Core Java 8

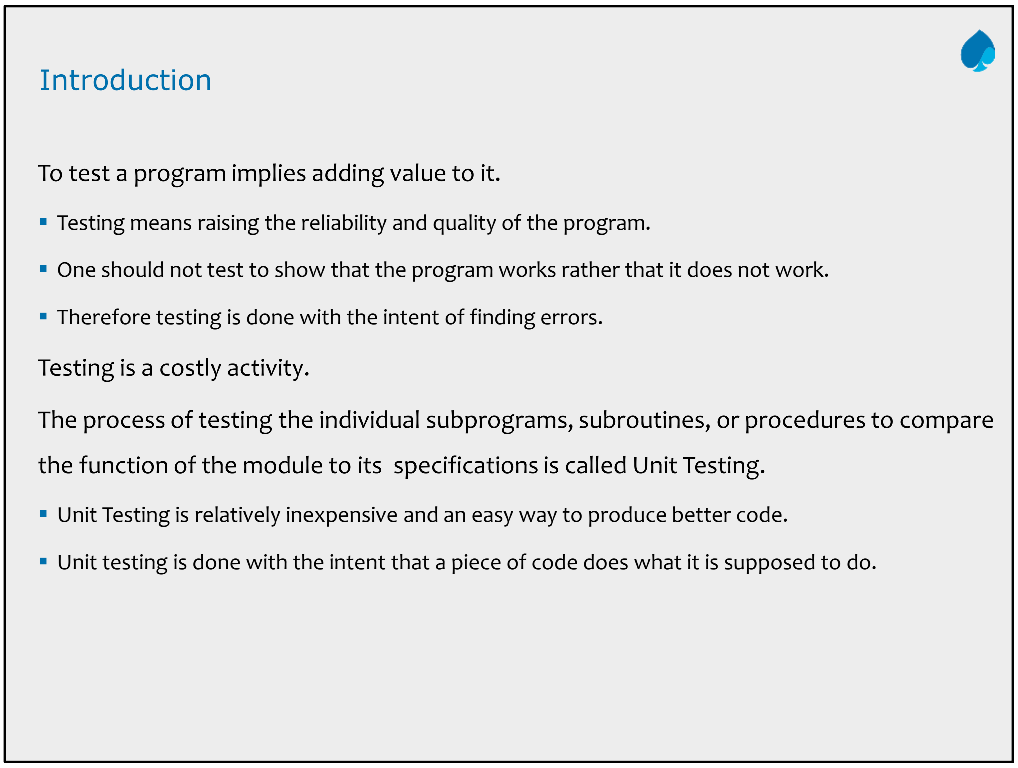
TDD with Junit 5

Capgemini

Lesson Objectives

After completing this lesson, participants will be able to

- Understand importance of Test Driver Development
- Writing Tests
- Creating dynamic and parameterized tests
- Extending Junit
- Integrating Junit
- Migrating from Junit
- TDD
- Mocking Concept



Introduction

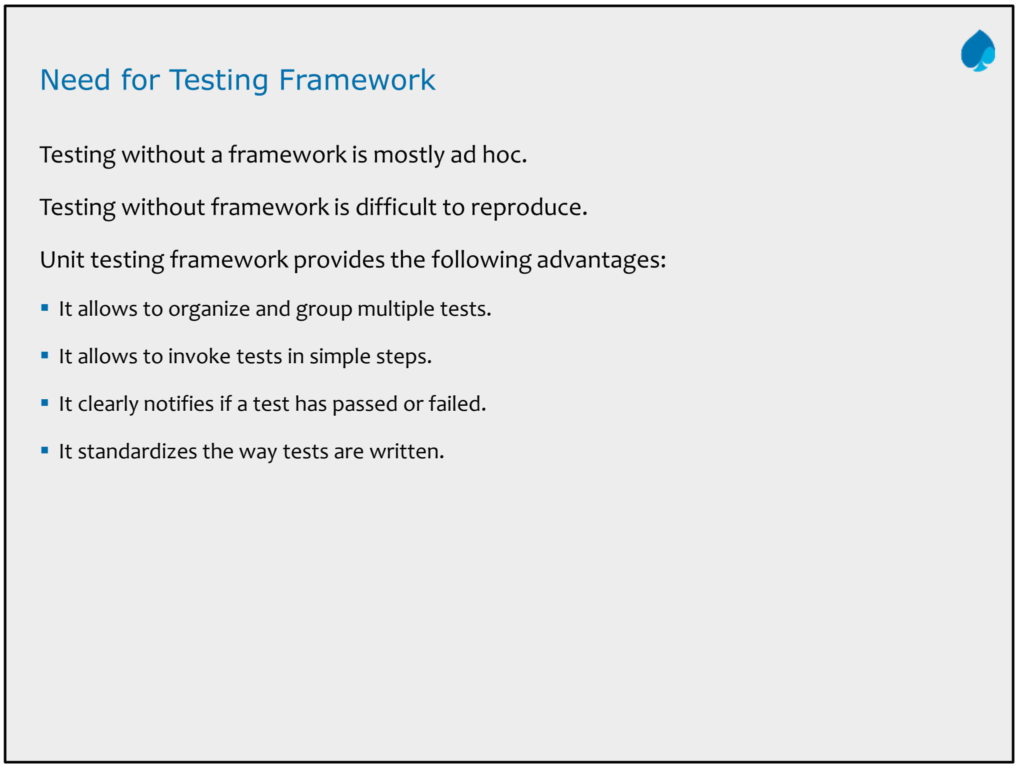
To test a program implies adding value to it.

- Testing means raising the reliability and quality of the program.
- One should not test to show that the program works rather that it does not work.
- Therefore testing is done with the intent of finding errors.

Testing is a costly activity.

The process of testing the individual subprograms, subroutines, or procedures to compare the function of the module to its specifications is called Unit Testing.

- Unit Testing is relatively inexpensive and an easy way to produce better code.
- Unit testing is done with the intent that a piece of code does what it is supposed to do.



Need for Testing Framework

Testing without a framework is mostly ad hoc.

Testing without framework is difficult to reproduce.

Unit testing framework provides the following advantages:

- It allows to organize and group multiple tests.
- It allows to invoke tests in simple steps.
- It clearly notifies if a test has passed or failed.
- It standardizes the way tests are written.

Why use JUnit?
Need for Testing Framework:
After having understood the need for Unit testing, there is a requirement to understand how to do Unit testing.
Mostly the Unit Testing done by the developers is ad hoc. The tests done in this manner are not put across in code at all. If at all they are put up, then they are written in such a manner that they cannot be reused in future. If they can be used in future, then typically they might be reproduced differently every time they are used. “Hence it is said that testing without a framework is difficult to reproduce”.
With the help of a framework, the tests get documented in code and are reproduced in the same manner, whenever required.

What is JUnit

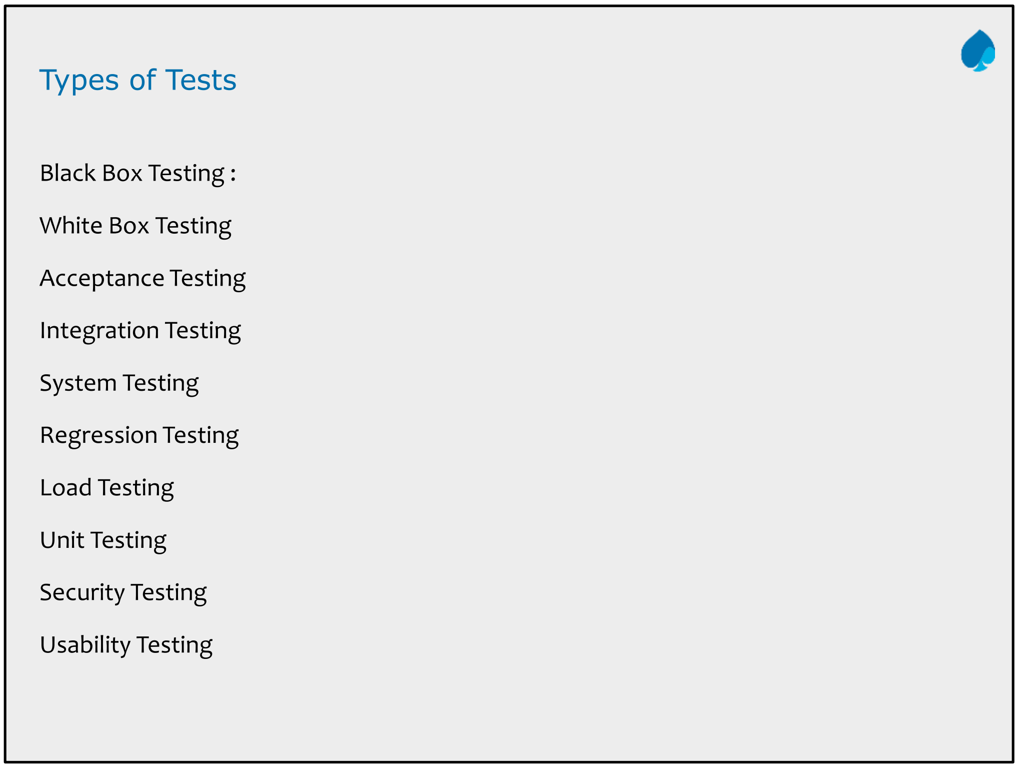
JUnit is a free, open source, software testing framework for Java.

