CODEGYM

Java Developer in 12 months

Mentor-supported training program from CodeGym

MODULE 3. JAVA PROFESSIONAL

Lesson #3 JUnit 5

CODEGYM

Lesson plan

- Testing in the life of a programmer
- Tests with JUnit
- Useful annotations in JUnit
- Parameterized tests in JUnit
- JUnit Assertions

MODULE 3. JAVA PROFESSIONAL

CODEGYM

Testing in the life of a programmer

In relation to software, testing can be defined as the task of verifying that a program:

- **Does what it is supposed to do**
- **Does not do what it is not supposed to do**

Testing is a process aimed at finding errors. These errors should be identified, described, and prioritized. Only after prioritizing the errors can we consider the software improvement process to be effective..

Test automation

Manual testing is a traditional method of assessing the quality of software where test cases are executed without the use of automation tools. Testers manually go through test scenarios and create error reports.

This process is labor-intensive and time-consuming, and it is typically used only for small projects without the combination of automated tests.

Automated testing, on the other hand, is performed using specialized scripts, reducing human intervention to a minimum and significantly improving the accuracy and speed of the checks.

Types of testing

In the field of testing, there are various types or directions that are commonly recognized.

For example:

- **Functional Testing:** It is performed to verify each individual function of a website or software application.
- **Integration Testing:** This type of testing focuses on testing the interaction and integration between larger modules or systems of your product.
- **Unit Testing:** It involves testing individual modules or components in isolation to ensure their proper functioning.
- **Performance Testing:** This type of testing is conducted to assess the performance and scalability of your website or application under different load conditions.

There can be even more types of testing: the more complex the product, the more aspects of its development need to be controlled.

CODEGYM

Tests with JUnit

To test Java code, we have a fantastic framework called JUnit. It works great, is constantly updated, and is very popular. And of course, it is tightly integrated with Intellij IDEA.

So, how do you add JUnit to your project? After learning Maven, it's simple: just add the necessary dependency to your pom.xml file.

```
<dependency>
   <groupId>org.junit.jupiter</groupId>
   <artifactId>junit-jupiter-engine</artifactId>
            <version>5.8.1</version>
            <scope>test</scope>
</dependency>
```

The @Test annotation

If we want to test the methods of a class, we need to create another class. Typically, this class has the same name as the original class with the suffix "Test" added to it. For each method, we need to add at least one test method.

The @Test annotation is placed above these test methods:

```
class CalculatorTest {
    @Test
    public void add() {
        Calculator calc = new Calculator();
        int result = calc.add(2, 3);
        assertEquals(result, 5);
    }

}
```

Here is an example of a typical test using **JUnit**. Interesting fact: it uses a special method **assertEquals()**, which compares the provided parameters, as indicated by the word "**equals**" in its name.

If the parameters passed to **assertEquals()** are not equal, the method will throw an exception and the test will be considered failed.

This exception will not prevent the execution of other tests.

CODEGYM

The annotations @BeforeEach and @AfterEach

The code that needs to be executed before each test, such as creating and configuring objects, can be extracted into a separate method and annotated with the @BeforeEach annotation. JUnit will then invoke this method before each test method.

Similarly, you can create a special method that will be called after each test method to clean up resources, write to logs, or perform other necessary actions. This method should be annotated with the @AfterEach annotation.

CODEGYM

The annotations @BeforeAll and @AfterAll

JUnit also allows adding a method that will be called once before all test methods. This method should be annotated with the @BeforeAll annotation.

Similarly, there is a corresponding @AfterAll annotation. JUnit will invoke the method marked with this annotation after all test methods have been executed.

CODEGYM

Useful annotations in JUnit:

@Disabled
allows disabling a specific test so that JUnit does not execute it. It is useful in cases where you notice that a test is not functioning correctly or if you make changes to the code and the test breaks accidentally.

@Nested
allows calling test methods within nested classes.

@TestInstance
allows invoking each test in a separate test instance, using its own class loader. This helps avoid issues with static and even test-specific variables.

@ExtendWith
allows writing different plugins (extensions) to customize the behavior of JUnit. Some extensions can gather test statistics, others can emulate an in-memory file system, and others can simulate working inside a web server, and so on.

@Timeout
allows setting a timeout for test execution. If the test execution exceeds the specified time in the annotation, it is considered as failed.

CODEGYM

Parameterized tests in JUnit

The @ParameterizedTest annotation

allows you to invoke a test multiple times with different parameters. These parameters can include different values, input arguments, or user names.

The @ValueSource annotation

is used to specify parameter values for the parameterized test. However, this annotation does not support null values. To handle null values, you can use the special workaround annotation **@NullSource**.

The @EnumSource annotation

is used to pass all values of a specific **Enum** to the test method.

