## Links for reference

<https://docs.spring.io/spring-framework/docs/current/reference/html/testing.html#unit-testing-spring-mvc>

<https://github.com/springframeworkguru/intro-junit5>

<https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html>

|  |  |
| --- | --- |
| Junit user guide | <https://junit.org/junit5/docs/current/user-guide/#writing-tests> |
| AssertJ documentation | <https://assertj.github.io/doc/#assertj-overview>  https://joel-costigliola.github.io/assertj/ |
| Hamcrest – pretty older | <https://hamcrest.org/JavaHamcrest/> |
| Git hub links for code reference | <https://github.com/springframeworkguru/testing-java-junit5>  <https://github.com/manideep-vv/Forked-testing-java-junit5> |
| Spring framework guru repos for reference | <https://github.com/springframeworkguru/junit4-legacy-app/tree/maven-deps> |

## Annotations for web layer

1. **@TestPropertySource(locations= "classpath: application-test.yml",**

**properties={ ccs.mongo.connection.enabled-true", "spring.data.mongodb.authMechanism-PLAIN"}**

this is a class level annotation

When spring is trying to start while running junit, instead of loading the properties from properties file or yml file we can provide and override and with this we can give the property file location also so that all those properties will be loaded

1. **@Mock vs @MockBean**

@Mock will just create a mock and it is mandatory to add @InjectMocks else these mock objects will be in test class only, it will not be injected to the target class

So use this @Mock along with the @InjectMocks only then mocked objects will be kept in that target class

@MockBean –This will create a mock of that class and it keeps that mocked bean in the spring context.

1. **@ContextConfiguration (classes={RestTemplateConfigMock.class))**

This annotation alone is enough to start the container while running junit test cases with this class this is same like @Configuration annotation this will be picked up or loaded when container is getting started

* If u want a mocked bean to be injected into spring container then use @MockBean
* Lets say u want to create a real bean object and inject into spring container then as usual , create a configuration class annotated with **@ContextConfiguration**

Like

**@ContextConfiguration** (classes=RestTemplateConfigMock.class) and have methods annotated with @Bean and create that object inside that method and these @Bean methods will be called and those will be kept into spring container

1. **@SpringJUnitConfig(classes-(RestTemplateConfigMock.class))** these classes will be

loaded or detected by the spring

**@SpringJUnitConfig=@ExtendWith(SpringExtension.class)+ @ContextConfiguration**

1. @SpringJUnitWebConfig
2. @WebMucTest(DataController.class)
3. @ActiveProfiles(“test../any profile”)

How to load data from files for junit testing

If u want to load the data from the file in src/test/resources – then use below class loader to load that file

|  |  |  |  |
| --- | --- | --- | --- |
| Best way to read file from class path is using class loader | |  |  | | --- | --- | | when u are using maven all these will be already in classpath  src/main/java & src/main/resources  src/test/java & src/test/resources | public void init() throws IOException {  *InputStream* inputStream= getClass().getClassLoader().getResourceAsStream("cars.txt");  ObjectMapper mapper=new ObjectMapper();  String fileData = mapper.readTree(inputStream).toString();  System.***out***.println(fileData); } |   this class loader can detect all the files that is present in the class path  after loading use only objectMapper to convert the content into sting |

Junit 4 vs 5

When u have both junit 4 and junit 5 , junit-vintage-engine jars in ur project then refer

<https://github.com/springframeworkguru/junit4-legacy-app/blob/maven-deps/pom.xml> (this is a project which has both junit 4 and 5 al together)

<https://github.com/springframeworkguru/junit4-legacy-app/blob/junit4to5/pom.xml>

|  |  |
| --- | --- |
|  |  |
| @Before | @BeforeEach |
| @After | @AfterEach |
| @BeforeClass | @BeforeAll |
| @AfterClass | @AfterAll |
| @Ignored | @Disabled |
| @Category | @Tag |
| @RunWith(SpringJunit4ClassRunner.class) | @ExtendWith(SpringExtension.class) |
| @Test(expected =Runtime.class) | Assertions.assertThrows(FooException.class) |
| @Test(timeout=1) | Assertions.assertTimeout(Duration) |
| @Rule and @ClassRule doesn’t exists | @ExtendWith (Rules are not supported in junit 5) |
| @Test  This anno is from package import org.junit.Test; | @Test anno is from package “org.junit.jupiter.api.Test” |

