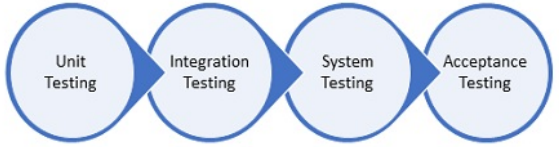
# JUnit 5 (Testing Tool)

## Introduction to software testing

why? → prevention is better than cure

types → automated testing & manual testing




**Unit Testing**

In Unit Testing, all independent parts or components, known as Units, are tested. It needs to be done during the development phase by developers and is the first level of testing.

**Integration Testing**

Once it is confirmed that all units are working fine individually, it's time to integrate them and test again to check if they can communicate with each other as expected or not, this is known as Integration Testing.

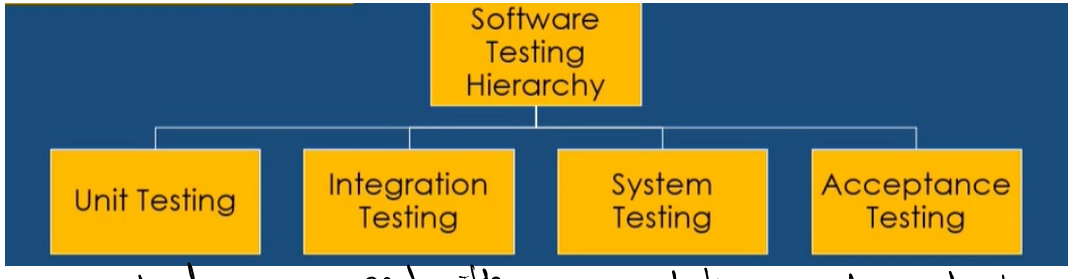
**System Testing**

Once all modules pass integration testing, it's time to have an end to end testing of complete software known as System Testing.

**Acceptance Testing**

In Acceptance Testing, the software is evaluated as per business requirements and analyzed whether it is ready for deployment or not.

## Hyrarchy of Testing




individual testing

connect with each other and then test

connect to server and then test

check client requirements and final product

# Introduction to JUnit 5

## Why JUnit5 ?

| Task | Without Framework | With Framework |
|---|---|---|
| Preparation | Manually | Manually |
| Provide Test inputs | Manually | Manually |
| Run the tests | Manually | Framework |
| Provide Expected output | Manually | Manually |
| Verify result | Manually | Framework |
| Notify developer if tests fail | Manually | Framework |

## what is Junit 5 ?

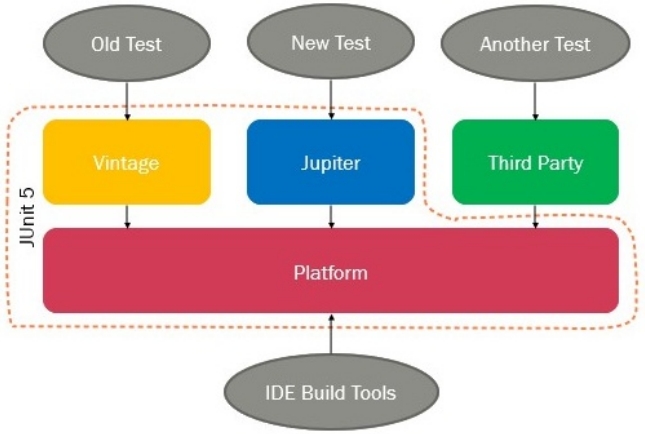
- JUnit 5 is the latest version of JUnit which works with a minimum JDK 8.
- Popular IDEs such as Eclipse and IntelliJ support JUnit 5 test case executions.
- It supports all new Java Language features including Lambdas.
- It provides various assertions and assumptions.
- Unlike previous versions of JUnit, it provides a whole new architecture that is comprised of modules, making it suitable to be used in small memory-based systems.

It provide libraries that provide console launcher

provide APIs which we used to interact with

Junit doesn't provide backward capability, instead this module provide APIs to write test cases

**JUnit 5** = JUnit Platform + JUnit Jupiter + JUnit Vintage




## → Environment setup for Junit 5

**Step1:** RightClick the project-> Build Path-> Configure build path

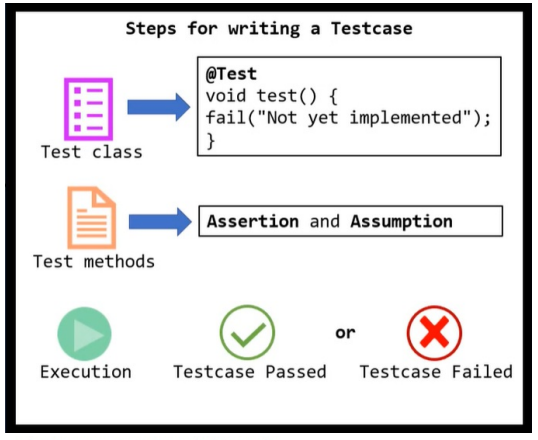
**Step2:** Click on AddLibrary -> Select JUnit and click on next

**Step3:** You will get the JUnit library version as below. Choose Junit5 and click on the finish button

**Step 4:** JUnit 5 jars will be added to the classpath of your project as shown below.

# Getting Started with Junit 5

## ↳ How to write testcase?




What is @Test annotation? Why it is used?