It is a library put in a jar file.

It is not an automated testing tool.

JUnit tests are Java classes that contain one or more unit test methods.

Why use JUnit?
What is JUnit?
JUnit is an open source, software testing framework for Java developed by Kent Beck and Erich Gamma. JUnit allows developers to write Unit test cases for your Java code. It is library put in a jar file.
JUnit is not an automated testing tool. The developer has to write the test files and execute. JUnit offers some support so that the developer can easily write test files. It includes a tool which is called test runner to run your test files.
JUnit provides an easy way to state how the code should work. By expressing your intentions in code, you can use the JUnit test runners to verify that your code behaves according to your intentions.



Types of Tests

Black Box Testing :

White Box Testing

Acceptance Testing

Integration Testing

System Testing

Regression Testing

Load Testing

Unit Testing

Security Testing

Usability Testing

Black Box Testing : Testing without any knowledge on internal structure of the system or component
White Box Testing : Testing based on internal structure of the system or component
Unit Testing :They will test a piece of code by invoking it and checking the correctness of function.
A unit test can test a function or a class containing multiple methods.
A good test should be automated, fast and repeatable .

Why Unit Tests Are Important?

Reduce debugging time

Serve as documentation

Help to improve the design

Easy to run

What is JUnit

JUnit is a free, open source, software testing framework for Java.

It is a library put in a jar file.

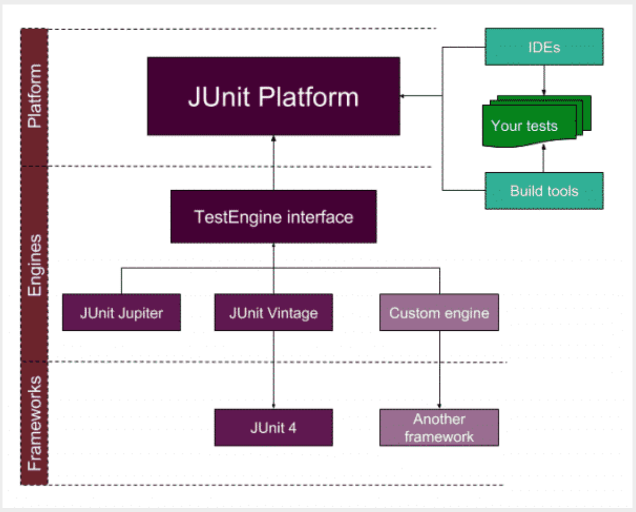
It is not an automated testing tool.

JUnit tests are Java classes that contain one or more unit test methods.

Why use JUnit?
What is JUnit?
JUnit is an open source, software testing framework for Java developed by Kent Beck and Erich Gamma. JUnit allows developers to write Unit test cases for your Java code. It is library put in a jar file.
JUnit is not an automated testing tool. The developer has to write the test files and execute. JUnit offers some support so that the developer can easily write test files. It includes a tool which is called test runner to run your test files.
JUnit provides an easy way to state how the code should work. By expressing your intentions in code, you can use the JUnit test runners to verify that your code behaves according to your intentions.

Junit 5 Architecture

Junit 5 comprises several building blocks . The architecture is illustrated below



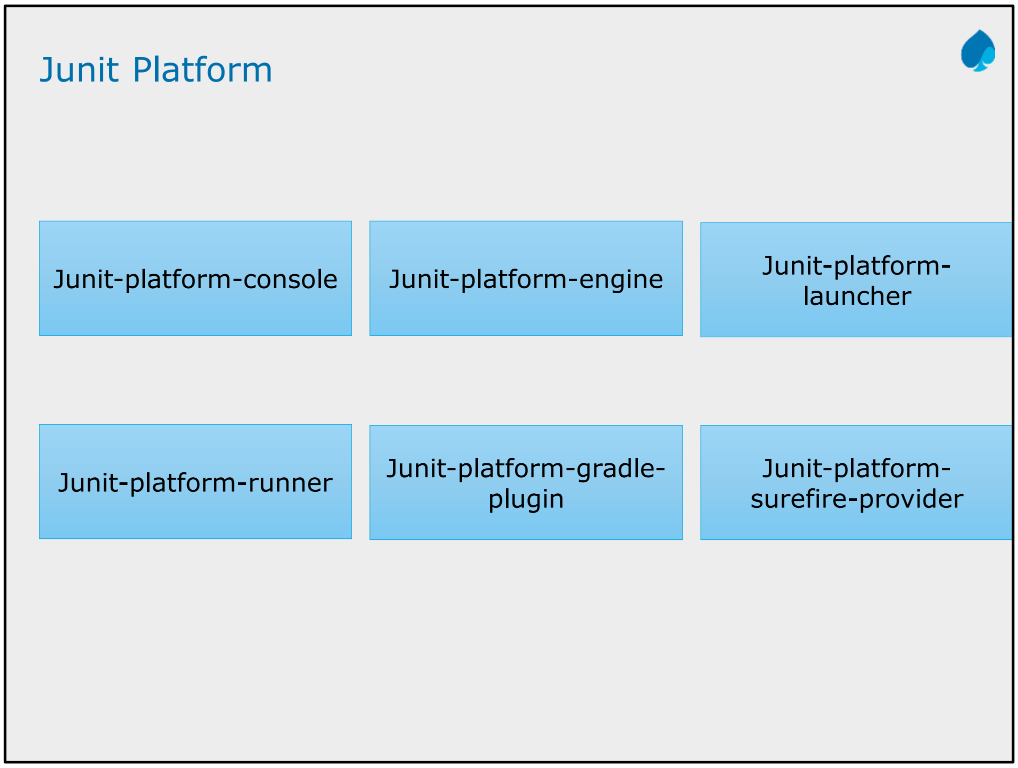
JUnit Platform
Its main responsibility is to launch the testing frameworks on the JVM. As you see in the architecture diagram it is an interface between build tools, tests, IDE and JUnit. It also defines TestEngine API to create a testing framework which operates on JUnit platform and in this way we can use external libraries in JUnit ecosystem by implementing custom engines.
JUnit Jupiter
This is the new extension and building block of JUnit. It comprises new extensions and libraries for JUnit 5. The new annotations of the JUnit 5 are summarized as follows:
@BeforeAll: Annotated method runs before all test methods in the current class.
@AfterAll: Annotated method runs after all test methods in the current class.
@BeforeEach: Annotated method runs before each test method.
@AfterEach: Annotated method runs after each test method.
@Disable: Disables a test class or a method.
@Tag: We can tag the tests methods such as Smoke, Regression, Critical, etc.
@Nested: Annotated class is a nested, non-static test class.
@DisplayName: We can declare a custom display name for a test class or a test method.
@ExtendWith: it is used to register custom extensions
@TestFactory: Declares that the method that is a test factory method for dynamic testıng.
JUnit Vintage
It is a supporting library for JUnit 5. By using **Junit Vintage** we can also run JUnit 3 and JUnit 4 based tests on the JUnit 5 platform.
JUnit 5 Maven Repositories
You can reach all JUnit maven libraries here.
JUnit Jupiter: https://mvnrepository.com/artifact/org.junit.jupiter
JUnit Platform: https://mvnrepository.com/artifact/org.junit.platform
JUnit Vintage: https://mvnrepository.com/artifact/org.junit.vintage
This is a brief overview of JUnit 5 architecture and its building blocks.

Junit 5 Building Blocks

Junit Platform : It will provide the API to launch the test either from console or IDE or build tools

Junit Jupiter : is the combination of the new programming model and extension model for writing tests and extensions in JUnit 5. The Jupiter sub-project provides a Test Engine for running Jupiter based tests on the platform.

JUnit Vintage: provides a Test Engine for running JUnit 3 and JUnit 4 based tests on the platform.



