# Test-Driven Development

## 1 More on testing

### 1.1 Additional annotations

In the former section we have seen how basic testing is performed. A testing method is marked with the annotation `@Test`, and this indicates to JUnit that the method will be run as a test. The testing method contains one (sometimes more) *assertions*, which is nothing more than a call to one of the methods of class `org.junit.Assert`, like `assertTrue(...)` and `assertEquals(...,...)`.

In this section we are going to learn how to create slightly more complicated tests.

#### Constraints: timeout and expected

Sometimes a bug does not result in an incorrect return value, sometimes it results in an infinite loop or an extremely long response time. If we want to specify a maximum time for our testing methods to run, we can do it by passing a "timeout" parameter to annotation `@Test`, as in the following example:

```
@Test(timeout = 1000)
public void testsThatFinishedBeforeOneSecond() {
    // ...
}
```

If the method does not return a value before 1000 milliseconds have passed, the test will fail.

We can also expect a method to return an exception, as in the following example:

```
@Test(expected = IndexOutOfBoundsException.class)
public void testsNegativeIndecesFail() {
    // ...
}
```

If the method does not throw an `IndexOutOfBoundsException` in this test, the test will fail. Note that this is quite the contrary of the usual behaviour: when a testing method throws an exception, it is usually a symptom of something not working as expected; but this not always the case. Situations in which we may want to expect an exception include requests to lists beyond their size, parsing strings that are not in the right format, and passing negative parameters to methods that only accept positive integers.

**Initialisation: Before and After**

Testing methods usually require the creation of some object. It is quite common that this is needed for all testing method in a testing class.

Instead of repeating the same code in each and every method, which is boring and error-prone, we can create a method to do that before every testing method is called. This method must be annotated with `@Before`. In the same way, if some cleanup must be performed after each testing method ends and before the next testing method starts, this should be marked with the `@After` annotation. See the example:

```
@Before
public void buildUp() {
    // A file is created here to be used in every test.
}
@After
public void cleanUp() {
    // The file is deleted here, after each test ends
}
```

Assuming that your testing class contains three testing methods, the execution path would be: buildUp, first test, cleanUp, buildUp, second test, cleanUp, buildUp, third test, cleanUp. Note that you can change the names of the methods (buildUp and cleanUp are common choices).

### 1.1.1 Heavy initialisation: BeforeClass and AfterClass

Using `@Before` and `After` is appropriate in those cases in which initialisation and clean-up are fast. However, if the resources needed are costly to allocate and release, and if they are not changed inside each testing method, then it is better to just do some initialisation at the beginning of all tests and some clean-up at the end of all test.

This is done by using the annotations `@BeforeClass` and `@AfterClass`. Examples of resources that are typically acquired and released once per class include network connections and database connections.

## 1.2 Test-Driven Development

Test-Driven Development (TDD) is a programming methodology that advocates that tests should be driven before the actual program. This is radically different from writing the program first and then writing the tests as we have done in the past.

The TDD methodology consists of three steps that are repeated in a loop:

1. Write the tests for the next functionality/feature of the program. Make sure they fail. If they do not fail, that means they are not testing anything that was not tested before (and are redundant) or they are incorrect (and should be fixed).

2. Write the minimal code that passes all the new tests.

3. Refactor the code to make it clearer and simpler. Run the tests at the end to make sure the final functionality is right.

There are four main benefits for this style of programming:

- As the tests are written in advance, the *production* code is written in a way that is easy to test.
- As the tests are written in advance, all the code is tested by at least one testing method. Otherwise, programmers can forget to test some methods.
- Writing the tests first makes the programmers think about the real specification of the class or method, focusing on *what* needs to be done before their short-term memory is filled with *how* to do it.
- Errors are detected early, when they are cheap to fix. It is more difficult for bug to remain undetected until later in the development, when it can be more costly to fix them.

The mains shortcoming is that it is difficult to see any progress at the beginning of the project. Time is spent writing tests and nothing "tangible" can be shown to managers or clients. But the effort pays in the long run, when the code evolves in a controlled way, with a strong battery of tests that ensures that bugs do not re-appear and that the functionality is always moving forward.

The TDD methodology can be combined with the "find bugs once" strategy we already know. A program can be developed using TDD, finding most bugs in the process, but some bugs may appear later in the development cycle; if that happens, the "find bugs once" strategy results in adding new tests that will find them as soon as they reappear.