LEARNING junit

#junit

Table of Contents

About

Chapter 1: Getting started with junit

Remarks

JUnit is a simple framework to write repeatable tests for Java programming language. It is an instance of the xUnit architecture for unit testing frameworks.

Main features consist of:

- **Assertions**, that let you customize how to test values in your tests
- **Test runners**, that let you specify how to run the tests in your class
- **Rules**, that allow you to flexibly modify the behaviour of tests in your class
- **Suites**, that allow you to build together a suite of tests from many different classes

Useful extension for JUnit:

- AssertJ: Fluent assertions for java
- Mockito: Mocking framework for java

Versions

Version	ReleaseDate
JUnit 5 Milestone 2	2016-07-23
JUnit 5 Milestone 1	2016-07-07
JUnit 4.12	2016-04-18
JUnit 4.11	2012-11-14
JUnit 4.10	2011-09-28
JUnit 4.9	2011-08-22
JUnit 4.8	2009-12-01
JUnit 4.7	2009-07-28
JUnit 4.6	2009-04-14

Examples

Installation or Setup

Since JUnit is a Java library, all you have to do to install it is to add a few JAR files into the classpath of your Java project and you're ready to go.

You can download these two JAR files manually: junit.jar & hamcrest-core.jar.

If you're using Maven, you can simply add in a dependency into your `pom.xml`:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
```

Or if you're using Gradle,add in a dependency into your `build.gradle`:

```
apply plugin: 'java'

dependencies {
    testCompile 'junit:junit:4.12'
}
```

After this you can create your first test class:

```
import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class MyTest {
    @Test
    public void onePlusOneShouldBeTwo() {
        int sum = 1 + 1;
        assertEquals(2, sum);
    }
}
```

and run it from command line:

- Windows `java -cp .;junit-X.YY.jar;hamcrest-core-X.Y.jar org.junit.runner.JUnitCore MyTest`
- Linux or OsX `java -cp .:junit-X.YY.jar:hamcrest-core-X.Y.jar org.junit.runner.JUnitCore MyTest`

or with Maven: `mvn test`

Basic unit test example

This example is a basic setup for unittesting the StringBuilder.toString() using junit.

```
import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class StringBuilderTest {

    @Test
    public void stringBuilderAppendShouldConcatinate()  {
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.append("String");
        stringBuilder.append("Builder");
        stringBuilder.append("Test");

        assertEquals("StringBuilderTest", stringBuilder.toString());
    }

}
```

@Before, @After

An annotated method with `@Before` will be executed before every execution of `@Test` methods. Analogous an `@After` annotated method gets executed after every `@Test` method. This can be used to repeatedly set up a Test setting and clean up after every test. So the tests are independent and preparation code is not copied inside the `@Test` method.

Example:

```
import static org.junit.Assert.assertEquals;

import java.util.ArrayList;
import java.util.List;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class DemoTest {

    private List<Integer> list;

    @Before
    public void setUp() {
        list = new ArrayList<>();
        list.add(3);
        list.add(1);
        list.add(4);
        list.add(1);
        list.add(5);
        list.add(9);
    }

    @After
    public void tearDown() {
        list.clear();
    }

    @Test
    public void shouldBeOkToAlterTestData() {
        list.remove(0); // Remove first element of list.
        assertEquals(5, list.size()); // Size is down to five
    }

    @Test
    public void shouldBeIndependentOfOtherTests() {
        assertEquals(6, list.size());
    }
}
```

Methods annotated with `@Before` or `@After` must be `public void` and with zero arguments.

Catch expected exception

It is possible to easily catch the exception without any `try catch` block.

