JAVA 1: BUILD TOOLS, JUNIT + MOCKITO

LESSON 8

<APACHE ANT>

ANT OVERVIEW

- Ant was the **first** among “modern” build tools. In many aspects it is similar to Make. It was released in 2000 and in a short period of time became the most popular build tool for Java projects.
- It has very low learning curve thus allowing anyone to start using it without any special preparation.
- It is based on procedural programming idea.

- Major drawback was XML as the format to write build scripts. XML, being hierarchical in nature, is not a good fit for procedural programming approach Ant uses. Another problem with Ant is that its XML tends to become unmanageably big when used with all but very small projects.
- Later on, as dependency management over the network became a must, Ant adopted Apache Ivy.
- Main benefit of Ant is its control of the build process.

Maven™

MAVEN OVERVIEW

- Apache Maven is a dependency management and a build automation tool, primarily used for Java applications. **Maven continues to use XML files just like Ant but in a much more manageable way.** The name of the game here is convention over configuration.
- While Ant gives the flexibility and requires everything to be written from scratch, **Maven relies on conventions and provides predefined commands (goals).**

- Simply put, Maven allows us to focus on what our build should do, and gives us the framework to do it. Another positive aspect of Maven was that it provided built-in support for dependency management.
- Maven's configuration file, containing build and dependency management instructions, is by convention called *pom.xml*. Additionally, Maven also prescribes strict project structure, while Ant provides flexibility there as well.

- **Maven's strict conventions come with a price of being a lot less flexible than Ant.** Goal customization is very hard, so writing custom build scripts is a lot harder to do, compared with Ant.
- Although Maven has made some serious improvements regarding making application's build processes easier and more standardized, it still comes with a price due to being a lot less flexible than Ant. This lead to the creation of Gradle which combines best of both worlds – Ant's flexibility and Maven's features.

Gradle

GRADLE OVERVIEW

- Gradle is a dependency management and a build automation tool which **was built upon the concepts of Ant and Maven.**
- **One of the first things we can note about Gradle is that it's not using XML files, unlike Ant or Maven.**
- Over time, developers became more and more interested in having and working with a domain specific language – which, simply put, would allow them to solve problems in a specific domain using a language tailored for that particular domain.

- This was adopted by Gradle, which is using a DSL based on Groovy. This led to smaller configuration files with less clutter since the language was specifically designed to solve specific domain problems. Gradle's configuration file is by convention called ***build.gradle***

Maven example: pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://maven.apache.org/POM/4.0.0"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.javaguru</groupId>
    <artifactId>todolist</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>todo-list</name>
    <description>JavaGuru Java 2 course demo project</description>
    <properties>
        <java.version>1.8</java.version>
    </properties>
    <dependencies>
        <!-- https://mvnrepository.com/artifact/junit/junit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.5.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

```
<modelVersion>4.0.0</modelVersion>

<groupId>com.javaguru</groupId>

<artifactId>todolist</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>todo-list</name>
```

Declares to which version of project descriptor this POM conforms.

A universally unique identifier for a project.
It is normal to use a fully-qualified package name to distinguish it from other projects with a similar name (eg. `org.apache.maven`).

The identifier for this artifact that is unique within the group given by the group ID.
An artifact is something that is either produced or used by a project.
Examples of artifacts produced by Maven for a project include: JARs, source and binary distributions, and WARs.

The current version of the artifact produced by this project.

The full name of the project.

```
<dependencies>
        <!-- https://mvnrepository.com/artifact/junit/junit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
</dependencies>
```

This element describes all of the dependencies associated with a project. These dependencies are used to construct a classpath for your project during the build process. They are automatically downloaded from the repositories defined in this project.

http://maven.apache.org/ref/3.3.3/maven-model/maven.html

Gradle example: build.gradle and settings.gradle

build.gradle

```
apply plugin: 'java'

group = 'com.javaguru'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '1.8'

repositories {
    mavenCentral()
}

dependencies {
    // https://mvnrepository.com/artifact/junit/junit
    testCompile group: 'junit', name: 'junit', version: '4.12'
}
```

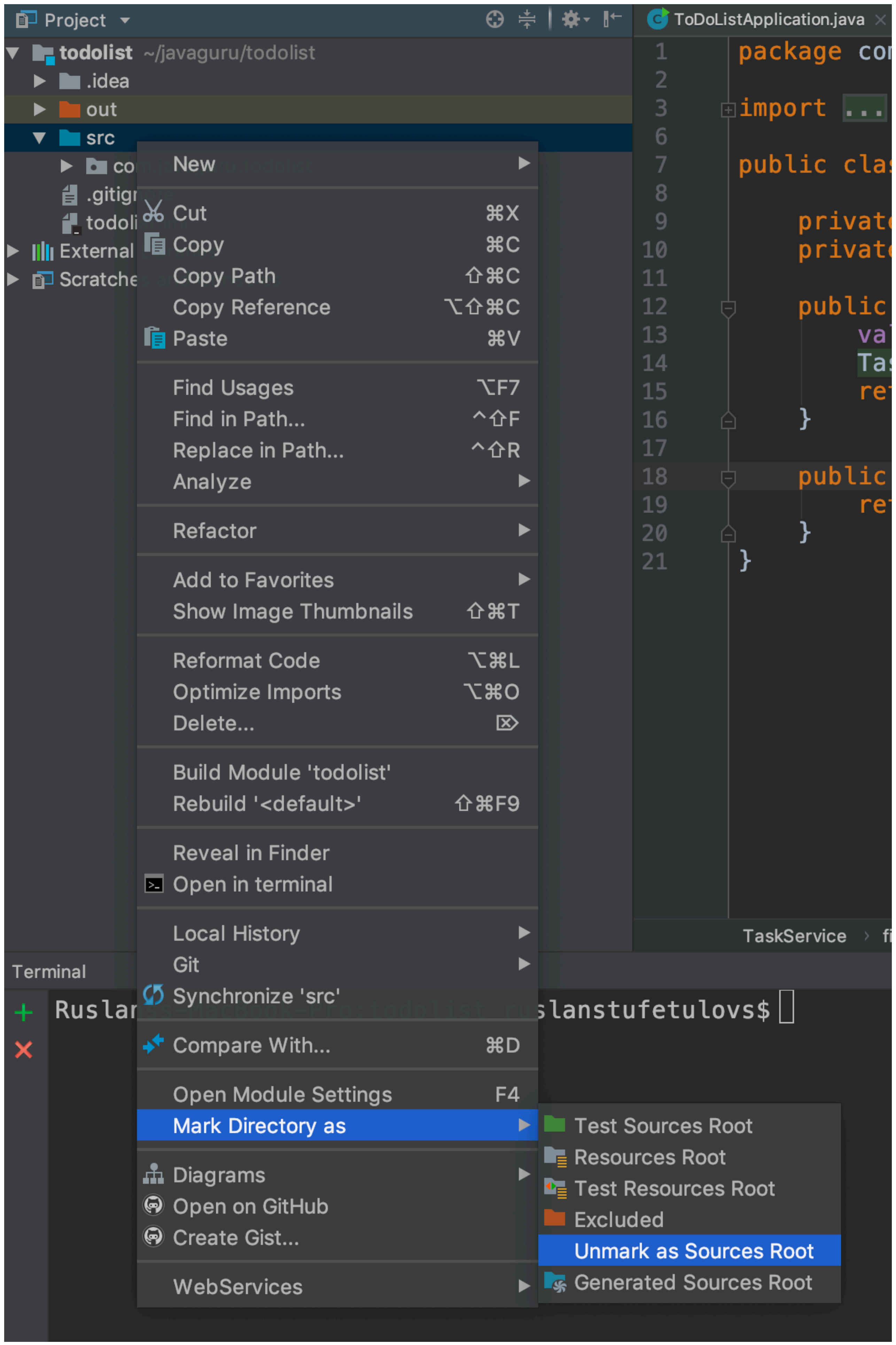
settings.gradle

```
rootProject.name = ‘todo-list'
```

