Testing

JUnit

Consider a situation that you have written a code and tested manually; everything works fine. In future other person is making changes (may be trying to optimize the code, may be the new developer find some condition of the code superfluous, may remove that). In such situation doing all testing manually is not feasible every time. Here comes the need of writing Test cases and learning the JUnit. As the new person is making the changes means he is going to write more test cases so that's why it is termed as ongoing process.

Steps for testing

1. Preparation of Test Cases
2. Provide the Test Input
3. Run the tests
4. Provide Expected Input
5. Verify the result
6. Do something to alert the developer if test fails.

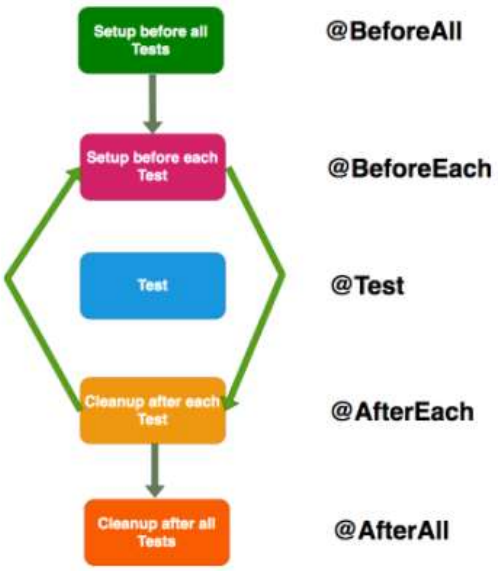
Naturally the step number 1,2 and 4 are separate for every method or code but the JUnit is actually going to help us in the step number 3, 5 and 6.

The Junit will run the test case for you and it will verify the result. Apart from that it will generate a nice test report so that developer will find the exact cause of Test case failure and make the necessary correction. So overall it making the work easier for the Developer by doing the things automated instead of doing them manually.

All core annotations are located in the ***org.junit.jupiter.api package*** in the junit-jupiter-api module. Let us take a look at the important annotations

The common annotations of JUnit are as follow-

- @BeforeEach: Denotes that the annotated method should be executed before each @Test or @RepeatedTest method in the current class.
- @AfterEach: Denotes that the annotated method should be executed after each @Test or @RepeatedTest method in the current class.
- @BeforeAll: Denotes that the annotated method should be executed before all @Test or @RepeatedTest methods in the current class; Such methods must be static (unless the "per-class" test instance lifecycle is used).
- @AfterAll: Denotes that the annotated method should be executed after all @Test or @RepeatedTest methods in the current class. Such methods must be static (unless the "per-class" test instance lifecycle is used).
- @Test: Denotes that a method is a test method.
- @RepeatedTest(value = n, name = “name of test case”): Denotes that a method is a test template for a repeated test.



(Test Life Cycle)

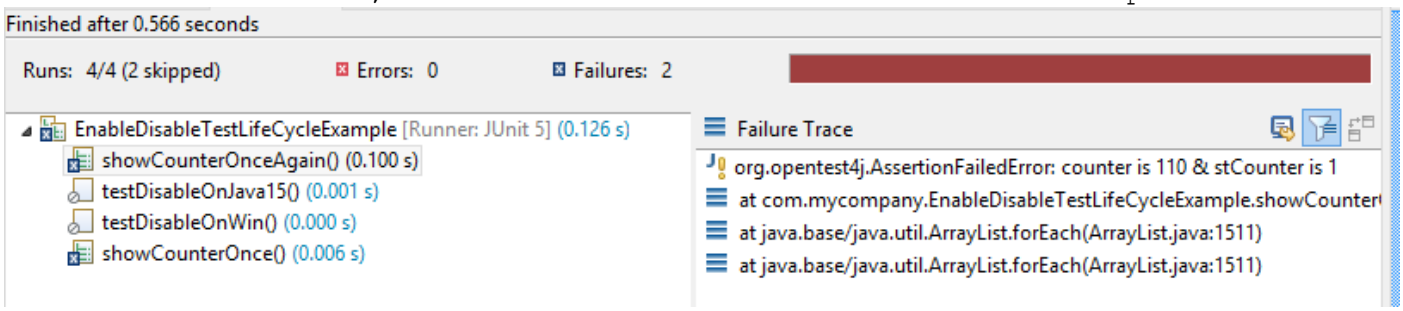
Do not use instance variable in the Test class as it may lead to make test cases having high coupling because if one test case changes the value of the instance variable then the other test case may not have awareness of the changes and eventually test cases become dependent on one another. Yet in Junit-5 a new class instance is created for every test run i.e. method run so the changes made by one method are not applicable so again better not to use instance variable. Yet you can change the behaviour by using the annotation that is @TestInstance(), the TestInstance.Lifecycle.PER_METHOD is the default value another value is TestInstance.Lifecycle.PER_CLASS

