JUnit Testing Part I: Setup With Simple Example

(/users/369193/odalinjo.html) by Dusan Odalovic (/users/369193/odalinjo.html) · Mar. 25, 16 · Java Zone (/java-jdk-development-tutorials-tools-news)

👍 Like (9)   💬 Comment (0)   ⭐ Save   🐦 Tweet   👁 10.26k Views

JUnit is one of available *Java* libraries we can use to test our application code. Let's get started by buliding simple project using *Maven (https://maven.apache.org/)*.

Generating Project With JUnit Support

Go generate simple *Java* artifact using *Maven, which* can be done quite easilly using command line.

```
mvn archetype:generate -DgroupId={GROUP_ID} -DartifactId={ARTIFACT_ID} -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

Just execute upper line in your console, giving it some `{GROUP_ID}` and `{ARTIFACT_ID}`. Make sure you haven *Maven* installed and on your path, so that you can run it as described. This is not related to JUnit testing, just give it any values, I will use `com.mydomain` as `{GROUP_ID}`, and `junit-demo` as `{ARTIFACT_ID}`. That should generate new java project ready to demo *JUnit* tests. You can import generated application into IDE of your choice.

To confirm we have all we need, open generated `pom.xml` file in the root of your project and verify there's dependency to *JUnit* library in there (I'm using 4.12 version which is the latest stable version of *JUnit* at the time of writing this article).

```
1  <dependency>
2      <groupId>junit</groupId>
3      <artifactId>junit</artifactId>
4      <version>4.12</version>
5      <scope>test</scope>
6  </dependency>
```

Once we confirm there's given dependency, we will proceed with some theory alongside example code. We will create simple calculator class which can do some basic operations as adding, subtracting, multiplying, dividing numbers. We want to create *JUnit* tests to make sure the calculator class is working as expected and has production ready quality.

Creating System Under Test (SUT) and Test case

JUnit testing takes care of testing our application at low level, meaning we want to make sure our classe methods are tested against different conditions, and they provide expected results in such cases. Our methods we want to test can, during application runtime, receive different values for input parameters. We want to make sure our method is tested against these. Of course - not against all possible values, since that's not possible, but against some meaningful values, as well as some edge cases where our methods can receive, let's say *null* as parameter, or some other value that could potentially crush our application. Let's start writing `com.mydomain.Calculator.java` and *JUnit* tests for it, `com.mydomain.CalculatorTest.java`. Test classes are also known as ***Test cases.***
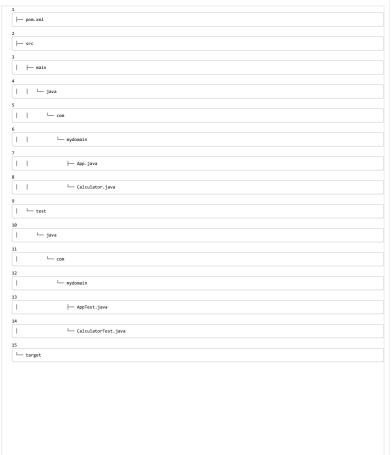
It's a common practice to name test class after class we're testing (sometimes referred to as **System Under Test - SUT**), by appending ***Test*** suffix to it. In our case, our *SUT* is `Calculator.java`, so our test class should be `CalculatorTest` .java. Also, common practice is that test class should reside in same package as *SUT* one (in our case, both are in `com.mydomain` package, but test class is in `src/test/java` tree, and *SUT* is in `src/main/java`. These are maven defaults which state that application code should go to X, whereas test code should be inside `src/test` packages. Our *Calculator* class should look like:

```
1  package com.mydomain;
2
3  public class Calculator {
4
5      public long add(long first, long second) {
6          return first + second;
7      }
8
9      public long subtract(long first, long second) {
10         return first - second;
11     }
12
```

```
13
    public long multiply(long first, long second) {
14
        return first * second;
15
    }
16

17
    public long divide(long first, long second) {
18
        return first / second;
19
    }
20
}
```

Without writing *JUnit* tests for given piece of code, we can only *HOPE* our code will work as expected. We should be able to exercise this piece of code before deploying it to production so that we can be confident code is working as expected.

Our application should have layout such as given below (target directory is incomplete here, but that's irrelevant):