```
public class ListTest {
  private final List<Object> list = new ArrayList<>();

  @Test(expected = IndexOutOfBoundsException.class)
  public void testIndexOutOfBoundsException() {
    list.get(0);
  }
}
```

The example above should suffice for simpler cases, when you don't want/need to check the message carried by the thrown exception.

If you want to check information about exception you may want to use try/catch block:

```
@Test
public void testIndexOutOfBoundsException() {
    try {
        list.get(0);
        Assert.fail("Should throw IndexOutOfBoundException");
    } catch (IndexOutOfBoundsException ex) {
        Assert.assertEquals("Index: 0, Size: 0", ex.getMessage());
    }
}
```

For this example you have to be aware to always add `Assert.fail()` to ensure that test will be failed when no Exception is thrown.

For more elaborated cases, JUnit has the `ExpectedException @Rule`, which can test this information too and is used as follows:

```
public class SimpleExpectedExceptionTest {
     @Rule
     public ExpectedException expectedException = ExpectedException.none();

     @Test
     public void throwsNothing() {
         // no exception expected, none thrown: passes.
     }

     @Test
     public void throwsExceptionWithSpecificType() {
         expectedException.expect(NullPointerException.class);

         throw new NullPointerException();
     }
```

```
    @Test
    public void throwsExceptionWithSpecificTypeAndMessage() {
        expectedException.expect(IllegalArgumentException.class);
        expectedException.expectMessage("Wanted a donut.");

        throw new IllegalArgumentException("Wanted a donut.");
    }
}
```

Testing exceptions in JUnit5

To achieve the same in JUnit 5, you use a completely new mechanism:

The tested method

```
public class Calculator {
    public double divide(double a, double b) {
        if (b == 0.0) {
            throw new IllegalArgumentException("Divider must not be 0");
        }
        return a/b;
    }
}
```

The test method

```
public class CalculatorTest {
    @Test
    void triangularMinus5() { // The test method does not have to be public in JUnit5
        Calculator calc = new Calculator();

        IllegalArgumentException thrown = assertThrows(
            IllegalArgumentException.class,
            () -> calculator.divide(42.0, 0.0));
        // If the exception has not been thrown, the above test has failed.

        // And now you may further inspect the returned exception...
        // ...e.g. like this:
        assertEquals("Divider must not be 0", thrown.getMessage());
}
```

Ignoring Tests

To ignore a test, simply add the `@Ignore` annotation to the test and optionally provide a parameter to the annotation with the reason.

```
@Ignore("Calculator add not implemented yet.")
@Test
public void testPlus() {
    assertEquals(5, calculator.add(2,3));
}
```

Compared to commenting the test or removing the `@Test` annotation, the test runner will still report this test and note that it was ignored.

It is also possible to ignore a test case conditionally by using JUnit *assumptions*. A sample use-case would be to run the test-case only after a certain bug is fixed by a developer. Example:

```
import org.junit.Assume;
import org.junit.Assert;
...

@Test
public void testForBug1234() {

    Assume.assumeTrue(isBugFixed(1234));//will not run this test until the bug 1234 is fixed

    Assert.assertEquals(5, calculator.add(2,3));
}
```

The default runner treats tests with failing assumptions as ignored. It is possible that other runners may behave differently e.g. treat them as passed.

JUnit – Basic annotation examples

Here’re some basic JUnit annotations you should understand:

```
@BeforeClass – Run once before any of the test methods in the class, public static void
@AfterClass – Run once after all the tests in the class has been run, public static void
@Before – Run before @Test, public void
@After – Run after @Test, public void
@Test – This is the test method to run, public void
```

Read Getting started with junit online: https://riptutorial.com/junit/topic/1838/getting-started-with-junit

Chapter 2: Custom Test Rules

Remarks

There are benefits for either. Extending `ExternalResource` it's convenient, especially if we only require a `before()` to set something up.

However, we should be aware that, since the `before()` method is executed outside of the `try...finally`, any code that is required to do clean up in `after()` won't get executed if there is an error during the execution of `before()`.

This is how it looks inside `ExternalResource`:

```
before();
try {
    base.evaluate();
} finally {
    after();
}
```

Obviously, if any exception is thrown in the test itself, or by another nested rule, the after will still get executed.

Examples

Custom @TestRule by implementation

This is especially useful if we have a class that we want to extend in the rule. See example below for a more convenient method.

