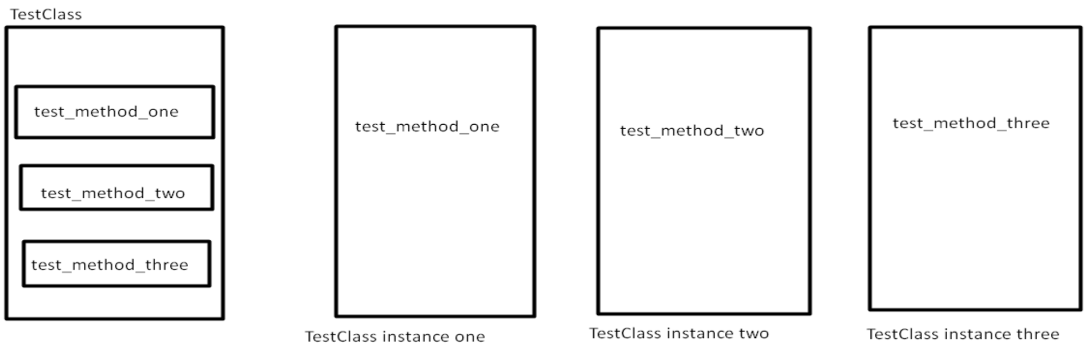
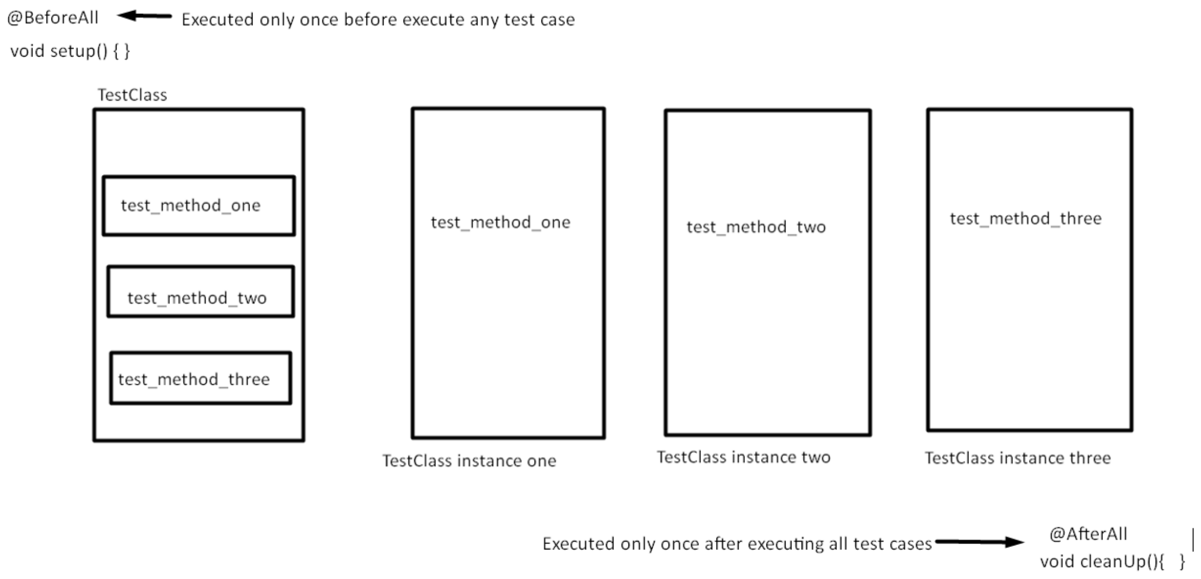
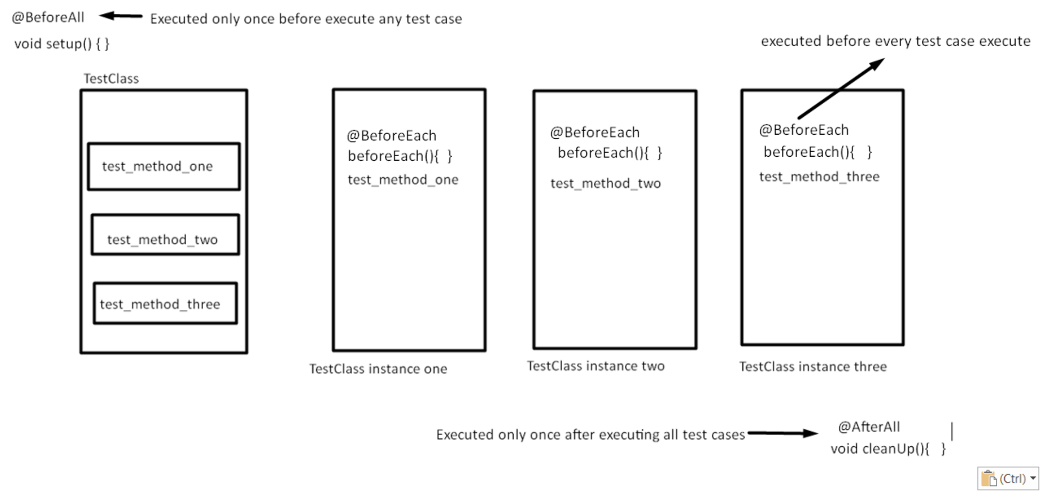
Life cycle methods:

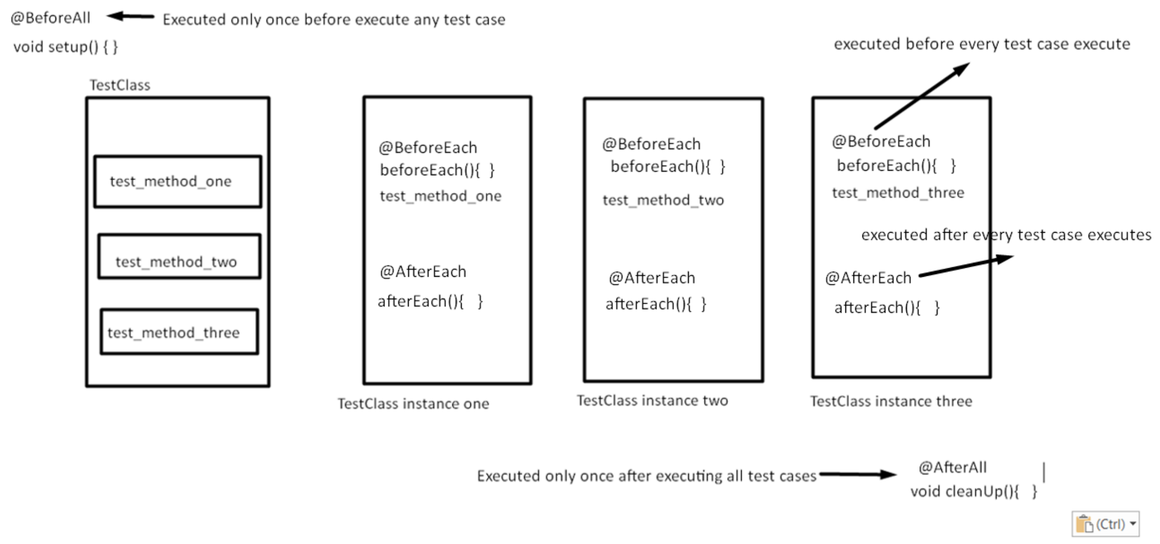
When we execute test methods in test class, by default **for each test method JUNIT will create new instance of class.** The execution of order is by default is not obvious.



Life cycle methods:







How to disable unit test ?

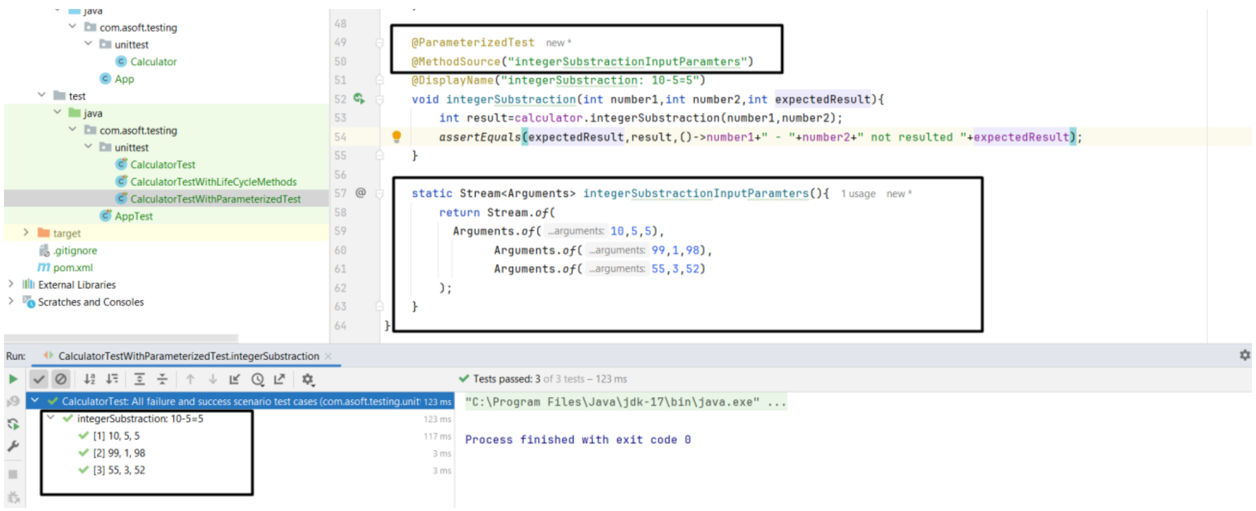
```
@Disabled("Not yet implemented")
   @Test
   void testIntegerDivision_WhenFourDivideByZero_ShouldResultArithmeticException(){
      fail("Not yet implemented");
   }
```

**Assert an Exception**

```
@Test

@DisplayName("testIntegerDivision- DivideByZero scenario")

void testIntegerDivision_WhenFourDivideByZero_ShouldResultArithmeticException(){

        String expectedExceptionMessage="/ by zero";

        ArithmeticException actualException= assertThrows(ArithmeticException.class,()->{

                calculator.integerDivision(4,0);

        });

        assertEquals(expectedExceptionMessage,actualException.getMessage());

}
```