@Test annotation will be placed along with a method. With the help of @Test annotation, JUnit will understand the void test () method has to run as a test case. @Test annotation does not take any attributes. They are from org.junit.jupiter.api.Test class.

```java
package com.infosys.test;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import com.infosys.Client.Calculator;
class CalculatorTest {

    Calculator cobj = new Calculator();

    @Test
    void additionTest() {
        assertEquals(20, cobj.addition(10, 10));
    }
}
```

← method to compare real and expected values

## ↳ @DisplayName // for better readability

Why @DisplayName?

In certain scenarios, while generating the test reports if a Developer/Tester wants to use some custom names for the test classes/methods instead of the default method/class names defined, then they can use the JUnit Display Name feature of JUnit Jupiter.

## ↳ Assertion( used to remove conditional statements)

Commonly used Assert methods

| Methods | Description |
|---|---|
| fail() | To make the test case as failed without any failure message |
| assertTrue(boolean condition) | Checks that the provided condition is true |
| assertFalse(boolean condition) | Checks that the provided condition is false |
| assertEquals(Object expected, Object actual) | Check that if actual and expected objects are equal |
| assertNotEquals(Object unexpected, Object actual) | Check that if expected and actual objects are not equal |
| assertNull(Object actual) | Checks if an object is null |
| assertNotNull(Object actual) | Checks if an object is not null |
| assertSame(Object expected, Object actual) | Checks if two object references are pointing to the same object |
| assertNotSame(Object unexpected, Object actual) | Checks if two object references are not pointing to the same object |
| assertArrayEquals( Object [] expected, Object [] actual) | Checks if expected array and actual array are equal to each other |

```java
class CalculatorTest {

//Assert that actual is null.
    @Test
    void assertNullTest() {
        String var = "Spring";
        assertNull("Expecting a null value for var ", var);
    }

//Assert that the supplied condition is false.
    @Test
    void assertFalseTest() {
        int expected = Arrays.asList("x", "y", "z").size();
        int result = Arrays.asList("a", "b", "c", "d").size();
        assertFalse(expected > result);
    }
}
```

← assertion example

## ↳ Assumptions // used to check or put conditions on operating system.

Commonly used assumption class methods

| Methods | Description |
|---|---|
| assumeFalse(boolean assumption) | Confirm the given assumption is false. |
| assumeTrue(boolean assumption) | Confirm the given assumption is true. |
| assumingThat(boolean assumption, Executable executable) | Perform the provided Executable only if the assumption is valid. |

```java
package com.infosys.test;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assumptions.assumeTrue;
import org.junit.jupiter.api.Test;
import com.infosys.service.Calculator;
class CalculatorTest {
    private Calculator cobj = new Calculator();
    // Assumes that the y value is >0. If y<=0 test case will not execute
    @Test
    void assumeTrueTest() {
        int x = 100;
        int y = 0;
        assumeTrue(y > 0);
        assertEquals(5, cobj.addition(x, y));
    }
}
```

← assumptions example

## ↳ @Disabled

simply used to skip particular testcase(s) or the entire testcase class

```java
class TestDemo {

    Calculator c = new Calculator();
    @Disabled("Until the bug fixed")
    @Test
    void testAdditionFail() {
        assertEquals(20, c.add(1, 1));
    }
}
```

# ⇒ Test fixture

Why are Test Fixtures needed?

Suppose in our test we need to test all the methods (subtraction(),division(),addition() and multiplication()) of Calculator class. For this testing, we need to initialize the variables a and b with the values 100 and 300 before each test case. How can we achieve that without repeating the initialization code?

What is Test Fixture?

Tests need to run against a set of objects called Test Fixture. Test Fixtures can contain a common set of statements to be executed and can be used by several tests. Test Fixtures can handle common resources required by tests.

↳ all the annotations are

@BeforeAll

The method which is annotated with @BeforeAll annotation is run only once. It is expected to execute before all test method of the test class is executed. This is helpful in the initialization of expensive resources like database connection, connection to a server. The @BeforeAll annotated method will be always a static method.

@AfterAll

The method which is annotated with @AfterAll annotation is run only once. When all the test methods of the test class executed, the @AfterAll annotated method will be called. This can be used in some operations like closing connections or closing files that are already opened. The @AfterAll annotated method will be always a static method.

@BeforeEach

The method which is annotated with @BeforeEach annotation is executed before each test. This is helpful in the initialization of variables, resources.

@AfterEach

The method which is annotated with @AfterEach annotation is executed after each test. This is helpful in releasing resources initialized with @BeforeEach.

TestInfo

TestInfo is an interface in org.junit.jupiter.api package. It is used to inject information about the current test or container into to @BeforeEach, @AfterAll,@AfterEach, @BeforeAll, and @Test annotated methods. So all the test fixture annotated methods and @Test annotated methods can take arguments as TestInfo which is used to get the info about the test.