Junit Platform

Junit-platform-console

Junit-platform-engine

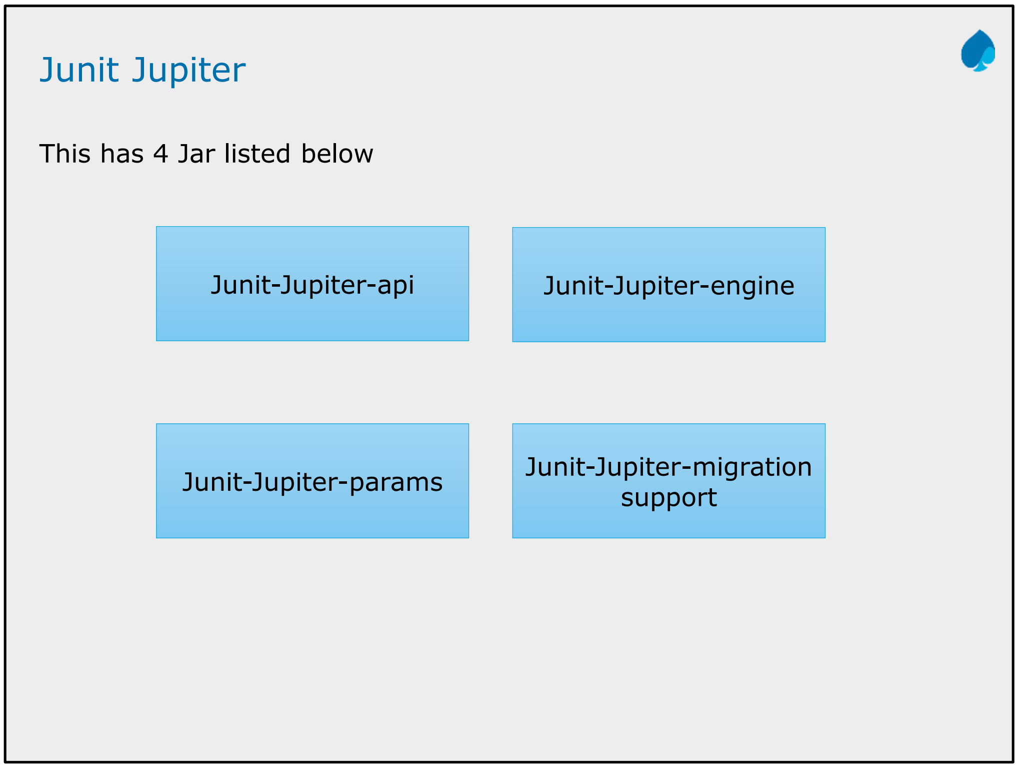
Junit-platform-launcher

Junit-platform-runner

Junit-platform-gradle-plugin

Junit-platform-surefire-provider

Junit-platform-console : Support for discovering and executing tests on the JUnit Platform from the console.
Junit-platform-engine : Public API for test engines
Junit-platform-launcher : Public API for configuring and launching test plans — typically used by IDEs and build tools.
Junit-platform-runner : Runner for executing tests and test suites on the JUnit Platform in a JUnit 4 environment.
Junit-platform-gradle-plug-in : Plug-in for discovering and executing test using gradle .
Junit-platform-surefire-provider : for discovering and executing test using maven surefire.



Junit Jupiter

This has 4 Jar listed below

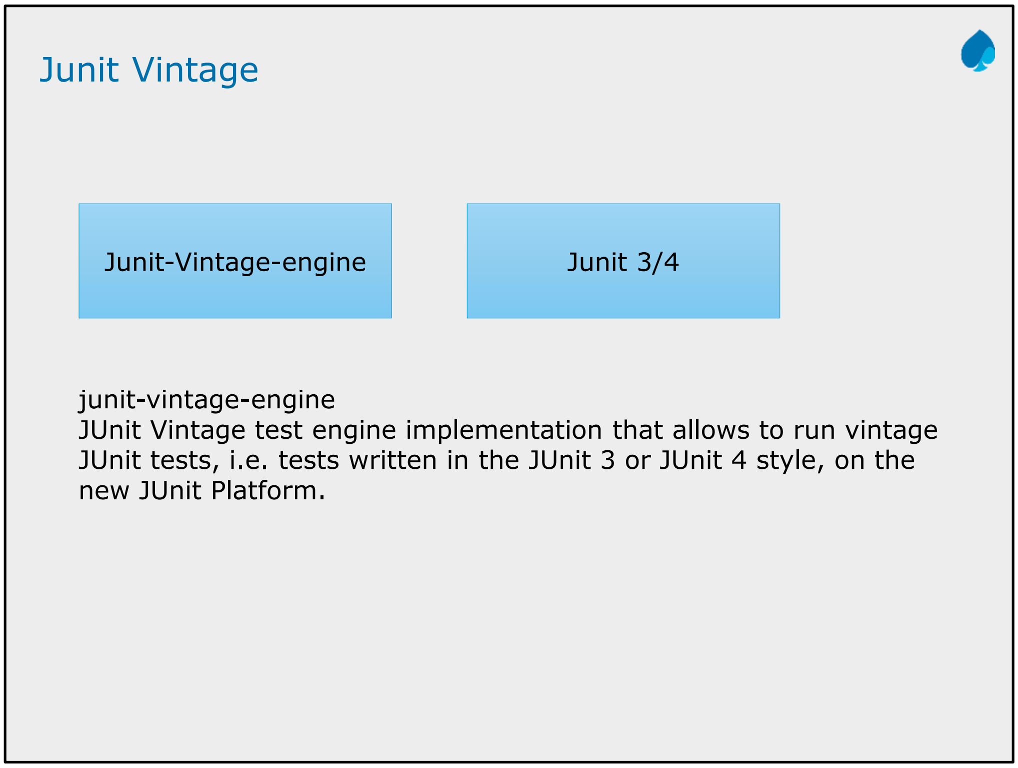
Junit-Jupiter-api

Junit-Jupiter-engine

Junit-Jupiter-params

Junit-Jupiter-migration support

Junit-Jupiter-api : For writing test and extensions
Junit-Jupiter-engine : Engine implementation
Junit-Jupiter-params : for writing parameterized test
Junit-Jupiter-migration support : it provides migration support from JUnit 4 to JUnit Jupiter, only required for running selected JUnit 4 rules.



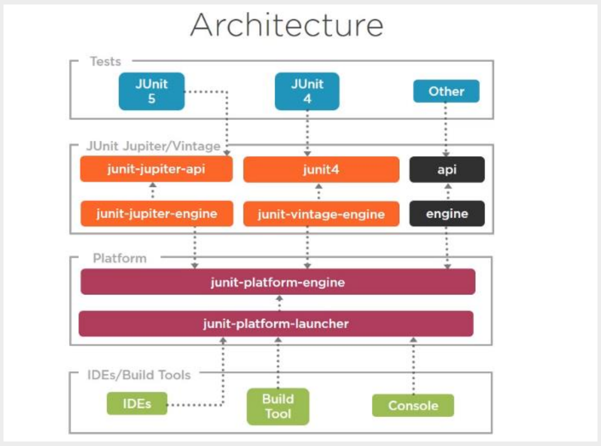
Junit Vintage

Junit-Vintage-engine

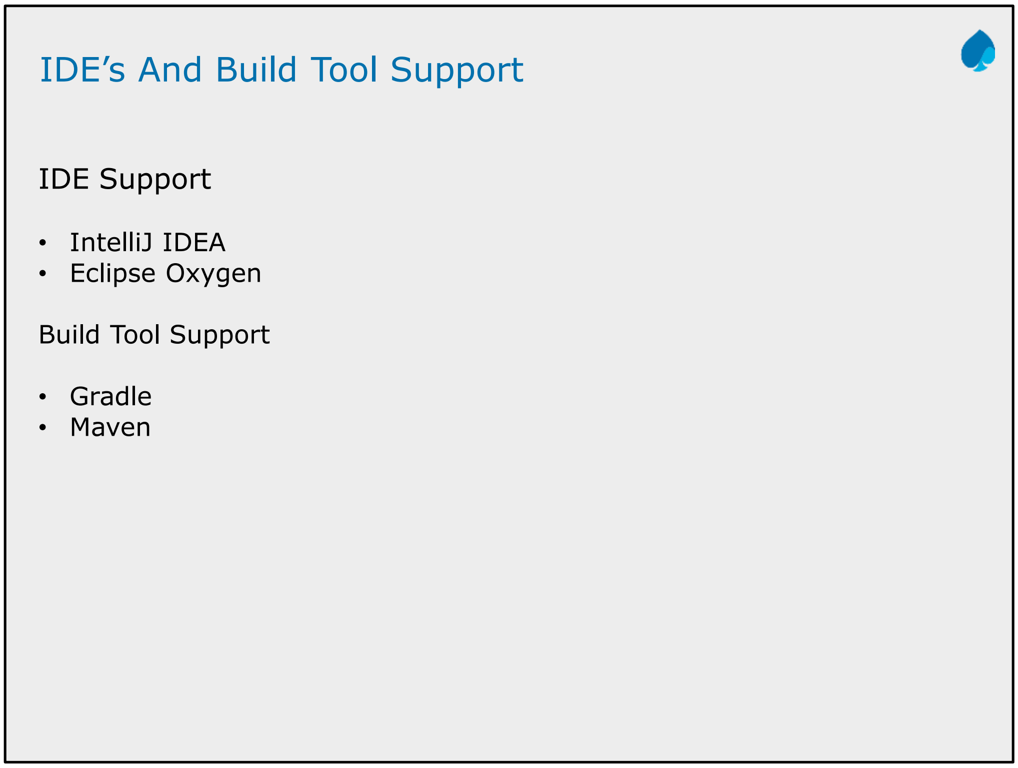
Junit 3/4

junit-vintage-engine
JUnit Vintage test engine implementation that allows to run vintage JUnit tests, i.e. tests written in the JUnit 3 or JUnit 4 style, on the new JUnit Platform.