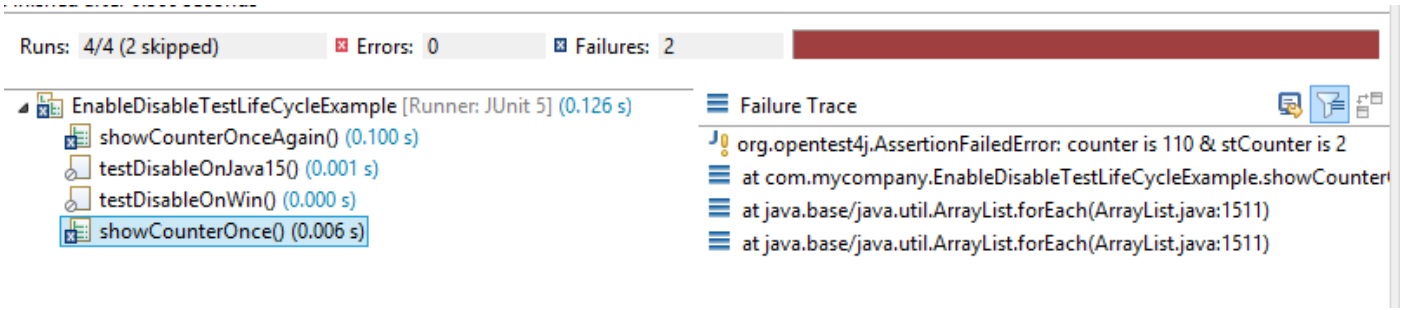
An Example

```
package com.masaischool;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.condition.DisabledOnJre;
import org.junit.jupiter.api.condition.DisabledOnOs;
import org.junit.jupiter.api.condition.EnabledIf;
import org.junit.jupiter.api.condition.JRE;
import org.junit.jupiter.api.condition.OS;

class EnableDisableTestLifeCycleExample {
	static int stCounter = 0;
	int counter;

	EnableDisableTestLifeCycleExample(){
		counter = 100;
	}

	@Test
	@DisabledOnOs(OS.WINDOWS)
	void testDisableOnWin() {
		fail("This is inside testDisableOnWin");
	}

	@Test
	@DisabledOnJre(JRE.JAVA_15)
	void testDisableOnJava15() {
		fail("This is inside testDisableOnJava15");
	}

	@Test
	void showCounterOnce() {
		if(counter == 100) {
			increaseCounterBy10();
			stCounter++;
			fail("counter is " + counter + " & stCounter is " + stCounter);

		}
	}

	void increaseCounterBy10() {
		counter+=10;
	}

	@Test
	@EnabledIf("isCounter100")
	void showCounterOnceAgain() {
		if(counter == 100) {
			increaseCounterBy10();
			stCounter++;
			fail("counter is " + counter + " & stCounter is " + stCounter);
```

```
            }
      }

      boolean isCounter100() {
            return (counter == 100);
      }
}
```

