Tutorial on Junit 5

Junit 5 is one of the popular **testing frameworks** in the Java Development World. JUnit is an automated unit-testing framework for Java. We will learn Junit5 using an application example. In this tutorial, Eclipse is used as an IDE.

We will use "Contact Manager Application". You will find out the source code of this application in Moodle. Download the source code. The application consists of two classes. ***Contact.java*** and ***ContactManager.java***.

Open Eclipse and import/copy the source code of both the classes.

Write the first Test Case:

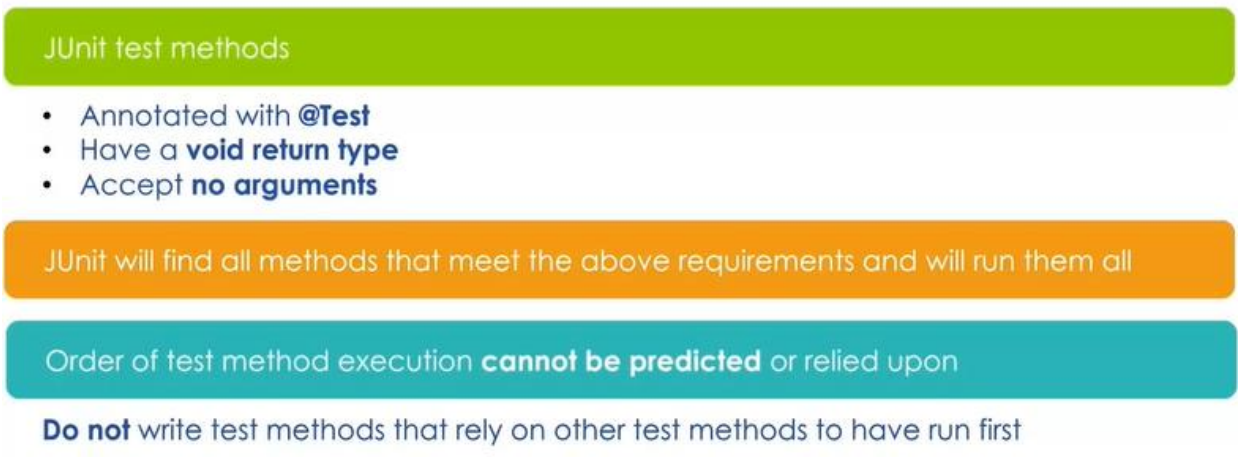
Right click ***ContactManager*** class. Then press New -> JUnit Test Case.

Give the name of the test class as **ContactManagerTest.java** or as per your choice. Write the following code inside the file.

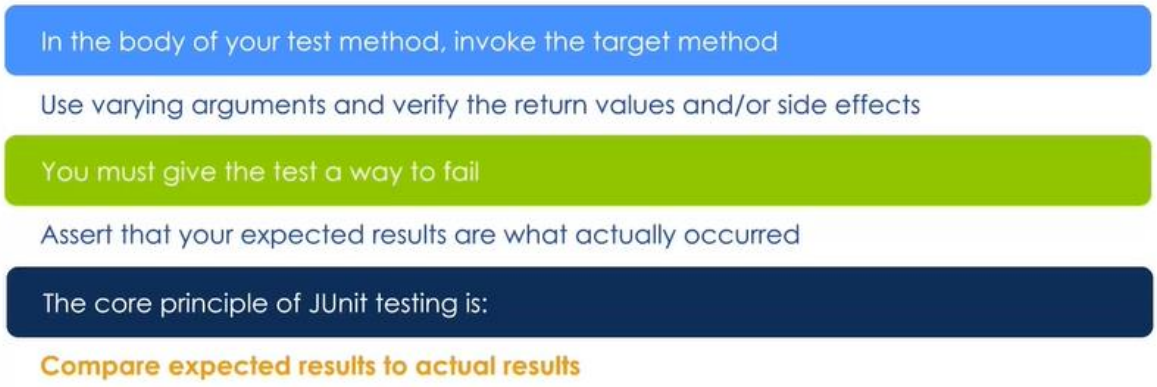
```
class ContactManagerTest
{
   @Test
   @DisplayName("Should Create Contact")
    public void shouldCreateContact() {
        ContactManager contactManager = new ContactManager();
        contactManager.addContact("John", "Doe", "0123456789");
        assertFalse(contactManager.getAllContacts().isEmpty());
        assertEquals(1, contactManager.getAllContacts().size());
    }
}
```

How to Test using JUnit?