```
import org.junit.rules.TestRule;
import org.junit.runners.model.Statement;

public class AwesomeTestRule implements TextRule {

    @Override
    public Statement apply(Statement base, Description description) {
        return new AwesomeStatement(base);
    }

    private static class AwesomeStatement extends Statement {

        private Statement base;

        public AwesomeStatement(Statement base) {
            this.base = base;
        }

        @Override
        public void evaluate() throws Throwable {
            try {
                // do your magic
                base.evaluate(); // this will call Junit to run an individual test
            } finally {
                // undo the magic, if required
            }
        }
    }

}
```

Custom @TestRule by extension

JUnit has an abstract implementation of `@TestRule` that lets you write a rule in a more simpler way. This is called `ExternalResource` and provides two protected methods that can be extended like this:

```
public class AwesomeTestRule extends ExternalResource {

    @Override
    protected void before() {
        // do your magic
    }

    @Override
    protected void after() {
        // undo your magic, if needed
    }

}
```

Read Custom Test Rules online: https://riptutorial.com/junit/topic/9808/custom-test-rules

Chapter 3: Generate Junit test cases skeleton for existing code

Introduction

Sometimes you need to generate the skeleton for the test cases based on the classes you have in your project.

Examples

Generate Junit test cases skeleton for existing code in Eclipse

Here are the steps to generate test skeleton for existing code:

1. Open **Eclipse**, and choose the project you want to create test cases for
2. In the **Package Explorer**, select the java class you want to generate the Junit test for
3. Go to **File** -> **New** -> **Junit Test Cases**
4. Change the **Source folder** to point to the test using Browse (Note: It is better to separate the source code from the testing code)
5. Change the **Package** based on the destination package you want
6. In the **Class under test**, make sure you enter the class you want to generate the test cases for.
7. Click **Next**
8. Select the methods you want to test for
9. Click **Finish**

Now, you will have a Junit class generated for testing the source class you have

Read Generate Junit test cases skeleton for existing code online: https://riptutorial.com/junit/topic/8648/generate-junit-test-cases-skeleton-for-existing-code

Chapter 4: Ignore test cases in Junit

Introduction

Sometimes you want to ignore some of the test cases you have in Junit. For instance, they are partially done and you want to come back to them later.

Examples

Ignore test case in Junit

1. Go to the test method you want to ignore
2. Before the `@Test` annotation, enter `@Ignore`
3. optional: You can add description why are you ignoring this test method, something like: `@Ignore ("ignoring this test case for now")`

a sample method would be:

```
@Ignore ("not going to test this method now")
@Test
public void test() {
     assertfalse(true);
}
```

Read Ignore test cases in Junit online: https://riptutorial.com/junit/topic/8640/ignore-test-cases-in-junit

Chapter 5: Paramaterizing Tests

Introduction

Sometimes you have a test you need to run multiple times, each time with different data. Parameterizing the test allows you to do this in an easy and maintainable way.

Syntax

- @RunWith(Parameterized.class) //annotation for test class

  @Parameters//annotation for data

Remarks

One benefit to using parameters is that if one set of data fails, execution will just move to the next set of data instead of stopping the whole test.

Examples

Using a Constructor

```
import static org.junit.Assert.assertThat;
import static org.hamcrest.CoreMatchers.is;
import java.util.*;
import org.junit.*;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)
public class SimpleParmeterizedTest {
    @Parameters
    public static Collection<Object[]> data(){
        return Arrays.asList(new Object[][]{
                {5, false}, {6, true}, {8, true}, {11, false}
        });
    }

    private int input;
    private boolean expected;

    public SimpleParmeterizedTest(int input, boolean expected){
        this.input = input;
        this.expected = expected;
    }

    @Test
    public void testIsEven(){
        assertThat(isEven(input), is(expected));
        }
}
```

In data() you supply the data to be used in the tests. Junit will iterate through the data and run the test with each set of data.

Read Paramaterizing Tests online: https://riptutorial.com/junit/topic/9425/paramaterizing-tests

Chapter 6: Test Execution Order

Syntax

- @FixMethodOrder // Runs test using default method sorter
- @FixMethodOrder(MethodSorters) // Runs test using MethodSorter associated with the MethodSorters enum.

Examples

Default Order

Use the annotation -- `@FixMethodOrder(MethodSorters.DEFAULT)`. This runs all tests within the class in a deterministic and somewhat predictable order. The implementation hashes the method names and compares them. In the scenario of a tie, it sorts by lexicographical order.

Code Segment Below Taken from JUnit Github -- MethodSorter.java