Commonly used TestInfo methods

| Method | Usage |
|---|---|
| String getDisplayName() | It will give the name of the test |
| Optional<Method> getTestMethod() | Gives the method related to the current test |

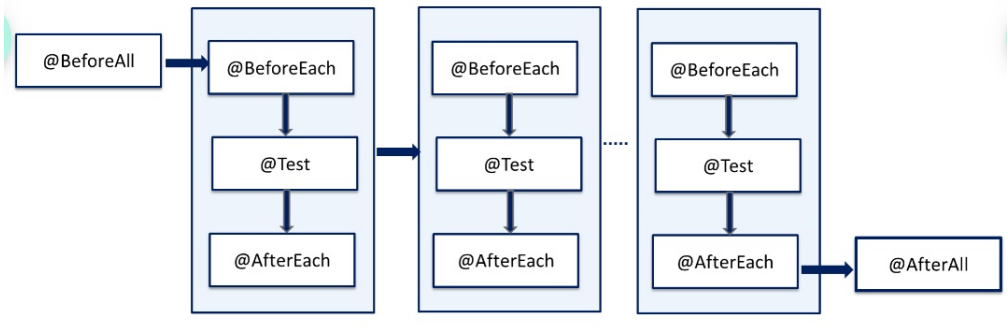
Why @BeforeEach and @AfterEach annotation used?

Suppose in our test we need to test the addition() and multiplication() methods of the Calculator class. For this testing, we need to initialize the variables a and b with the values 20 and 30 before each test case. How can we achieve that without repeating the initialization code? @BeforeEach will help to resolve it.

Why @BeforeAll and @AfterAll annotations used?

Consider a situation where we need to open a DB connectivity or file before starting any of the test cases and we need to close the DB connection or file after finishing all of our test case execution. In that situation, we can use @BeforeAll and @AfterAll annotations.

@BeforeAll annotated method will execute only once and it will execute before all the test cases start. @AfterAll annotated method will execute only once and it will execute after all the test cases end.



# ⟹ Test Suite

### Why Test Suite?

Suppose if we want to run multiple test cases spread over multiple classes and packages at the same time so that we do not have to execute each one of those test cases manually. What would be the solution for this?

### What is Test Suite?

A test suite in JUnit 5 is a collection of multiple packages or classes so that we do not have to execute test cases in multiple packages or classes manually.

We can create TestSuites in Junit5 using primarily two annotations: @SelectPackages & @SelectClasses.

Additional capabilities of filtering test packages, classes, and test methods are also provided by JUnit 5.

We will make use of @RunWith(JUnitPlatform.class) to run a test suite.

## ↳ Test suite annotations

### @SelectPackages

This annotation helps us to specify the names of the packages to run as a test suite with the help of @RunWith(JUnitPlatform.class)

```
@RunWith(JUnitPlatform.class)

@SelectPackages(com.infosys.additionTest)

public class SuiteTest {

}
```

↳ We can select multiple packages as well in the form of array

```
@SelectPackages({"com.infosys.additionTest","com.infosys.multiplicationTest","com.infosys.divisionTest"})
```

### @SelectClasses

This annotation helps us to specify the names of the test classes to run with the help of @RunWith(JUnitPlatform.class)

```
@RunWith(JUnitPlatform.class)

@SelectClasses(CalculatorMultiplicationTest.class)

public class ClassTest {

}
```

### @IncludePackages & @ExcludePackages

@SelectPackages annotation allows us to include a package and its sub-packages which will be scanned for test classes. Similarly, to include or exclude sub-packages we have @IncludePackages and @ExcludePackages annotations respectively.

```
@RunWith(JUnitPlatform.class)

@SelectPackages("com.infosys")

@IncludePackages("com.infosys.additionTest")

public class IncludepackageTest {

}
```

```
@RunWith(JUnitPlatform.class)

@SelectPackages("com.infosys")

@ExcludePackages("com.infosys.divisionTest")

public class ExcludePackageTest {

}
```

### @IncludeClassNamePatterns and @ExcludeClassNamePatterns

These annotations allow us to filter test classes based on class names. So, if including the entire package is not feasible then the above-mentioned annotations come in handy.

```
@RunWith(JUnitPlatform.class)

@SelectPackages("com.infosys")

@IncludeClassNamePatterns(".*Calculator.*")

public class IncludeClassNameTest {

}
```

```
@RunWith(JUnitPlatform.class)

@SelectPackages("com.infosys")

@ExcludeClassNamePatterns(".*Division.*")

public class ExcludeClassNameTest {

}
```

# ⇒ Tagging

Why Tagging is needed?

Consider a scenario where we need to group the success and failure test cases separately for the CalculatorTest class, and we need to run only the success Test cases

Where we can use the tag?