How It works?



We have test written using junit API's and executed by junit platform which is also used by IDE's, build tools.
We have a junit test class which depends on only junit Jupiter api and does not know about engines or platforms. Junit Jupiter also includes the engine This engine depends on junit Jupiter API to run the test and is the implementation of platform engine.
ON the other hand we have IDE's , build tool and console which uses the platform launcher to discover ,build and execute the test case .This also provides API to platform engine
Following the architecture we can have junit 4 test class that depends on junit 4 api and this is another engine implementation – junit vintage engine.
Here junit platform engine does not know whether test is from junit 5 or junit 4.
In fact other test frameworks can be plugged into this architecture by implementing API's and engine implementation which talks with platform engine to run those test.



IDE’s And Build Tool Support

IDE Support

- IntelliJ IDEA
- Eclipse Oxygen

Build Tool Support

- Gradle
- Maven

Creating Test Case using Eclipse

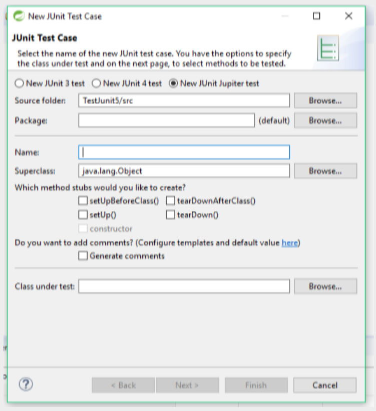
Step 1 :Create new Java project in Eclipse

Step 2 : To create Test case, right click on src ->new ->junit test case.

Step 3 : Select the radio button for

New Junit Jupiter Test option

Step 4: Enter Name for Test case class.



New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

New JUnit 3 test New JUnit 4 test [✓ New JUnit Jupiter test]

Source folder: TestJunit5/src

Package: (default)

Name:

Superclass: java.lang.Object

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()

☐ setUp() ☐ tearDown()

☐ constructor

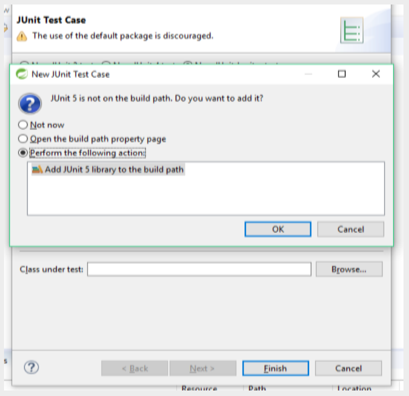
Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

Class under test:

Creating Test Case in Eclipse

Step 5: After clicking on Finish button it will ask you to add junit 5 to build path if you are creating test case first time


JUnit Test Case

The use of the default package is discouraged.

New JUnit Test Case

JUnit 5 is not on the build path. Do you want to add it?

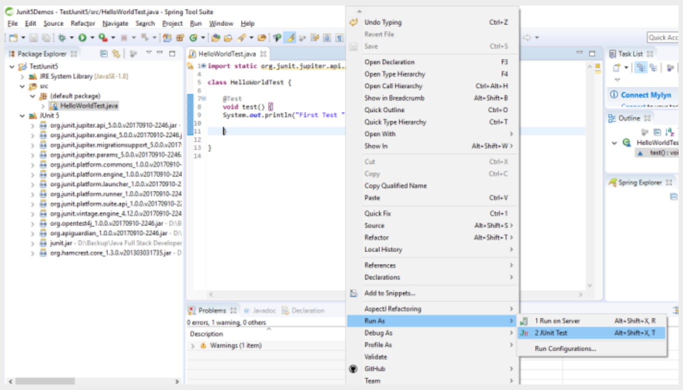
Not now [ ]
Open the build path property page [ ]
[✓ Perform the following action:]

Add JUnit 5 library to the build path

Class under test:

Creating Test Case in Eclipse

To run the Test case
Right click on Test Case class -> run as -> junit Test


Junit5Demos - TestJunit5/src/HelloWorldTest.java - Spring Tool Suite
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
TestJunit5
JRE System Library [JavaSE-1.8]
src
(default package)
HelloWorldTest.java
JUnit 5
org.junit.jupiter.api_5.0.0.v20170910-2246.jar
org.junit.jupiter.engine_5.0.0.v20170910-2246.j
org.junit.jupiter.migrationsupport_5.0.0.v2017
org.junit.jupiter.params_5.0.0.v20170910-2246.
org.junit.platform.commons_1.0.0.v20170910-
org.junit.platform.engine_1.0.0.v20170910-224
org.junit.platform.launcher_1.0.0.v20170910-2
org.junit.platform.runner_1.0.0.v20170910-224
org.junit.platform.suite.api_1.0.0.v20170910-2
org.junit.vintage.engine_4.12.0.v20170910-224
org.opentest4j_1.0.0.v20170910-2246.jar - D:\B
org.apiguardian_1.0.0.v20170910-2246.jar - D:\
junit.jar - D:\Backup\Java Full Stack Developer
org.hamcrest.core_1.3.0.v201303031735.jar - D
HelloWorldTest.java
Undo Typing Ctrl+Z
Revert File
Save Ctrl+S
Open Declaration F3
Open Type Hierarchy F4
Open Call Hierarchy Ctrl+Alt+H
Show in Breadcrumb Alt+Shift+B
Quick Outline Ctrl+O
Quick Type Hierarchy Ctrl+T
Open With
Show In Alt+Shift+W
Cut Ctrl+X
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Quick Fix Ctrl+1
Source Alt+Shift+S
Refactor Alt+Shift+T
Local History
References
Declarations
Add to Snippets...
AspectJ Refactoring
Run As
Debug As
Profile As
Validate
GitHub
Team
1 Run on Server Alt+Shift+X, R
2 JUnit Test Alt+Shift+X, T
Run Configurations...
Task List
Connect Mylyn
Outline
HelloWorldTes
test() : voi
Spring Explorer
Quick Acc
Problems
Javadoc
Declaration
Description
Warnings (1 item)

```
import static org.junit.jupiter.api..

class HelloWorldTest {

    @Test
    void test() {
    System.out.println("First Test "

}
```

0 errors, 1 warning, 0 others

Setting up Junit with Maven

Step 1: Create Maven Project
Step 2 : Add below dependency in pom.xml file

```
        <dependency>
                <groupId>org.junit.jupiter</groupId>
                <artifactId>junit-jupiter-api</artifactId>
                <version>5.0.1</version>
                 <scope>test</scope>
    </dependency>
    <dependency>
                <groupId>org.junit.jupiter</groupId>
                <artifactId>junit-jupiter-engine</artifactId>
                 <version>5.0.1</version>
                 <scope>test</scope>
    </dependency>
```

Setting up Junit with Maven

Step 3 : Below dependency is required only to run test in an IDE that bundles old version

```
    <dependency>
<groupId>org.junit.platform</groupId>
<artifactId>junit-platform-launcher</artifactId>
<version>1.0.1</version>
<scope>test</scope>
</dependency>
```

Step 4 :Create the junit Test case in src/test/java folder

Step 5 : Run the project using Maven Test Goal.