```
public int compare(Method m1, Method m2) {
    int i1 = m1.getName().hashCode();
    int i2 = m2.getName().hashCode();
    if(i1 != i2) {
        return i1 < i2 ? -1 : 1;
    }
    return NAME_ASCENDING.compare(m1,m2);
}
```

Example

```
@FixMethodOrder(MethodSorters.DEFAULT)
public class OrderedTest {
    @Test
    public void testA() {}

    @Test
    public void testB() {}

    @Test
    public void testC() {}
}
```

Suppose hashes for `testA`, `testB`, and `testC` are 3, 2, and 1 respectively. Then the execution order is

1. testC
2. testB
3. testA

Suppose hashes for all tests are the same. Since all hashes are the same, execution order is based on lexicographical order. The execution order is

1. testA
2. testB
3. testC

Lexicographical Order

Use the annotation `@FixMethodOrder` with the method sorter `MethodSorters.NAME_ASCENDING`. This will run all tests within the class in a deterministic and predictable order. The implementation compares the method names and in the case of a tie, it compares the methods' `toString()`.

Code Segment Below Taken from JUnit Github -- MethodSorter.java

```
public int compare(Method m1, Method m2) {
    final int comparison = m1.getName().compareTo(m2.getName());
    if(comparison != 0) {
        return comparison;
    }
    return m1.toString().compareTo(m2.toString());
}
```

Example

```
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class OrderedTest {
    @Test
    public void testA() {}

    @Test
    public void testB() {}

    @Test
    public void testC() {}
}
```

The execution order is

1. testA
2. testB
3. testC

Read Test Execution Order online: https://riptutorial.com/junit/topic/5905/test-execution-order

Chapter 7: Testing with DataProviders

Examples

Installation and usage

Installation:

In order to use DataProviders, you need junit-dataprovider .jar :

Github

Direct download

Hamcrest-core-1.3.jar :

Github

Direct download

And add both of this .jar to your project.

Usage:

Add this `import` to your code:

```
import com.tngtech.java.junit.dataprovider.DataProvider;
import com.tngtech.java.junit.dataprovider.DataProviderRunner;
import com.tngtech.java.junit.dataprovider.UseDataProvider;
```

Before the declaration of your class:

```
@RunWith(DataProviderRunner.class)
```

So it looks like this:

```
@RunWith(DataProviderRunner.class)
public class example {
    //code
}
```

How to create DataProviders:

Before whichever function you want it to be a DataProvider, add this decorator:

```
@DataProvider
```

So it would look like this:

```
@DataProvider
public static Object[][] testExampleProvider() {
    return new Object[][]{
        {"param1", "param2", number1}
        {"param1", "param2", number1}
        //You can put as many parameters as you want
    };
}
```

How to use DataProviders:

Before any function you want it to get those params that we return from the DataProvider, add this decorator:

```
@UseDataProvider("testExampleProvider")
```

So your function to test looks like this:

```
@Test
@UseDataProvider("testExampleProvider")
public void testAccount(String param1, String param2, int number) {
    //System.out.println("exampleOfDataProviders");
    //assertEquals(...);
    //assertEquals(...);
}
```

Read Testing with DataProviders online: https://riptutorial.com/junit/topic/7469/testing-with-dataproviders

Chapter 8: Tests

Remarks

Parameter	Context	Details
@BeforeClass	Static	Executed when the class is first created
@Before	Instance	Executed before *each test* in the class
@Test	Instance	Should be declared each method to test
@After	Instance	Executed after *each test* in the class
@AfterClass	Static	Executed before destruction of the class

Example Test Class Format

```
public class TestFeatureA {

    @BeforeClass
    public static void setupClass() {}

    @Before
    public void setupTest() {}

    @Test
    public void testA() {}

    @Test
    public void testB() {}

    @After
    public void tearDownTest() {}

    @AfterClass
    public static void tearDownClass() {}

    }
}
```

Examples

Unit testing using JUnit

Here we have a class `Counter` with methods `countNumbers()` and `hasNumbers()`.