Press Alt + Shift + X,T to run the test and observe the both outputs





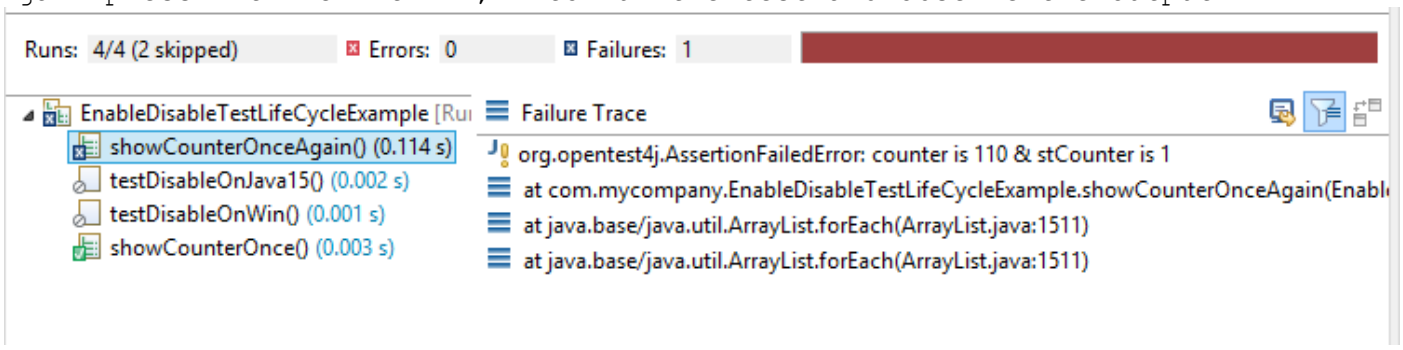
Now add following lines-

```
import org.junit.jupiter.api.TestInstance;
```

and at the top of the 'class EnableDisableTestLifeCycleExample' statement

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
```

Again press Alt + Shift + X, T to run the test and observe the output-



Assertions: It is a collection of utility methods that support asserting conditions in tests. Unless otherwise noted, a failed assertion will throw an AssertionFailedError or a subclass thereof. Some important assertion related methods (defined in org.junit.jupiter.api.Assertions) are as follow-

Assertion Method	Description
public static void assertEquals(expected, actual)	Asserts that expected and actual are equal.
public static void assertNotEquals(expected, actual)	Asserts that expected and actual are not equal. Fails if both are null, Fails with the supplied failure message.
public static void assertFalse(condition, message)	Asserts that the supplied condition is not true. If necessary, the failure message will be retrieved lazily from the supplied messageSupplier.
public static void assertTrue(condition, message)	It says all okay is the condition is false, you can use assertEquals also for the same: Asserts that the boolean condition supplied by booleanSupplier is true. If necessary, the failure message will be retrieved lazily from the supplied messageSupplier.

public static void assertNull(expected, actual)	Asserts that actual is null.
public static void assertNotNull(actual)	Asserts that actual is not null.
public static void assertThrows(Class obj, lambda Expression, String message)	Asserts that execution of the supplied executable throws an exception of the expectedType and returns the exception. If no exception is thrown, or if an exception of a different type is thrown, this method will fail.
public static void assertAll(Executable... executables)	Asserts that all supplied executables do not throw exceptions.

Pareto Rule: 80% of the errors comes due to 20% of the code

JUnit Coding Convention

1. Name of the test class must end with Test
   e.g. for class SimpleInterest, The test class name should be SimpleInterestTest
2. Name of the method must begin with the test
   e.g. for method calculateInterest() the test method should be named as testCalculateInterest()
3. test method must not return anything so return type must be void; must not throw any exception; must not accept any parameter; must not be private; must be annotated with @Test/@RepeatedTest annotation

An Example
Create a maven project (change the jdk version from 1.7 to any of your choice) and add junit-jupitor dependency as follow-