Demo

Refer Lesson 15 Junit5 folder for demos

Life Cycle Methods

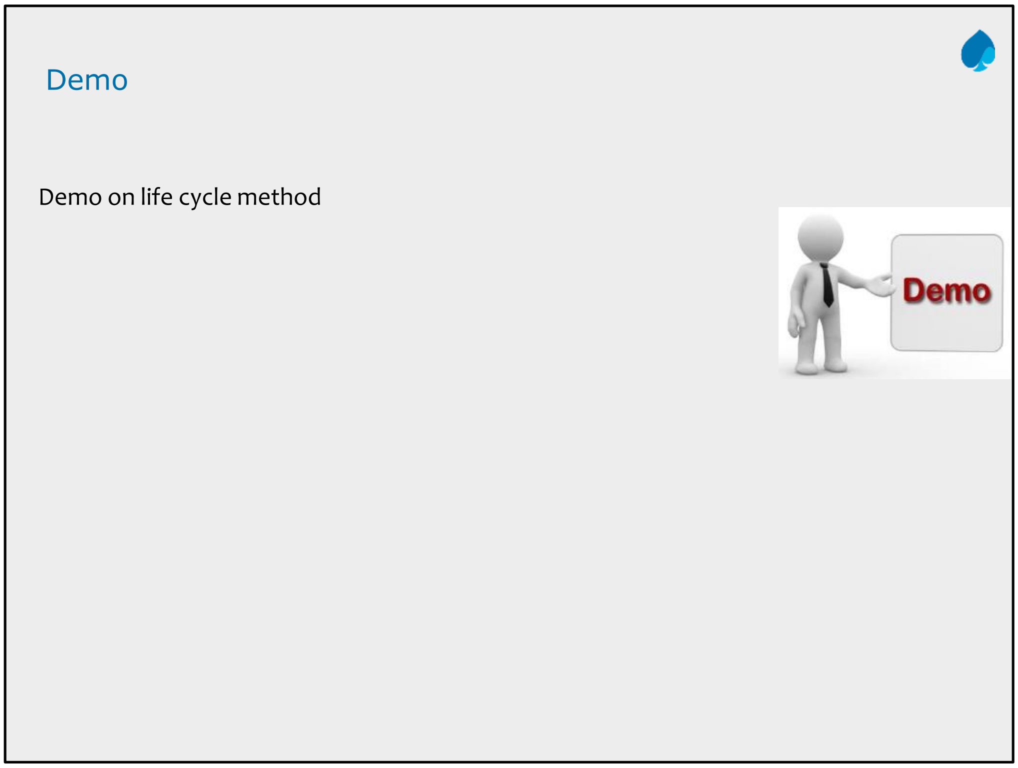
In JUnit 5, test lifecycle is driven by 4 primary annotations i.e. @BeforeAll, @BeforeEach, @AfterEach and @AfterAll. Along with it, each test method must be marked with @Test annotation. @Test annotation is virtually unchanged, although it no longer takes optional arguments.

@BeforeAll : It is used to signal that the annotated method should be executed before all tests in the current test class.

@BeforeEach : It is used to signal that the annotated method should be executed before each test in the current test class.

@AfterAll : It is used to signal that the annotated method should be executed after all tests in the current test class.

@AfterEach: It is used to signal that the annotated method should be executed after each test in the current test class.



Demo

Demo on life cycle method

Assertions

JUnit Jupiter comes with many of the assertion methods that JUnit 4 has and adds a few that lend themselves well to being used with Java 8 lambdas. All JUnit Jupiter assertions are static methods in the org.junit.jupiter.api.Assertions class.

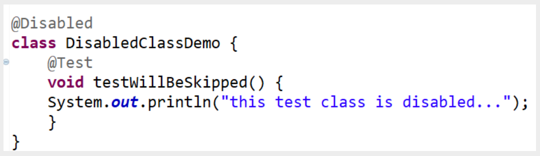
AssertAll
AssertNotSame
AssertArrayEquals
assertNull
AssertFalse
AssertThrow
assertIterableEquals
AssertNotNull

assertTrue
fail
assertTimeout
assertLinesMatch

Disabling Test

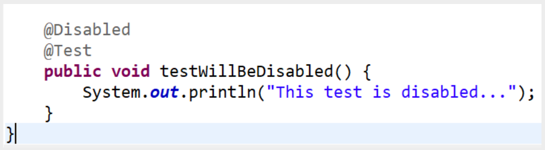
Entire test classes or individual test methods may be disabled via the @Disabled annotation, via one of the annotations discussed in Conditional Test Execution, or via a custom ExecutionCondition.

Disabled Class



```
@Disabled
class DisabledClassDemo {
    @Test
    void testWillBeSkipped() {
    System.out.println("this test class is disabled...");
    }
}
```

Disabled Test Method



```
    @Disabled
    @Test
    public void testWillBeDisabled() {
        System.out.println("This test is disabled...");
    }
}
```

Assumptions

JUnit Jupiter comes with a subset of the assumption methods that JUnit 4 provides and adds a few that lend themselves well to being used with Java 8 lambdas. All JUnit Jupiter assumptions are static methods in the org.junit.jupiter.api.Assumptions class.

assumeTrue(Boolean assumption) : will evaluate the given assumption and if it results in true then the given test is allowed to execute

assumeFalse(Boolean assumption) : Will evaluate the given assumption and test will run if result is false.

assumeThat (Boolean assumption, Executable executable)

It will evaluate the lambda executable only if given assumption is true .

AssumeFalse(Boolean assumption)
AssumeFalse(Boolean assumption, String message)
AssumeFalse(BooleanSupplier assumptionSupplier)

assumeTrue(Boolean assumption)

assumeTrue(Boolean assumption,String message)

AssumeTrue(BooleanSupplier assumptionSupplier)

Demo

Demo on

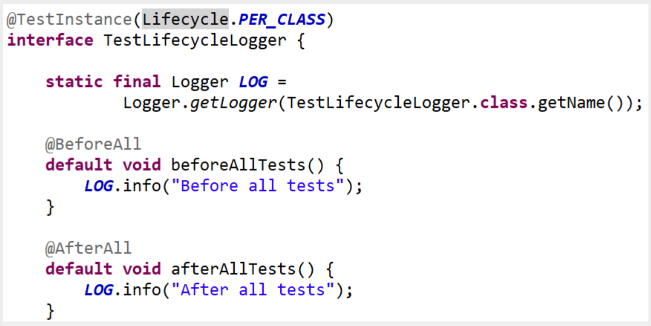
Assertions

Disabling Tests

Assumptions

Test interfaces and Default Methods

JUnit Jupiter allows @Test, @RepeatedTest, @ParameterizedTest, @TestFactory, @TestTemplate, @BeforeEach, and @AfterEach to be declared on interface default methods. @BeforeAll and @AfterAll can either be declared on static methods in a test interface or on interface default methods if the test interface or test class is annotated with @TestInstance(Lifecycle.PER_CLASS)



```
@TestInstance(Lifecycle.PER_CLASS)
interface TestLifecycleLogger {

    static final Logger LOG =
            Logger.getLogger(TestLifecycleLogger.class.getName());

    @BeforeAll
    default void beforeAllTests() {
        LOG.info("Before all tests");
    }

    @AfterAll
    default void afterAllTests() {
        LOG.info("After all tests");
    }
```

19.4: Testing with JUnit

Demo

Demo on

Test Interface and Default Methods

Repeating Tests

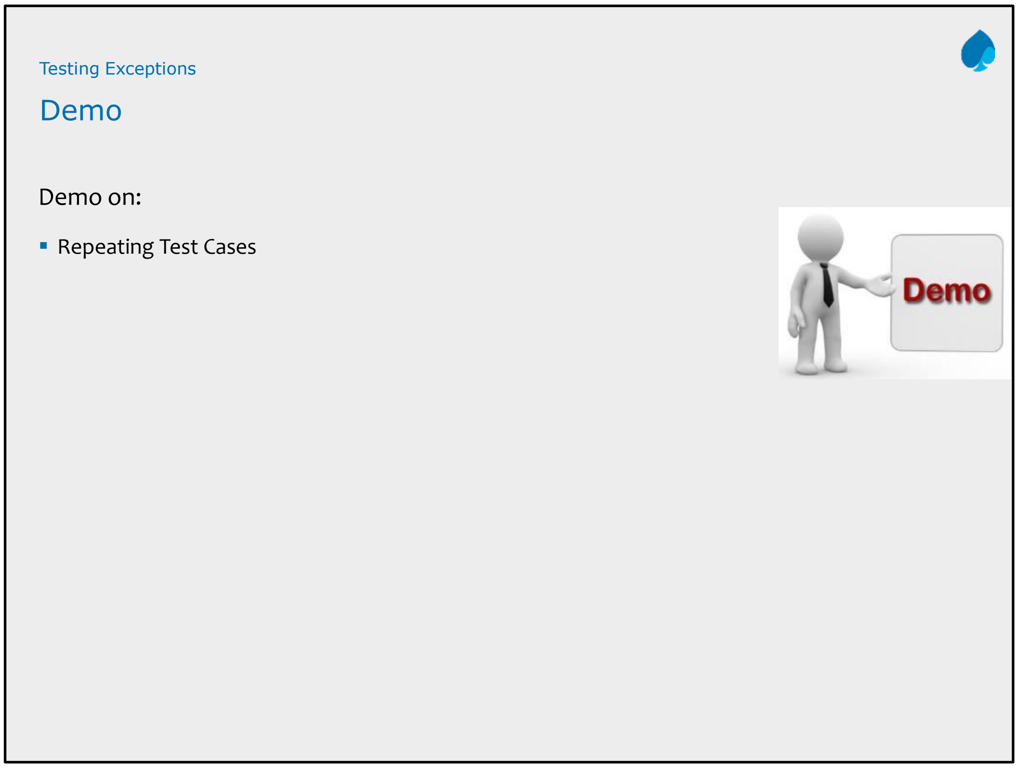
JUnit Jupiter provides the ability to repeat a test a specified number of times simply by annotating a method with @RepeatedTest and specifying the total number of repetitions desired. Each invocation of a repeated test behaves like the execution of a regular @Test method with full support for the same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

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

```
@RepeatedTest(10)
void repeatedTest() {
   // ...
}
```

In addition to specifying the number of repetitions, a custom display name can be configured for each repetition via the name attribute of
the @RepeatedTest annotation. Furthermore, the display name can be a pattern composed of a combination of static text and dynamic placeholders. The following placeholders are currently supported.
{displayName}: display name of the @RepeatedTest method
{currentRepetition}: the current repetition count
{totalRepetitions}: the total number of repetitions

The default display name for a given repetition is generated based on the following pattern: "repetition {currentRepetition} of {totalRepetitions}". Thus, the display names for individual repetitions of the previous repeatedTest() example would be: repetition 1 of 10, repetition 2 of 10, etc. If you would like the display name of the @RepeatedTest method included in the name of each repetition, you can define your own custom pattern or use the
predefined RepeatedTest.LONG_DISPLAY_NAME pattern. The latter is equal to "{displayName} :: repetition {currentRepetition} of {totalRepetitions}" which results in display names for individual repetitions likerepeatedTest() :: repetition 1 of 10, repeatedTest() :: repetition 2 of 10, etc.