Migration to Maven

Changing project structure

Step 1: Unmark you source directory (“src”)



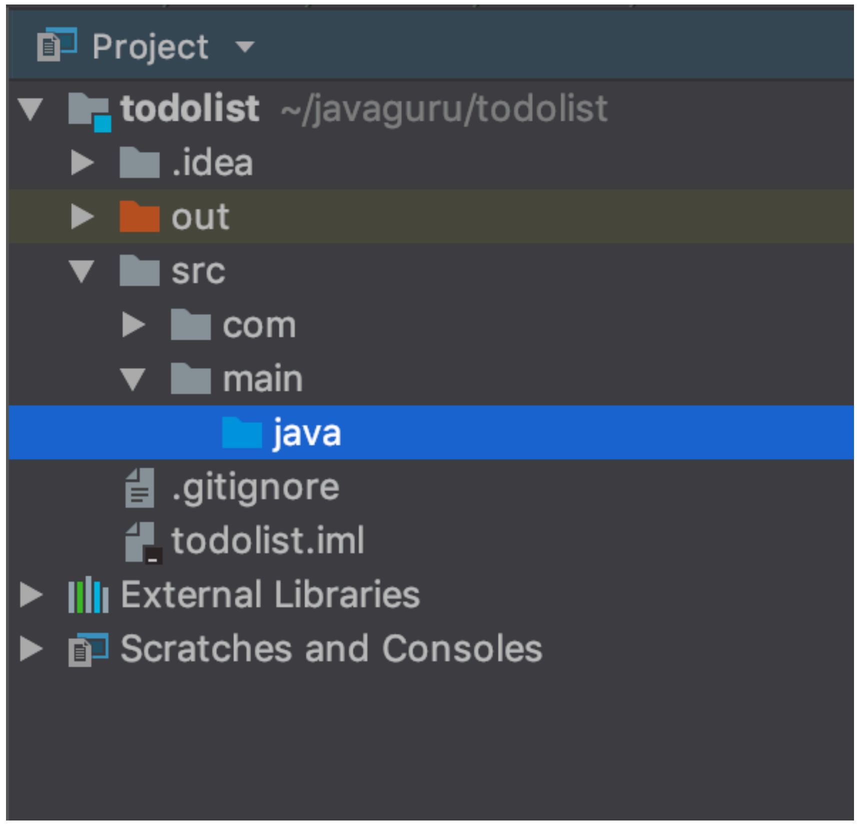
```
package co

import ...

public clas

    private
    private

    public
        va
        Ta
        re
    }

    public
        re
    }
}
```

```
Ruslar                    slanstufetulovs$
```

Step 2: Create main directory in src directory

Step 3: Create java directory in main directory

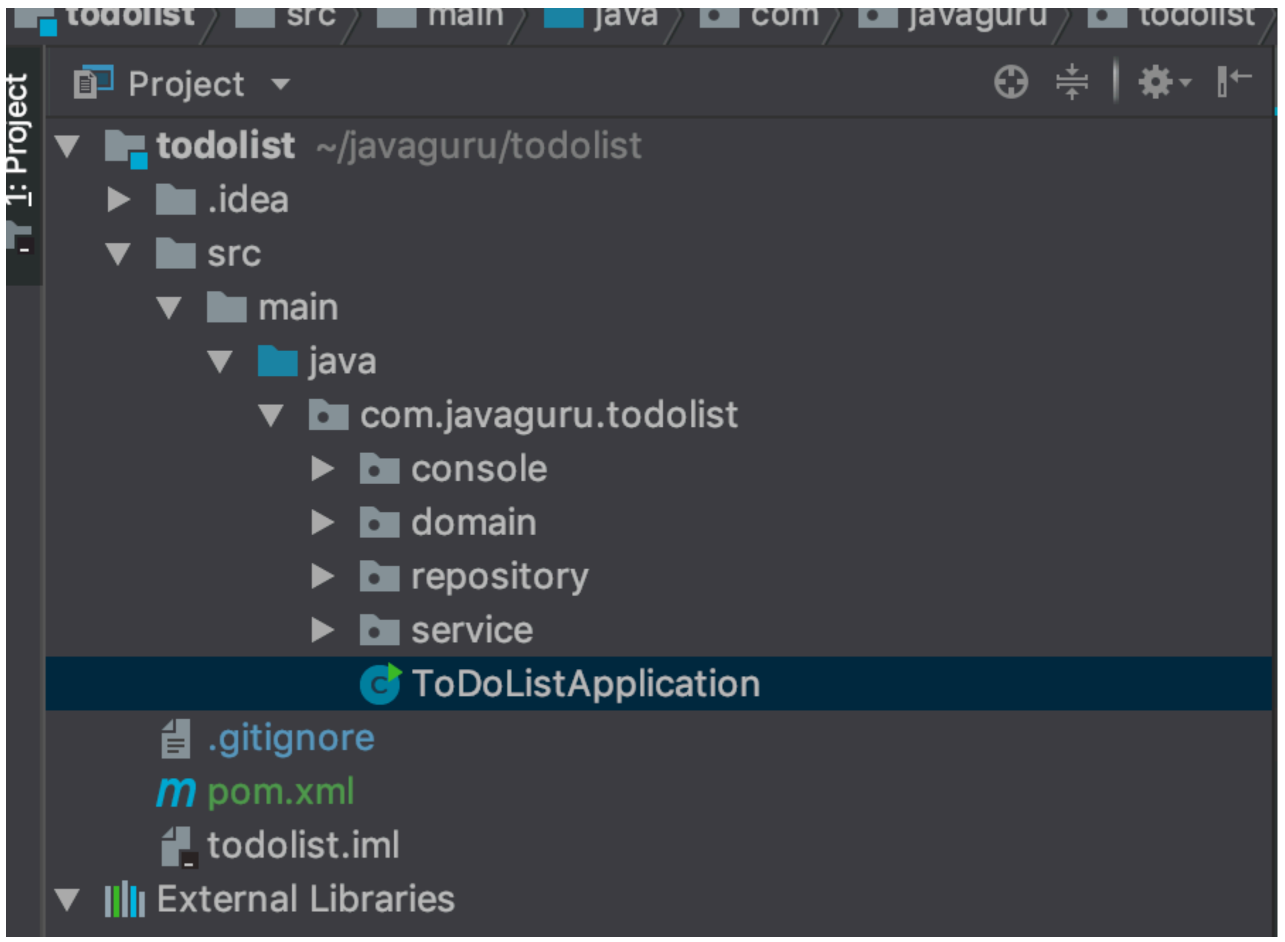
Step 4: Mark java directory as sources root



```
Project ▾
▼ todolist ~/javaguru/todolist
  ▶ .idea
  ▶ out
  ▼ src
    ▶ com
    ▼ main
        java
    .gitignore
    todolist.iml
▶ External Libraries
▶ Scratches and Consoles
```