**How to write unit test that accept input parameter:**

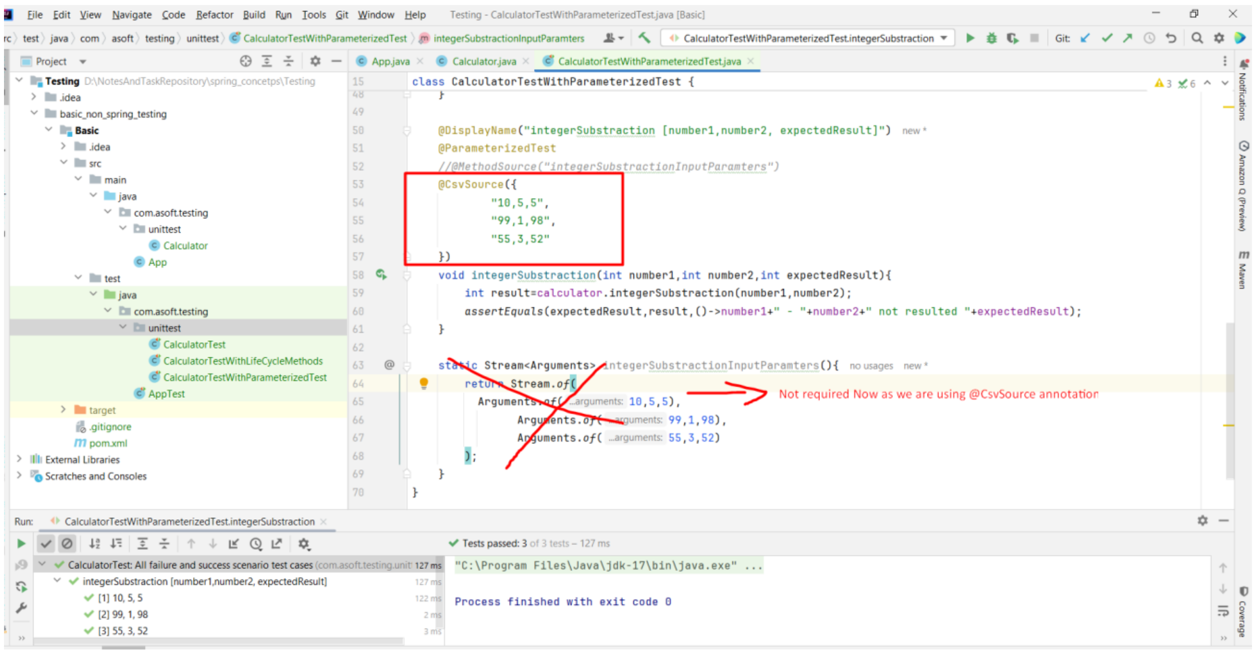
Using **@ParameterizedTest** and **@MethodSource** annotation. Here we need to provide proper method that will used to prepare input parameters.



```
    @ParameterizedTest
    @MethodSource("integerSubstractionInputParamters")
    @DisplayName("integerSubstraction: 10-5=5")
    void integerSubstraction(int number1,int number2,int expectedResult){
        int result=calculator.integerSubstraction(number1,number2);
        assertEquals(expectedResult,result,()->number1+" - "+number2+" not resulted "+expectedResult);
    }

    static Stream<Arguments> integerSubstractionInputParamters(){
        return Stream.of(
          Arguments.of( 10,5,5),
                Arguments.of( 99,1,98),
                Arguments.of( 55,3,52)
        );
    }
}
```

```
Tests passed: 3 of 3 tests – 123 ms

CalculatorTest: All failure and success scenario test cases (com.asoft.testing.unit  123 ms
  integerSubstraction: 10-5=5   123 ms
    [1] 10, 5, 5   117 ms
    [2] 99, 1, 98   3 ms
    [3] 55, 3, 52   3 ms

"C:\Program Files\Java\jdk-17\bin\java.exe" ...

Process finished with exit code 0
```

**How to pass multiple parameter with comma (i.e ,) separated values?**



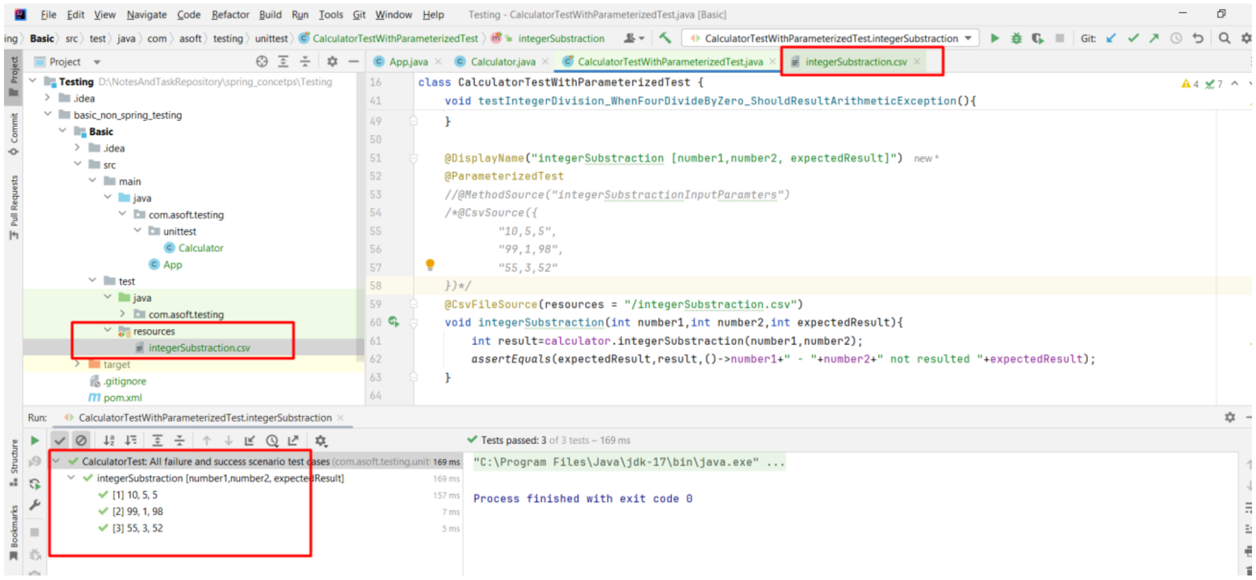
```java
class CalculatorTestWithParameterizedTest {
    }

    @DisplayName("integerSubstraction [number1,number2, expectedResult]")
    @ParameterizedTest
    //@MethodSource("integerSubstractionInputParamters")
    @CsvSource({
            "10,5,5",
            "99,1,98",
            "55,3,52"
    })
    void integerSubstraction(int number1,int number2,int expectedResult){
        int result=calculator.integerSubstraction(number1,number2);
        assertEquals(expectedResult,result,()->number1+" - "+number2+" not resulted "+expectedResult);
    }

    static Stream<Arguments> integerSubstractionInputParamters(){
        return Stream.of(
          Arguments.of( 10,5,5),
                Arguments.of( 99,1,98),
                Arguments.of( 55,3,52)
        );
    }
}
```

Not required Now as we are using @CsvSource annotation

```
Tests passed: 3 of 3 tests – 127 ms

CalculatorTest: All failure and success scenario test cases  127 ms
  integerSubstraction [number1,number2, expectedResult]      127 ms
    [1] 10, 5, 5                                             122 ms
    [2] 99, 1, 98                                              2 ms
    [3] 55, 3, 52                                              3 ms

"C:\Program Files\Java\jdk-17\bin\java.exe" ...

Process finished with exit code 0
```