Testing Exceptions

Demo

Demo on:

- Repeating Test Cases

.

Dynamic Tests

The methods annotated using @Test are static in the sense that they are fully specified at compile time ,and their behavior cannot be changed by anything happening runtime.

In addition to these standard tests a completely new kind of test programming model has been introduced in JUnit Jupiter. This new kind of test is a dynamic test which is generated at runtime by a factory method that is annotated with @TestFactory.

In contrast to @Test methods, a @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.

Restriction :

These methods should not be private or static in nature .

Dynamic Tests

@TestFactory method must return a Stream, Collection, Iterable, Iterator, or array of DynamicNode subclass instances.

A DynamicTest is a test case generated at runtime. It is composed of a display name and an Executable. Executable is a @FunctionalInterface which means that the implementations of dynamic tests can be provided as lambda expressions or method references.

@BeforeEach And @AfterEach methods will not be executed for every dynamic test.

Dynamic Test Lifecycle
The execution lifecycle of a dynamic test is quite different than it is for a standard @Test case. Specifically, there are no lifecycle callbacks for individual dynamic tests. This means that @BeforeEach and @AfterEach methods and their corresponding extension callbacks are executed for the @TestFactorymethod but not for each *dynamic test*. In other words, if you access fields from the test instance within a lambda expression for a dynamic test, those fields will not be reset by callback methods or extensions between the execution of individual dynamic tests generated by the same @TestFactory method.

Test Fixtures

Demo

Demo on:

- Dynamic Test

Parameterized Tests

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.