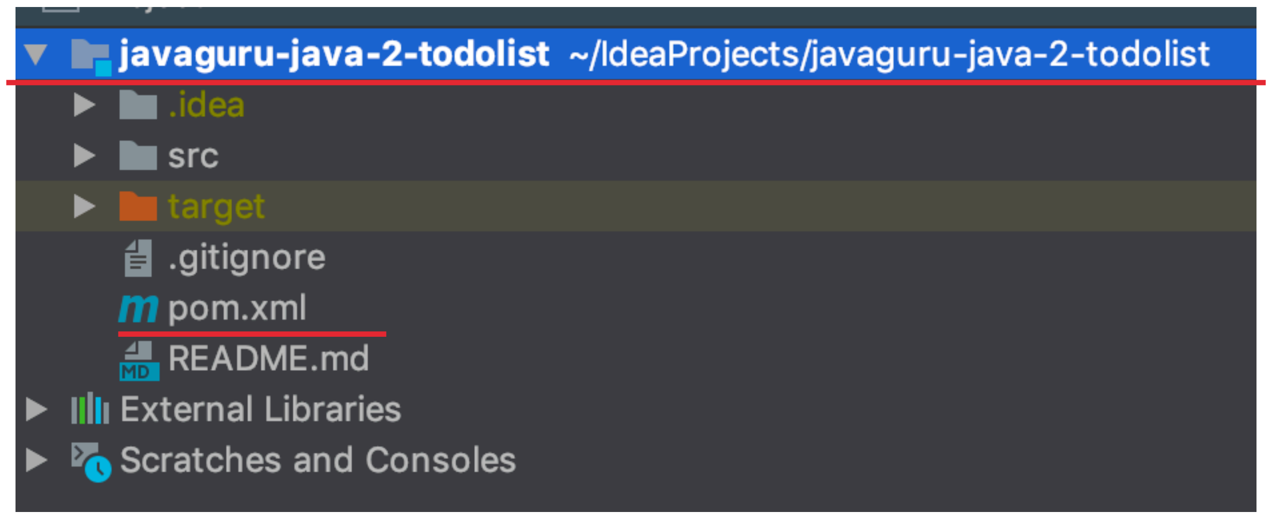
Step 5: Then move com directory

(and all that include that directory) into java

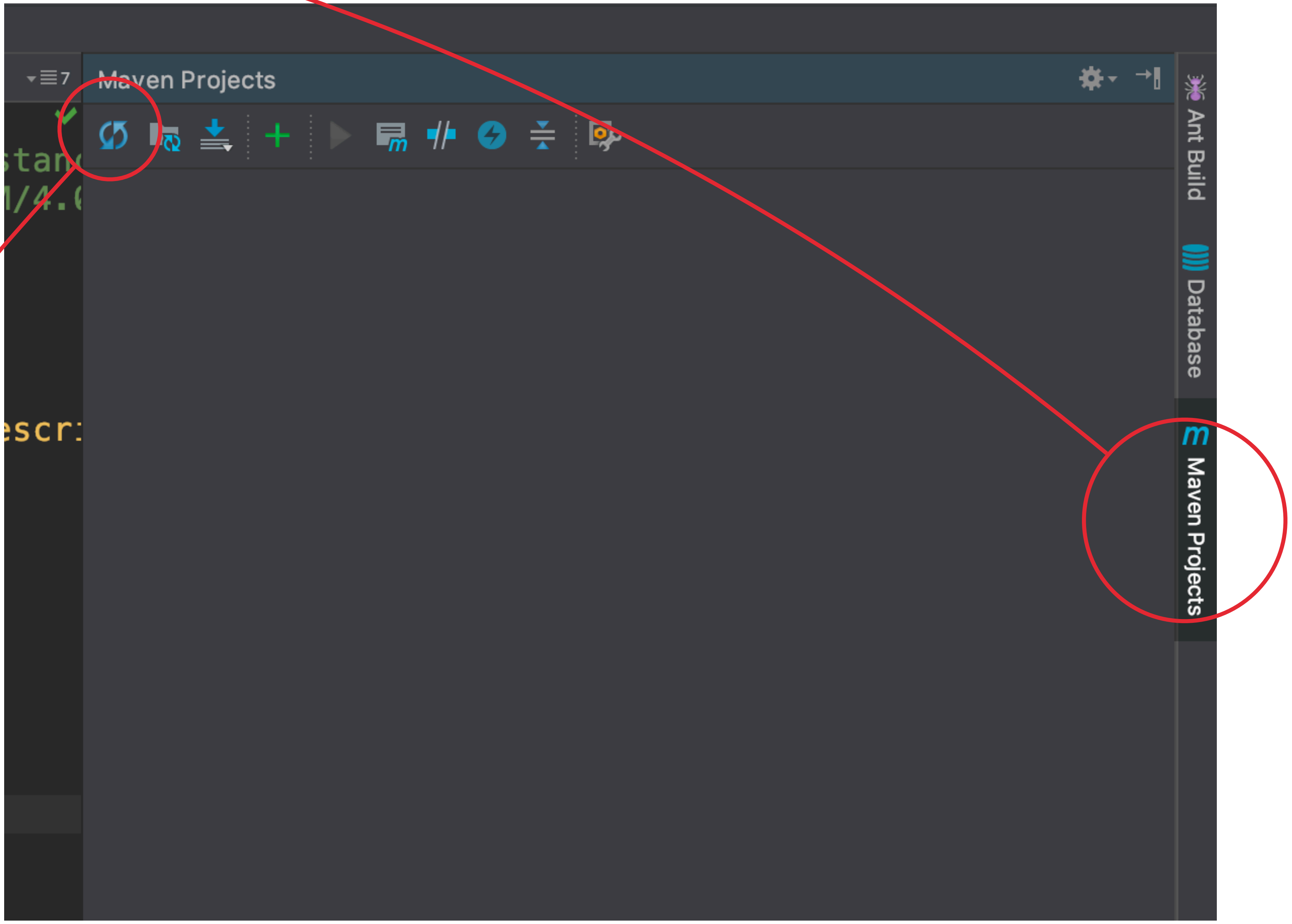


```
todolist / src / main / java / com / javaguru / todolist
1: Project
Project ▾
▼ todolist ~/javaguru/todolist
  ▶ .idea
  ▼ src
    ▼ main
      ▼ java
        ▼ com.javaguru.todolist
          ▶ console
          ▶ domain
          ▶ repository
          ▶ service
            ToDoListApplication
    .gitignore
    pom.xml
    todolist.iml
▼ External Libraries
```

Step 6. Copy pom.xml to your project (**root** directory)