**@ParameterizedTest with csv file:**

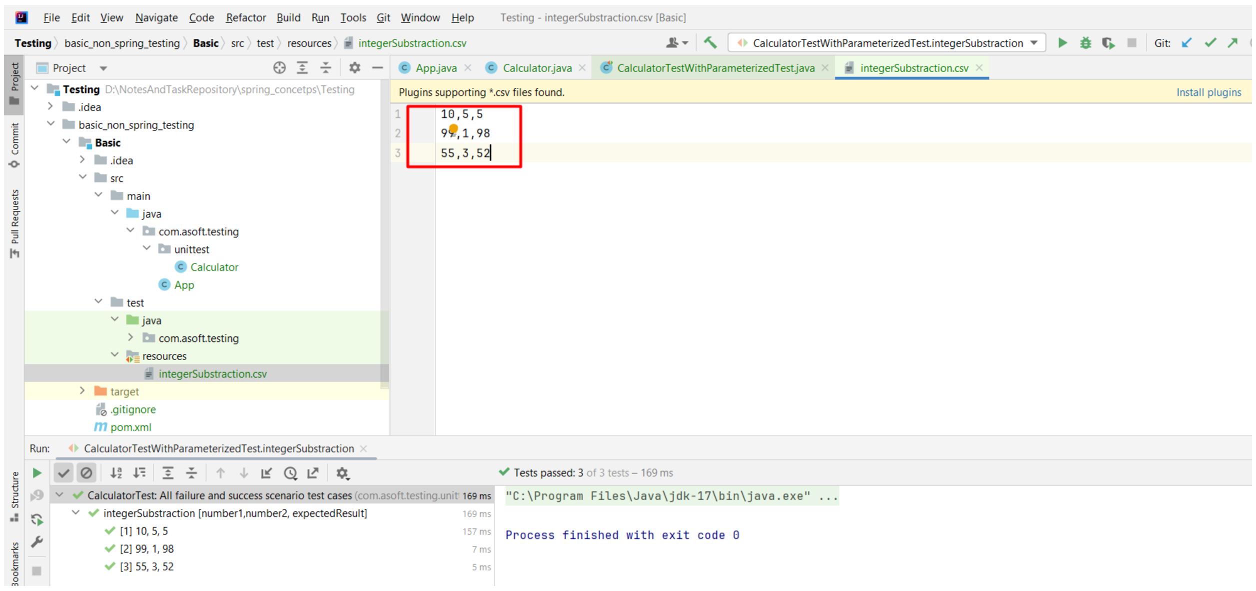
Use **@CsvFileSource** annotation.



```java
class CalculatorTestWithParameterizedTest {
    void testIntegerDivision_WhenFourDivideByZero_ShouldResultArithmeticException(){
    }

    @DisplayName("integerSubstraction [number1,number2, expectedResult]")
    @ParameterizedTest
    //@MethodSource("integerSubstractionInputParamters")
    /*@CsvSource({
            "10,5,5",
            "99,1,98",
            "55,3,52"
    })*/
    @CsvFileSource(resources = "/integerSubstraction.csv")
    void integerSubstraction(int number1,int number2,int expectedResult){
        int result=calculator.integerSubstraction(number1,number2);
        assertEquals(expectedResult,result,()->number1+" - "+number2+" not resulted "+expectedResult);
    }
```

```
Tests passed: 3 of 3 tests – 169 ms

CalculatorTest: All failure and success scenario test cases  169 ms
  integerSubstraction [number1,number2, expectedResult]      169 ms
    [1] 10, 5, 5                                             157 ms
    [2] 99, 1, 98                                              7 ms
    [3] 55, 3, 52                                              5 ms

"C:\Program Files\Java\jdk-17\bin\java.exe" ...

Process finished with exit code 0
```

Csv file:


File Edit View Navigate Code Refactor Build Run Tools Git Window Help
Testing - integerSubstraction.csv [Basic]
Testing 〉 basic_non_spring_testing 〉 Basic 〉 src 〉 test 〉 resources 〉 integerSubstraction.csv
CalculatorTestWithParameterizedTest.integerSubstraction
Project
App.java
Calculator.java
CalculatorTestWithParameterizedTest.java
integerSubstraction.csv
Plugins supporting *.csv files found.
Install plugins
Testing D:\NotesAndTaskRepository\spring_concepts\Testing
.idea
basic_non_spring_testing
Basic
.idea
src
main
java
com.asoft.testing
unittest
Calculator
App
test
java
com.asoft.testing
resources
integerSubstraction.csv
target
.gitignore
pom.xml
Run: CalculatorTestWithParameterizedTest.integerSubstraction

```
10,5,5
99,1,98
55,3,52
```

```
Tests passed: 3 of 3 tests – 169 ms
CalculatorTest: All failure and success scenario test cases (com.asoft.testing.unit 169 ms
  integerSubstraction [number1,number2, expectedResult]   169 ms
    [1] 10, 5, 5    157 ms
    [2] 99, 1, 98     7 ms
    [3] 55, 3, 52     5 ms
```

```
"C:\Program Files\Java\jdk-17\bin\java.exe" ...

Process finished with exit code 0
```

**@ParameterizedTest + @ValueSource**

Used when method has single parameter.