```
1
├── pom.xml
2
├── src
3
│   ├── main
4
│   │   └── java
5
│   │       └── com
6
│   │           └── mydomain
7
│   │               ├── App.java
8
│   │               └── Calculator.java
9
│   └── test
10
│       └── java
11
│           └── com
12
│               └── mydomain
13
│                   ├── AppTest.java
14
│                   └── CalculatorTest.java
15
└── target
```

Let's now write some *JUnit* tests in order to exercise our application. Our `CalculatorTest` class could look like:

```
1
package com.mydomain;
2

3
import org.junit.Test;
4

5
import static org.hamcrest.CoreMatchers.is;
6
```

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

7

8
```
public class CalculatorTest {
```

9
```
@Test
```

10

11
```
final long result = new Calculator().add(2, 3);
```

12
```
assertThat(result, is(5L));
```

13
```
}
```

14

15
```
@Test
```

16
```
public void twoAndZeroIsTwo() throws Exception {
```

17
```
final long result = new Calculator().add(2, 0);
```

18
```
assertThat(result, is(2L));
```

19
```
}
```

20

21
```
@Test
```

22
```
public void twoAndMinusTwoIsZero() throws Exception {
```

23
```
final long result = new Calculator().add(2, -2);
```

24
```
assertThat(result, is(0L));
```

25
```
}
```

26

27
```
@Test
```

28
```
public void threeMinusTwoIsOne() throws Exception {
```

29
```
final long result = new Calculator().subtract(3, 2);
```

30
```
assertThat(result, is(1L));
```

31

32

33
```
@Test
```

34
```
public void threeMinusThreeIsZero() throws Exception {
```

35
```
final long result = new Calculator().subtract(3, 3);
```

36
```
assertThat(result, is(0L));
```

37
```
}
```

38

39
```
@Test
```

40
```
public void threeMinusMinusThreeIsSix() throws Exception {
```

41
```
final long result = new Calculator().subtract(3, -3);
```

42
```
assertThat(result, is(6L));
```

43
```
}
```

44

45
```
@Test
```

46
```
public void threeXThreeIsNine() throws Exception {
```

```
47
        final long result = new Calculator().multiply(3, 3);
48
        assertThat(result, is(9L));
49
    }
50

51
    @Test
52
    public void threeXZeroIsZero() throws Exception {
53
        final long result = new Calculator().multiply(3, 0);
54
        assertThat(result, is(0L));
55
    }
56

57
    @Test
58
    public void threeXMinusThreeIsMinusNine() throws Exception {
59
        final long result = new Calculator().multiply(3, -3);
60
        assertThat(result, is(-9L));
61
    }
62

63
}
```

Idea is to exercise our public methods present in `Calculator` class. We should give it various inputs, and *assert* that results are as expected. In order to create *JUnit* test, we should just mark our method with `@Test` *JUnit* annotation (`org.junit.Test`). When we execute this set of tests we wrote, *JUnit* will executed all methods marked with `@Test` annotation.

In each test method we have two phases:

1. Call *SUT* method for particular input (parameters)
2. Assert that result matches our expectancy (like we assert that 3 x 3 should be 9)

Running Tests and Verifying Results

The easy way to run all the tests is to package our application, using *Maven* command in our comand line:

```
1
    mvn clean package
```

In console output, after our application code was compiled, we can see that *JUnit* tests are executed:

```
1
    ----------------------------------------------------

2
    T E S T S

3
    ----------------------------------------------------

4
    Running com.mydomain.AppTest

5
    Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.013 sec - in com.mydomain.AppTest

6
    Running com.mydomain.CalculatorTest

7
    Tests run: 9, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.004 sec - in com.mydomain.CalculatorTest
```

⬆

DZone (/)

/ Java Zone (/java-jdk-development-tutorials-tools-news)

```
1
    Results :

    Tests run: 10, Failures: 0, Errors: 0, Skipped: 0
```

Over a million developers have joined DZone. Sign In / Join 👤 🔍

REFCARDZ (/REFCARDZ)   **GUIDES** (/GUIDES)   **ZONES** (/PORTALS) | AGILE (/AGILE-METHODOLOGY-TRAINING-TOOLS-NEWS)

We can see that there's total of 9 tests being run in `CalculatorTest` and that there were no failures (Failures: 0 in log output). If some tests were failing, we would be presented with an error message, and *Maven* build would have failed. Let's change implementation of `threeMinusMinusThreeIsSix` method to wrongly assert that 3 multiplied by -3 should be 2.