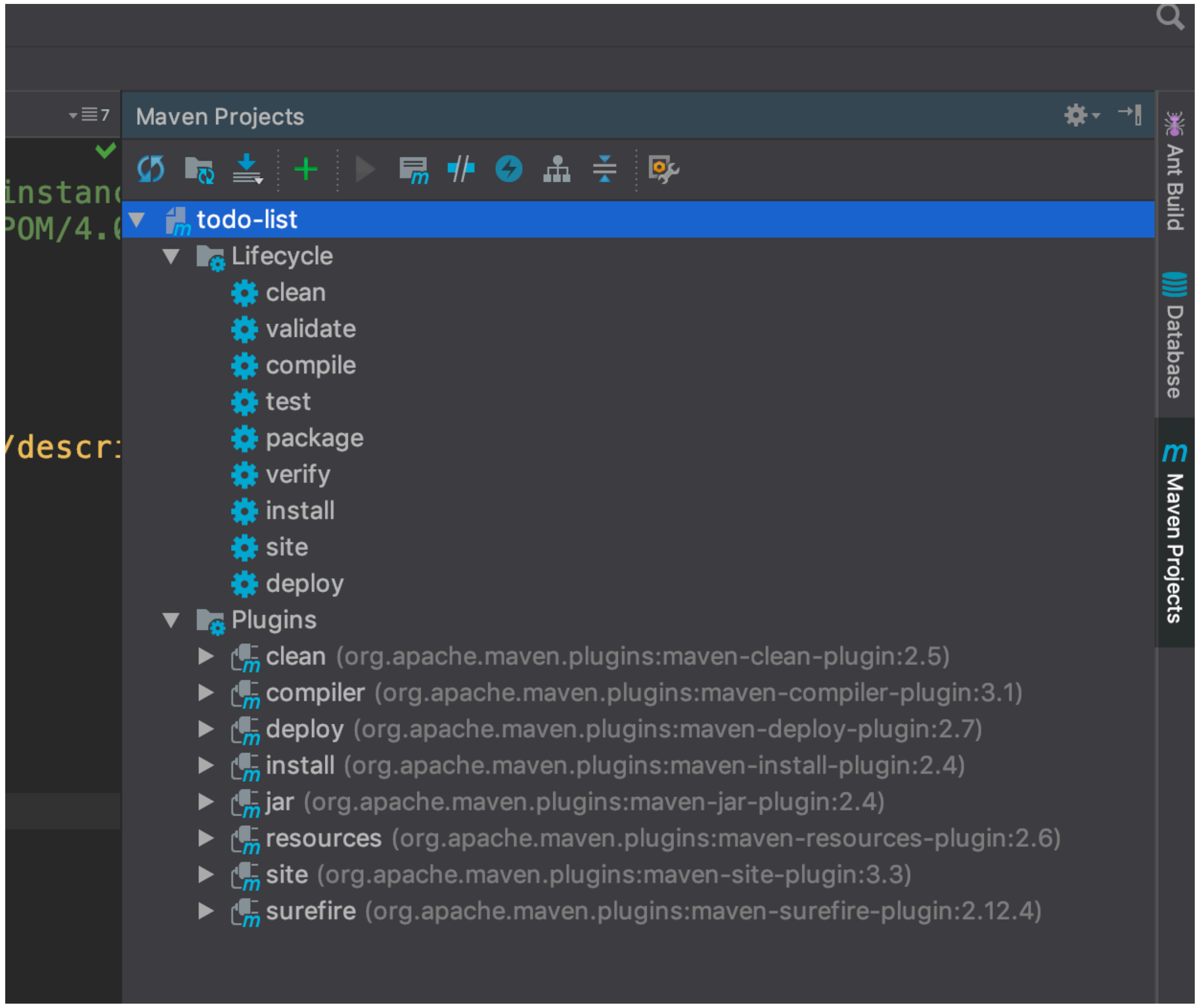


Step 7: Go to Maven Projects



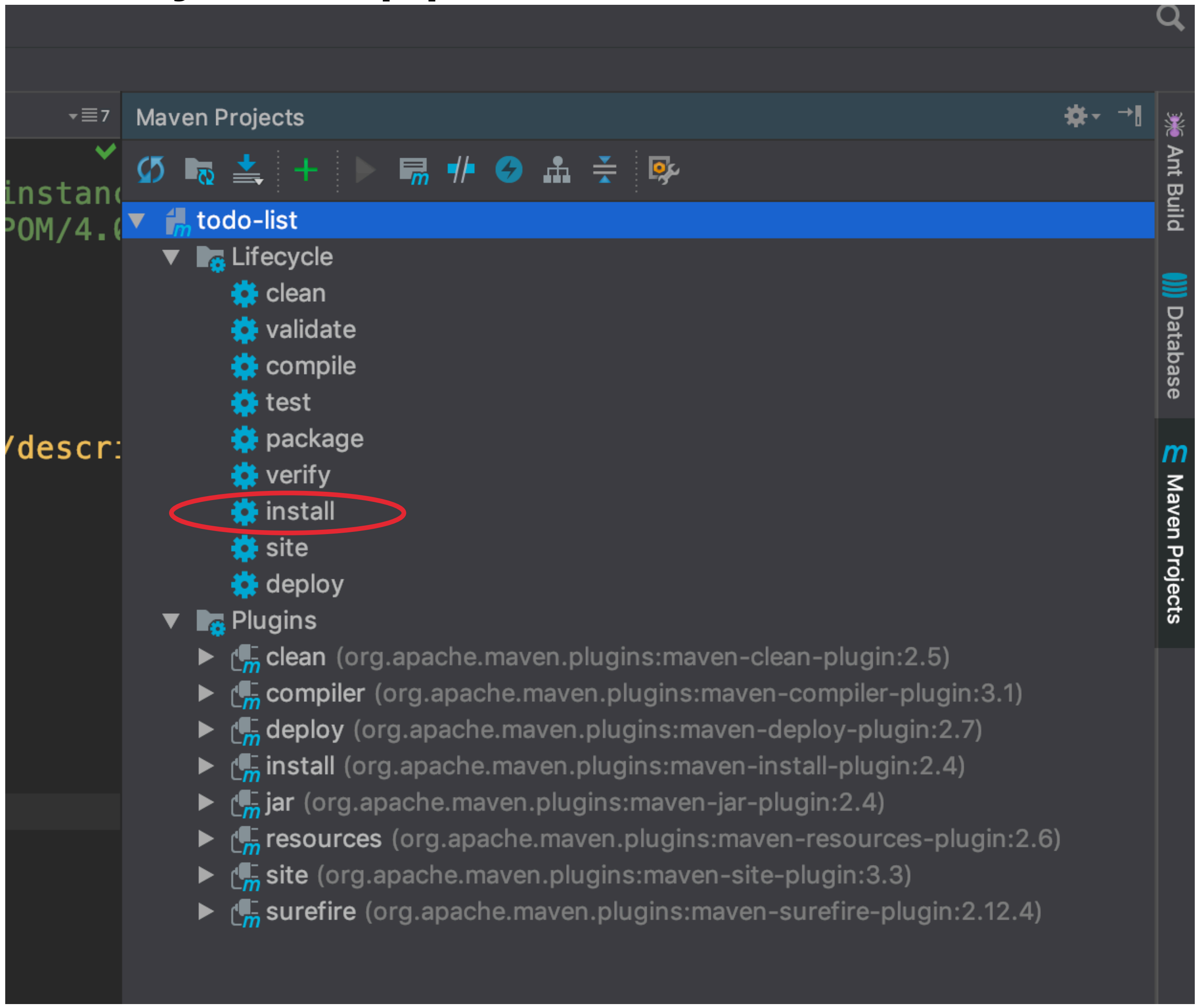
Step 8: Click on “Reimport all maven projects”