Example: Below method will execute 3 times with given parameter

**@ParameterizedTest**

**@ValueSource(strings = {"abhishek","rudra","aai"})**

void valueSourceDemo(String **name**){

System.out.println(name);

assertNotNull(name);

}

**How to Repeat Test?**

**@RepeatedTest(3) // With annotation we can invoke below test method 3 times.**

void testIntegerDivision_WhenFourDivideByTwo_ShouldResultTwo(){

int result=calculator.integerDivision(4,2);

assertEquals(2,result,"4/2 is not resulting correct result");

}

**<u>Methods Order - Random order</u>**

**How to apply order to all test methods in class?**

Apply Random order using **@TestMethodOrder(MethodOrderer.Random.class)** at class level

```
@TestMethodOrder(MethodOrderer.Random.class)

public class MethodOrderedRandomlyTest{

  @Test

  void testA(){

    System.out.println("Running test A");

  }

  @Test

  void testB(){

    System.out.println("Running test B");

  }

  @Test

  void testC(){      System.out.println("Running test C");    }

}
```

Apply **@TestMethodOrder(MethodOrderer.MethodName.class).** Just replace previous annotation with …> Junit will sort execution order by method name

**Custom order:**

```
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)

public class MethodOrderedByOrderIndexTest{

  @Order(1)

  @Test

  void testA(){      System.out.println("Running test A");    }

  @Order(3)

  @Test

  void testB(){      System.out.println("Running test B");    }

  @Order(2)

  @Test

  void testC(){      System.out.println("Running test C");    }

}
```

**How to order execution of test classes? >** Write a **@Order** Annotation on classes and configure **junit.jupiter.testclass.order.default=org.junit.jupiter.api.ClassOderer$OrderAnnotation in junit-platform.properties** in test **resource directory. NOTE THAT junit.jupiter.testclass.order.default=org.junit.jupiter.api.ClassOderer$OrderAnnotation is property equals to @TestMethodOrder(MethodOrderer.OrderAnnotation.class) in previous example.** Used while integration testing. For example, we have in below flow, first create product then place an order and make a payment. So, maintain execution order in that way.

```
@Order(1)

class ProductServiceTest {

        @Test

        void createProduct(){ }

}

@Order(3)

class PaymentService {

        @Test

        void makeAPayment(){}

}

@Order(2)

class OrderServiceTest {

        @Test

        void placeAndOrder(){}

}
```

<u>**How to create only one instance of TestClass?**</u>

We can use **@TestInstance(TestInstance.Lifecycle.PER_CLASS)** annotation on class. By default for each test method the new TestClass instance created.

```
@TestInstance(TestInstance.Lifecycle.PER_ CLASS)

public class TestClass{

  public TestClass() {}

  @Test

  void test_m1(){ }
  @Test
  void test_m2(){ }
}
```

# Mokito

Mokito is f/w that used to create mock /fake objects.

**Example**: We have below service. We want to test ONLY **createUser method** as we are writing JUNIT test. Not the implementation of **usersRepository**.**save**(user);

```
public class UserServiceImpl implements UserService {
  UsersRepository usersRepository;
  EmailVerificationService emailVerificationService;
  public UserServiceImpl(UsersRepository usersRepository) {
    this.usersRepository = usersRepository;
  }
  @Override
  public User createUser(String firstName, String lastName, String email, String password, String repeatPassword) {
    if(firstName == null || firstName.trim().length() == 0) {     throw new IllegalArgumentException("User's first name is empty");
    }
    User user = new User(firstName, lastName, email, UUID.randomUUID().toString());
    boolean isUserCreated;
    try {          isUserCreated = usersRepository.save(user);      } catch (RuntimeException ex) {
      throw new UserServiceException(ex.getMessage());
    }
    if(!isUserCreated) throw new UserServiceException("Could not create user");
    return user;
  }
}
```

**Unit Test for above class:**

```
@ExtendWith(MockitoExtension.class)
class UserServiceTest {
    @InjectMocks
    UserServiceImpl userService;
    @Mock
    UsersRepository usersRepository;
    String firstName;
    @BeforeEach
    void init() {
        firstName = "Sergey";
    }
    @DisplayName("User object created")
    @Test
    void testCreateUser_whenUserDetailsProvided_returnsUserObject() {
        //Arrange
        Mockito.when(usersRepository.save(Mockito.any(User.class)))
                .thenReturn(false);
        // Act
        User user = userService.createUser(firstName);
        // Assert
        assertNotNull(user, "The createUser() should not have returned null");
        assertEquals(firstName, user.getFirstName(), "User's first name is incorrect.");
    }
}
```

**How to verify that mock object invoke method or verify how many time it invoke the method?**

Added below line to end of testCreateUser_whenUserDetailsProvided_returnsUserObject()  method

```
Mockito.verify(usersRepository,Mockito.times(1)).save(Mockito.any(User.class));
```

**Exception stubbing: It means making mock object to throw exception when it invoke. For example: In previous code when** userService.**createUser**(firstName); is invoked our mocke object return “true” but instead of that now we need exception.

Code: NOTE Ref UserService class for CreateUser method implementation

```
@Test

void testCreateUser_whenFirstNameIsEmpty_throwsIllegalArgumentException() {

        Mockito.when(usersRepository.save(Mockito.any(User.class))).thenThrow(RuntimeException.class);

        assertThrows(UserServiceException.class,()->{

                userService.createUser(firstName, lastName, email, password, repeatPassword);

        });

}
```

**How to stub void methods?**

**>> Remember that in scheduleEmailConfirmation method is void in EmailNotificationService class.**

```
@Test

@DisplayName("EmailNotificationException is handled")

void testCreateUser_whenEmailNotificationExceptionThrown_throwsUserServiceException(){

        Mockito.when(usersRepository.save(Mockito.any(User.class))).thenReturn(true);

        //Invoking void scheduleEmailConfirmation...

        Mockito.doThrow(EmailNotificationServiceException.class)

                        .when(emailVerificationService)

                        .scheduleEmailConfirmation(Mockito.any(User.class));

        assertThrows(EmailNotificationServiceException.class,()->{

          userService.createUser(firstName, lastName, email, password, repeatPassword);

        });

        Mockito.verify(emailVerificationService,Mockito.times(1))

                        .scheduleEmailConfirmation(Mockito.any(User.class));

}
```

# Spring boot Testing REST Controller

Code:

**@WebMvcTest**

public class UsersControllerWebLayerTest {

}

What does **@WebMvcTest** do?

The @WebMvcTest indicates that the tests within that class are focused on testing MVC controllers in your Spring application. It loads only the relevant parts of the Spring application context needed for testing **MVC controllers**. This annotation **does not load the entire application context**, which can **improve the performance of the tests**. It automatically **configures and mocks Spring MVC infrastructure, such as DispatcherServlet**, allowing you to focus on testing your **controllers in isolation**.