```
<!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api -->
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.9.2</version>
    <scope>test</scope>
</dependency>
```

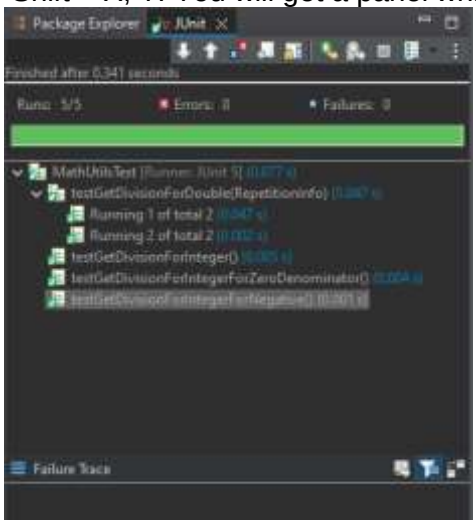
Create a class MathUtils that has following method

```
package com.masai.junit_example;

public class MathUtils {
    /**
     * return the quotient of a to b
     * @param a the first number of type double that is numerator
     * @param b the second number of type double that is denominator
     * @return quotient of first parameter to the second parameter
     */
    public double getDivisionForDouble(double a, double b){
        return (a/b);
    }

    /**
     * return the quotient of a to b
     * @param a the first number of type int that is numerator
     * @param b the second number of type int that is denominator
     * @return quotient of first parameter to the second parameter
     */
    public int getDivisionForInteger(int a, int b){
        return (a/b);
    }
}
```

Say we want to write test cases for the methods of MathUtils class
Create a class MathUtilsTest inside src/test/java folder with following code

```
package com.masai.junit_example;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.RepeatedTest;
import org.junit.jupiter.api.RepetitionInfo;
import org.junit.jupiter.api.Test;

class MathUtilsTest {
  @Test
  void testGetDivisionForInteger() {
    MathUtils mathUtils = new MathUtils();
    assertEquals(6, mathUtils.getDivisionForInteger(12, 2));
    assertEquals(7 ,mathUtils.getDivisionForInteger(15, 2));
  }

  @Test
  void testGetDivisionForIntegerForNegative() {
    MathUtils mathUtils = new MathUtils();
    assertEquals(6, mathUtils.getDivisionForInteger(-12, -2));
    assertEquals(-8, mathUtils.getDivisionForInteger(-17, 2));
    assertEquals(-8, mathUtils.getDivisionForInteger(17, -2));
  }

  @Test
  void testGetDivisionForIntegerForZeroDenominator() {
    MathUtils mathUtils = new MathUtils();
    assertThrows(ArithmeticException.class, () ->
mathUtils.getDivisionForInteger(15, 0));
  }

  @RepeatedTest(value = 2, name = "Running {currentRepetition} of total
{totalRepetitions}")
  void testGetDivisionForDouble(RepetitionInfo rp) {
    MathUtils mathUtils = new MathUtils();
    if(rp.getCurrentRepetition() == 1) {
      assertTrue(Double.valueOf(mathUtils.getDivisionForDouble(15,
0)).isInfinite());
    }else {
      assertTrue(Double.valueOf(mathUtils.getDivisionForDouble(0, 0)).isNaN());
    }
  }
}
```

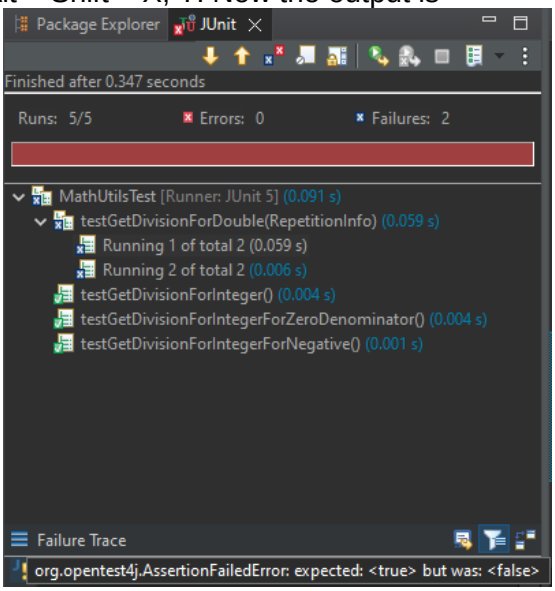
Run the above file by pressing Alt + Shift + X, T. You will get a panel which as output as follow



If any of the test case fails then it will show output with proper error message. Say we have written the following code