Junit 4 tests can be run via junit 5 with the help of “junit-vintage-engine” jar (which brings junit 4 too) which enables support for junit 3,4

Junit-vintage-engine is like an adapter jar which takes junit 4 test and run that test in junit 5 jar

If junit 4 is not there in Ur project u can happily remove **junit-vintage-engine** jar also

Junit 5 supports only java 1.8+

Note: we can have both junit 4 code and junit 5 code in single project, we don’t need to migrate instantly by changing all junit4’s @Before to junit 5’s @BeforeEach..

Ex:- in old project if we have junit 4 already u can definitely add junit5 with java 8 we can run both junit 4,5 test cases happily in same project



if u just add vintage engine jar, junit-4 will come automatically, so incase if u want junit 4 along with junit5 just add vintage engine jar which will bring junit 4 automatically

Junit 5

Junit vintage engine for running junit 3,4 tests

JUnit 5 was on September 10th, 2017

### **Maven**

|  |  |
| --- | --- |
|  |  |
| //These jars are for junit 5  <dependency>  <groupId>org.junit.jupiter</groupId>  <artifactId>junit-jupiter-api</artifactId>  <version>5.10.2</version>  <scope>test</scope>  </dependency> | // This Jupiter jar is for junit 5  <dependency>  <groupId>org.junit.jupiter</groupId>  <artifactId>junit-jupiter-engine</artifactId>  <version>5.10.0</version>  <scope>test</scope>  </dependency> |
| “Junit vintage engine jar “ to run junit 4 test cases | <dependency>  <groupId>org.junit.vintage</groupId>  <artifactId>junit-vintage-engine</artifactId>  <version>${junit-platform.version}</version>  </dependency> |
| assertJ  <dependency>  <groupId>org.assertj</groupId>  <artifactId>assertj-core</artifactId>  <version>3.11.1</version>  <scope>test</scope>  </dependency> | Hamcrest  <!-- https://mvnrepository.com/artifact/org.hamcrest/hamcrest-library -->  <dependency>  <groupId>org.hamcrest</groupId>  <artifactId>hamcrest-library</artifactId>  <version>1.3</version>  <scope>test</scope>  </dependency> |
| To run the same test with diff parameters or diff inputs then use this jar for below anno  @ParameterizedTest  @ValueSource  @CSVSource  @MethodSource | <!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-params -->  <dependency>  <groupId>org.junit.jupiter</groupId>  <artifactId>junit-jupiter-params</artifactId>  <version>5.10.0</version>  <scope>test</scope>  </dependency> |

Running the test cases

Surefire plugin is mandatory, so that when u run the build it will run the tests [INFO] --- maven-surefire-plugin:2.22.0:test (default-test) @ testing-java-junit5 ---

See here surefire plugin is running the test cases

Maven commands

|  |  |
| --- | --- |
| mvn clean test //This is to clean and execute test cases using installed maven | .\mvnw.cmd test  //This is like we are running the tests using maven wrapper instead of installed maven software |
|  |  |

### **Gradle**

To run tests in gradle u can use gradle wrapper command for windows “./gradlew.bat test” we should use batch file for windows

plugins {

id 'java'

}

group 'guru.springframework'

version '1.0-SNAPSHOT'

sourceCompatibility = 1.8

repositories {

mavenCentral()

}

dependencies {

testCompile('org.junit.jupiter:junit-jupiter-api:5.3.1')

testCompile('org.junit.jupiter:junit-jupiter-engine:5.3.1')

}

test {

useJUnitPlatform()

testLogging.showStandardStreams = true

// if u don’t keep above nothing will be logged

testLogging {

events "passed", "skipped", "failed"