- When we need to group tests based on the type of automated tests: UnitTests, IntegrationTests, SmokeTests, RegressionTests, PerformanceTests.
- When we need to group tests based on how quick the tests execute: SlowTest, QuickTest
- When we need to group based on the state of the test: UnstableTests, InProgressTests

Example

```
class CalculatorMultiplicationTest{
@Test
    @Tag("Slow")
    @Tag("Failure")
    void multiplicationTest() {
        // code
    }
}
```

Steps to create tagging

Step1

```
@Test
@Tag("Success")
void additionSuccessTest() {
    actual = cobj.addition(a, b);
    assertEquals(50, actual);
}

@Test
@Tag("Failure")
void additionFailureTest() {
    actual = cobj.addition(a, b);
    assertEquals(40, actual);
}
@Test
@Tag("Cloud")
@Tag("Failure")
void additionCloudTest() {
    actual = cobj.addition(a, b);
    assertNotEquals(50, actual);
}
```

Step 2

```
@RunWith(JUnitPlatform.class)
@SelectPackages("com.infosys.test")
//@SelectClasses({CalculatorAddtionTest .class,CalculatorMultiplicationTest.class})
@IncludeTags("Success")
//@IncludeTags({ "Success", "Cloud" })
//@ExcludeTags("Failure")
public class TagTest {

}
```

# ⇒ Parameterized Test

Why Parameterized Test?

Suppose we want to provide multiple inputs from single/various sources automatically. What would be the solution for this?

What is Parameterized Test?

Parameterized tests are extremely important if we want to provide multiple inputs and from various sources. These tests help us to automate the process of providing different arguments and running the tests multiple times. They help us to reduce the number of test cases involved for specific functionality.

These tests are declared like any other regular test methods but instead of **@Test**, we use **@ParameterizedTest** annotation. We need to provide at least one source from which will provide arguments to the test case.

# ⇒ Parameterized Test annotations

1)

@CsvSource

This annotation is used when the input data is expressed as comma-separated values (i.e., String literals).

```
@CsvSource({ "2, 1, 2", "3, 2, 6" })
@ParameterizedTest
void multiplicationValidTest(int a, int b, int ans) {
assertEquals(ans, cobj.multiplication(a, b));
}
```

2)

@CsvFileSource

This annotation lets you read data of a CSV file from the classpath's location. The parameterized test is invoked for each line in that CSV file.

```
@ParameterizedTest
@CsvFileSource(resources = "/two-column.csv", numLinesToSkip = 1, delimiter=';')
void testFileSource (String sampleCountry, int sampleRef) {
    assertNotEquals(0, sampleRef);
    assertNotNull(sampleCountry);
}
```

3)

@ValueSource

One of the simplest sources to work with as it allows you to define an array of literal values. It goes one by one through that array and provides them as arguments per each invocation of the parameterized test case.

```
@ParameterizedTest
@ValueSource(strings = { "V", "X" })
void testWithValueSource(String argument) {
     assertTrue(RomanNumeral.contains(argument));
}
```

4)

- **@NullSource**: it is used to provide a null value as an argument to a method annotated with @Parameterized annotation.
  - @NullSource is not applicable for primitive type arguments.
- **@EmptySource**: a parameterized method can be passed a singular empty argument using this annotation. This annotation works only for the below mentioned types: java.lang.String, java.util.List, java.util.Set, java.util.Map, arrays containing primitive data types (e.g., int[], char[][], etc.), object arrays (e.g.,String[], Integer[][], etc.).
  - No support is provided for the subtypes of the supported types.
- **@NullAndEmptySource**: this annotation combines the functionality of the @Null Source and @Empty Source annotations.

5)

@EnumSource

This annotation is an easy and efficient way to make use of Enum constants in parameterized tests. It uses the constants specified in an Enum as an ArgumentsSource.

```
@ParameterizedTest
@EnumSource(ChronoUnit.class)
void test (TemporalUnit sample) {
    assertNotNull(sample);
}
```

6)

@MethodSource

This annotation helps us to use one or more than one factory method present in the same test class or in an external class.

```
@ParameterizedTest
@MethodSource("nameProvider")
void testSource(String args) {
    assertNotNull(args);
}

static Stream<String> nameProvider() {
    return Stream.of("apple", "banana");
}
```

7)

To specify a custom and reusable ArgumentsProvider we can use **@ArgumentsSource**.

```
@ParameterizedTest
@ArgumentsSource(CustomProvider.class)
void testWithArgumentsSource(String argument) {
    assertNotNull(argument);
}
public class CustomProvider implements ArgumentsProvider {
    @Override
    public Stream<? extends Arguments> provideArguments(ExtensionContext context) {
        return Stream.of("apple", "banana").map(Arguments::of);
    }
}
```

# ⇒ Dynamic Test

alternative of parameterized test so instead of creating multiple test method for some business logic, we can have only one test method with multiple inputs which act as factory of test methods.

new data type provided by Junit 5

```
@TestFactory
Collection<DynamicTest> resultTest() {
  Collection<DynamicTest> dynamicTests
                        = new ArrayList<>();
  List<String> inputList
               = //contains test input values
  List<String> expectedOutputList
            = //contains expected output values
  for (int i = 0; i < inputList.size(); i++) {
    //logic to test business method
  }
  return dynamicTests;
}
```