```
@RepeatedTest(value = 2, name = "Running {currentRepetition} of total
{totalRepetitions}")
void testGetDivisionForDouble(RepetitionInfo rp) {
  MathUtils mathUtils = new MathUtils();
  if(rp.getCurrentRepetition() == 1) {
    assertTrue(Double.valueOf(mathUtils.getDivisionForDouble(15, 0)).isNaN());
  }else {
    assertTrue(Double.valueOf(mathUtils.getDivisionForDouble(0, 0)).
isInfinite());
  }
}
```

Run the above file by pressing Alt + Shift + X, T. Now the output is-



Important:
Using junit we can test the methods whose body is already defined, we used to call such method using object of the class i.e. the code to be tested has to be defined completely but this may not be the case all time. Say your code is depending on some external library/resource that may not be available at the time of testing in such condition testing the code using junit only is not possible and here comes the need of mockito.

Mockito

Mock *means something not authentic or unreal, but without the intention to deceive anyone.*

Original

Mock

What is mock object?

1. A **mock object** is a dummy implementation for an interface or a class. Mocks are used as a replacement for a dependency.
2. The main purpose of using a mock object is to simplify the development of a test by mocking external dependencies and using them in the code.

3. You should mock objects in unit tests when the real object is yet to be implemented or it has a non-deterministic behaviour or it is a callback function.

What is mockito?

- Mockito is a popular open source framework for mocking objects in software test.
- Mockito provides an implementation for every method of the mock object.
- Using Mockito greatly simplifies the development of tests for classes with external dependencies.

The mockito dependency can be added to the maven project as follow-

```
<!-- https://mvnrepository.com/artifact/org.mockito/mockito-junit-jupiter -->
<dependency>
   <groupId>org.mockito</groupId>
   <artifactId>mockito-junit-jupiter</artifactId>
   <version>5.3.1</version>
   <scope>test</scope>
</dependency>
```

No need to add junit-jupiter-api dependency additionally because mockito itself depend on this so additional dependencies are downloaded by the maven itself.

How to create mock of an object?
The ***Mockito*** class defined in the ***org.mockito*** package is a utility class has various versions **mock()** method that are used to create Mock object.

```
public static <T> T mock(Class<T> classToMock)
```

- We can even call methods using mock. By default, all methods of a mock return "uninitialized" or "empty" values, e.g., zeros for numeric types (both primitive and boxed), false for booleans, and nulls for most other types.
- You can change the behaviour of a method (*If its return type is not void*) by defining "when this method is called, then do something." This strategy uses Mockito's thenReturn call:

  ```
  when(mock-object.method-call(arguments)).thenReturn(some-value);
  ```

  In thenReturn, you can also specify multiple values which will be returned as the results of consecutive method calls. The last value will be used as a result for all further method calls.
- You may throw exception from a method also

  ```
  when(mock-object.method-call(arguments)).thenThrow(Exception-object);
  ```
  or
  ```
  when(mock-object.method-
  call(arguments)).thenThrow(ExceptionClassName.class);
  ```

- To change the behaviour for method whose return type is void, we have another strategy

  ```
  doNothing().when(mock-object).method-call(arguments);
  ```
  or
  ```
  doReturn(value-to-return).when(mock-object).method-call(arguments);
  ```
  or
  ```
  doAnswer(Answer a).when(mock-object).method-call(arguments);
  ```
- You may throw exception from such a method also