```
1
    @Test

2
    public void threeMinusMinusThreeIsSix() throws Exception {

3
        final long result = new Calculator().subtract(3, -3);

4
        assertThat(result, is(2L));

5
    }
```

When repeating upper maven command to package app, we will get output:

```
1
    ----------------------------------------------------

2
    T E S T S

3
    ----------------------------------------------------

4
    Running com.mydomain.AppTest

5
    Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.013 sec - in com.mydomain.AppTest

6
    Running com.mydomain.CalculatorTest

7
    Tests run: 9, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.027 sec <<< FAILURE! - in com.mydomain.CalculatorTest

8
    threeMinusMinusThreeIsSix(com.mydomain.CalculatorTest) Time elapsed: 0.005 sec <<< FAILURE!

9
    java.lang.AssertionError:

10
    Expected: is <2L>

11
    but: was <6L>

12
    ....
```

⬆

DZone (/)

/ Java Zone (/java-jdk-development-tutorials-tools-news)

```
13
    ....
```

Over a million developers have joined DZone. Sign In / Join 👤 🔍

REFCARDZ (/REFCARDZ)   **GUIDES** (/GUIDES)   **ZONES** (/PORTALS) | AGILE (/AGILE-METHODOLOGY-TRAINING-TOOLS-NEWS)

```
15
    Tests run: 10, Failures: 1, Errors: 0, Skipped: 0

16

17
    [INFO] ----------------------------------------------------

18
    [INFO] BUILD FAILURE
```

»

We clearly see that build failed, and what's causing build to fail. It's quite handy we can quickly execute tests and easilly see if something is failing and why. Test can fail if *SUT* has issues, or *JUnit* tests are not properly written (such as we did above - just for demo purpose).

Bare in mind that *JUnit* test should have quality standards same as application code, so make sure you apply all the good practices you already apply to application code. Also, *JUnit* tests should be independent of each other, and must be able to run in any order given. They should also be quick to execute and easy to verify results.

Key Takeaways

- *JUnit* **is library for automated unit testing**
- **Idea is to excercise the application code for various cases before application goes live, thus preventing issues that might happen during application runtime.**
- **Each** *JUnit* **test should be independent of other tests, and should be able to run fast, so that we can execute them frequently.**

The next post (https://odalinjo.wordpress.com/2016/03/22/junit-testing-part-ii-using-mockito-for-dependencies-configuration/) will be related to extending *JUnit* test using *Mockito* library.

Hope you liked the post and feel free to download application sources at GitHub (https://github.com/dodalovic/blog-junit-testing).

Stay tuned and please - don't forget to subscribe in case you're eager to find out what's coming next in upcoming posts.

*The Java Zone is brought to you in partnership with AppDynamics (/go?
i=91824&u=http%3A%2F%2Fwww.appdynamics.com%2F2Ffree%3Futm_source%3Ddzone%26utm_medium%3Dsponsorship%26utm_campaign%3Ddzone%2520java%2520zone%26utm_content%3Dpost-rol%26utm_term%3Dsstrial). AppDynamics helps you gain the fundamentals behind application performance, and implement best practices so you can proactively analyze and act on performance problems as they arise, and more specifically with your Java applications. Start a Free Trial (/go?
i=91824&u=http%3A%2F%2Fwww.appdynamics.com%2F2Ffree%3Futm_source%3Ddzone%26utm_medium%3Dsponsorship%26utm_campaign%3Ddzone%2520java%2520zone%26utm_content%3Dpost-rol%26utm_term%3Dsstrial).*

Topics: JUNIT, MAVEN, JAVA

Published at DZone with permission of Dusan Odalovic . (https://odalinjo.wordpress.com/2016/03/13/junit-testing-part-i-setup-with-simple-example/)
Opinions expressed by DZone contributors are their own.

Java Partner Resources

Java Code Challenge: Bus Gossip Solution

(/users/1338295/samberic.html) by Sam Atkinson (/users/1338295/samberic.html) 🏅 MVB ④ · Jun 23, 16 · Java Zone (/java-jdk-development-tutorials-tools-news)

👍 Like (4)   💬 Comment (0)   ⭐ Save   🐦 Tweet   👁 2,756 Views

Check out this 8-step guide (/go?i=124041&u=http%3A%2F%2Fpages.zeroturnaround.com%2FRocket-Powered-White-Paper_Rocket-Powered-White-Paper.html%3Futm_source%3Ddzone%26utm_medium%3Djavazone_partner_resources%26utm_campaign%3Drocketpowered) to see how you can increase your productivity by skipping slow application redeploys and by implementing application profiling, as you code! Brought to you in partnership with ZeroTurnaround (/go?i=124041&u=http%3A%2F%2Fpages.zeroturnaround.com%2FRocket-Powered-White-Paper_Rocket-Powered-White-Paper.html%3Futm_source%3Ddzone%26utm_medium%3Djavazone_partner_resources%26utm_campaign%3Drocketpowered).

If you've not seen the original challenge, then check the article out here (https://dzone.com/articles/java-code-challenge-bus-gossip.).

Challenge #2 in the series is much more difficult than challenge one. This thing can get complex if you don't plan it out a bit. Well done to everyone who submitted a solution, and remember that none of the below is a criticism of the participants, but hopefully can help everyone to improve their code and to help start a discussion.

For me (and most of the solutions created) the challenge breaks down into two sections:

- The "simulation" (or "day"), where we run through a whole chunk of for loops to find the answer, specifically loop through all the minutes of the day, loop through the drivers to advance their stop, loop through to exchange gossip, loop through to see if have a solution.

- The Bus Driver, who is most likely stateful for knowing which gossip they've collected.

With that many loop-throughs, it's easy to produce code that's hard to understand and maintain. Take for example this solution for the simulation:

```
 1    public class Gossip {
 2
 3        private int mins = 0;        // elapsed minutes
 4        private int[][] routes;      // the bus routes
 5        private int allGossip;       // all gossip (200-1)
 6        private int[] currentLocation; // current location of each driver
 7        private int[] currentGossip;   // current gossip set
 8        private String result = null; // result of evaluation;
 9        private boolean isDebug = false;
10
11        public Gossip(final int[][] routes) {
12            Objects.requireNonNull(routes, "Illegal number of routes");
13            if(routes.length ==0 || routes.length > 32) {
14                throw new IllegalArgumentException("Invalid number of routes");
15            }
16            this.routes = routes;
17            this.allGossip = (int) Math.pow(2, routes.length)-1;
18            this.currentLocation = new int[routes.length];
19            this.currentGossip = new int[routes.length];
20            for(int i=0; i<routes.length; ++i) {
21                currentGossip[i] = 1 << i;
22            }
23        }
24
25        public String getResult() {
26            // only eval once
27            if(result == null) {
28                // loop until all gossip is disseminated, or until we've tried for 8 hours
```

```
29          for(mins = 0; mins <= 60 * 8; ++mins) {

30              // exchange gossip

31              for(int i = 0; i < routes.length; ++i) {

32                  for(int j = 0; j < routes.length; ++j) {

33                      if(i == j) {

34                          continue;

35                      }

36                      if(routes[i][currentLocation[i]] == routes[j][currentLocation[j]]) {

37                          currentGossip[i] |= currentGossip[j];

38                      }

39                  }

40              }

41              if(isDebug) {

42                  logState();

43              }

44              // evaluate completion criteria

45              if(IntStream.of(currentGossip).allMatch(g -> g == allGossip)) {

46                  result = (mins + 1) + "";
```

```
47              }

48          }

              // move all drivers

50          for(int i = 0; i < routes.length; ++i) {

51              currentLocation[i] = ++currentLocation[i] % routes[i].length;