// if u miss above console will not be soo clear

}

}

Important notes on Junit

1. By default junit creates a new object of test class before every test method executes due to default behaviour of

@TestInstance(TestInstance.Lifecycle.PER\_METHOD)

If 10 test methods = 10 times test class object will be created after each new obj then only test method will be exec for isolation

If u want to only 1 test class obj to be created (irrespective of number of methods present in test class) then u can annotate the class with @TestInstance(TestInstance.Lifecycle.*PER\_CLASS*)

1. For every test method u can add a param named TestInfo which is an interface

public void t2(*TestInfo info*) {  
 System.*out*.println("Test method 2 is fired"+*info*.getTags());  
}

sample output

this is a cons

Before all method is fired

Test method 2 is fired[controller]

## Assertions

### Assertions with AssertJ

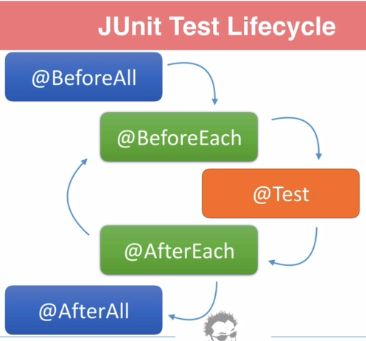
### Assertions with hamcrest

Hamcrest is pretty older as it is released in 2012 and no updates but tons of old code is already written in hamcrest

MatchAssert.AssertThat

## Annotations

|  |  |
| --- | --- |
| @Test | Marks a method as a test method |
| @TestInstance(TestInstance.Lifecycle.PER\_ *CLASS*) | This can be placed on top of a class and bec of which only once the test class obj will be created  In this case as only 1 instance, all the @BeforeAll, @AFterall methods can be non-static  @SpringBootTest Ⓐ  @TestInstance(TestInstance.Lifecycle.PER\_CLASS) Ⓑ  abstract class SpringDataJpaApplicationTests {  use case:- especially when containers starts again and again for every test method (then container will be created that many times) it will be more time consuming, |
| @TestInstance(TestInstance.Lifecycle.*PER\_METHOD*) | Why per method needs a static @BeforeAll method?  In this scenario before every test method exec a new test class will be created means cons will be exec, Means this @beforeAll must be exec before cons means it should / it can only be a static method |
| @ParameterizedTest  @ValueSource(strings = { "ab", "cd",})  [@NullSource](https://junit.org/junit5/docs/current/api/org.junit.jupiter.params/org/junit/jupiter/params/provider/NullSource.html)  [@EmptySource](https://junit.org/junit5/docs/current/api/org.junit.jupiter.params/org/junit/jupiter/params/provider/EmptySource.html)  [@NullAndEmptySource](https://junit.org/junit5/docs/current/api/org.junit.jupiter.params/org/junit/jupiter/params/provider/NullAndEmptySource.html):  @EnumSource(ChronoUnit.class)  @MethodSource("stringProvider") | Marks method as param test  @valueSource anno is to provide multiple inputs |
| @RepeatedTest(10 or some number) | Repeats test n times |
| @TestFactory | Test factory method for dynamic tests |
| @TestInstance | Used to configure test instance life cycle |
| @TestTemplate | Creates a template to be used by multiple test cases |
| @DisplayName | Human friendly name for test |
| @BeforeEach | Method to run before each test case |
| @AfterEach | Method to run after each test case |
| @BeforeAll | Static method to run before all test cases in current class |
| @AfterAll | Static method to run after all test cases in current class |
| @Nested | Creates a nested test class |
| @Tag | Declare ‘tags’ for filtering tests |
| @Disabled | Disable test or test class  @Disabled(value = "this test has been disabled to improve coverage")  // u can write why it has been disabled |
| @ExtendWith | Used to register extensions |
| @EnabledIfSystemProperty | @EnabledIfSystemProperty(named = "os.arch", matches = ".\*64.\*")  Run the test only if that matches |
|  | @EnabledIfEnvironmentVariable(named = "ENV", matches = "staging-server") |
| @EnabledIf("customCondition") | @EnabledIf("customCondition") void enabled() { // ... }  boolean customCondition() { return true; } |
| @DisabledIf("customCondition") | Based on the method output it will run |
| @EnabledIf("example.ExternalCondition#customCondition") void enabled() { // ... } | That method can be located even in another class to |