1. Here, ***shouldCreateContact()*** is a test method that is testing, the contact creation.



2. Junit understands that this method is a Test, by looking at the ***@Test*** annotation.



3. By looking at the method name ***shouldCreateContact()*** we understand what the method is trying to do, but we can also **provide a custom name** for this method using the ***@DisplayTest*** annotation.

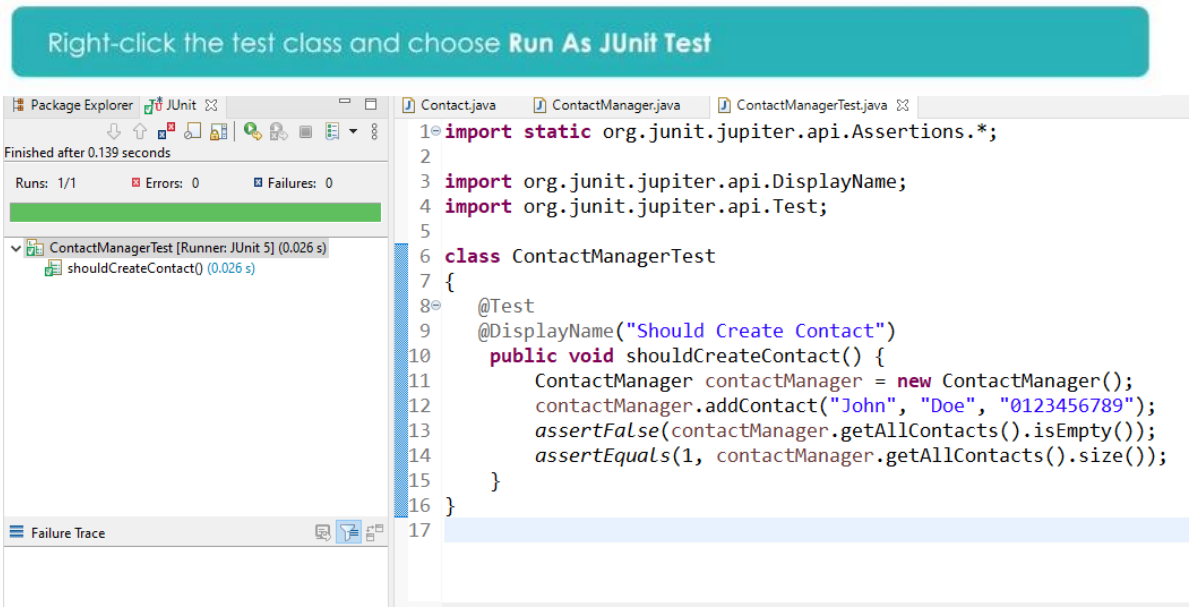
4. Inside the method, we created a contact by providing the first name, last name and phone number details to the ***addContact()*** method of the ***ContactManager*** class.