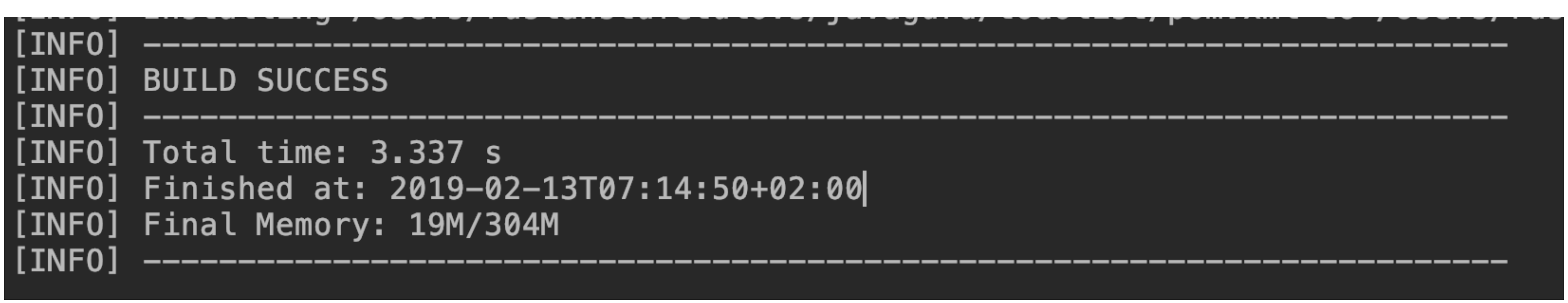
Maven project successfully imported



```
▼ todo-list
  ▼ Lifecycle
      clean
      validate
      compile
      test
      package
      verify
      install
      site
      deploy
  ▼ Plugins
    ▶ clean (org.apache.maven.plugins:maven-clean-plugin:2.5)
    ▶ compiler (org.apache.maven.plugins:maven-compiler-plugin:3.1)
    ▶ deploy (org.apache.maven.plugins:maven-deploy-plugin:2.7)
    ▶ install (org.apache.maven.plugins:maven-install-plugin:2.4)
    ▶ jar (org.apache.maven.plugins:maven-jar-plugin:2.4)
    ▶ resources (org.apache.maven.plugins:maven-resources-plugin:2.6)
    ▶ site (org.apache.maven.plugins:maven-site-plugin:3.3)
    ▶ surefire (org.apache.maven.plugins:maven-surefire-plugin:2.12.4)
```

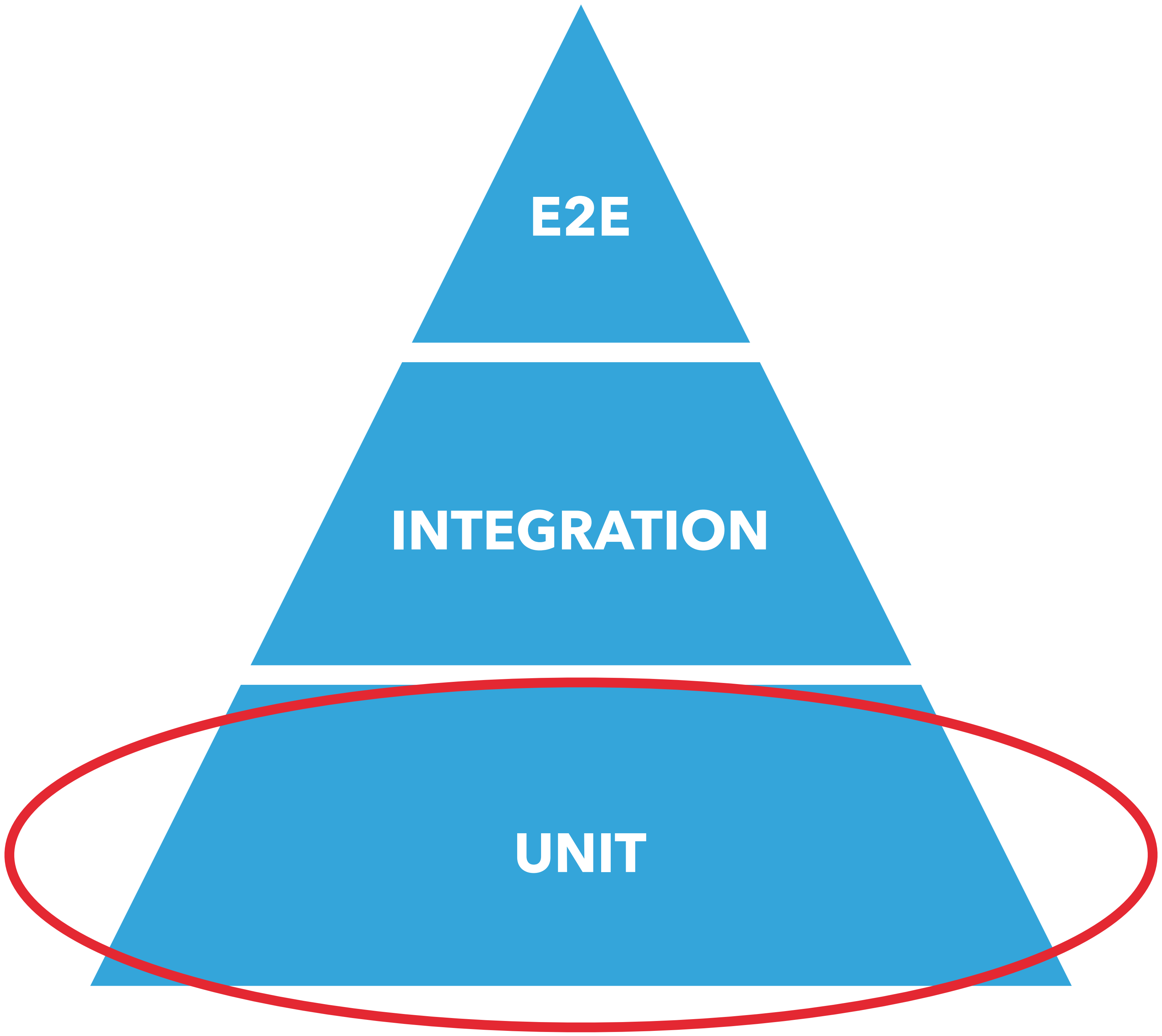
2x Click on ‘install’ and check that your application ‘build’ are success


Maven Projects
Ant Build
Database
Maven Projects
todo-list
Lifecycle
clean
validate
compile
test
package
verify
install
site
deploy
Plugins
clean (org.apache.maven.plugins:maven-clean-plugin:2.5)
compiler (org.apache.maven.plugins:maven-compiler-plugin:3.1)
deploy (org.apache.maven.plugins:maven-deploy-plugin:2.7)
install (org.apache.maven.plugins:maven-install-plugin:2.4)
jar (org.apache.maven.plugins:maven-jar-plugin:2.4)
resources (org.apache.maven.plugins:maven-resources-plugin:2.6)
site (org.apache.maven.plugins:maven-site-plugin:3.3)
surefire (org.apache.maven.plugins:maven-surefire-plugin:2.12.4)



```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3.337 s
[INFO] Finished at: 2019-02-13T07:14:50+02:00
[INFO] Final Memory: 19M/304M
[INFO] ------------------------------------------------------------------------
```

UNIT TESTING OVERVIEW


E2E
INTEGRATION
UNIT

UNIT TESTING

- Is a level of software testing where individual units/ components of a software are tested.
- The purpose is to validate that each unit of the software performs as designed. A unit is the smallest testable part of any software.
- In object-oriented programming, the smallest unit is a method, which may belong to a base/super class, abstract class or derived/child class.