Assertions:- <https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html>

|  |  |
| --- | --- |
|  |  |
| assertThrows(class,Executable) | Assertions.*assertThrows*(RuntimeException.class,()->*greeting*.helloWorld("null")); |
| public static void assertEquals(short expected,  short actual,  [String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html?is-external=true) message) | That message will be displayed or executed only when failure condition is met |
| Grouped assertions | Person person=new Person(1L,"Manideep","kumar"); Assertions.*assertAll*("this is a grouped assertion checking",  ()-> *assertEquals*("Manideep1",person.getFirstName(),"First name match failed"),  ()-> *assertEquals*("kumar1",person.getLastName(),"Last name match failed")  ); |
| To check in how much time test cases are getting executed  Within that time test case should be executed | [**assertTimeout**](https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html#assertTimeout-java.time.Duration-org.junit.jupiter.api.function.ThrowingSupplier-java.util.function.Supplier-)([Duration](https://docs.oracle.com/javase/8/docs/api/java/time/Duration.html?is-external=true) timeout, [ThrowingSupplier](https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/function/ThrowingSupplier.html)<T> supplier, [Supplier](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html?is-external=true)<[String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html?is-external=true)> messageSupplier)  Asserts that execution of the supplied supplier completes before the given timeout is exceeded.  Here test case will not be aborted , it will be run completely  @Test @DisplayName("dont interrupt") public void testTimeoutWithoutAbortion(){  *assertTimeout*(Duration.*ofSeconds*(1),()->{  Thread.*sleep*(5000);  System.*out*.println("test case ran completely");  }); } |
| [**assertTimeoutPreemptively**](https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html#assertTimeoutPreemptively-java.time.Duration-org.junit.jupiter.api.function.Executable-) **– means test case will be prevented to run beyond certain time**  Here if within that time test case is not executed then test case will be aborted | [**assertTimeoutPreemptively**](https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html#assertTimeoutPreemptively-java.time.Duration-org.junit.jupiter.api.function.Executable-)([Duration](https://docs.oracle.com/javase/8/docs/api/java/time/Duration.html?is-external=true) timeout, [Executable](https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/function/Executable.html) executable)  Asserts that execution of the supplied executable completes before the given timeout is exceeded  @Test @DisplayName("test case will be aborted within time") public void testTimeoutWithAbortion(){  *assertTimeoutPreemptively*(Duration.*ofSeconds*(1),()->{  Thread.*sleep*(5000);  System.*out*.println("test case ran completely");  }); } |
| Junit assumptions –  Assumptions.assumeTrue()  this is like  @ConditionalOnClass, @ConditionalOnBean | If that condition is satisfied then only continue to execute that test case, else ignore that test case  @Test @DisplayName("using assumptions") public void assumptionDemo(){  Assumptions.*assumeTrue*(System.*getenv*().get("JAVA\_HOME").contains("18"),"java version is not 11 and hence ignoring this test case");  System.*out*.println("test case ran successfully"); }  Here in this case, since condition is not satisfied testcase will be aborted and skipped |
| *@EnabledIfSystemProperty*  @EnabledIfEnvironmentVariable  Run the test only if condition matched else ignore or don’t run | @Test *// @EnabledIfSystemProperty(named = "java.specification.version",matches = "17")* @EnabledIfEnvironmentVariable(named = "java.specification.version",matches = "18")  void enabledIf() {  System.*out*.println("i ran as condition got matched...");  } |

@Nested



In john code refer

<https://github.com/springframeworkguru/testing-java-junit5/blob/nested-tests2/src/test/java/guru/springframework/sfgpetclinic/services/map/OwnerMapServiceTest.java>