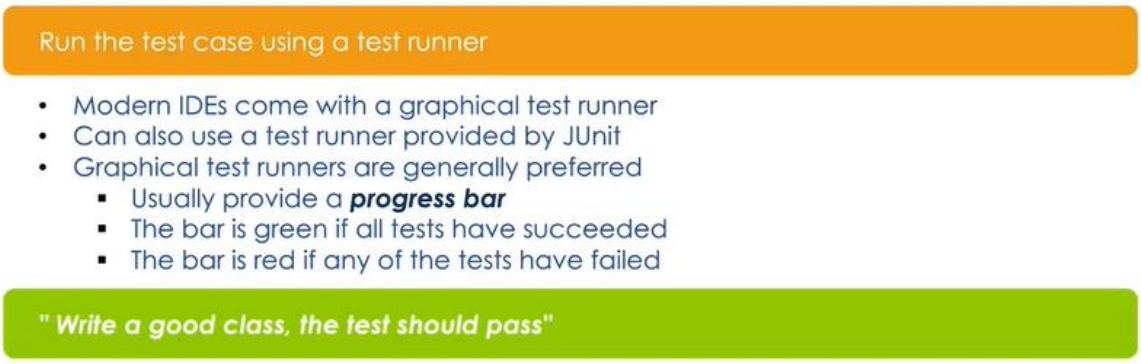
5. We can retrieve all the contact information through the ***getAllContacts()*** method.

6. As a last step, we are verifying that the result of ***getAllContacts()*** is not empty, and the size of the ***getAllContacts()*** List is exactly 1.

The process of ***verifying the actual output with the expected output*** is called as performing assertions. Each test method executes a target method and asserts value on the return.

If you run the above test in Eclipse, the result will be green just as displayed in the following figure. This indicates that the test case has passed. In this example, the test case is [“John”, “Doe”, “0123456789”]. That means this test case does not find out any bugs in the code.





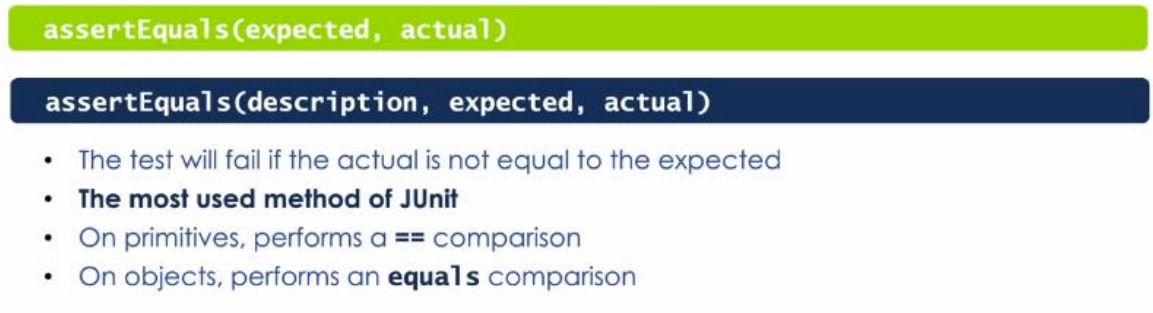
Junit5 provides many different methods to perform **assertions** for different cases. We have ***Assertions*** class which provides number of static methods, we can use in our tests.

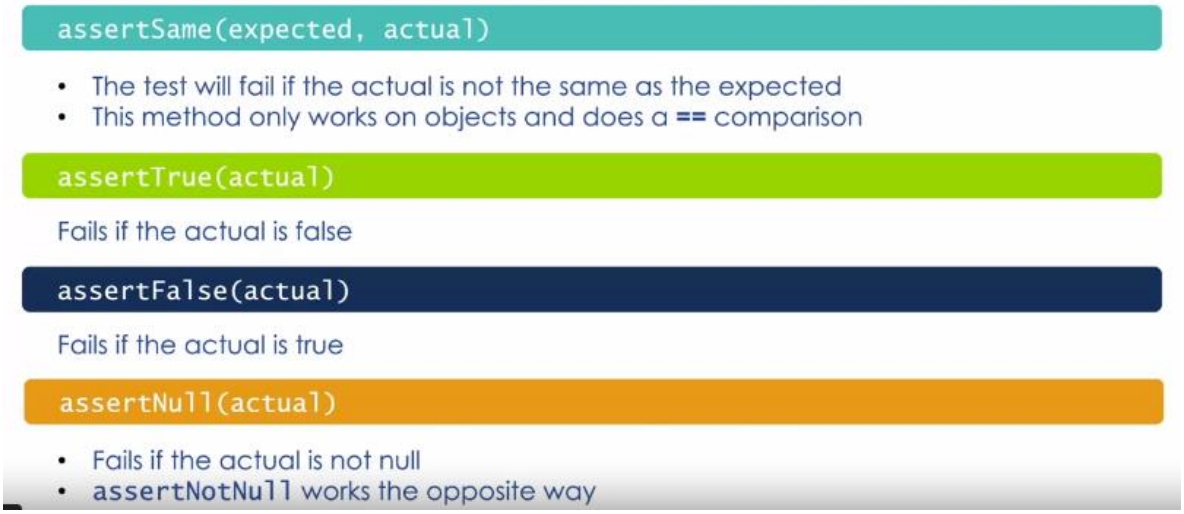
List of assertion methods:

JUnit Reference documentation

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

https://howtodoinjava.com/junit5/junit-5-assertions-examples/





Now let’s explore some other scenarios.

Here are some validations, which are performed when creating a Contact:

- First Name should not be null
- Last Name should not be null
- Phone Number Should :
  - Be Exactly 10 Digits Long
  - Contain only Digits
  - Start with 0

Let’s write test cases to test some of the above cases,

```
class ContactManagerTest
{
    @Test
    @DisplayName("Should Create Contact")
     public void shouldCreateContact() {
         ContactManager contactManager = new ContactManager();
         contactManager.addContact("John", "Doe", "0123456789");
         assertFalse(contactManager.getAllContacts().isEmpty());
         assertEquals(1, contactManager.getAllContacts().size());
     }
    @Test
    @DisplayName("Should Not Create Contact When First Name is Null")
    public void shouldThrowRuntimeExceptionWhenFirstNameIsNull() {
        ContactManager contactManager = new ContactManager();
        assertThrows(RuntimeException.class, () -> {
            contactManager.addContact(null, "Doe", "0123456789");
        });
    }
}
```

The 2nd Test Case, tests whether the Contact Creation is Failed when we enter the First Name as Null.

We are asserting that the addContact() method throws a RuntimeException.

We can assert the Exceptions using the assertThrow() method from the Assertions class

The assertThrow() method takes the Exception Type as the first parameter and the Executable which is throws the Exception as the second parameter.

We can write similar test cases also for the fields Last Name and Phone Number as follows:

```
class ContactManagerTest
{
    @Test
    @DisplayName("Should Create Contact")
     public void shouldCreateContact() {
         ContactManager contactManager = new ContactManager();
         contactManager.addContact("John", "Doe", "0123456789");
         assertFalse(contactManager.getAllContacts().isEmpty());
         assertEquals(1, contactManager.getAllContacts().size());
     }
    @Test
    @DisplayName("Should Not Create Contact When First Name is Null")
    public void shouldThrowRuntimeExceptionWhenFirstNameIsNull() {
        ContactManager contactManager = new ContactManager();
        assertThrows(RuntimeException.class, () -> {
            contactManager.addContact(null, "Doe", "0123456789");
        });
    }

    @Test
    @DisplayName("Should Not Create Contact When Last Name is Null")
    public void shouldThrowRuntimeExceptionWhenLastNameIsNull() {
        ContactManager contactManager = new ContactManager();
        Assertions.assertThrows(RuntimeException.class, () -> {
            contactManager.addContact("John", null, "0123456789");
        });
    }

    @Test
    @DisplayName("Should Not Create Contact When Phone Number is Null")
    public void shouldThrowRuntimeExceptionWhenPhoneNumberIsNull() {
        ContactManager contactManager = new ContactManager();
        Assertions.assertThrows(RuntimeException.class, () -> {
            contactManager.addContact("John", "Doe", null);
        });
    }
}
```

Understanding Test Lifecycle

Now let's go ahead and understand the Lifecycle of Junit Tests. Each Test undergoes different phases as part of it's execution, each phase is represented by an Annotation.