Dynamic Test, as the name suggests are dynamic in nature, which means they are automatically created during runtime. Automatic doesn't mean that we don't need to write any test method but at least not so many test methods. These methods are not annotated with @Test instead they are annotated with @TestFactory annotation.

```
class CalculatorTest {
    private Calculator cobj = new Calculator();

    @TestFactory
    Collection<DynamicTest> convertToNumberTest() {
        Collection<DynamicTest> dynamicTests = new ArrayList<>();
        return dynamicTests;
    }

}
```

```
class CalculatorTest {
    private Calculator cobj = new Calculator();

    @TestFactory
    Collection<DynamicTest> convertToNumberTest() {
        Collection<DynamicTest> dynamicTests = new ArrayList<>();
        List<String> romanSymbolsList = new ArrayList<>(Arrays.asList("I", "V", "X", "L", "C", "D", "M", "K", " ", ""));
        List<Integer> numbersList = new ArrayList<>(Arrays.asList(1, 5, 10, 50, 100, 500, 1000, null, null, null));
        for (int i = 0; i < romanSymbolsList.size(); i++) {
            String romanSymbol = romanSymbolsList.get(i);
            Integer number = numbersList.get(i);
            Executable execution = () -> assertEquals(number, cobj.convertToNumber(romanSymbol));
            String testName = "Testing input " + romanSymbol;
            DynamicTest dynamicTest = DynamicTest.dynamicTest(testName, execution);
            dynamicTests.add(dynamicTest);
        }
        return dynamicTests;
    }

}
```

Note:-

Now with this method, any number of inputs can be tested for business logic. But hold on, you need to leave some other features of JUnit if you use this. Like you can't use @BeforeEach and @AfterEach annotated methods with @TestFactory annotated test method for each input instead they will get executed only once because @TestFactory doesn't support life cycle like @Test.

## ⇒ Exception Testing ( JUnit 5 provides ways to test exceptions also using Assertions. )

What is Exception Testing?

Exception Testing means writing test methods with input parameters such that they result in throwing exceptions. JUnit 5 provides the **assertThrows()** method to assert these exception types with their appropriate messages.

```
class CalculatorTest {
    private Calculator cobj = new Calculator();

    @Test
    void divisionValidTest() {
        int quotient = cobj.division(10, 5);
        assertEquals(2, quotient);
    }

    @Test
    void divisionInValidTest() {
        Exception exception = assertThrows(ArithmeticException.class, () -> cobj.division(10, 0));
        assertEquals("divide by zero", exception.getMessage());
    }

}
```

## ⇒ Timeout Testing

What is Timeout Testing?

Timeout Testing means writing test methods with input parameters such that they result in the failure of test methods due to timeout. JUnit 5 provides 3 ways for timeout testing:

1. **@Timeout annotation** to declare that a test method, testfactory method, test template, or life cycle method will fail if timeout exceeds. By default, the value is always in seconds. But you can even provide other time units provided by TimeUnit enum. @Timeout annotation can also be applied at class level if you want the same timeout for all methods.

```
@Timeout(3) // Timeout after 3 seconds
@Test
void annotationDelayValidTest() throws InterruptedException {
    cobj.delay(2); // Delay for 2 seconds
}

@Timeout(value = 3, unit = TimeUnit.NANOSECONDS) // Timeout after 3 nanoseconds
@Test
void annotationDelayInValidTest() throws InterruptedException {
    cobj.delay(2); // Delay for 2 seconds
}
```

2. **assertTimeout() method** can be used within the test method. It takes a method to be tested as an Executable object and time needs to be provided using Duration and ChronoUnit libraries. Execution will **NOT** be aborted if timeout exceeds.

```
@Test
void assertDelayValidTest() {
    Executable execution = () -> cobj.delay(2); // Delay for 2 seconds
    assertTimeout(Duration.of(3, ChronoUnit.SECONDS), execution); // Timeout after
3 seconds
}

@Test
void assertDelayInValidTest() {
    Executable execution = () -> cobj.delay(2); // Delay for 2 seconds
    assertTimeout(Duration.ofNanos(3), execution); // Timeout after 3 nanoseconds
}
```

3. **assertTimeoutPreemptively() method** can be used within the test method. It takes a method to be tested as an Executable object and time needs to be provided using Duration and ChronoUnit libraries. Execution will be preemptively aborted if timeout exceeds.

```
@Test
void preemptiveAssertDelayValidTest() {
    Executable execution = () -> cobj.delay(2); // Delay for 2 seconds
    assertTimeoutPreemptively(Duration.ofSeconds(3), execution); // Timeout after 3 seconds
}

@Test
void preemptiveAssertDelayInValidTest() {
    Executable execution = () -> cobj.delay(2); // Delay for 2 seconds
    assertTimeoutPreemptively(Duration.of(3, ChronoUnit.NANOS), execution); // Timeout after 3
nanoseconds
}
```

⇒ Extension // generic way of test fixture
// @before, @after

What is Extension?

In JUnit 5, one of the principles is to prefer extensions over features. Extensions are used in JUnit 5 to provide some additional functionality to all test methods of a test class but are defined outside the test class.

