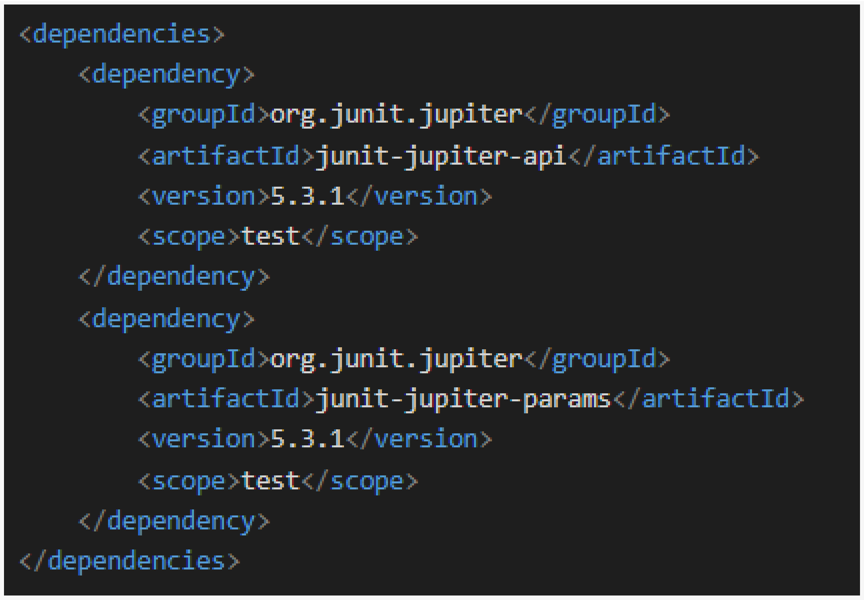
JUnit-5

JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks. Its main use is to write repeatable tests for your application code units.

Installation

To include JUnit into your project, you need to include its dependency into classpath.



```xml
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.3.1</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-params</artifactId>
        <version>5.3.1</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Annotations in Java

Annotations are used to provide supplement information about a program.

- Annotations start with ‘@’.
- Annotations do not change action of a compiled program.
- Annotations help to associate *metadata* (information) to the program elements i.e. instance variables, constructors, methods, classes, etc.

```
class Base
{
     public void display()
     {
         System.out.println("Base display()");
     }
}
class Derived extends Base
{
     @Override
     public void display(int x)
     {
         System.out.println("Derived display(int )");
     }

     public static void main(String args[])
     {
         Derived obj = new Derived();
         obj.display();
     }
}
```

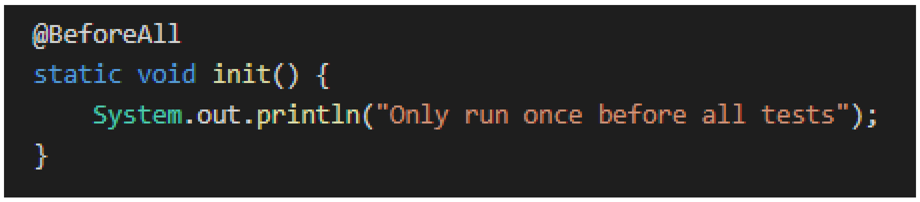
Output :