@BeforeAll

The method marked with this annotation will be executed before any of the @Test methods are executed inside the Test class.

@BeforeEach

The method marked with this annotation will be executed before each @Test method in the Test class.

@AfterEach

The method marked with this annotation will be executed after each @Test method in the Test class.

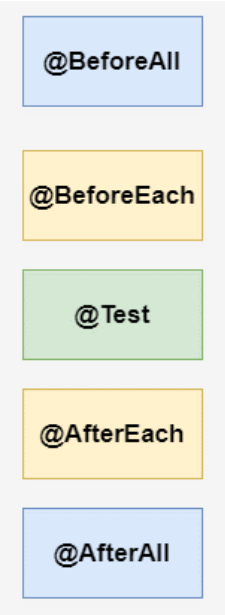
@AfterAll

The method marked with this annotation will be executed after all the @Test methods are executed in the Test class.

The Before methods (@BeforeAll, @BeforeEach) perform some initialization actions like setting up of test data or environment data, before executing the tests.

The After methods (@AfterAll, @AfterEach) perform clean up actions like cleaning up of created Environment Data or Test Data.

@BeforeAll and @AfterAll are called only once for the entire test, and the methods are usually marked as static.



Implementing Lifecycle Annotations in our Test

So now, let's see how we can use the above mentioned Lifecycle annotations, in our test.

In our 4 tests, we can observe that we are creating an instance of ***ContactManager*** at the start of each test. This logic can go inside the method, which is marked with either @BeforeAll or @BeforeEach

```
@BeforeAll
public static void setupAll() {
    System.out.println("Should Print Before All Tests");
}

@BeforeEach
public void setup() {
    System.out.println("Instantiating Contact Manager");
    contactManager = new ContactManager();
}
```

```
@AfterEach
public void tearDown() {
    System.out.println("Should Execute After Each Test");
}

@AfterAll
public static void tearDownAll() {
    System.out.println("Should be executed at the end of the Test");
}
```

Conditional Executions

We can execute the @Test methods in our class based on a specific condition, for example: imagine if you developed a specific functionality on Linux for our Contact Manager application. After saving the contact, we are performing some additional logic, which is specific to MAC Operating System.

Then we have to make sure that the test should only run when it's running only on MAC Operating System.

You can enable this conditional execution using different annotations:

@EnabledOnOs

@DisabledOnOs

These annotations take the values for usual Operating Systems like MAC, LINUX, WINDOWS. You can see the example usage of this annotation below:

```java
@Test
@DisplayName("Should Create Contact on MAC")
@EnabledOnOs(value = OS.MAC, disabledReason = "Should Run only on MAC")
public void shouldCreateContactOnMac() {
    contactManager.addContact("John", "Doe", "0123456789");
    assertFalse(contactManager.getAllContacts().isEmpty());
    assertEquals(1, contactManager.getAllContacts().size());
}
```

Parameterized Tests

You can also run parameterized tests by running a test multiple times and providing different set of inputs for each repetition. We have to add the ***@ParameterizedTest*** annotation to mark a test method as Parameterized Test.

```java
@Test
@DisplayName("Phone Number should start with 0")
public void shouldTestPhoneNumberFormat() {
    contactManager.addContact("John", "Doe", "0123456789");
    assertFalse(contactManager.getAllContacts().isEmpty());
    assertEquals(1, contactManager.getAllContacts().size());
}
```

In the above testcase, we are checking whether the provided phone number is in the required format or not. i.e.. If the Phone Number is starting with 0.

We can run this test against different set's of input and make sure that the test is working as expected or not.

You can provide the input to the test using different ways:

Value Source

You can provide input to the Parameterized Test using the ***@ValueSource*** annotation where you can provide a set of string, long, double, float literals to our test.