UNIT TESTING BENEFITS

- Unit testing increases confidence in changing/ maintaining code.
- Codes are more reusable.
- Development is faster.
- The cost of fixing a defect detected during unit testing is lesser.
- Debugging is easy.

UNIT TESTING INCREASES CONFIDENCE IN CHANGING/MAINTAINING CODE

- If good unit tests are written and if they are run every time any code is changed, we will be able to promptly catch any defects introduced due to the change.
- Also, if codes are already made less interdependent to make unit testing possible, the unintended impact of changes to any code is less.

CODES ARE MORE REUSABLE

- In order to make unit testing possible, codes need to be modular. This means that codes are easier to reuse.

DEVELOPMENT IS FASTER

- If you do not have unit testing in place, you write your code and perform that fuzzy 'developer test' (You set some breakpoints, fire up the GUI, provide a few inputs that hopefully hit your code and hope that you are all set.)
- But, if you have unit testing in place, you write the test, write the code and run the test.
- Writing tests takes time but the time is compensated by the less amount of time it takes to run the tests

THE COST OF FIXING A DEFECT DETECTED DURING UNIT TESTING IS LESSER

- The cost of fixing a defect detected during unit testing is lesser in comparison to that of defects detected at higher levels.
- Compare the cost (time, effort, destruction, humiliation) of a defect detected during acceptance testing or when the software is live.

DEBUGGING IS EASY

- When a test fails, only the latest changes need to be debugged. With testing at higher levels, changes made over the span of several days/weeks/months need to be scanned.

UNIT TESTING TIPS

- Find a tool/framework for your language.
- Do not create test cases for everything. Instead, focus on the tests that impact the behavior of the system.
- Use test data that is close to that of production.
- In addition to writing cases to verify the behavior, write cases to ensure the performance of the code.
- Perform unit tests continuously and frequently.

WHAT IS UNIT TEST CASE

- A Unit Test Case is a part of code, which ensures that another part of code (method) works as expected. To achieve the desired results quickly, a test framework is required.
- A formal written unit test case is characterized by a known input and an expected output, which is worked out before the test is executed. The known input should test a precondition and the expected output should test a post-condition.
- There must be at least two unit test cases for each requirement – one positive test and one negative test. If a requirement has sub-requirements, each sub-requirement must have at least two test cases as positive and negative.

JUNIT OVERVIEW

JUNIT

- JUnit is a unit testing framework for Java programming language.
- JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks collectively known as xUnit, that originated with JUnit.

FEATURES OF JUNIT

- JUnit is an open source framework, which is used for writing and running tests.
- Provides annotations to identify test methods.
- Provides assertions for testing expected results.
- Provides test runners for running tests.
- JUnit tests allow you to write codes faster, which increases quality.
- JUnit is elegantly simple. It is less complex and takes less time.

FEATURES OF JUNIT 2

- JUnit tests can be run automatically and they check their own results and provide immediate feedback. There's no need to manually comb through a report of test results.
- JUnit tests can be organized into test suites containing test cases and even other test suites.
- JUnit shows test progress in a bar that is green if the test is running smoothly, and it turns red when a test fails.

EXAMPLES

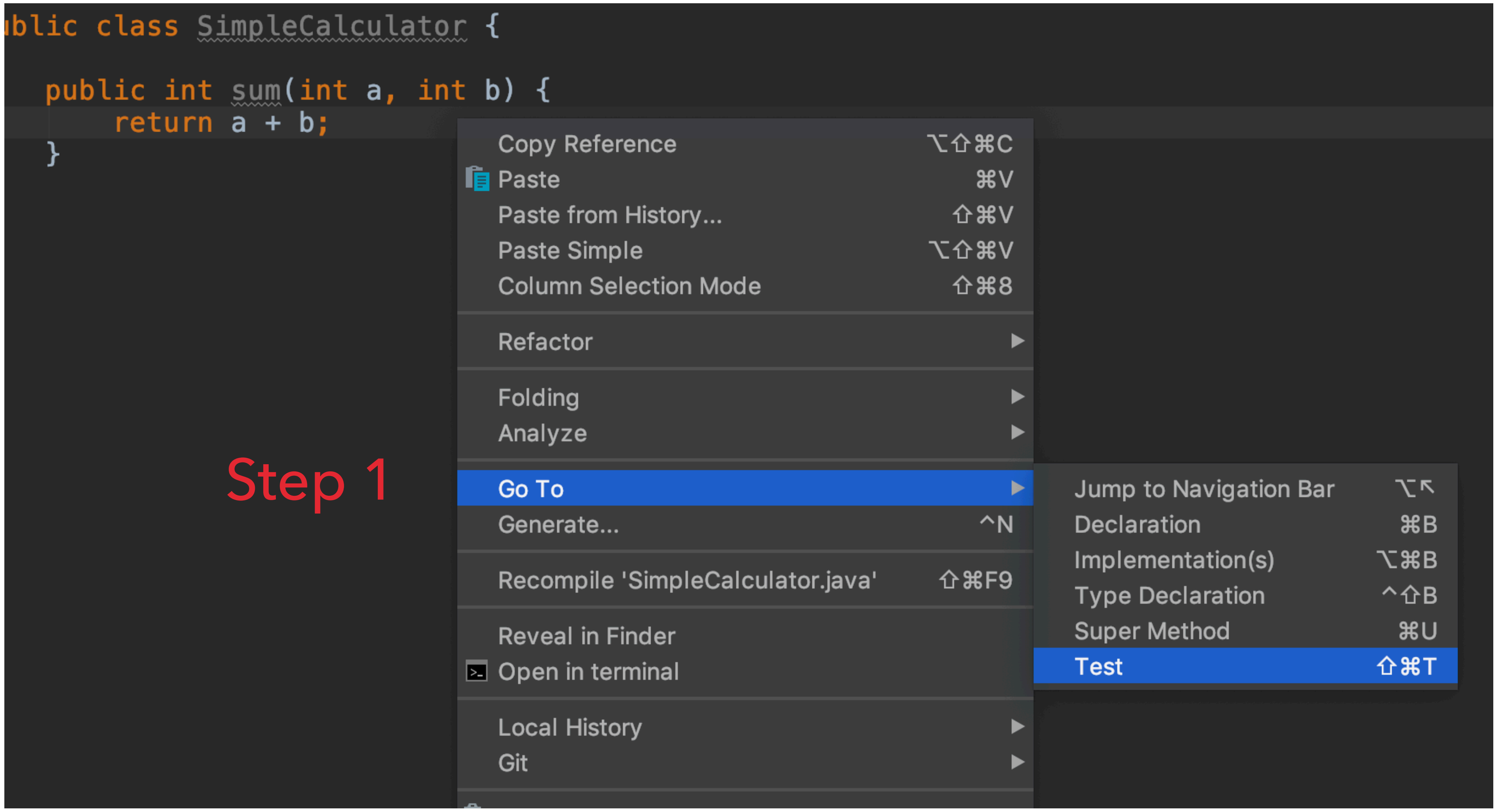
+1 dependency to pom.xml

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

NUMBER SERVICE

```java
public class SimpleCalculator {

    public int sum(int a, int b) {
        return a + b;
    }
}
```