@TestInstance(TestInstance.Lifecycle.PER\_CLASS) // Optional for reuse across nested tests

public class VendingMachineTest {

private VendingMachine vendingMachine;

private int initialAmount;

@BeforeAll

public void setUp() {

vendingMachine = new VendingMachine();

initialAmount = 100;

vendingMachine.insertMoney(initialAmount);

}

@Nested

public class WithMoneyInserted {

@Test

public void testSelectProduct() {

// Test selecting a product with money inserted

}

@Test

public void testReturnMoney() {

// Test returning money when money is already inserted

}

}

@Nested

public class NoMoneyInserted {

@Test(expected = IllegalStateException.class)

public void testSelectProduct() {

// Test selecting a product without money inserted (expected exception)

}

@Test

public void testReturnMoney() {

// Test returning money when no money is inserted

}

}

}

@Tag

In maven to run only those tests annotated with tags when we run maven test, then we should add <!--<groups>model</groups>-->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.22.0</version>

<configuration>

<argLine>

--illegal-access=permit

</argLine>

<groups>model</groups> // means this will run those test cases annotated with @Tag(“model”)

<!--<excludedGroups>controllers</excludedGroups>--> means these tags will not be executed

<!This means only tests which are tagged with model will be executed @Tag(“model”)> -><excludedGroups>controllers</excludedGroups>-->

</configuration>

</plugin>

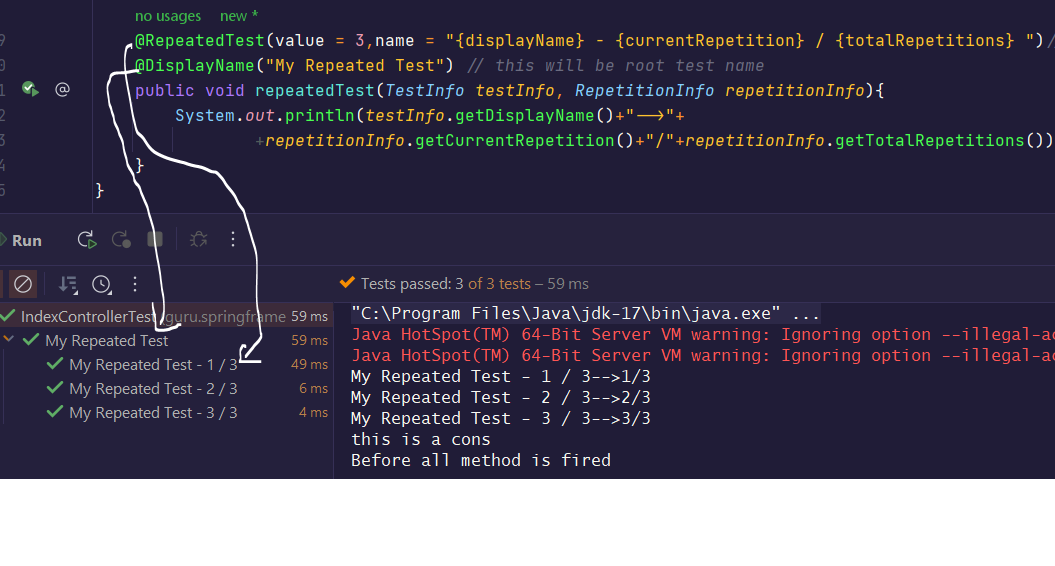
|  |  |
| --- | --- |
| @Tag("model") in intellij u can run all those tagged classes at once using configurations | If parent is annotated with @Tag all the child’s doesn’t need to add that tag again as those are childs  @Tag("service") interface parent{ } class Child1Test implements parent{  @Test  public void t1(){  System.*out*.println("child 1");  } }  class Child2Test implements parent{  @Test  public void t1(){  System.*out*.println("child 2");  } }  //Now if we run that @tag all those 2 child’s will also will run |

@RepeatedTest

If u use this anno then no need of @Test annotation also, and u can use these 3 interfaces

|  |  |
| --- | --- |
| Interface name |  |
| *TestInfo* | Provides info about test name, method name, class, tags |
| *RepetitionInfo* | Provides info about current test repetition number |