**It scans only the components relevant to Spring MVC, such as controllers, and excludes other components like services or repositories.**

**@MockBean: (Belons to spring)**

@MockBean is a Spring Boot annotation **used in integration tests** (typically with **@SpringBootTest** or @WebMvcTest). It creates a **mock of a bean or a component in the Spring application context**. @MockBean is often **used to mock services, repositories, or other Spring components that your class under test depends on**. Mocked beans created with **@MockBean are added to the Spring application context during test execution**.

*If you are using @MockBean then you need to use @Autowire to inject the bean.*

Summary: **Used to create mock object of in spring boot context and after creating those object/beans will be put into application context.**

**Example:** Writing integration test only for WebLayer.

**@Mock: (Belongs to** Mockito framework)

@Mock is from the Mockito framework, not Spring. It creates a mock object of a class or interface. **@Mock is typically used in unit tests**, not in integration tests. *If you are using @Mock then you need to use @InjectMock to inject the bean.*

**In summary,** @MockBean is Spring-specific and used for mocking Spring beans in integration tests, while @Mock is a general-purpose mocking annotation provided by Mockito for creating mock objects in unit tests.

**@WebMvcTest vs @SpringBootTest**

@WebMvcTest is used for testing a specific slice of the application (typically web layer) while @SpringBootTest **loads the whole application context** including all beans and configurations. @SpringBootTest can be used to perform end-to-end testing, where you want to test the behavior of the entire application, including its integration with the database, external services, and other components. It starts the embedded servlet container (if the application is a web application) and provides a fully functional environment for testing.

**In summary,** @WebMvcTest is used for focused testing of Spring MVC controllers, while @SpringBootTest is used for broader integration testing of the entire Spring application, including all beans and configurations. Choose

@WebMvcTest for testing MVC components in isolation, and @SpringBootTest for end-to-end integration testing of the application.

**Example of @WebMvcTest**

```
@WebMvcTest(controllers = UsersController.class,
excludeAutoConfiguration = SecurityAutoConfiguration.class)
public class UsersControllerWebLayerTest {
   @Autowired
   private MockMvc mockMvc;
   @MockBean
   private UsersService usersService;
   UserDetailsRequestModel requestModel;
   UserDto userDto;
   @BeforeEach
   void beforeTest(){
      requestModel=new UserDetailsRequestModel();
      requestModel.setFirstName("Abhishek");
      requestModel.setLastName("Patil");
      requestModel.setEmail("abc@gmail.com");
      requestModel.setPassword("1234");
      requestModel.setRepeatPassword("1234");
      userDto=new ModelMapper().map(requestModel,UserDto.class);
      userDto.setUserId(UUID.randomUUID().toString());
   }
   @Test
   void testCreateUser_whenValidUserDetailProvided_returnCreatedUserDetail() throws Exception {
      //Arrange
      Mockito.when(usersService.createUser(Mockito.any(UserDto.class))).thenReturn(userDto);
      RequestBuilder requestBuilder=MockMvcRequestBuilders
            .post("/users")
            .accept(MediaType.APPLICATION_JSON)
            .contentType(MediaType.APPLICATION_JSON)
            .content(new ObjectMapper().writeValueAsString(requestModel));



        //act
        MvcResult result=mockMvc.perform(requestBuilder).andReturn();
        String response=result.getResponse().getContentAsString();
        UserRest userRest=new ObjectMapper().readValue(response,UserRest.class);
        //Assert
        Assertions.assertNotNull(userRest);
        Assertions.assertEquals(requestModel.getFirstName(),userRest.getFirstName(),
            "The returned users first name is incorrect");
    }
}
```

# Testing all layers of Spring boot application.

**@SpringBootTest:**

- @SpringBootTest is a broader annotation used for integration testing the entire Spring application.
- It loads the complete application context, including all beans and configurations.
- @SpringBootTest can be used to perform end-to-end testing, where you want to test the behavior of the entire application, including its integration with the database, external services, and other components.
- It starts the embedded servlet container (if the application is a web application) and provides a fully functional environment for testing.
- @SpringBootTest is more heavyweight compared to @WebMvcTest, as it loads the entire application context and might take longer to execute tests.

@SpringBootTest(**webEnvironment = SpringBootTest.WebEnvironment.MOCK**)

//webEnvironment: It is default type of web environment. If you don't define then it is MOCK. And MOCK environment not load entire spring context by default. It will not start embedded server by default.

@SpringBootTest(**webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT**)

Embedded server will start on port number that is defined in application.properties and load entire spring context.

**NOTE: WE CAN USE @SpringBootTest WITH AN DEFAULT CONFIGURATION TO DEVELOP UNIT TEST. AS YOU KNOW WITH DEFAULT CONFIGURATION IT DOES NOT START WHOLE CONTEXT.**

**Q.** Let say your spring boot application main class present in package com.luv2code.**component** and your test class present in com.luv2code.**test.** Then does @SpringBootApplication annotation load the context for you?

> No, To fix this problem you need to do below configuration in your test class

**@SpringBootTest(classes= MySampleExampleApplication.class)//Provide your application main class name**

public classes **UserControllerTest**{ }

**How to redefine configuration while testing? Or how to use alternative configuration or What is @TestPropertySource annotation.**

**@TestPropertySource** annotation allow us manually override defined properties in default file and also allow us to add new file. I below example application.properties and **application-test.properties both will get loaded. But application-test.properties will have high precedence over any other properties. Ex: server.port:8080 defined in application.properties and server.port:9999 defined in application-test.properties, then test will consider port no as 9999.**

Ex:

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT)
@TestPropertySource(locations="/application-test.properties")
public class UsersControllerIntegrationTest{
  @Test
  void contextLoad(){
  }
}
```

**TestRestTemplate:**

**TestRestTemplate** is a class provided by the Spring Framework for testing RESTful services in **integration tests**.

**Integration with Spring Boot:** TestRestTemplate is particularly useful in Spring Boot applications. When you use **@SpringBootTest, a TestRestTemplate instance is automatically configured** and available for use in your test classes.

Advantage of using **TestRestTemplate** is when user authentication involved and we need to include username and password.

**Full Completed class:**

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
//@TestPropertySource(locations="/application-test.properties"
//,properties = "server.port=7777")
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
@TestInstance(TestInstance.Lifecycle.PER_CLASS) // used as we need only one instance of test class
//the reason is we are using authorizationToken instance variable in 3 and 4 test case.
public class UsersControllerIntegrationTest {
```

```
@Autowired
private TestRestTemplate testRestTemplate;
private String authorizationToken;
@Test
@Order(1)
void testCreateUser_whenValidDetailsProvided_returnUserDetails() throws JSONException {
   //Arrange
   JSONObject userDetailRequest=new JSONObject();
   userDetailRequest.put("firstName","Abhishek");
   userDetailRequest.put("lastName","Patil");
   userDetailRequest.put("email","abc@gmail.com");
   userDetailRequest.put("password","1234");
   userDetailRequest.put("repeatPassword","1234");

   HttpHeaders headers=new HttpHeaders();
   headers.setContentType(MediaType.APPLICATION_JSON);
   headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));

   HttpEntity<String> request=new HttpEntity<>(userDetailRequest.toString(),headers);

   //act
   ResponseEntity<UserRest> createdUserDetailEntity
         =testRestTemplate.postForEntity("/users",request,UserRest.class);
   UserRest userRest=createdUserDetailEntity.getBody();

   //assert
   Assertions.assertEquals(HttpStatus.OK,createdUserDetailEntity.getStatusCode());
   Assertions.assertEquals(userDetailRequest.get("firstName"),userRest.getFirstName()
   ,"Returned User first name is incorrect");
   Assertions.assertEquals(userDetailRequest.get("lastName"),userRest.getLastName()
         ,"Returned last first name is incorrect");
```

```
    }

    //testing protected API's -test case-1
    @Test
    @Order(2)
    void testGetUsers_whenMissingJWT_return403(){
        //arrange
        HttpHeaders headers=new HttpHeaders();
        headers.set("Accept","application/json");

        HttpEntity request=new HttpEntity(null,headers);

        //act
        ResponseEntity<List<UserRest>> listResponseEntity=testRestTemplate
.exchange("/users", HttpMethod.GET, request, new ParameterizedTypeReference<List<UserRest>>() { });

        //List<UserRest> userRestList=listResponseEntity.getBody();

        //assert
        Assertions.assertEquals(HttpStatus.FORBIDDEN,listResponseEntity.getStatusCode(),
        "Http status code 403 forbidden should return");
    }

    //testing protected API's -test case-2-
    @Test
    @Order(3)
    void testUserLogin_whenValidCredentialsProvided_returnJWTAuthorizationHeader() throws JSONException {
        //arrange
        JSONObject loginCredential=new JSONObject();
        loginCredential.put("email","abc@gmail.com");
        loginCredential.put("password","1234");
        HttpEntity request=new HttpEntity(loginCredential.toString());

        //act
        ResponseEntity response=testRestTemplate.postForEntity("/users/login" ,request  ,null);

        authorizationToken=response.getHeaders()
            .getValuesAsList(SecurityConstants.HEADER_STRING).get(0);

        //assert
        Assertions.assertEquals(HttpStatus.OK,response.getStatusCode());
        Assertions.assertNotNull(authorizationToken
        ,"Response should contain JWT Authorization header");

        Assertions.assertNotNull(response.getHeaders()
                .getValuesAsList("UserID").get(0) ,"Response should contain UserID response header");
    }

    @Test
    @Order(4)
    void testGetUsers_whenValidJWTProvided_returnUsers(){
        //arrange
        HttpHeaders headers=new HttpHeaders();
        headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
        headers.setBearerAuth(authorizationToken);

        HttpEntity request=new HttpEntity(headers);

        //act
        ResponseEntity<List<UserRest>> response=testRestTemplate
.exchange("/users" ,HttpMethod.GET,request ,new ParameterizedTypeReference<List<UserRest>>(){});
```

```
        //asserts

        Assertions.assertEquals(HttpStatus.OK,response.getStatusCode());

        Assertions.assertTrue(response.getBody().size()==1);

    }

}
```

## Testing Data Layer: JPA Entity

**@DataJpaTest** is an annotation provided by Spring Boot for testing JPA repositories in Spring applications. Used for testing the interaction between your **Spring Data JPA repositories** and the **underlying database.**

- **@DataJpaTest** loads a limited Spring application context
- It excludes configuration related to web layers, security, and other components **not necessary for testing JPA repositories.**
- @DataJpaTest **scans for @Entity classes and configures Spring Data JPA repositories** in the test context. *It does not scan other components like controllers or services, focusing solely on JPA-related components.*
- @DataJpaTest is transactional it means test method runs within a transaction that is rolled back at the end of the test, ensuring that changes made during the test do not persist in the database.
- ***@DataJpaTest automatically configures an embedded database and sets up a TestEntityManager bean.*** The **TestEntityManager** provides methods for interacting with the database during tests, such as persisting entities or querying data.

Summary:

-Loads Application context ONLY with JPA-related component.