  ```
  doThrow(Exception-object).when(mocked-object).method-name(arg-list);
  ```
  or
  ```
  doThrow(ExceptionClassName.class).when(mocked-object).method-name(arg-
  list);
  ```

Some other methods in Mockito Framework are as follow-
any(): method of ***org.mockito.ArgumentMatchers*** class is used to match against any arguments of given type; excluding null. Similar to this method we have methods anyString(), anyByte(), anyInt(), anyCollection() etc,
argThat(ArgumentMatcher<T> matcher): Allows creating custom argument matchers.

An Example
Create a maven project (change the jdk version from 1.7 to any of your choice) and add mockito-junit-jupiter dependency as follow-

```
<!-- https://mvnrepository.com/artifact/org.mockito/mockito-junit-jupiter -->
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>5.3.1</version>
    <scope>test</scope>
</dependency>
```

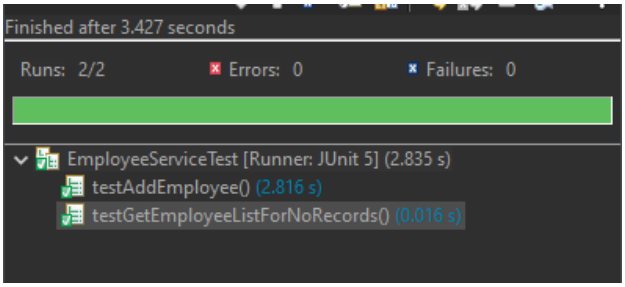
Write following code

```
package com.masai.mockito_example.dto;

public class Employee {
  private int empId;
  private String name;
  private double salary;
  public Employee() {
    super();
  }

  public Employee(int empId, String name, double salary) {
    super();
    this.empId = empId;
    this.name = name;
    this.salary = salary;
  }

  public int getEmpId() {
    return empId;
  }
  public String getName() {
    return name;
  }

  public double getSalary() {
    return salary;
  }
  public void setEmpId(int empId) {
    this.empId = empId;
  }

  public void setName(String name) {
    this.name = name;
  }
  public void setSalary(double salary) {
    this.salary = salary;
  }

  @Override
  public int hashCode() {
    return empId;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
```

```
    Employee other = (Employee) obj;
    return empId == other.empId;
  }
}


package com.masai.mockito_example.exception;

public class NoRecordFoundException extends Exception {
  public NoRecordFoundException(String message) {
    super(message);
  }
}


package com.masai.mockito_example.exception;

public class SomeThingWrongException extends Exception {
  public SomeThingWrongException(String message) {
    super(message);
  }
}


package com.masai.mockito_example.dao;
import java.util.List;

import com.masai.mockito_example.dto.Employee;
import com.masai.mockito_example.exception.NoRecordFoundException;
import com.masai.mockito_example.exception.SomeThingWrongException;

public interface EmployeeDAO {
  public void addEmployee(Employee emp) throws SomeThingWrongException;
  public List<Employee> getEmployeeList() throws NoRecordFoundException;
}


//No implementation of the EmployeeDAO because it will be tested using mockito

package com.masai.mockito_example.service;

import java.util.List;

import com.masai.mockito_example.dto.Employee;
import com.masai.mockito_example.exception.NoRecordFoundException;
import com.masai.mockito_example.exception.SomeThingWrongException;

public interface EmployeeService {
  public void addEmployee(Employee emp) throws SomeThingWrongException;
  public List<Employee> getEmployeeList() throws NoRecordFoundException;
}

package com.masai.mockito_example.service;

import java.util.List;

import com.masai.mockito_example.dao.EmployeeDAO;
import com.masai.mockito_example.dto.Employee;
import com.masai.mockito_example.exception.NoRecordFoundException;
import com.masai.mockito_example.exception.SomeThingWrongException;
```

```
public class EmployeeServiceImpl implements EmployeeService {
  EmployeeDAO empDAO;

  public EmployeeServiceImpl(EmployeeDAO empDAO){
    this.empDAO = empDAO;
    //Make empDAO point to some implementation if it is available
    //In this example we don't have any implementation
    //only mock implemetation can be used as of now
  }