These extensions can be registered in the following 3 ways:

- Declaratively with @ExtendWith
- Programmatically with @RegisterExtension
- Automatically with the Service Loader

1) @ExtendWith annotation is used upon test classes and methods where extensions need to be registered declaratively. It's a repeatable annotation that means it can be used to extend multiple extensions and can be used both upon class and method level.

```
@ExtendWith(LoggerExtension.class)
class CalculatorTest {
    private Calculator cobj = new Calculator();

    @Test
    void additionValidTest() {
        System.out.println("Inside addValidTest()");
        int sum = cobj.addition(1, 1);
        assertEquals(2, sum);
    }

}
```

Note:- The sequence registering matter in @ExtendWith & @RegisterExtension

2)

@RegisterExtension annotation is used for registering extensions programmatically and can be used to pass arguments to the extension's parameterized constructors. This annotation is used upon the field of the test class.

```
class CalculatorTest {
    private Calculator cobj = new Calculator();

    @RegisterExtension
    LoggerExtension le = new LoggerExtension("Extension registered for " + this.getClass());

    @Test
    void additionValidTest() {
        System.out.println("Inside addValidTest()");
        int sum = cobj.addition(1, 1);
        assertEquals(2, sum);
    }

}
```

3) With 3rd way, we dont need to write any code but just turn on this functionality

```
-ea -Djunit.jupiter.extensions.autodetection.enabled=true
```

## ⇒ Conditional Testing

When developers want to enable or disable some features on the basis of some conditions.

- Operating System Condition
- Java Runtime Environment condition
- System Property Condition
- Environment Variable condition

### 1) Operating System conditions

- @EnableOnOS
- @DisableOnOS

Note: Annotated test classes/methods will be enabled/disabled based on the OS version of the specified system.

### 2) Java Runtime Environment Conditions

- @EnableOnJre
- @EnableOnJreRange
- @DisableOnJre
- @DisableOnJreRange

**Note:-**

default values of @EnableOnJreRange and @DisableOnJreRange are Java_8 (which is minimum) and Java.OTHER (which is maximum)

### 3) System Property Conditions:-

- @EnableIfSystemProperty
- @DisableIfSystemProperty

These annotations will accept

- named
- matches

### 4) Environment Variable Conditions

- @EnableIfEnvironmentVariable
- @DisableIfEnvironmentVariable

These annotations will accept

- named
- matches

**Note:-** gmp

- All the conditional annotations can be used as a class level and method level. When the annotation is used as a class level then the condition in the conditional annotations will be applied to all the test methods in the test class.
- All the conditional annotations can be used to create a custom composed annotation.

# ⇒ Repeated Tests

When we want to run a same method multiple time.

Using the annotation @RepeatedTests

```
class CalculatorTest {
    private Calculator cobj = new Calculator();

    @RepeatedTest(5)
    void additionTest() {
        assertEquals(2, cobj.addition(1, 1));
    }

    @Test
    void multiplicationTest() {
        assertEquals(4, cobj.multiplication(2, 2));
    }

}
```

⇒ # Nested Tests

When we need to create Test class inside a Test class.

Why Nested Tests?

We met certain scenarios where we need to nest/group the test cases in the same test class for better maintainability.

What are Nested Tests?

When a Developer/Tester needs more clarity to establish a relationship among similar types of tests which may be a class or a function, and which will be more maintainable then Nested tests give the capability to express the same.

using annotation @Nested

```
@Nested
class CalculatorAdditionTest {
    @Test
    void additionValidTest() {
        assertEquals(4, cobj.addition(2, 2));
    }

    @Test
    void additionInvalidTest() {
        assertNotEquals(5, cobj.addition(2, 2));
    }
}
```

} example

Note:- In future if needed, we can seperate the logic

# ⇒ Test Interface

### Why Test Interfaces?

In certain scenarios when there is a lot of code like pre-test, post-test methods must be provided for a particular test case, instead of writing the same in the test class, we can create an interface to write all the pre and post-test works for better maintainability.

JUnit Jupiter allows certain annotations to be declared in an interface as default methods. Some of them are:

- @Test
- @RepeatedTest
- @ParameterizedTest
- @TestFactory
- @TestTemplate
- @BeforeEach
- @AfterEach

Out of these annotations, some of the annotations can also be declared as static methods also. They are:

- @BeforeAll
- @AfterAll

To declare all these methods as default or static methods based on the requirements the interface should be annotated with @TestInstance(Lifecycle.PER_CLASS).

demo

Step 1

```java
@TestInstance(Lifecycle.PER_CLASS)
public interface CalculatorTest {
    public static final Logger logger = Logger.getLogger(CalculatorImplTest.class.getName());

    @BeforeAll
    default void beforeAllTests(TestInfo info) {
        logger.info("Inside BeforeAll method: " + info.getDisplayName());
    }

    @BeforeEach
    default void beforeEachTest(TestInfo info) {
        logger.info("Inside BeforeEach in method: " + info.getDisplayName());
    }

    @AfterEach
    default void afterEachTest(TestInfo info) {
        logger.info("Inside AfterEach in method: " + info.getDisplayName());
    }

    @AfterAll
    default void afterAllTests(TestInfo info) {
        logger.info("Inside AfterAll method: " + info.getDisplayName());
    }
}
```

Step 2

now implement this in seperate class for better readability

```java
class CalculatorImplTest implements CalculatorTest {
    private Calculator cobj = new Calculator();

    @Test
    void additionTest() {
        assertEquals(2, cobj.addition(1, 1));
    }

    @Test
    void multiplicationTest() {
        assertEquals(1, cobj.multiplication(1, 1));
    }
}
```

# ⇒ Dependency Injection

ParameterResolver is a powerful API that resolves the constructor/method parameter dynamically at the runtime with the help of a registered parameter resolver from JUnit Jupiter.

