annotation

```
public class Numbers {
    public static boolean isOdd(int number) {
```

```
        return number % 2 != 0;
    }
}
```

```
class TestOddNumber{

 @ParameterizedTest
 @ValueSource(ints = {1, 3, 5, 23, 87, 999}) // six numbers
 void isOdd_ShouldReturnTrueForOddNumbers(int number) {
    assertTrue(Numbers.isOdd(number));
 }
}
```

this above code test isOdd method of Numbers class with each parameter of ints values. And each time, it assigns a different value from the `@ValueSource` array to the number method parameter.

Argument Sources

As we should know by now, a parameterized test executes the same test multiple times with different arguments.

And we can hopefully do more than just numbers, so let’s explore.

Simple Values

With the @ValueSource annotation, we can pass an array of literal values to the test method.

Suppose we’re going to test our simple isBlank method:

```
public class Strings {
    public static boolean isBlank(String input) {
```

```
        return input == null || input.trim().isEmpty();
    }
}
```

We expect from this method to return true for null for blank strings. So, we can write a parameterized test to assert this behavior:

```
@ParameterizedTest
@ValueSource(strings = {"", "  "})
void isBlank_ShouldReturnTrueForNullOrBlankStrings(String input) {
    assertTrue(Strings.isBlank(input));
}
```

Null and Empty Values

we can pass a single null value to a parameterized test method using `@NullSource`:

```
@ParameterizedTest
@NullSource
void isBlank_ShouldReturnTrueForNullInputs(String input) {
    assertTrue(Strings.isBlank(input));
}
```

Since primitive data types can't accept null values, we can't use the `@NullSource` for primitive arguments.

Quite similarly, we can pass empty values using the `@EmptySource` annotation:

```
@ParameterizedTest
@EmptySource
void isBlank_ShouldReturnTrueForEmptyStrings(String input) {
    assertTrue(Strings.isBlank(input));
```

```
}
```

Assignment03: Palindrome

Modul: Testing principles

TDD

- 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.

TDD workflow

1. Add a test
    - the adding of a new feature begins by writing a test
    - the developer can discover these specifications by asking about use cases and user stories
    - focus on requirements before writing code
2. Run all tests. The new test should fail for expected reasons
    - This shows that new code is actually needed for the desired feature
    - validates that the test harness is working correctly
    - rules out the possibility that the new test is flawed and will always pass

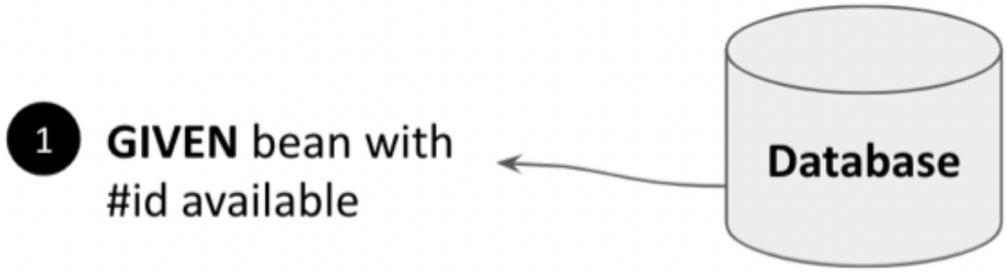
3. Write the simplest code that passes the new test
    - inelegant or hard code is acceptable, as long as it passes the test
    - no code should be added beyond the tested functionality.
4. All tests should now pass
    - If any fail, the new code must be revised until they pass
    - This ensures the new code meets the test requirements and does not break existing features
5. Refactor as needed, using tests after each refactor to ensure that functionality is preserved
    - Code is refactored for readability and maintainability
    - hard-coded test data should be removed
    - Running the test suite after each refactor helps ensure that no existing functionality is broken

GWT

- Testing pattern
    - GIVEN a context
    - WHEN some condition
    - THEN expect some output

GWT

This is a typical format of GWT



3 **THEN** expect bean to returned

2 **WHEN** bean found in database

```
Bean findBean(int id){
   if(foundBeanInDatabase){
      return bean;
   } else{
      return null;
   }
}
```

GWT

With GWT template, it will be easier to understand what the function/method do. Once you defined Given-Then-When, that’s mean you already finish thinking about the control flow of your process and eventually, coding will be easier (But this is not as easy as it’s said). GWT can be useful when refactoring the code to become more readable == Module: Mockito

What is Mocking?

- Mocking is a way to test the functionality of a class in isolation
- Mock objects do the mocking of the real service
- A mock object returns a dummy data corresponding to some dummy input passed to it.

Benefits of Mockito

- No Handwriting ¬ No need to write mock objects on your own.
- Refactoring Safe ¬ Renaming interface method names or reordering parameters will not break the test code as Mocks are created at runtime.
- Return value support ¬ Supports return values.
- Exception support ¬ Supports exceptions.
- Order check support ¬ Supports check on order of method calls.
- Annotation support ¬ Supports creating mocks using annotation.

Features of mockito

mock()

- It is used to create mock objects of a concrete class or an interface. It takes a class or an interface name as a parameter.
- Has multiple implementations with many other possibilities
- Most commonly used as

```
ToDoService doService = mock(ToDoService.class);
```

when()

It enables stubbing methods. It should be used when we want to mock to return specific values when particular methods are called. In simple terms, "When the XYZ() method is called, then return ABC." It is mostly used when there is some condition to execute.

```
when(mock.someCode()).thenReturn(5);
```

verify()

The verify() method is used to check whether some specified methods are called or not. In simple terms, it validates the certain behavior that happened once in a test. It is used at the bottom of the testing code to assure that the defined methods are called.