52          }

53      }

54      if(result == null) {

55          result = "never";

56      }

57  }

58  return result;

59  }

60

61  // logs the time, gossip-set, route, and current location for each driver

62  void logState() {

63      System.out.println("t = "+ mins);

64      for(int i=0; i<routes.length; ++i) {

65          System.out.print(i);

66          System.out.print("\t[");

67          for(int j=0; j<currentGossip.length; ++j) {

68              System.out.print((currentGossip[i] & 1<<j)>>j);
```

```
70        System.out.print("]");
```

Over a million developers have joined DZone. Sign In / Join

REFCARDZ (/REFCARDZ)   GUIDES (/GUIDES)   ZONES (/PORTALS) | AGILE (/AGILE-METHODOLOGY-TRAINING-TOOLS-NEWS)

```
              for(int j=0; j<routes[i].length; ++j) {
```

```
72            System.out.print("\t");
```

```
73            System.out.print(routes[i][j]);
```

```
74            if(j == currentLocation[i]) {
```

```
75                System.out.print("*"); // current_loc stop
```

```
76                }
```

```
77            }
```

```
78        System.out.println();
```

```
79        }
```

```
80      }
```

```
81
```

```
82      public Gossip setDebug(boolean isDebug) {
```

```
83        this.isDebug = isDebug;
```

```
84        return this;
```

```
85      }
```

```
86
```

```
87
```

```
88  }
```

DZone (/)

/ Java Zone (/java-jdk-development-tutorials-tools-news)

Over a million developers have joined DZone. Sign In / Join

REFCARDZ (/REFCARDZ) GUIDES (/GUIDES) ZONES (/PORTALS) | AGILE (/AGILE-METHODOLOGY-TRAINING-TOOLS-NEWS)

For me, although the solution may be correct, it's hard to understand and would turn into a maintenance nightmare. The good idea is correct but the variable names do not describe what is occurring, just loop-and-switch style. It would be much better to pull this out into smaller methods.

This also would make the comments redundant. Comments are a bad thing for maintainable code. They quickly become outdated, incorrect and out of place. Most developers (myself included) routinely ignore them now. Instead, create smaller methods and use method names as documentation.

```
1   // move all drivers
2           for(int i = 0; i < routes.length; ++i) {
3               currentLocation[i] = ++currentLocation[i] % routes[i].length;
4           }
5   …becomes….
6
7       public void advanceAllDriversToNextLocation(){
8           for(int i = 0; i < routes.length; ++i) {
9               currentLocation[i] = ++currentLocation[i] % routes[i].length;
10          }
11      }
```

A really good example of this comes from GitHub user @lutzhorn who has broken down the solution into lots of nice small helper methods.

```
1   /**
2    * A day with a set of drivers. The only public method can be used to calculate
3    * the minute after which all drivers know all gossips.
4
5
6    */
7   public class Day {
8
9       private final List<BusDriver> drivers = new ArrayList<>();
10      private final List<Integer> allGossips = new ArrayList<>();
11
12      public Day(BusDriver... drivers) {
13          this.drivers.addAll(Arrays.asList(drivers));
14          for (final BusDriver driver : this.drivers) {
15              allGossips.add(driver.getGossip());
16          }
17      }
18
19      public Day(List<BusDriver> drivers) {
20          this.drivers.addAll(drivers);
21          for (final BusDriver driver : this.drivers {
```

DZone (/)

/ Java Zone (/java-jdk-development-tutorials-tools-news)

Over a million developers have joined DZone. Sign In / Join

REFCARDZ (/REFCARDZ) GUIDES (/GUIDES) ZONES (/PORTALS) | AGILE (/AGILE-METHODOLOGY-TRAINING-TOOLS-NEWS)

```
22
        allGossips.add(driver.getGossip());
23
      }
24
    }
25
```

```
26
     * Calculates the minute after which all drivers know all gossips.
27
28
```

```
29
     * @return the minutea after which all drivers know all gossips.
30
     * @throws GossipsAreNeverKnownByAllDriversException when not all drivers
31
     * know all gossips after the complete day
32
     */
33
    public Integer minuteAllDriversKnowAllGossips() throws GossipsAreNeverKnownByAllDriversException {
34
      if (drivers.size() == 1) {
35
        return 0;
36
      }
37