```
10: error: method does not override or implement
    a method from a supertype
```

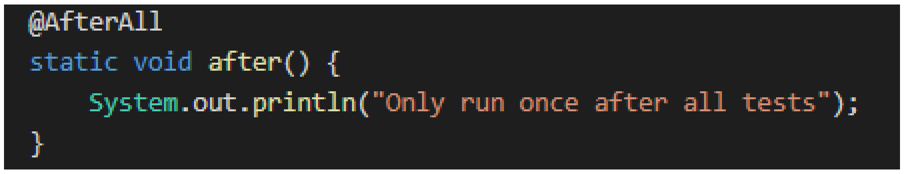
If we remove parameter (int x) or we remove @override, the program compiles fine.

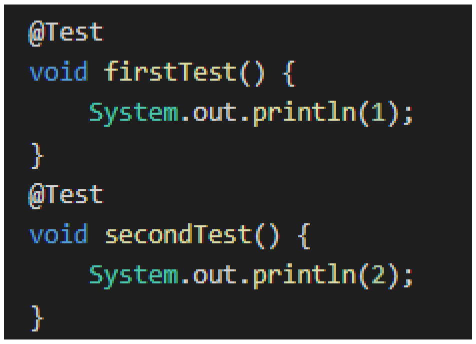
JUnit Annotations

JUnit offers the following annotations to write tests.

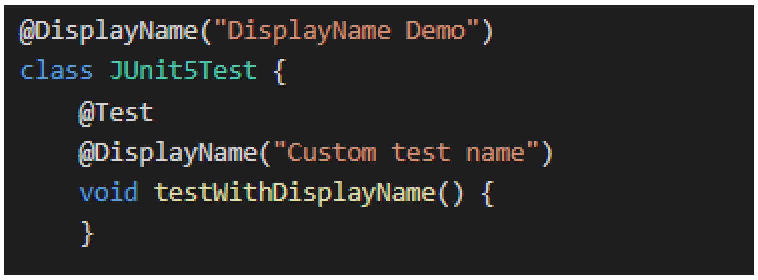
ANNOTATION	DESCRIPTION
`@BeforeEach`	The annotated method will be run before each test method in the test class.
`@AfterEach`	The annotated method will be run after each test method in the test class.
`@BeforeAll`	 `@BeforeAll static void init() { System.out.println("Only run once before all tests"); }`

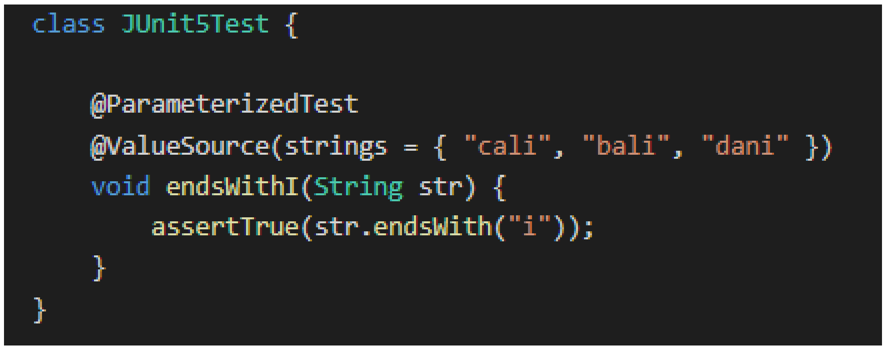
```java
@BeforeAll
static void init() {
    System.out.println("Only run once before all tests");
}
```

`@AfterAll`	
	`@AfterAll`
	`static void after() {`
	`    System.out.println("Only run once after all tests");`
	`}`

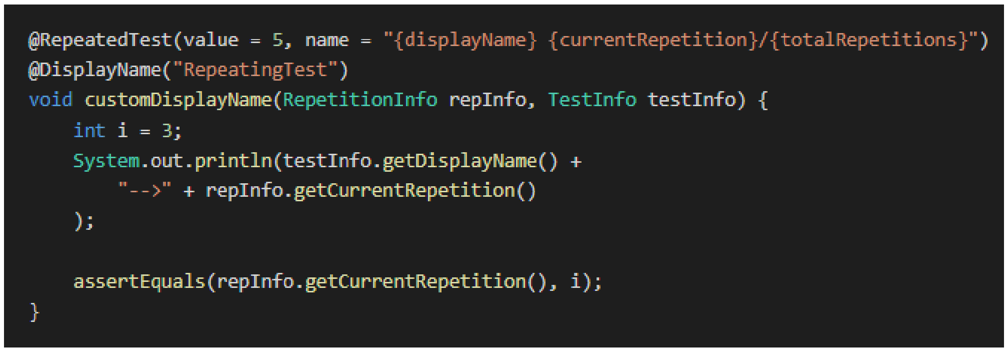
`@Test`	It is used to mark a method as junit test
	
	`@Test`
	`void firstTest() {`
	`    System.out.println(1);`
	`}`
	`@Test`
	`void secondTest() {`
	`    System.out.println(2);`
	`}`

`@Disabled`	It is used to disable or ignore a test class or method from test suite.

@DisplayName	
	```
	@DisplayName("DisplayName Demo")
	class JUnit5Test {
	    @Test
	    @DisplayName("Custom test name")
	    void testWithDisplayName() {
	    }
	```

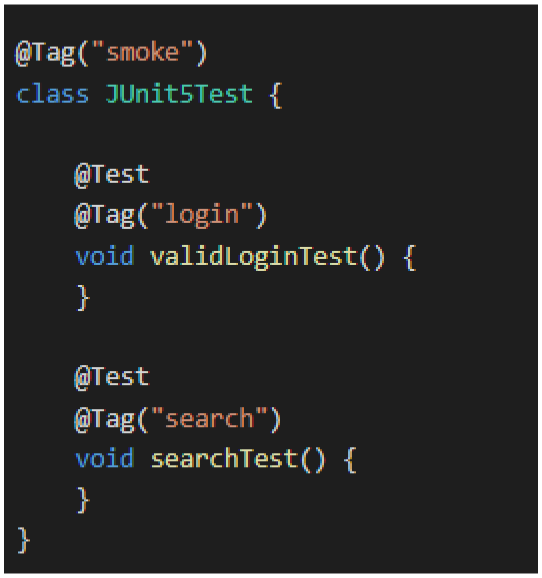
@ParameterizedTest	
	```
	class JUnit5Test {