Mockito framework keeps track of all the method calls with their parameters for mocking objects. After mocking, we can verify that the defined conditions are met or not by using the verify() method. This type of testing is sometimes known as behavioral testing. It checks that a method is called with the right parameters instead of checking the result of a method call.

The verify() method is also used to test the number of invocations. So we can test the exact number of invocations by using the times method, at least once method, and at most method for a mocked method.

```
List mockList = Mockito.mock(ArrayList.class);

mockList.add("one");

Mockito.verify(mockList).add("one");
```

spy()

Mockito provides a method to partially mock an object, which is known as the spy method. When using the spy method, there exists a real object, and spies or stubs are created of that real object. If we don't stub a method using spy, it will call the real method behavior. The main function of the spy() method is that it overrides the specific methods of the real object. One of the functions of the spy() method is it verifies the invocation of a certain method.

You can spy an object by using spy method of the mockito or using @Spy annotation

```
DummyService service =spy(new DummyService());
```

OR with @Spy annotation

```
@Spy
DummyService service =new DummyService();
```

Example

```
// here we spied the Arraylist
@Spy
List<String> spiedList = new ArrayList<>();
```

```
@Test
public void whenUseSpyAnnotation_thenSpyIsInjectedCorrectly() {

    /* or you can spy list as
    * List<String> spiedList=spy(new ArrayList<>());
     */

    // we add 2 values into the list
    spiedList.add("one");
    spiedList.add("two");


    Mockito.verify(spiedList).add("one");
    Mockito.verify(spiedList).add("two");

    // our list will contains 2 values
    assertEquals(2, spiedList.size());

    // here we stub the list  behaviour
    Mockito.doReturn(100).when(spiedList).size();
    assertEquals(100, spiedList.size());
}
```

times()

It is used to verify the exact number of method invocations, which means it declares how many times a method is invoked. The signature of the times() method is:

```
Mockito.verify(loginService, Mockito.times(1)).saveInSession(Mockito.any(Customer.class));
```

Additional functions

- reset()
    - it’s better to create new mocks rather than using reset()
    - rarely used, not recommended
- verifyNoMoreInteractions()
    - use this method after verifying all the mock, to make sure that nothing else was invoked on the mocks
    - detects the unverified invocations that occur before the test method
- verifyZeroInteractions()
    - verifies that no interaction has occurred on the given mocks.

Additional functions

- doThrow()
    - used to stub a void method to throw an exception
- doCallRealMethod()
    - used when we want to call the real implementation of a method
- doAnswer()
    - used when we want to stub a void method with a generic Answer type
- doNothing()
    - used for setting void methods to do nothing.

    - by default, the void methods on mock instances do nothing

- doReturn()
    - the Mockito.when(object) method is always suggested for stubbing because it is argument type-safe and more readable as compare to the doReturn() method.

Additional functions

- inOrder()
    - used to create objects that allow the verification of mocks in a specific order
- ignoreStubs()
    - used to ignore the stubbed methods of given mocks for verification
- never()
    - is used to verify that the interaction did not happen
- atLeastOnce()
    - used to verify the invocation at-least-once that means the method should be invoked at least once.
- atLeast(x)
    - used to verify the invocation at least x number of times

Additional functions

- atMost(x)
    - used to verify the invocation at most x number of times

- calls()
    - allows a non-greedy verification in order. It can only be used with the inOrder() verification method. For eg., inOrder.verify(mock, calls(3)).xyzMethod("…");
- only()
    - it checks that the given method was the only invoked method
- timeout()
    - perform verification with a timeout
- after()
    - allows Mockito to verify over a given period of time
    - after() method waits for the full period unless the final result is declared whereas the timeout() method will stop as soon as the verification passes

Example test

```
import java.util.ArrayList;
import java.util.List;

import static org.mockito.Mockito.*;

public class PortfolioTester {

   public void testPortfolio(){

      //Create a portfolio object which is to be tested
      Portfolio portfolio = new Portfolio();
```

```
      //Creates a list of stocks to be added to the portfolio
      List<Stock> stocks = new ArrayList<Stock>();
      Stock googleStock = new Stock("1","Google", 10);
      Stock microsoftStock = new Stock("2","Microsoft",100);

      stocks.add(googleStock);
      stocks.add(microsoftStock);

      //Create the mock object of stock service
      StockService stockServiceMock = mock(StockService.class);

      // mock the behavior of stock service to return the value of various stocks
      when(stockServiceMock.getPrice(googleStock)).thenReturn(50.00);
      when(stockServiceMock.getPrice(microsoftStock)).thenReturn(1000.00);

      //add stocks to the portfolio
      portfolio.setStocks(stocks);

      //set the stockService to the portfolio
      portfolio.setStockService(stockServiceMock);

      double marketValue = portfolio.getMarketValue();

      //verify the market value to be
      //10*50.00 + 100* 1000.00 = 500.00 + 100000.00 = 100500
      System.out.println("Market value of the portfolio: "+ marketValue);
   }
}
```

Assignment 04: Garage