Currently, we have 3 built-in parameter resolvers in JUnit 5 which get registered automatically:

1. TestInfoParameterResolver
2. RepetitionInfoParameterResolver
3. TestReporterParameterResolver

TestInfoParameterResolver:

1) If a constructor/method inside a test class is accepting the parameter of type TestInfo class then the TestInfoParameterResolver will provide an instance of TestInfo from org.junit.jupiter.api package. This parameter resolver will help the programmer to retrieve the information of a test case such as the tags associated with a test class/method, the display name associated, etc.

RepetitionInfoParameterResolver:

2) Some of the test methods like @RepeatedTest, @BeforeEach, and @AfterEach methods are get repeated multiple times during the execution of test cases in JUnit Jupiter. So to keep a track of how many times these methods are getting repeated we can use some built-in parameter resolver provided by JUnit Jupiter. The parameter resolver which helps to keep a track of repeating methods in the JUnit 5 is RepetitionInfoParameterResolver.

TestReporterParameterResolver:

3) TestReporter is an instance of TestReporterParameterResolver resides in org.junit.jupiter.api package. This parameter resolver can be used as a parameter in the methods/constructors of a test case and used to give some additional information about the test case execution to the developers/testers.

3)

```java
class CalculatorTest {
    private Calculator cobj = new Calculator();

    @Test
    void additionTest(TestReporter testReporter) {
        testReporter.publishEntry("Status Msg : Inside additionTest()");
        assertEquals(2, cobj.addition(1, 1));
    }

    @Test
    void subtractionTest(TestReporter testReporter) {
        testReporter.publishEntry("Custom Message", "Inside subtractionTest()");
        assertEquals(4, cobj.subtraction(6, 2));
    }

    @Test
    void multiplicationTest(TestReporter testReporter) {
        Map<String, String> myMap = new HashMap<>();
        myMap.put("Custom Message 1", "Inside multiplicationTest()");
        myMap.put("Custom Message 2", "Inputs are 2 and 2");
        testReporter.publishEntry(myMap);
        assertEquals(4, cobj.multiplication(2, 2));
    }
}
```

1)

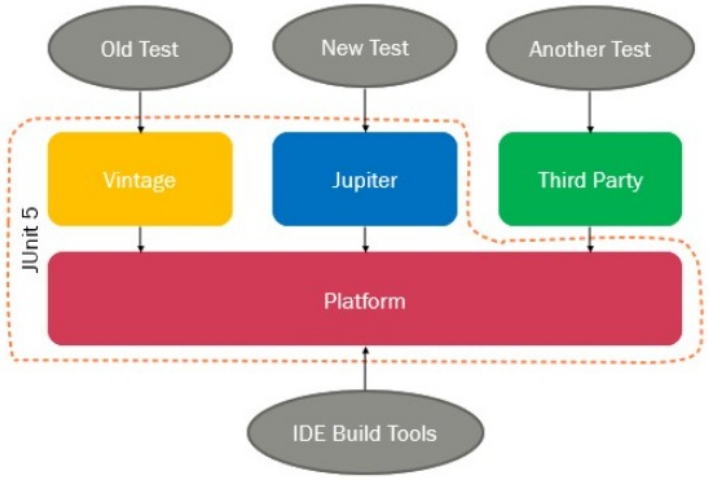
```java
import com.infosys.service.Calculator;

@DisplayName("Test class to demonstrate TestInfo")
class CalculatorTest {
    private Calculator cobj = new Calculator();

    public CalculatorTest(TestInfo testInfo) {
        assertEquals("Test class to demonstrate TestInfo", testInfo.getDisplayName());
    }

    @BeforeEach
    public void setUp(TestInfo testInfo) {
        String displayName = testInfo.getDisplayName();
        assertTrue(displayName.equals("Addition Test") || displayName.equals("Multiplication Test"));
    }

    @DisplayName("Addition Test")
    @Tag("addition-tag")
    @Test
    void additionTest(TestInfo testInfo) {
        assertEquals("Addition Test", testInfo.getDisplayName());
        assertTrue(testInfo.getTags().contains("addition-tag"));
        assertEquals(2, cobj.addition(1, 1));
    }

    @DisplayName("Multiplication Test")
    @Test
    void multiplicationTest() {
        assertEquals(4, cobj.multiplication(2, 2));
    }
}
```

2)

```java
class CalculatorTest {
    private static final Logger logger = Logger.getLogger(CalculatorTest.class.getName());

    private Calculator cobj = new Calculator();

    @BeforeEach
    void beforeEach(TestInfo testInfo, RepetitionInfo repetitionInfo) {
        int currentRepetition = repetitionInfo.getCurrentRepetition();
        int totalRepetitions = repetitionInfo.getTotalRepetitions();
        logger.info("Current Repetition: " + currentRepetition);
        logger.info("Total Repetition: " + totalRepetitions);
    }

    @RepeatedTest(5)
    void additionTest(RepetitionInfo repetitionInfo) {
        assertEquals(5, repetitionInfo.getTotalRepetitions());
        assertEquals(2, cobj.addition(1, 1));
    }

}
```

# ⇒ Backward Compatibility

architecture




In JUnit 3 and 4 the @Rule annotation can be used on the variables as well as on the test methods. So if the Developer/Tester, while migrating from JUnit 3 / 4 to JUnit 5 wants to enable a few of the Rule extensions in the JUnit Jupiter then he/she can add the below class-level annotations.