@RepeatedTest(value = 3,name = "internal method - {currentRepetition} / {totalRepetitions} ")*// this will be individual test name*@DisplayName("My Repeated Test Root Test ") *// this will be root test name*public void repeatedTest(*TestInfo testInfo*, *RepetitionInfo repetitionInfo*){  
 System.*out*.println("input is "+**allInputs**.get(*repetitionInfo*.getCurrentRepetition()));  
 System.*out*.println(*testInfo*.getDisplayName()+"-->"+  
 +*repetitionInfo*.getCurrentRepetition()+"/"+*repetitionInfo*.getTotalRepetitions());  
}



Sample another

@RepeatedTest(value = 3,name = "internal method - {currentRepetition} / {totalRepetitions} ")*// this will be individual test name*@DisplayName("My Repeated Test Root Test ") *// this will be root test name*public void repeatedTest(*TestInfo testInfo*, *RepetitionInfo repetitionInfo*){  
 System.*out*.println(*testInfo*.getDisplayName()+"-->"+  
 +*repetitionInfo*.getCurrentRepetition()+"/"+*repetitionInfo*.getTotalRepetitions());  
}

Output:-

internal method - 1 / 3-->1/3

internal method - 2 / 3-->2/3

internal method - 3 / 3-->3/3

this is a cons

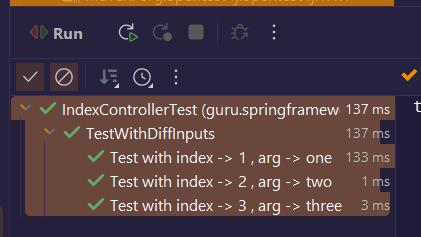
Before all method is fired

@ValueSource ,@parameterizedTest

Means running the same test with diff parameters

@ValueSource disadvantage is single test can be run with only single input at a time

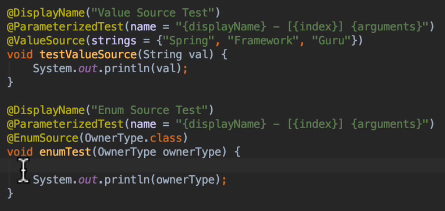
@DisplayName("TestWithDiffInputs") *//This will be the root test name*@ParameterizedTest(name = " Test with index -> {index} , arg -> {arguments}")*// sub test name*@ValueSource(strings = {"one","two","three"} )  
public void testMethodWithDiffInputs(String *input*){  
 System.*out*.println(*input*);  
}



// now this same test is executed with diff parameters

@ParameterizedTest  
@ValueSource(strings = {"true","false"})  
void processCreationFormMockitoStyle(boolean *hasErrors*) {  
 System.***out***.println("test method invoked wiht value -->"+*hasErrors*);  
 if(*hasErrors*){  
 Mockito.*when*(**bindingResult**.hasErrors()).thenReturn(*hasErrors*);  
 }else{  
 Mockito.*when*(**ownerService**.save(ArgumentMatchers.*argThat*(*arg*->*arg*.getLastName().equals("vv"))))  
 .thenReturn(**owner**);  
 }  
 System.***out***.println(**ownerController**.processCreationForm(**owner**,**bindingResult**));  
}

### @EnumSource



### @CSVSource

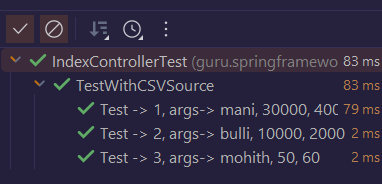
In @CSVSource in **single iteration/execution we can pass multiple inputs**

This is the most recommended as we can hardcode multiple inputs / input and output

@CSVSource means single test can be run with single input at a time

@DisplayName("TestWithCSVSource")*// This will be root test name*@ParameterizedTest(name = "Test -> {index}, args-> {arguments} ")*// sub test name*