	    @ParameterizedTest
	    @ValueSource(strings = { "cali", "bali", "dani" })
	    void endsWithI(String str) {
	        assertTrue(str.endsWith("i"));
	    }
	}
	```
	 Parameterized tests make it possible to run a test multiple times with different arguments.

@RepeatedTest @Tag	JUnit 5 has the ability to repeat a test a specified number of times simply by annotating a method with `@RepeatedTest` and specifying the total number of repetitions desired. 

```java
@RepeatedTest(value = 5, name = "{displayName} {currentRepetition}/{totalRepetitions}")
@DisplayName("RepeatingTest")
void customDisplayName(RepetitionInfo repInfo, TestInfo testInfo) {
    int i = 3;
    System.out.println(testInfo.getDisplayName() +
        "-->" + repInfo.getCurrentRepetition()
    );

    assertEquals(repInfo.getCurrentRepetition(), i);
}
```

	We can use this annotation to declare tags for filtering tests, either at the class or method level. 

```java
@Tag("smoke")
class JUnit5Test {

    @Test
    @Tag("login")
    void validLoginTest() {
    }

    @Test
    @Tag("search")
    void searchTest() {
    }
}
```

Writing Tests in JUnit

In JUnit, test methods are marked with `@Test` annotation. To run the method, JUnit first constructs a fresh instance of the class then invokes the annotated method. Any exceptions thrown by the test will be reported by JUnit as a failure. If no exceptions are thrown, the test is assumed to have succeeded.

Assertions

Assertions help in validating the expected output with actual output of a testcase. All the assertions are in the org.junit.jupiter.api.Assertions class. All assert methods are `static`, it enables better readable code.

assertEquals()

The assertEquals() method compares two objects for equality, using their equals() method.

assertTrue() + assertFalse()

The assertTrue() and assertFalse() methods tests a single variable to see if its value is either true, or false.

assertNull() + assertNotNull()

The assertNull() and assertNotNull() methods test a single variable to see if it is null or not null.

assertThat()

The assertThat() method compares an object to an org.hamcrest.Matcher to see if the given object matches whatever the Matcher requires it to match.

https://mvnrepository.com/artifact/org.hamcrest/hamcrest-all/1.3

JUnit (Hamcrest) comes with a few builtin matchers you can use.

Core Matchers

Before you start implementing your own Matcher's, you should look at the core matchers that come with JUnit already. Here is a list of the matcher methods:

Core		
	any()	Matches anything
	is()	A matcher that checks if the given objects are equal.
	describedAs()	Adds a description to a Matcher
Logical		
	allOf()	Takes an array of matchers, and all matchers must match the target object.
	anyOf()	Takes an array of matchers, and at least one of the matchers must report that it matches the target object.
	not()	Negates the output of the previous matcher.
Object		
	equalTo()	A matcher that checks if the given objects are equal.
	instanceOf()	Checks if the given object is of type X or is compatible with type X

notNullValue() + nullValue()	Tests whether the given object is null or not null.
sameInstance()	Tests if the given object is the exact same instance as another.

Actually, all of the above are static methods which take different parameters, and return a Matcher.

You will have to play around with matchers a little, before you get the hang of them. They can be quite handy.