  @Override
  public void addEmployee(Employee emp) throws SomeThingWrongException {
    empDAO.addEmployee(emp);
  }

  @Override
  public List<Employee> getEmployeeList() throws NoRecordFoundException{
    return empDAO.getEmployeeList();
  }
}

package com.masai.mockito_example;

import org.junit.jupiter.api.Test;
import org.mockito.ArgumentMatcher;
import org.mockito.Mockito;

import static org.junit.jupiter.api.Assertions.assertThrows;
import static org.mockito.Mockito.*;

import com.masai.mockito_example.dao.EmployeeDAO;
import com.masai.mockito_example.dto.Employee;
import com.masai.mockito_example.exception.NoRecordFoundException;
import com.masai.mockito_example.exception.SomeThingWrongException;
import com.masai.mockito_example.service.EmployeeService;
import com.masai.mockito_example.service.EmployeeServiceImpl;

public class EmployeeServiceTest {
  EmployeeService es;

  private EmployeeDAO createMockForInsert() {
    EmployeeDAO empDAO = mock(EmployeeDAO.class);
    try {
      //throw NullPointerException if null is passed
      Mockito.doThrow(NullPointerException.class).when(empDAO).addEmployee(null);

      //throw SomeThingWrongException if any instance variable has incorrect value
      ArgumentMatcher<Employee> argMatch = emp -> emp.getEmpId() <= 0 ||
          (emp.getName() == null || emp.getName().equals("")) ||
          (emp.getSalary() < 1.8 || emp.getSalary() > 100.0);

Mockito.doThrow(SomeThingWrongException.class).when(empDAO).addEmployee(argThat(argMatch));

      //throw NoRecordFoundException if no Record in the table
      Mockito.when(empDAO.getEmployeeList()).thenThrow(NoRecordFoundException.class);
    }catch(SomeThingWrongException | NoRecordFoundException ex) {
      System.out.println(ex);
    }
    return empDAO;
```

```
  }

  @Test
  public void testAddEmployee() {
    EmployeeDAO empDAO = createMockForInsert();
    es = new EmployeeServiceImpl(empDAO);
    assertThrows(NullPointerException.class, () -> es.addEmployee(null));
    assertThrows(SomeThingWrongException.class, () -> es.addEmployee(new Employee(-1, "ABC", 2.0)));
    assertThrows(SomeThingWrongException.class, () -> es.addEmployee(new Employee(3, "", 2.0)));
    assertThrows(SomeThingWrongException.class, () -> es.addEmployee(new Employee(4, "ABC", 0.18)));
    try {
      verify(empDAO, times(1)).addEmployee(null);
      verify(empDAO, times(1)).addEmployee(new Employee(-1, "ABC", 2.0));
      verify(empDAO, times(1)).addEmployee(new Employee(3, "", 2.0));
      verify(empDAO, times(1)).addEmployee(new Employee(4, "ABC", 0.18));
    }catch(SomeThingWrongException ex) {
      ex.printStackTrace();
    }
  }

  @Test
  public void testGetEmployeeListForNoRecords() {
    EmployeeDAO empDAO = createMockForInsert();
    es = new EmployeeServiceImpl(empDAO);
    try {
      assertThrows(NoRecordFoundException.class, () -> empDAO.getEmployeeList());
      verify(empDAO, times(1)).getEmployeeList();
    }catch(NoRecordFoundException ex) {
      System.out.println(ex);
    }
  }
}
```



Conclusion

Test Driven Development Methodology.... Write test case first and then write the code. This thing is exactly that is going to match the agile methodology in which project owner and the scrum manager are writing use case so actually they are writing the test cases in which the expected output is written properly and the developer team has to set up everything properly for the same.

References

https://www.toptal.com/java/a-guide-to-everyday-mockito
https://site.mockito.org/
https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html
https://www.arhohuttunen.com/junit-5-mockito/
https://www.lambdatest.com/blog/junit5-mockito-tutorial/