```
public class Counter {

    /* To count the numbers in the input */
    public static int countNumbers(String input) {
        int count = 0;
        for (char letter : input.toCharArray()) {
            if (Character.isDigit(letter))
                count++;
        }
        return count;
    }

    /* To check whether the input has number*/
    public static boolean hasNumber(String input) {
        return input.matches(".*\\d.*");
    }
}
```

To unit test this class, we can use Junit framework. Add the `junit.jar` in your project class path. Then create the Test case class as below:

```
import org.junit.Assert; // imports from the junit.jar
import org.junit.Test;

public class CounterTest {

    @Test // Test annotation makes this method as a test case
    public void countNumbersTest() {
        int expectedCount = 3;
        int actualCount = Counter.countNumbers("Hi 123");
        Assert.assertEquals(expectedCount, actualCount); //compares expected and actual value
    }

    @Test
    public void hasNumberTest() {
        boolean expectedValue = false;
        boolean actualValue = Counter.hasNumber("Hi there!");
        Assert.assertEquals(expectedValue, actualValue);
    }
}
```

In your IDE you can run this class as "Junit testcase" and see the output in the GUI. In command prompt you can compile and run the test case as below:

```
\> javac -cp ,;junit.jar CounterTest.java
\> java  -cp .;junit.jar org.junit.runner.JUnitCore CounterTest
```

The output from a successful test run should look similar to:

```
JUnit version 4.9b2
..
Time: 0.019

OK (2 tests)
```

In the case of a test failure it would look more like:

```
Time: 0.024
```

```
There was 1 failure:
1) CountNumbersTest(CounterTest)
java.lang.AssertionError: expected:<30> but was:<3>
... // truncated output
FAILURES!!!
Tests run: 2,  Failures: 1
```

Fixtures

From Wikipedia:

> A test fixture is something used to consistently test some item, device, or piece of software.

It can also enhance readability of tests by extracting common initialisation / finalisation code from the test methods themselves.

Where common initialisation can be executed once instead of before each tests, this can also reduce the amount of time taken to run tests.

The example below is contrived to show the main options provided by JUnit. Assume a class `Foo` which is expensive to initialise:

```
public class Foo {
    public Foo() {
        // expensive initialization
    }

    public void cleanUp() {
        // cleans up resources
    }
}
```

Another class `Bar` has a reference to `Foo`:

```
public class Bar {
    private Foo foo;

    public Bar(Foo foo) {
        this.foo = foo;
    }

    public void cleanUp() {
        // cleans up resources
    }
}
```

The tests below expect an initial context of a List containing a single `Bar`.

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

import java.util.Arrays;
import java.util.List;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class FixturesTest {

    private static Foo referenceFoo;

    private List<Bar> testContext;

    @BeforeClass
    public static void setupOnce() {
        // Called once before any tests have run
        referenceFoo = new Foo();
    }

    @Before
    public void setup() {
        // Called before each test is run
        testContext = Arrays.asList(new Bar(referenceFoo));
    }

    @Test
    public void testSingle() {
        assertEquals("Wrong test context size", 1, testContext.size());
        Bar baz = testContext.get(0);
        assertEquals(referenceFoo, baz.getFoo());
    }

    @Test
    public void testMultiple() {
        testContext.add(new Bar(referenceFoo));
        assertEquals("Wrong test context size", 2, testContext.size());
        for (Bar baz : testContext) {
            assertEquals(referenceFoo, baz.getFoo());
        }
    }

    @After
    public void tearDown() {
        // Called after each test is run
        for (Bar baz : testContext) {
            baz.cleanUp();
        }
    }

    @AfterClass
    public void tearDownOnce() {
        // Called once after all tests have run
        referenceFoo.cleanUp();
    }
}
```

In the example the `@BeforeClass` annotated method `setupOnce()` is used to create the `Foo` object, which is expensive to initialise. It is important that it not be modified by any of the tests, otherwise the outcome of the test run could be dependent on the order of execution of the individual tests. The idea is that each test is independent and tests one small feature.

The `@Before` annotated method `setup()` sets up the test context. The context may be modified during test execution, which is why it must be initialised before each test. The equivalent effect could be achieved by including the code contained in this method at the start of each test method.