Eg: @ValueSource(strings = {"string1″,"string2″,"string3"})

Then you can access this string inside the test by first adding a String parameter to our test.

```java
@DisplayName("Phone Number should match the required Format")
@ParameterizedTest
@ValueSource(strings = {"0123456789", "1234567890", "+0123456789"})
public void shouldTestPhoneNumberFormat(String phoneNumber) {
    contactManager.addContact("John", "Doe", phoneNumber);
    assertFalse(contactManager.getAllContacts().isEmpty());
    assertEquals(1, contactManager.getAllContacts().size());
}
```

In the above example, we are providing the Phone Number String in different formats.

MethodSource

We can also use ***@MethodSource*** annotation to provide the input to our Parameterized Tests, using this annotation we will refer the method name, which returns the values required for our tests as output.

```
@DisplayName("Method Source Case - Phone Number should match the required Format")
@ParameterizedTest
@MethodSource("phoneNumberList")
public void shouldTestPhoneNumberFormatUsingMethodSource(String phoneNumber) {
    contactManager.addContact("John", "Doe", phoneNumber);
    assertFalse(contactManager.getAllContacts().isEmpty());
    assertEquals(1, contactManager.getAllContacts().size());
}
private List<String> phoneNumberList() {
    return Arrays.asList("0123456789", "1234567890", "+0123456789");
}
```

In the above test, we declared a method called as phoneNumberList() which returns the required input values as a List<String>.

We passed the name of this method as the value to the @MethodSource annotation.

annotation**.**

Disabled Tests

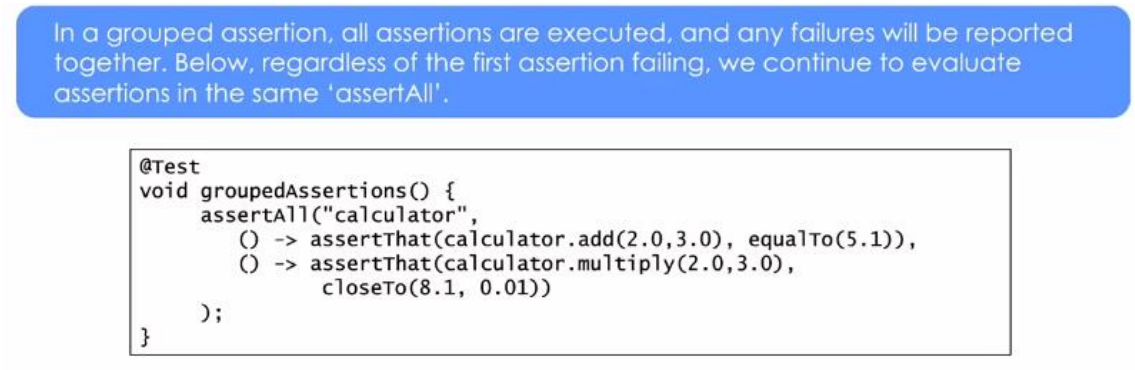
You can disable some tests from running by adding the @Disabled annotation**.**

```
1  @Test
2  @DisplayName("Test Should Be Disabled")
3  @Disabled
4  public void shouldBeDisabled() {
5      throw new RuntimeException("Test Should Not be executed");
6  }
```

You can see the following links:

https://www.baeldung.com/parameterized-tests-junit-5

Group Assertions



You can see the following links:

https://www.javaguides.net/2018/09/junit-5-assertall-example.html

AssertJ Library:

You also use AssertJ Library for a rich assertion methods. See the following links:

https://joel-costigliola.github.io/assertj/

https://assertj.github.io/doc/

Another Example:

Download "Circle-Radius.java" (inside Circle-Radious folder) from Moodle. This code will calculate circumference and area of the circle.

How do test main method inside "Circle-Radius.java"?

See the file "Circle-Radius-Test.java" (inside Circle-Radious folder).