@CsvSource({"mani,30000,400","bulli,10000,2000","mohith,50,60"})  
public void csvSource(String *empName*, int *salary*,int *bonus*){  
 System.*out*.printf("\n name-> %s sal-> %d bonus-> %d ",*empName*,*salary*,*bonus*);  
}



@CSVFileSource

You have to keep the “\*.csv” in src/test/resources

@DisplayName("@CSVFileSource anno demo")  
@ParameterizedTest(name = "Test {index} args--> {arguments}")  
@CsvFileSource(files = "src/test/resources/index.csv",numLinesToSkip = 1)  
public void csvFromFile(String *empName*,int *sal*,int *bonus*){  
 System.*out*.printf("\n empName -> %s sal -> %d bonus -> %d",*empName*,*sal*,*bonus*);  
}

### @NullSource

this annotation will provide 1 null value

[@EmptySource](https://junit.org/junit5/docs/current/api/org.junit.jupiter.params/org/junit/jupiter/params/provider/EmptySource.html)

[@NullAndEmptySource](https://junit.org/junit5/docs/current/api/org.junit.jupiter.params/org/junit/jupiter/params/provider/NullAndEmptySource.html)

### @MethodSource

Means same will be executed with diff inputs all those inputs will come from a method

With 1 input at a time

@ParameterizedTest

@MethodSource("stringProvider")

void testWithExplicitLocalMethodSource(String argument) {

assertNotNull(argument);

}

static Stream<String> stringProvider() {

return Stream.of("apple", "banana");

}

With Multiple inputs in single execution

Program -2 this is a best way we can pass multiple inputs / input and output in single shot

@DisplayName("@MethodSource With multiple inputs")  
@ParameterizedTest(name= "Test - {index} , arg- {arguments}")  
@MethodSource("getAllInputs")// this is a method name  
public void methodSourceWithMultipleInputs(String *name*, int *salary*, int *bonus*){  
 System.*out*.printf("\n name--> %s, salary--> %d, bonus -> %d",*name*,*salary*,*bonus*);  
}  
public *Stream*<*Arguments*> getAllInputs(){  
 return *Stream*.*of*(  
 *arguments*("mani",10000,1000), // or u can use Arguments.of()  
 *arguments*("santu",20000,2000),  
 *arguments*("mani",30000,3500)  
 );  
}

Here we will get data from a method where that method is returning employee class type objects

@DisplayName("Test @MethodSource ")  
@ParameterizedTest(name = "Test - {index}")  
@MethodSource("getAllEmployees")  
public void testMethodSource(Employee *emp*) {  
 System.*out*.println(*emp*);  
}  
  
public *Stream*<Employee> getAllEmployees() {  
 return *Stream*.*of*(  
 new Employee("mani", 10, 10),  
 new Employee("santu", 30, 20)  
 );  
}

## Spring boot testing

|  |  |
| --- | --- |
| @SpringBootTest | This annotation will search for main class which is annotated with @SpringBootApplication  this will create the create and does auto cfg and container starts as usual |
|  |  |

@SpringBootTest Ⓐ

@TestInstance(TestInstance.Lifecycle.PER\_CLASS) Ⓑ

abstract class SpringDataJpaApplicationTests {

@Autowired Ⓒ

UserRepository userRepository; Ⓒ

@BeforeAll Ⓓ

void beforeAll() { Ⓓ

userRepository.saveAll(generateUsers()); Ⓓ

}

The @SpringBootTest annotation, added by Spring Boot to the initially created class, tells Spring Boot to search the main configuration class (the @SpringBootApplication annotated class, for instance) and create the ApplicationContext to be used in the tests. Recall that the @SpringBootApplication annotation added by Spring Boot to the class containing the main method will enable the Spring Boot autoconfiguration mechanism, enable the scan on the package where the application is located, and allow the registration of extra beans in the context.

Ⓑ Using the @TestInstance(TestInstance.Lifecycle.PER\_CLASS) annotation, we ask JUnit 5 to create a single instance of the test class and reuse it for all test methods. This will allow us to make the @BeforeAll and @AfterAll annotated methods non-static and to directly use the autowired UserRepository instance field inside them.