- @EnableRuleMigrationSupport: This annotation enables all APIs and extensions for the above JUnit rules(The Rules discussed in the second point only).
- @EnableJUnit4MigrationSupport: The @EnableRuleMigrationSupport annotation only supports the migration for Rules where the @EnableJUnit4MigrationSupport annotation supports the above three Rules as well as JUnit 4 annotations like - @Ignore, @Category.

Rules are :-
- ↳ Catagary Support
- ↳ Rules of Junit 4 Support
- ↳ @Ignore Support

1. @EnableJUnit4MigrationSupport : To enable all JUnit 4 extensions in JUnit Jupiter this class level annotation must be added in the test class. The test class can be created as below:

Why JUnit Vintage?

Many of the larger organizations have large JUnit 4 test cases that make use of some JUnit 4 annotations and custom rules. So, for these kinds of organizations directly migrating from JUnit 3/4 to JUnit Jupiter will be difficult.

So what is JUnit Vintage?

JUnit Vintage provides a test engine to run certain test cases of JUnit 3/4 in JUnit Jupiter. The JUnit team provided a gradual migration path with support for some annotations and rules in JUnit 5 Jupiter with help of JUnit Vintage based on some adapters. These adapters are suitable to certain JUnit 3/4 annotations and rules with JUnit 5 and don't alter the implementations of the current test cases completely.

Some of the steps need to be taken care of while migrating the test cases from JUnit 3/4 to JUnit 5.

- Annotations reside in the org.junit.jupiter.api package.
- No @Before and @After methods, A Developer/Tester needs to use @BeforeEach and @AfterEach as a replacement.
- No @BeforeClass and @AfterClass methods; A Developer/Tester needs to use @BeforeAll and @AfterAll as a replacement.
- No @Ignore method: A Developer/Tester needs to use @Disabled or one of the other built-in execution conditions as a replacement
- No @Category; A Developer/Tester needs to use @Tag as a replacement.
- No @RunWith, A Developer/Tester needs to use @ExtendWith as a replacement.
- No @Rule and @ClassRule; A Developer/Tester needs to use @ExtendWith and @RegisterExtension as a replacement.

```java
@EnableJUnit4MigrationSupport
class CalculatorTest {
    private Calculator cobj = new Calculator();

    @Test
    @Ignore
    void additionTest() {
        assertEquals(2, cobj.addition(1, 1));
    }

}
```

previous version

with help of this annotation

# Best Practice in Junit 5

When we develop any test case we need to ensure the following

- All the test cases should execute very fast
- It will be good if more test cases executed frequently
- Test cases should be extremely consistent and reliable
- It should indicate the real issues in the production code
- Tests should fail only when there is a bug in the system
- Tests should be easily readable

Let us see some of the best practices while writing the JUnit 5 test cases

- Test cases are readable.
- The Name of the test class and test methods should be appropriate.
- Plan to run the test cases completely in-memory, avoid doing a test for availing file system and DB, and for HTTP requests
- Develop the JUnit test cases in a way that it should execute faster
- Avoid skipping test cases with the usage of @Disabled without any reason, a reason could also be passed for @Disabled for any developer to know why was a particular test case is skipped.
- Each unit test method should have only one assertion
- Maintain proper order of parameters in assertions.
- Practice the resilient assertions
- Don't go for unit-testing of any settings or configuration.
- First plan unit tests for functionalities that have fewer dependencies
- Plan for proper tests for all functionalities.
- Confirm that unit test code is parted from production code
- Do not print anything out in unit tests
- Avoid the usage of static members inside the test class
- Do not give your own exception that occurs only to fail a test

**Code coverage** is a measure used to give the degree to which the source code of a program is executed when a specific test suite runs.

To measure the percentage of code has been utilized by a test suite, one or more coverage condition can be used:

- Function coverage – Function coverage will check every function within the program been called?
- Statement coverage – Statement coverage will check every statement within the program been executed?
- Branch coverage – Branch coverage will check every branch of each control structure (such as in if and case statements) been executed?
- Condition coverage (or predicate coverage) – Condition coverage will check every Boolean sub-expression evaluated both to true and false?

### Why Sonar?

Code quality means how well software **code** is standard, performs, bug-free, stable, and well tested.

This means we need to ensure

- The application should be robust and has to properly respond to the users
- If the software code is a good-quality code then it will be easy to read and do any modification
- Application code should be maintainable

### What is Sonar?

Sonar is developed in Java and is an open-source tool to manage code quality.

It considers the architecture, design, comments, duplication, unit tests, complexity, potential bugs, and coding rules as they are important co-ordinates to analyze the code.