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 saw elba" })
void palindromes(String candidate) {
   assertTrue(isPalindrome(candidate));
}
```

Demo

Demo on:

- Parameterised Tests

Note:
TestPersonSuite.java
In this demo example, the three classes, namely TestPerson, TestPerson2, and TestPersonFixture, have been put together for execution. The class itself does not have any methods for testing.

Argument Sources

In parameterized Test arguments are provided by sources that you can use via annotations.

Source Rules :

1.For every test method you can use as many arguments as you want but there should be at least one source argument.

2.Each source must provide the value for all parameters.

3.Test will be executed for each group of arguments.

Argument Source Annotations

@ValueSource: It allows you to define Array of Types String,int ,long,double

This can be used for only those methods with single method parameters.

@EnumSource: To run the test with parameter of enum type. IT can be used on single parameter methods.

@MethodSource: Allows you to refer to one or more factory methods of the test class or external classes.

@CsvSource: allows you to express argument lists as comma-separated values (i.e., String literals).

@CsvFileSource : allows you to use CSV files from the classpath. Each line from a CSV file results in one invocation of the parameterized test.

@ArgumentsSource
@ArgumentsSource can be used to specify a custom, reusable ArgumentsProvider. Note that an implementation of ArgumentsProvider must be declared as either a top-level class or as a static nested class.

Demo

Demo on source Annotation

Argument Conversion

Widening Conversion

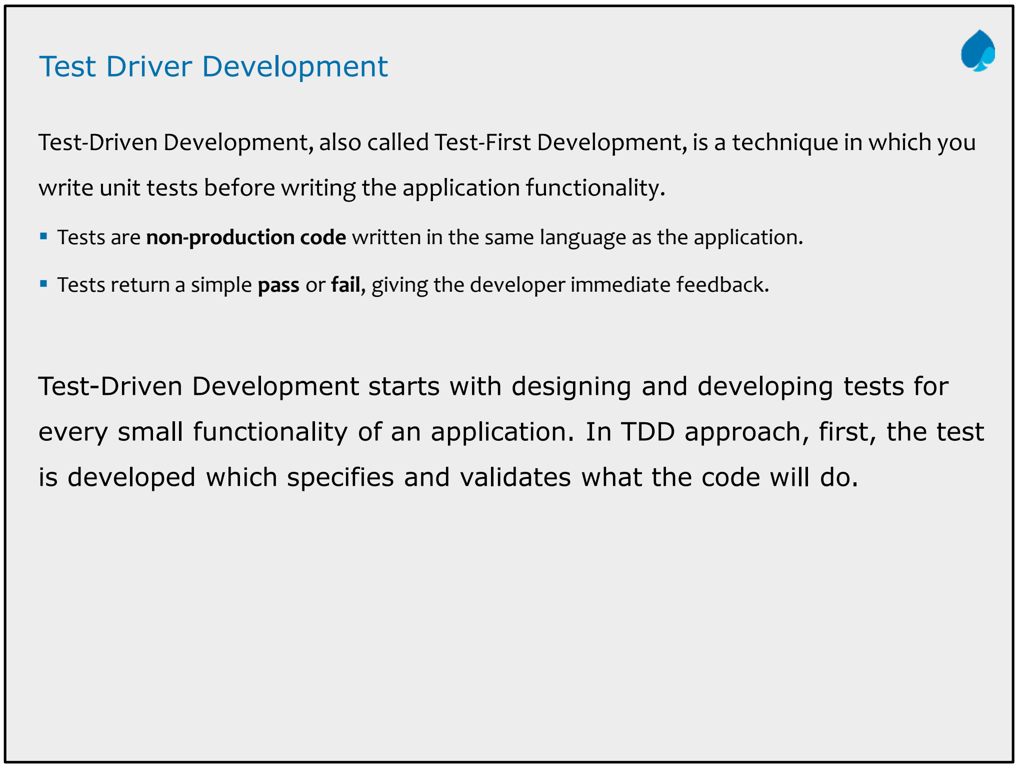
JUnit Jupiter supports Widening Primitive Conversion for arguments supplied to a @ParameterizedTest. For example, a parameterized test annotated with @ValueSource(ints = { 1, 2, 3 }) can be declared to accept not only an argument of type int but also an argument of type long, float, or double.

Implicit Conversion

To support use cases like @CsvSource, JUnit Jupiter provides a number of built-in implicit type converters. The conversion process depends on the declared type of each method parameter.

For example, if a @ParameterizedTest declares a parameter of type TimeUnit and the actual type supplied by the declared source is a String, the string will be automatically converted into the corresponding TimeUnit enum constant.

Fallback String-to-Object Conversion
In addition to implicit conversion from strings to the target types listed in the above table, JUnit Jupiter also provides a fallback mechanism for automatic conversion from a String to a given target type if the target type declares exactly one suitable *factory method* or a *factory constructor* as defined below.
factory method: a non-private, static method declared in the target type that accepts a single String argument and returns an instance of the target type. The name of the method can be arbitrary and need not follow any particular convention.
factory constructor: a non-private constructor in the target type that accepts a single String argument. Note that the target type must be declared as either a top-level class or as a static nested class.



Test Driver Development

Test-Driven Development, also called Test-First Development, is a technique in which you write unit tests before writing the application functionality.

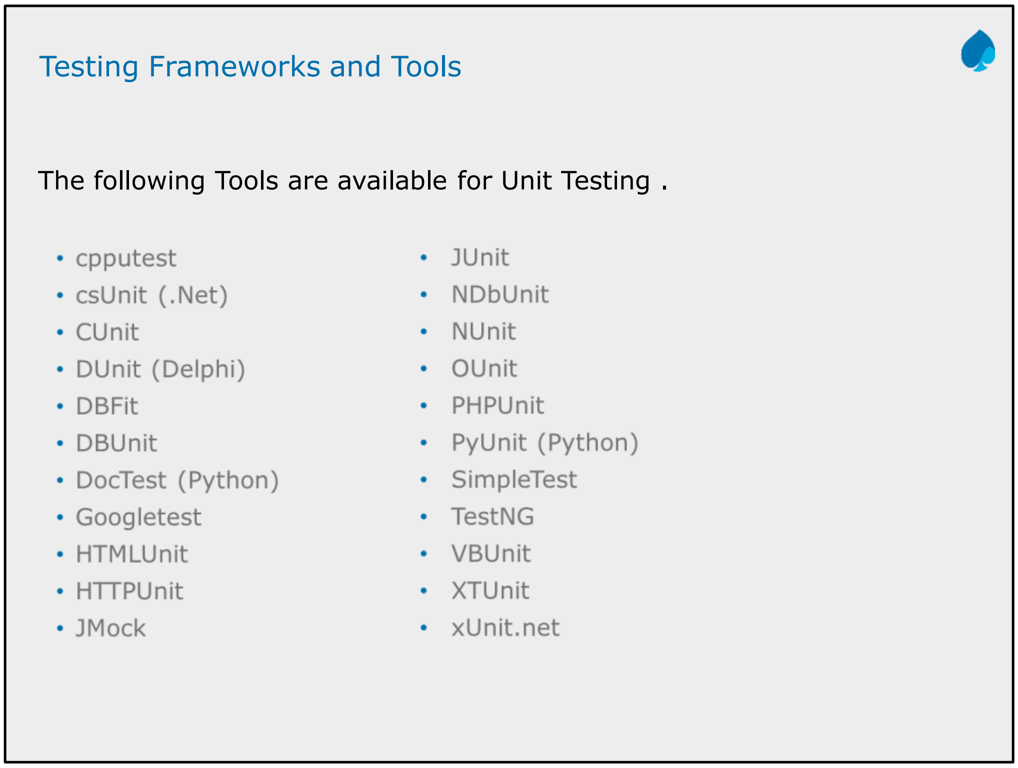
- Tests are **non-production code** written in the same language as the application.
- Tests return a simple **pass** or **fail**, giving the developer immediate feedback.

Test-Driven Development starts with designing and developing tests for every small functionality of an application. In TDD approach, first, the test is developed which specifies and validates what the code will do.

Why TDD

A significant advantage of TDD is that it enables you to take small steps when writing software.

- TDD is done at Unit level - i.e. testing the internals of a class
- Tests are written for every function
- Mostly written by developers using one of the tool specific to the application



Testing Frameworks and Tools

The following Tools are available for Unit Testing .

- cpputest
- csUnit (.Net)
- CUnit
- DUnit (Delphi)
- DBFit
- DBUnit
- DocTest (Python)
- Googletest
- HTMLUnit
- HTTPUnit
- JMock

- JUnit
- NDbUnit
- NUnit
- OUnit
- PHPUnit
- PyUnit (Python)
- SimpleTest
- TestNG
- VBUnit
- XTUnit
- xUnit.net

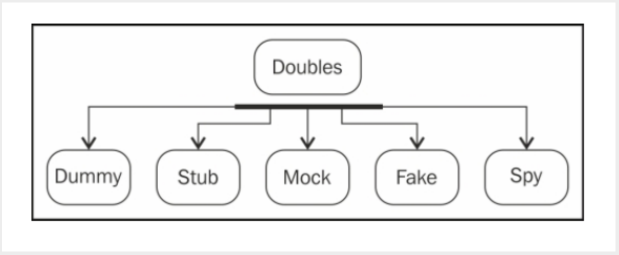
Mocking Concept

Unit Testing of any method should be ideally done in isolation from other methods.

The goal of unit testing is to test each method or path in isolation.

Complications can arise when a method depends on other classes or even worse, external resources.

This is where Mockito comes into play. It will allow you to completely mock a class or interface either inline or with Spring DI.



In software development there is an opportunity of ensuring that objects perform the behaviors that are expected of them.

One approach is to create a test automation framework that actually exercises each of those behaviors and verifies that it performs as expected,
even after it is changed.

However, the requirement to create an entire testing framework is often an onerous task that requires as much effort as writing the original objects that were supposed to be tested.

For that reason, developers have created mock testing frameworks. These effectively fake some external dependencies so that the object being tested has a consistent interaction with its outside dependencies.

Mockito intends to streamline the delivery of these external dependencies that are not subjects of the test

This lesson covers the concept of test doubles and explains various test double types, such as mock, fake, dummy, stub, and spy.
Sometimes, it is not possible to unit test a piece of code because of unavailability of collaborator objects or the cost of instantiation for the collaborator.
Test doubles alleviate the need for a collaborator.
We know about stunt doubles—a trained replacement used for dangerous action sequences in movies, such as jumping out of the
Empire State building, a fight sequence on top of a burning train, jumping from an airplane,
or similar actions. Stunt doubles are used to protect the real actors or chip in when the actor is not available.

While testing a class that communicates with an API, you don't want to hit the API for every single test; for example, when a piece of code is dependent on database access, it is not possible to unit test the code unless the database is accessible.