The @MethodSource annotation

How can we pass objects as parameters, especially if they have a complex construction algorithm? For that, you can simply specify a special helper method that returns a list **(Stream)** of such values.

Parameterized tests with multiple arguments

Of course, you might be wondering what to do if you want to pass multiple arguments to a method. For this, there is a very handy annotation called **@CSVSource**. It allows you to list the parameter values for the method separated by commas.

CODEGYM

JUnit Assertions

Asserts — are special checks that can be inserted at various points in the code. Their task is to determine if something has gone wrong or to verify that everything is going as expected. They allow you to specify "how things should be" in different ways.

The order of comparison is important here because **JUnit** will report something like "**Received value 1, but expected 3**" in the final report.

The general format of such a check is as follows:

```
assertEquals(expected, actual)
```

CODEGYM

The methods assertEquals, assertTrue, assertFalse

Here is a list of some popular assertion methods in JUnit:

assertEquals	Checks that two objects are equal
assertArrayEquals	Checks that two arrays are equal
assertNotNull	Checks that an object is not null
assertNull	Checks that an object is null
assertNotSame	Checks that two objects do not refer to the same object
assertSame	Checks that two objects refer to the same object
assertTrue	Checks that a condition is *true*
assertFalse	Checks that a condition is *false*

The assertAll method

allows you to perform multiple parameter checks. However, if any of the checks fail, the test will be considered as failed. It takes a comment as its first argument, which is a description to be included in the log, followed by any number of assertion functions.

The assertTimeout method

allows you to set constraints on the execution of a specific portion of code inside a method. It takes the time duration as its first parameter and the code (function) that should execute within the specified time as its second parameter.

The assertThrows method

allows you to assert that a specific code block throws the expected exception. It takes the exception class as its first parameter and the code (function) that is expected to throw the exception as its second parameter.

The @Suite annotation

A special annotation for grouping tests together is the @Suite annotation.
It can be used in conjunction with other annotations.

- SuiteDisplayName – sets the display name for the test suite group in the test log;
- SelectPackages – specifies a list of packages where the test classes should be searched;
- IncludeClassNamePatterns – sets the pattern for including test class names

The annotations @Order and @TestMethodOrder

The @TestMethodOrder annotation allows you to specify the order of test method invocations within a test class. It can be useful when you know that the execution order of methods affects their behavior, and you want to ensure a specific order.
The @Order annotation allows you to assign a sequential order number to each method

The @DisplayName annotation

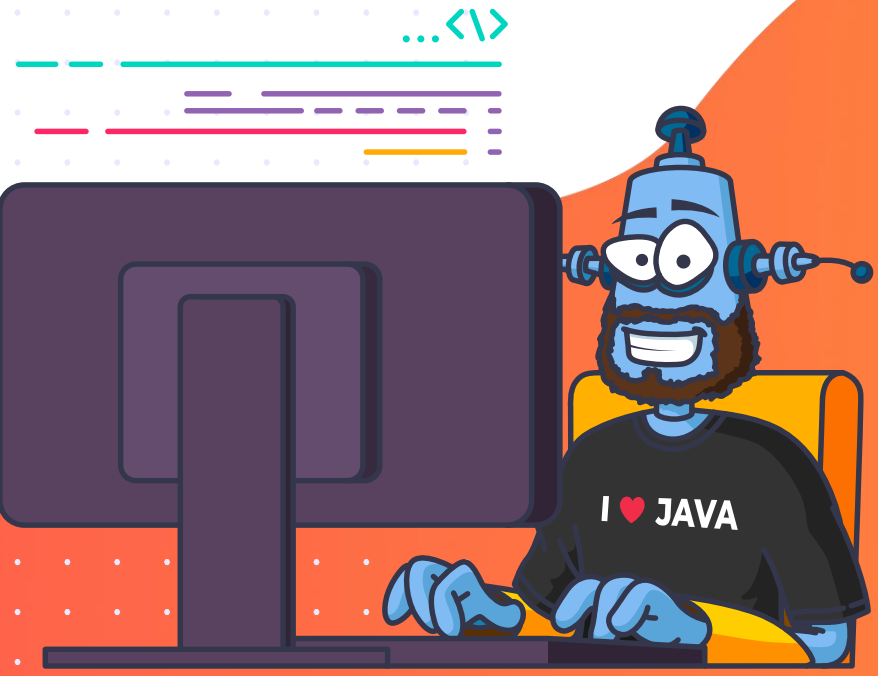
annotation allows you to assign a custom display name to each test. It can be useful when you have a large number of tests and want to create specific scenarios or subsets of tests.

CODEGYM

Homework

MODULE 3. JAVA PROFESSIONAL

Level 3. JUnit 5



MODULE 3. JAVA PROFESSIONAL

CODEGYM

Answers to questions

MODULE 3. JAVA PROFESSIONAL