-By default, test methods is Transaction and will rollback when completes

-By default, in memory database is used

**TestEntityManager:** Used for interacting with database, it is optional for EntityManager.

**Test class:**

```
@DataJpaTest

public class UserEntityIntegrationTest {

    @Autowired

    private TestEntityManager testEntityManager;
    UserEntity userEntity;


    @BeforeEach
    void setup() {
        userEntity = new UserEntity();
        userEntity.setUserId(UUID.randomUUID().toString());
        userEntity.setFirstName("Sergey");
        userEntity.setLastName("Kargopolov");
        userEntity.setEmail("test@test.com");
        userEntity.setEncryptedPassword("12345678");
    }
    @Test
    void testUserEntity_whenValidUserDetailsProvided_shouldReturnStoredUserDetails() {
        // Act
        UserEntity storedUserEntity = testEntityManager.persistAndFlush(userEntity);
        // Assert
        Assertions.assertTrue(storedUserEntity.getId() > 0);
        Assertions.assertEquals(userEntity.getUserId(), storedUserEntity.getUserId());
    }
}
```

## Testing Data Layer: JPA Repositories

**Let see directly example**

```
@DataJpaTest
public class UserRepositoryTest {
    @Autowired
    private TestEntityManager testEntityManager;
    @Autowired
    private UsersRepository usersRepository; //we can auto wired becz we use @DataJpaTest
    private final String userId1 = UUID.randomUUID().toString();
    private final String email1 = "test@test.com";

    @BeforeEach
    void beforeEachSetup(){
        UserEntity userEntity = new UserEntity();
        userEntity.setUserId(userId1);
        userEntity.setEmail(email1);
        userEntity.setEncryptedPassword("12345678");
        userEntity.setFirstName("Abhishek");
        userEntity.setLastName("Patil");
        testEntityManager.persistAndFlush(userEntity);
    }

    @Test
    void testFindByEmail_whenGivenCorrectEmail_returnUserEntity(){
        UserEntity userEntity= usersRepository.findByEmail(email1);
        Assertions.assertNotNull(userEntity);
        Assertions.assertEquals(email1,userEntity.getEmail());
    }
}
```

## Conditional Testing

**Scenarios:**

**Run test on particular operating system or specific java version or certain environment like DEV, QA etc.**

| Annotation | Applied on | description |
|---|---|---|
| @Disabled | Class/method | Disable a test method |
| @EnabledOnOs | Class/method | Enable test when running on a given operating system |
| @EnabledOnJre | | Enable test for a given Java version |
| @EnabledForJreRange | | Enabled test for a given java version range |
| @EnabledifSystemProperty | | Enable test based on system property |
| @EnabledIfEnvironmentVariable | | Enable test based on environment variable |

```
@Test
@EnabledIfSystemProperty(named="PROPERTY_1", matches="VALUE1")
void testOnlySystemProperty(){
  ....
}

@Test

@EnabledIfEnvironmentVariable(named="PROPERTY_1", matches="VALUE1")

void testOnlyEnvironmentValirable(){

  ....

}
```

What is benefit of using @SpringBootTest annotation?

- You can have entire application context available for you. So that we can use any bean in your test class and access properties file just like normal. Example

```
@SpringBootTest

public class MyTest {

   @Value("${my.property}")

   private String myProperty;

   @Test

   public void testPropertyAccess() {

      System.out.println("My Property: " + myProperty);

      // Use myProperty in your test

   }

}
```

Different ways to access properties in test class if you are using @SpringBootTest annotation?

**Using @Value Annotation**: Ref above example

**Using Environment:**

```
@SpringBootTest

public class MyTest {

   @Autowired

   private Environment env;

   @Test

   public void testPropertyAccess() {

      String myProperty = env.getProperty("my.property");

      System.out.println("My Property: " + myProperty);

   }
}
```

**Using Test Properties File:**

You can create a separate properties file for tests (e.g., application-test.properties) and put your test-specific properties there. Spring Boot will automatically load these properties for your test environment. You can access them using either @Value or Environment.

**How to access properties value if you are not using @SpringBootTest annotation, instead of that using @WebMvcTest and @DataJpaTest to just test weblayer and data access layer.?**

If you're not using @SpringBootTest, but instead opting for more specialized testing annotations like @WebMvcTest or @DataJpaTest, you won't have the entire Spring context loaded, which means property injection via @Value won't work out of the box. However, you can still access properties by loading them manually or using a different approach.

**Using Test Property Source Annotation:**

let's define a **test.properties** file in the **src/test/resources** directory

```
# test.properties

test.property=myTestPropertyValue
```

```
@WebMvcTest

@TestPropertySource(locations = "classpath:test.properties")

public class MyTest {

  @Test

  public void testPropertyAccess() {

    System.out.println("My Property: " + System.getProperty("test.property"));

  }

}
```

**Using Environment:**

```
@WebMvcTest

public class MyTest {

  @Autowired

  private Environment env;

  @Test

  public void testPropertyAccess() {
        String myProperty = env.getProperty("my.property");
    }
}
```

**Using @TestConfiguration with @PropertySource:**

If you want to load properties programmatically for a specific test, you can create a test configuration class annotated with **@TestConfiguration and use @PropertySource** to load properties.

```
@Configuration
@PropertySource("classpath:test.properties")
public class TestConfig {
}
```

*Then you can use this configuration class in your test:*

```
@WebMcvTest
@ContextConfiguration(classes = TestConfig.class)
public class MyTest {
    @Test
    public void testPropertyAccess() {
        // Access properties using your preferred method
    }
}
```

# *Testing using reflection*

Use cases:

- Need to access private fields
  - Read the field values
  - set the field value.
- Invoke private methods.

Spring provide a utility class: ReflectionTestUtils

- this class allows you to get/set private fields directly
- allow us to invoke private methods

Example:

```
class CollegeStudent{
  private int id;
  private String firstName;
  private String lastName;
  //setter-getter
  private String getFirstNameAndId(){
    return getFirstName()+" "+getId();
  }
}
```

**<u>Test class:</u>**

```
public class TestRelectionDemo {
     @BeforeEach
    void beforeEachSetup(){
            CollegeStudent colStud=new CollegeStudent();
            colStud.setFirstName("abhishek");
            colStud.setLastName("patil");
            RelectionTestUtils.setFieldValue(colStud,"id",1);// setting value to private variable
    }
    @Test
     Void testPrivateFieldGetValue(){
       assertEquals(1, RelectionTestUtils.getFieldValue(colStud ,“ id”));
     }
}
```

# ***Database integration testing***

Let say I have insertData.sql file (which has insert statements) in src/resource/ folder. So how do I call this insertData.sql before execute any test.

Here is code example:

```
@Sql("/insertData.sql")

@Test

void testBooks(){

  List<Book> books=bookservice.getBooks();

  assertEquals(5,books.size());

}
```