Similarly, while testing a class that communicates with a payment gateway, you can't submit payments to a real payment gateway to run tests.

Test doubles act as stunt doubles. They are skilled replacements for collaborator objects.
Gerard Meszaros coined the term test doubles and explained test doubles in his book *xUnit Test Patterns*, *Pearson Education*.

Test doubles are categorized into five types. The above diagram shows these types:

Mockito Concepts

Methods which are under test has often dependency.

Testing can become a challenge because if multiple developer testing simultaneously. There can be conflicts.

Incomplete dependency Implementation.

Mocking framework allows you to replace the dependency and implementation classes with mock implementation during test execution.

Mocking framework avoids dependent class object creation.

It leverage the proxy pattern.

- Methods which are under test has often has dependency.
  -Ex Service layer depends of Dao using JPA API. So testing with dependencies is a big challenge because of live database we may require.

- Multiple developer Testing simultaneously
  Testing can become a challenge because multiple developer testing simultaneously. There can be conflicts
  -There might me a challenge to multiple developer testing data access object independently.
  -But we may not want conflicts to happen while testing service since service is dependent on DAO.

- Another problem is Incomplete dependency implementation.
  -Here dependent component is not yet developed-We may have a contract of interface defined but not the implantation developed.

Benefits Of Mockito

No Handwriting – No need to write mock objects on your own.

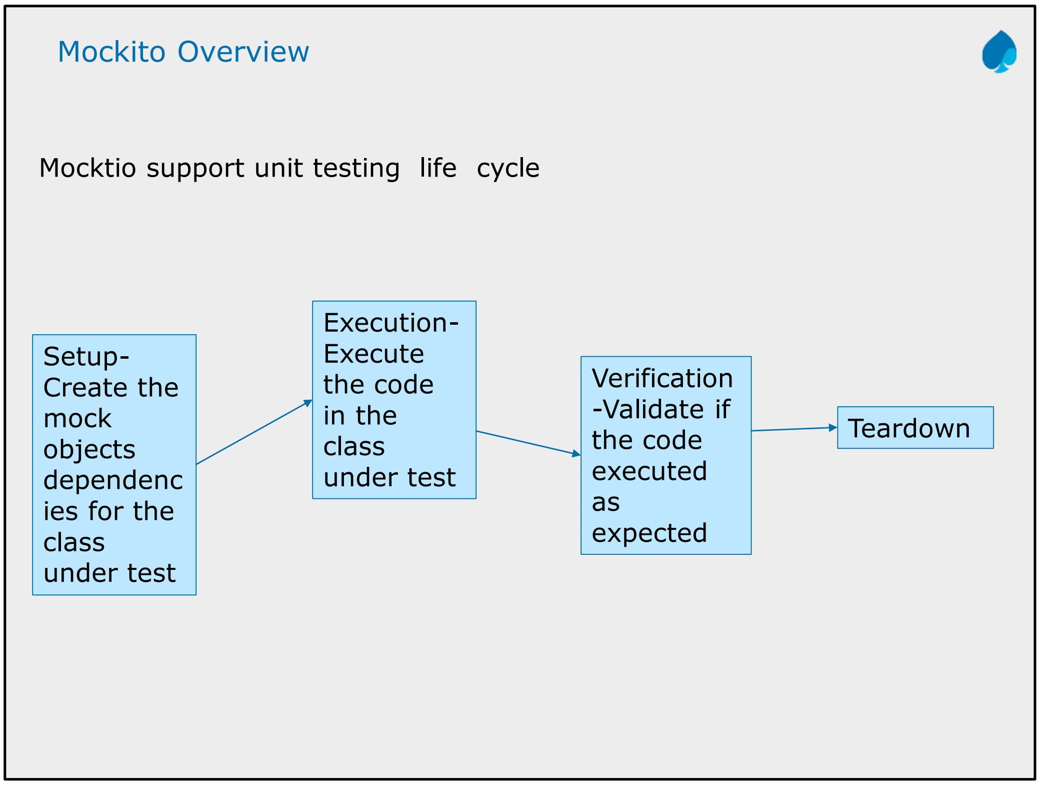
Refactoring Safe – Renaming interface method names or reordering parameters will not break the test code as Mocks are created at runtime.

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.



Mockito Overview

Mocktio support unit testing life cycle

Setup- Create the mock objects dependencies for the class under test
Execution- Execute the code in the class under test
Verification -Validate if the code executed as expected
Teardown

- **Setup**- In this phase we ask the framework to create the dependency using mock objects.
- **Execution** – During execution mocks go in to response to a method under test.
- **Verification**-provide capability to ensure that mock behave in the manner you intended.

Creating Mock Objects With Mockito

Mockito provides several methods to create mock objects:

Using the static mock() method.
Using the @Mock annotation.

SetUp- Creating The Mock

ICalculator mockCalcDao= Mockito.*mock(ICalculatorDao.**class);***

SetUp- Method Stubbing

Mockito.*when(mockCalcDao.add(7, 3)).thenReturn(10);*

Verification

Mockito.*verify(mockCalcDao).add(7, 3);*

SetUp- Creating The Mock -

SetUp- Method Stubbing- It follows when then pattern . It specify how the operation behave when it is called with specific set of values.
We don't do anything special in the execution phase.

Verification –You user Mockito verify method to assert that particular method was called with a matched set of inputs.

Mockito Functions

•The when then pattern

-Great! now we have successfully created and injected the mock, and now we should tell the mock how to behave when certain methods are called on it.

-when is a static method of the Mockito class and it returns an OngoingStubbing<T> (T is the return type of the method that we are mocking, in this case it is integer)

Ex-

Mockito.*when(mockCalcDao.add(7, 3)).thenReturn(10);*

Mockito.*when(mockCalcDao.add(7, 3)).thenReturn(10);*

-the above line of code tells the Mockito framework that we want the add() method of the mock dao instance to return 10 when 7 and 3 is passed as parameter

Mockito Functions

Following are some of the methods that we can call on the stub

thenReturn(returnValue)-

thenThrow(exception)-

Mockito.*when(mockCalcDao.getIntListFromDao(-1)).*
thenThrow(**new NegativeNumberException());**

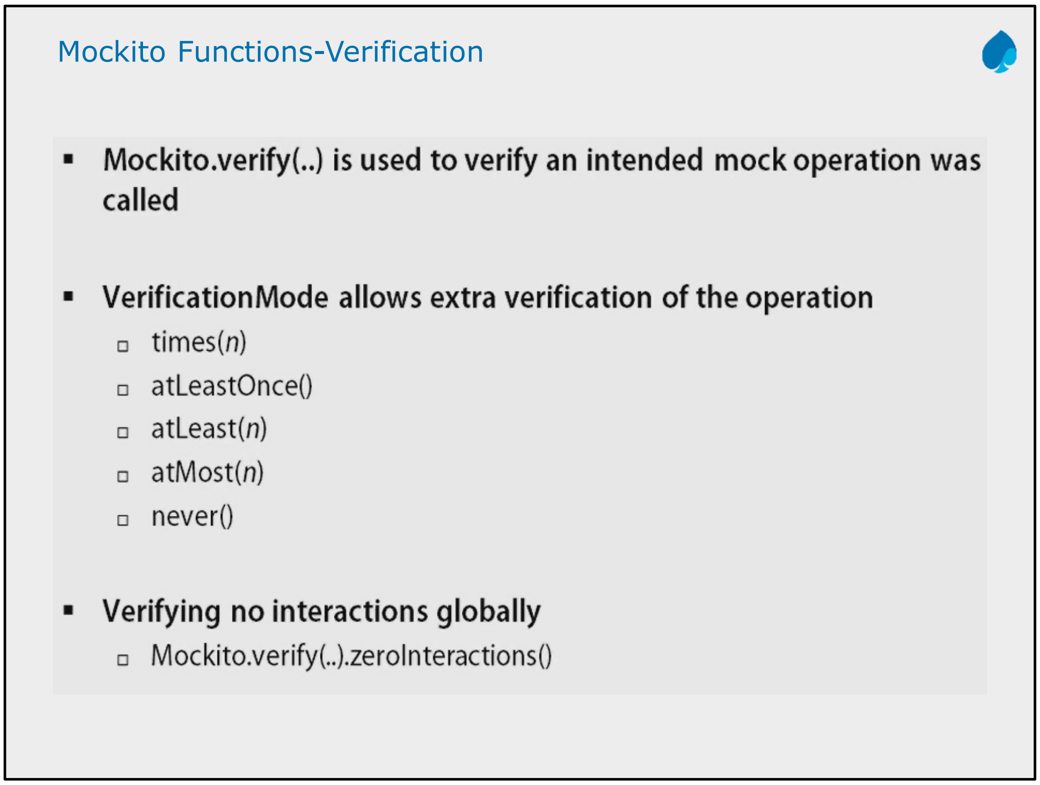
thenCallRealMethod()-

thenAnswer() - this could be used to set up smarter stubs and also mock behavior of void methods as well .

- **thenReturn(returnValue)-** Specify object or value to return when method is called
- **thenThrow(exception)-** Specify mock invocation should result in exception thrown.
- **doThrow(..)** -If we need to throws exception when a method whose return type is void is called, then we can use the alternate way of throwing exception , i.e. doThrow(..) of class org.mockito.Mockito

- **thenCallRealMethod()-** When we are mocking a class then delegate call to underlying instance with thenCallRealMethod()
- **thenAnswer() –** answering allows you to provide means to conditionally respond based on mock operation parameter.

- Methods with return values can be tested by asserting the returned value, but how to test void methods? The void method that you want to test could either be calling other methods to get things done or processing the input parameters or maybe generating some values or all of it.
- With Mockito, you can test all of the above scenarios.



Mockito Functions-Verification

- Mockito.verify(..) is used to verify an intended mock operation was called
- VerificationMode allows extra verification of the operation
  - times(*n*)
  - atLeastOnce()
  - atLeast(*n*)
  - atMost(*n*)
  - never()
- Verifying no interactions globally
  - Mockito.verify(..).zeroInteractions()

Stubbing Method Calls

Using stubbing we train the mock objects about what values to return when its methods are invoked. Mockito provides when–then stubbing pattern to stub a mock object's method invocation.

The mock API invocation goes into when() which is a static Mockito API method and the value that we want the want the mock to return goes into the then() API.

Mockito is an open source mock unit testing framework for Java. In this article, we will look into some of the stubbing examples.

Demo

Demo on Mockito and Stubbing Method Calls

Lab : Introduction to Junit

Introduction to Junit

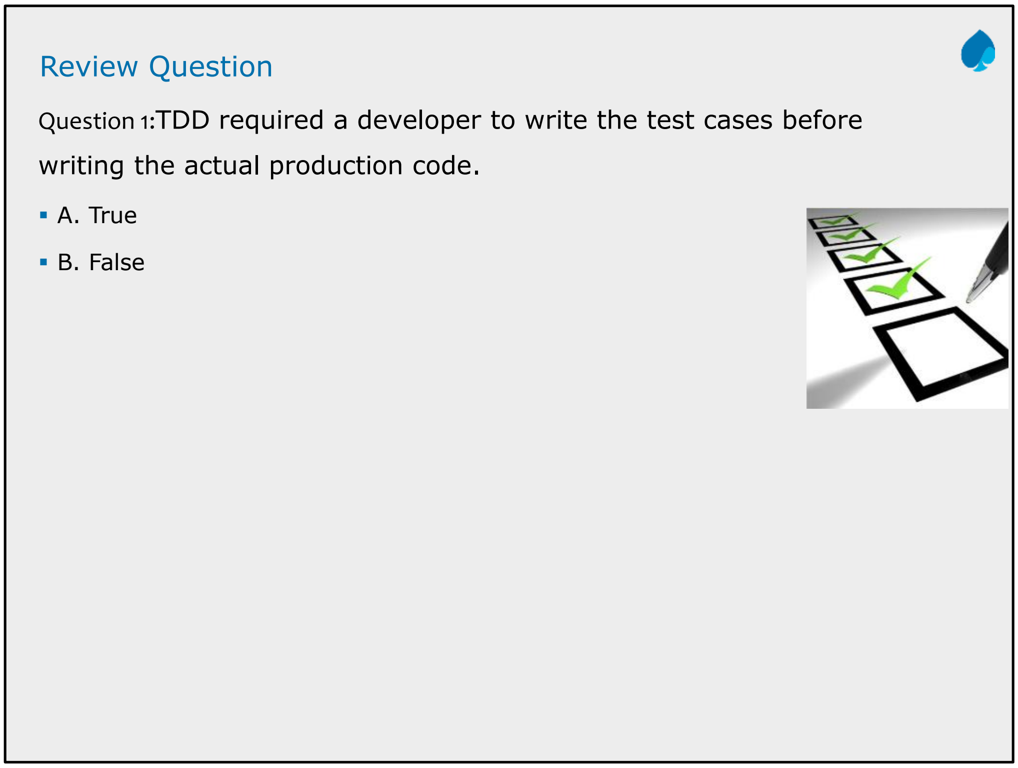
Review Question

Question 1: Why should one do Unit Testing?

- Option 1: Helps to write code better
- Option 2: Provides immediate feedback on the code
- Option 3: Because it is one of the testing methods that has to be carried out

Question 2: JUnit is a licensed product and can be purchased with Java.

- True / False



Review Question

Question 1:TDD required a developer to write the test cases before writing the actual production code.

- A. True
- B. False