38
      for (int minute = 1; minute <= 480; minute++) {
39
        final Map<Integer, Set<BusDriver>> stops = calculateDriversAtStops(minute);
40
        shareGossips(stops);
41
        if (allDriversKnowAllGossip()) {
42
          return minute;
43
        }
44
      }
45

46
      throw new GossipsAreNeverKnownByAllDriversException();
47
    }
```

```
48
```

```
    private boolean allDriversKnowAllGossip() {
50
```

```
51
      boolean allDrivesknowAllGossips = true;
52
      for (final BusDriver driver : drivers) {
53
        if (!allGossips.equals(driver.getKnownGossips())) {
54
          allDrivesknowAllGossips = false;
55
        }
56
      }
57
      return allDrivesknowAllGossips;
58
    }
59

60
    private Map<Integer, Set<BusDriver>> calculateDriversAtStops(int minute) {
61
      // For each driver get the stop where he is now.
62
```

```
         final Map<Integer, Set<BusDriver>> stops = new HashMap<>();
63
         for (final BusDriver driver : drivers) {
64
             final int currentStopOfDriver = driver.getStop(minute);
65
             final Set<BusDriver> driversAtStop = stops.getOrDefault(currentStopOfDriver, new HashSet<>());
66
             driversAtStop.add(driver);
67
             stops.put(currentStopOfDriver, driversAtStop);
68
         }
69
         return stops;
70
```

```
72
         private void shareGossip(final Map<Integer, Set<BusDriver>> stops) {
73
             // Loop over all stops and share gossips.
74
             for (final Integer stop : stops.keySet()) {
75
                 final Set<BusDriver> driversAtStop = stops.get(stop);
76
                 if (driversAtStop.size() > 1) {
77
                     for (BusDriver driver : driversAtStop) {
78
                         for (BusDriver otherDriver : driversAtStop) {
79
                             otherDriver.nowKnowsGossip(driver.getKnownGossips());
80
                         }
81
                     }
82
                 }
83
             }
84
         }
85
     }
```

There's still a few comments in there I would remove, but on the whole that's a good solution.

The other big question is the design of the bus/driver object. All the solutions converged on the fact the object would have the state of which gossips the driver knew (and that the gossip would be an "ID" assigned to each driver at construction).

The main variation was regarding the location of the bus. For the non-bonus question it was very feasible to have this knowledge be stateless and calculated on the fly.

```
1    public int getStop(int minute) {

2        int effectiveStop = (minute - 1) % route.size();

3        return route.get(effectiveStop);

4    }
```

DZone (/)
/ Java Zone (/java-jdk-development-tutorials-tools-news)
Over a million developers have joined DZone. Sign In / Join
REFCARDZ (/REFCARDZ)　GUIDES (/GUIDES)　ZONES (/PORTALS)　|　AGILE (/AGILE-METHODOLOGY-TRAINING-TOOLS-NEWS)

```
6    // It was also possible to keep the state on each driver, as so.

7

8    public int travelAStop() {

9        if (minutesTravelled >= totalMinutesToTravel) {

10           throw new CannotTravelException("Driver cannot travel. Total time to travel will be exceeded.");

11       }

12       minutesTravelled++;

13       currentStop = route[routeIndex++ % route.length];

14       return currentStop;

15   }
```

Generally it's a good idea to go for immutable objects, which is very much possible for the original question in this case, as the location of the driver can be calculated on the fly for each minute, and is probably the preferred solution in this case.

👍 Like (4)　💬 Comment (0)　⭐ Save　🐦 Tweet　　　　　　　　　　　　　👁 2,756 Views

Topics: CODE, TUTORIAL, CLEAN CODE

DZone
/ Java Zone (/java-jdk-development-tutorials-tools-news)
Over a million developers have joined DZone. Sign In / Join
REFCARDZ (/REFCARDZ)　GUIDES (/GUIDES)　ZONES (/PORTALS)　|　AGILE (/AGILE-METHODOLOGY-TRAINING-TOOLS-NEWS)