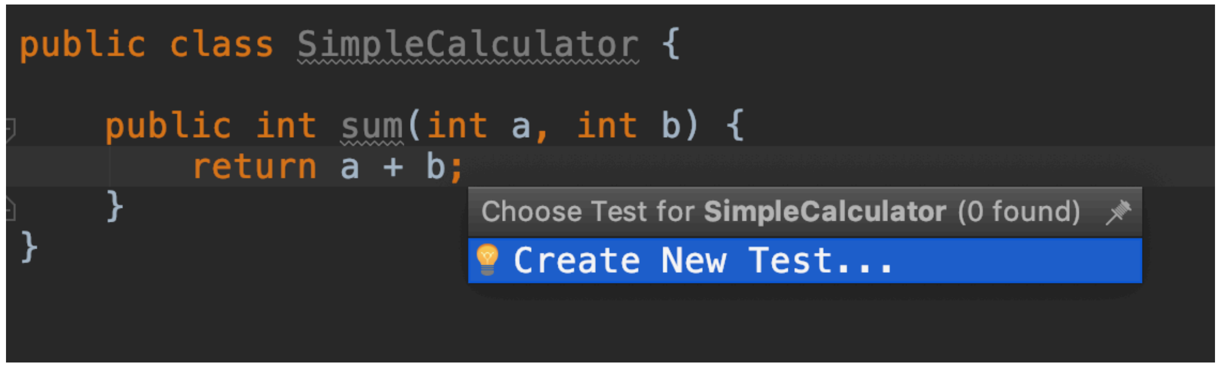
Creating first test in IntelliJ



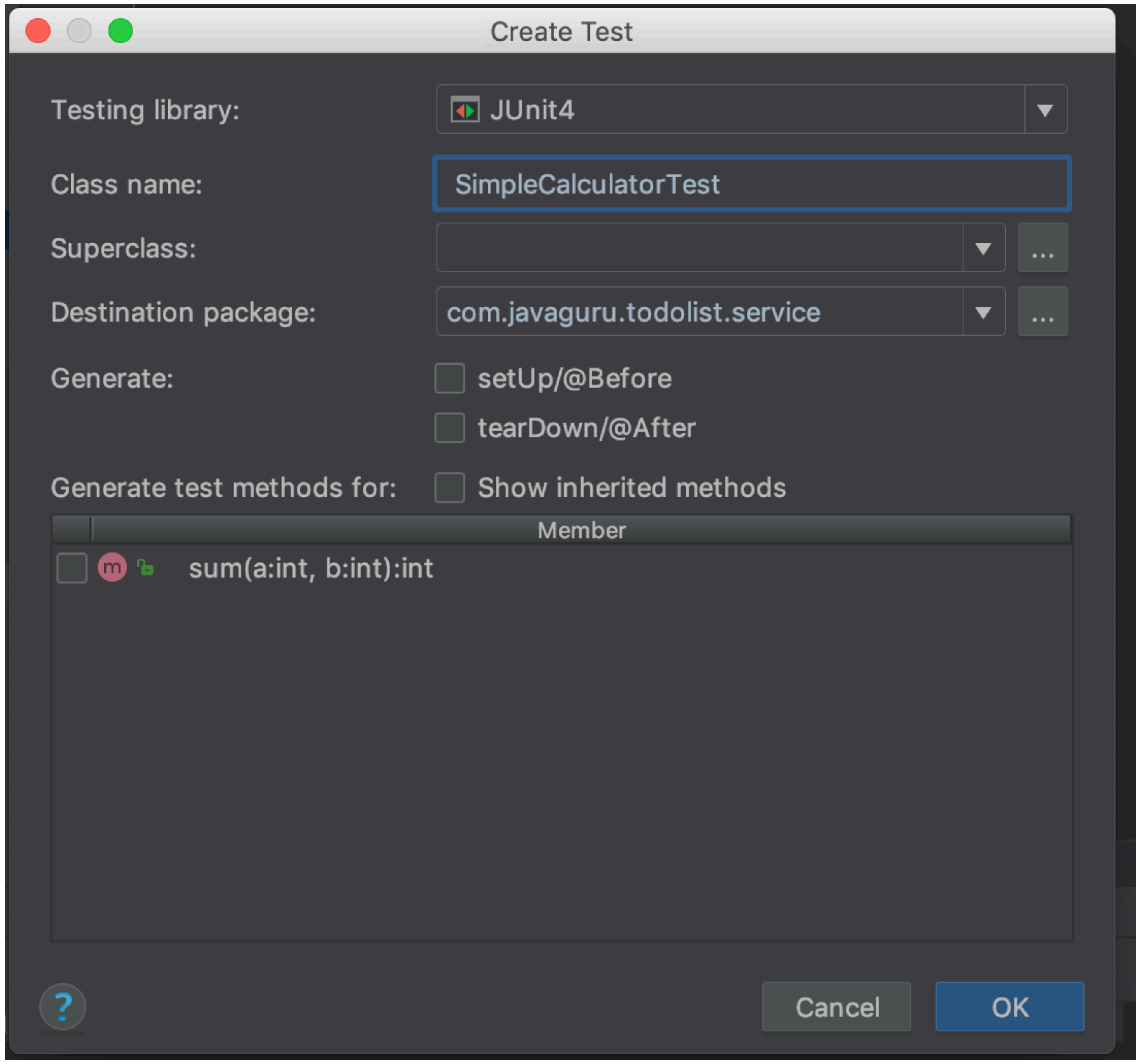
```
ublic class SimpleCalculator {

  public int sum(int a, int b) {
      return a + b;
  }
```

Step 1

Step 2



```
public class SimpleCalculator {

    public int sum(int a, int b) {
        return a + b;
    }
}
```