The `@After` annotated method `tearDown()` cleans up resources within the test context. It is called after each test invocation, and as such is often used to free resources allocated in a `@Before` annotated method.

The `@AfterClass` annotated method `tearDownOnce()` cleans up resources once all tests have run. Such methods are typically used to free resources allocated during initialisation or in a `@BeforeClass` annotated method. That said, it's probably best to avoid external resources in unit tests so that the tests don't depend on anything outside the test class.

Unit testing using Theories

From JavaDoc

> The Theories runner allows to test a certain functionality against a subset of an infinite set of data points.

Running theories

```
import org.junit.experimental.theories.Theories;
import org.junit.experimental.theories.Theory;
import org.junit.runner.RunWith;

@RunWith(Theories.class)
public class FixturesTest {

    @Theory
    public void theory(){
        //...some asserts
    }
}
```

Methods annotated with `@Theory` will be read as theories by Theories runner.

@DataPoint annotation

```
@RunWith(Theories.class)
public class FixturesTest {

    @DataPoint
    public static String dataPoint1 = "str1";
    @DataPoint
    public static String dataPoint2 = "str2";
    @DataPoint
    public static int intDataPoint = 2;

    @Theory
    public void theoryMethod(String dataPoint, int intData){
        //...some asserts
    }
}
```

Each field annotated with `@DataPoint` will be used as a method parameter of a given type in the theories. In example above `theoryMethod` will run two times with following parameters: `["str1", 2]` , `["str2", 2]`

@DataPoints annotation @RunWith(Theories.class) public class FixturesTest {

```
    @DataPoints
    public static String[] dataPoints = new String[]{"str1", "str2"};
    @DataPoints
    public static int[] dataPoints = new int[]{1, 2};

    @Theory
    public void theoryMethod(String dataPoint, ){
        //...some asserts
    }
}
```

Each element of the array annotated with `@DataPoints` annotation will be used as a method parameter of a given type in the theories. In example above `theoryMethod` will run four times with following parameters: `["str1", 1], ["str2", 1], ["str1", 2], ["str2", 2]`

Performance measurement

If you need to check if your testing method takes too long to execute, you can do that by mentioning your expected execution time using timeout property of @Test annotation. If the test execution takes longer than that number of milliseconds it causes a test method to fail.

```
public class StringConcatenationTest {

    private static final int TIMES = 10_000;

    // timeout in milliseconds
    @Test(timeout = 20)
    public void testString(){

        String res = "";

        for (int i = 0; i < TIMES; i++) {
            res += i;
        }

        System.out.println(res.length());
    }

    @Test(timeout = 20)
    public void testStringBuilder(){

        StringBuilder res = new StringBuilder();

        for (int i = 0; i < TIMES; i++) {
            res.append(i);
        }

        System.out.println(res.length());
    }
    @Test(timeout = 20)
    public void testStringBuffer(){

        StringBuffer res = new StringBufferr();

        for (int i = 0; i < TIMES; i++) {
            res.append(i);
        }

        System.out.println(res.length());
    }

}
```

In most cases without JVM warm-up `testString` will fail. But `testStringBuffer` and `testStringBuilder` should pass this test successfully.

Read Tests online: https://riptutorial.com/junit/topic/5414/tests

Credits

S. No	Chapters	Contributors
1	Getting started with junit	acdcjunior, Andrii Abramov, Ashish Bhavsar, cheffe, Community , Daniel Käfer, Honza Zidek, Lachezar Balev, NamshubWriter, nishizawa23, Roland Weisleder, Rufi, Simulant, Squidward, svgameren, t0mppa, Tim Tong, Tomasz Bawor
2	Custom Test Rules	pablisco
3	Generate Junit test cases skeleton for existing code	Hamzawey
4	Ignore test cases in Junit	Hamzawey
5	Paramaterizing Tests	Hai Vu, selotape, user7491506
6	Test Execution Order	Tim Tong
7	Testing with DataProviders	Manuel
8	Tests	Andrii Abramov, gar, Ram, Robert, Sergii Bishyr, Squidward