Create Test

Testing library: JUnit4

Class name: SimpleCalculatorTest

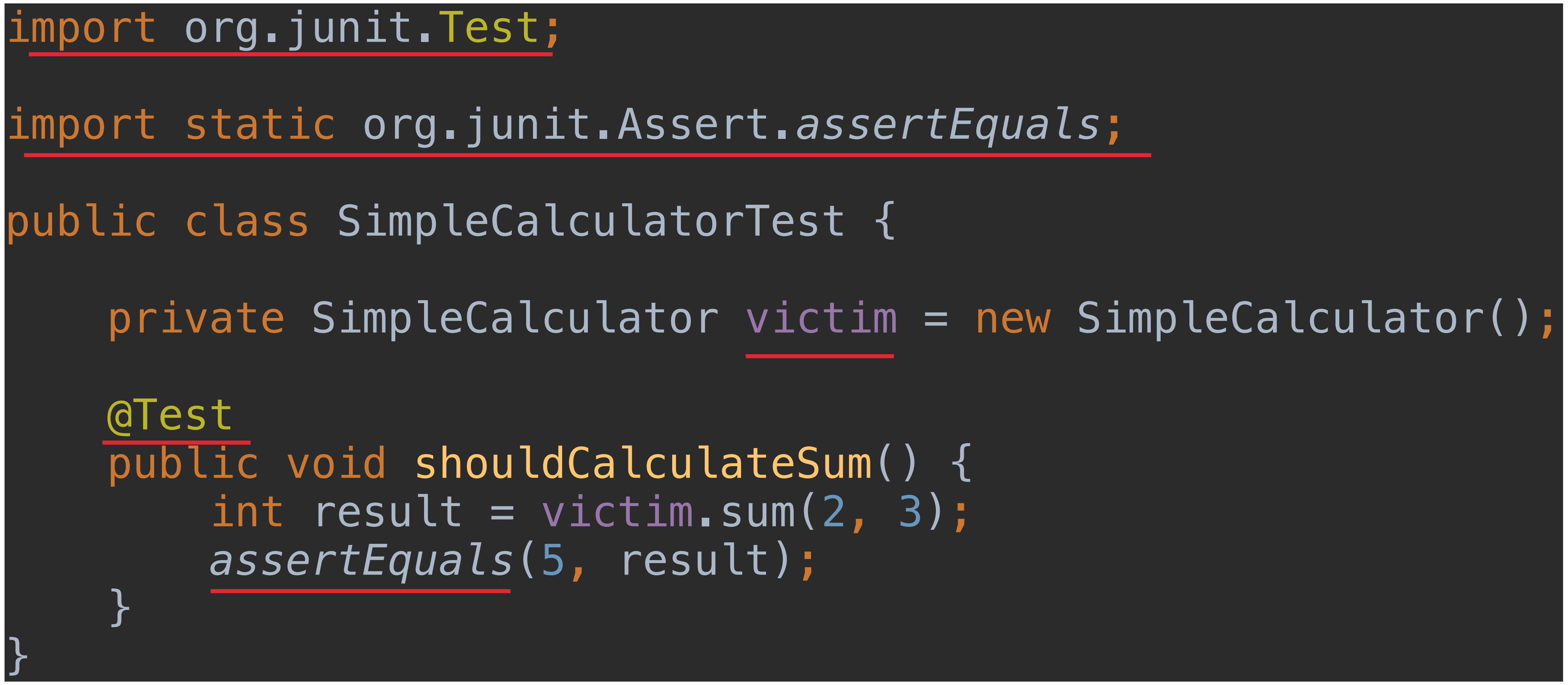
Superclass:

Destination package: com.javaguru.todolist.service

Generate: ☐ setUp/@Before
☐ tearDown/@After

Generate test methods for: ☐ Show inherited methods

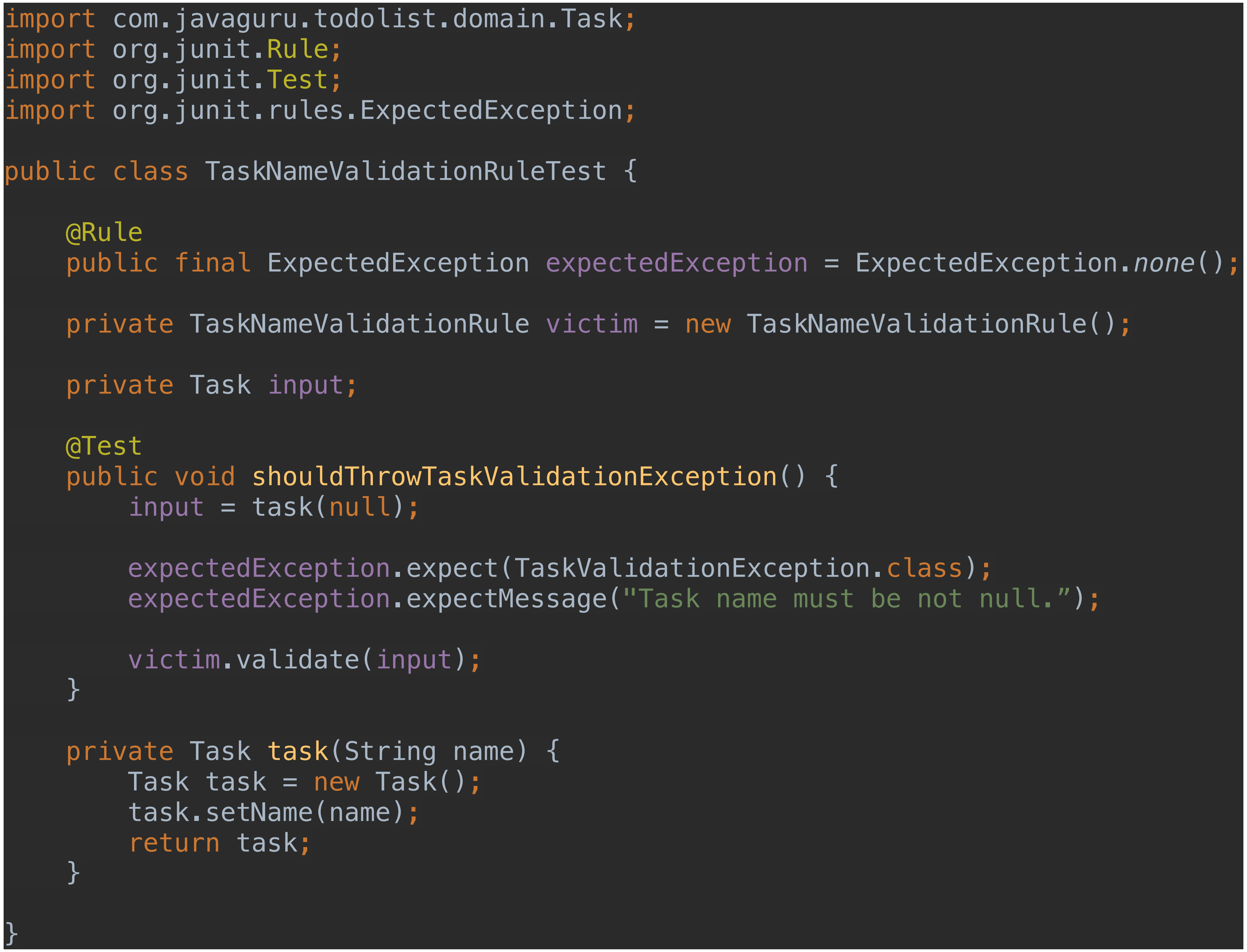
	Member
☐	sum(a:int, b:int):int



```java
import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class SimpleCalculatorTest {

    private SimpleCalculator victim = new SimpleCalculator();

    @Test
    public void shouldCalculateSum() {
        int result = victim.sum(2, 3);
        assertEquals(5, result);
    }
}
```

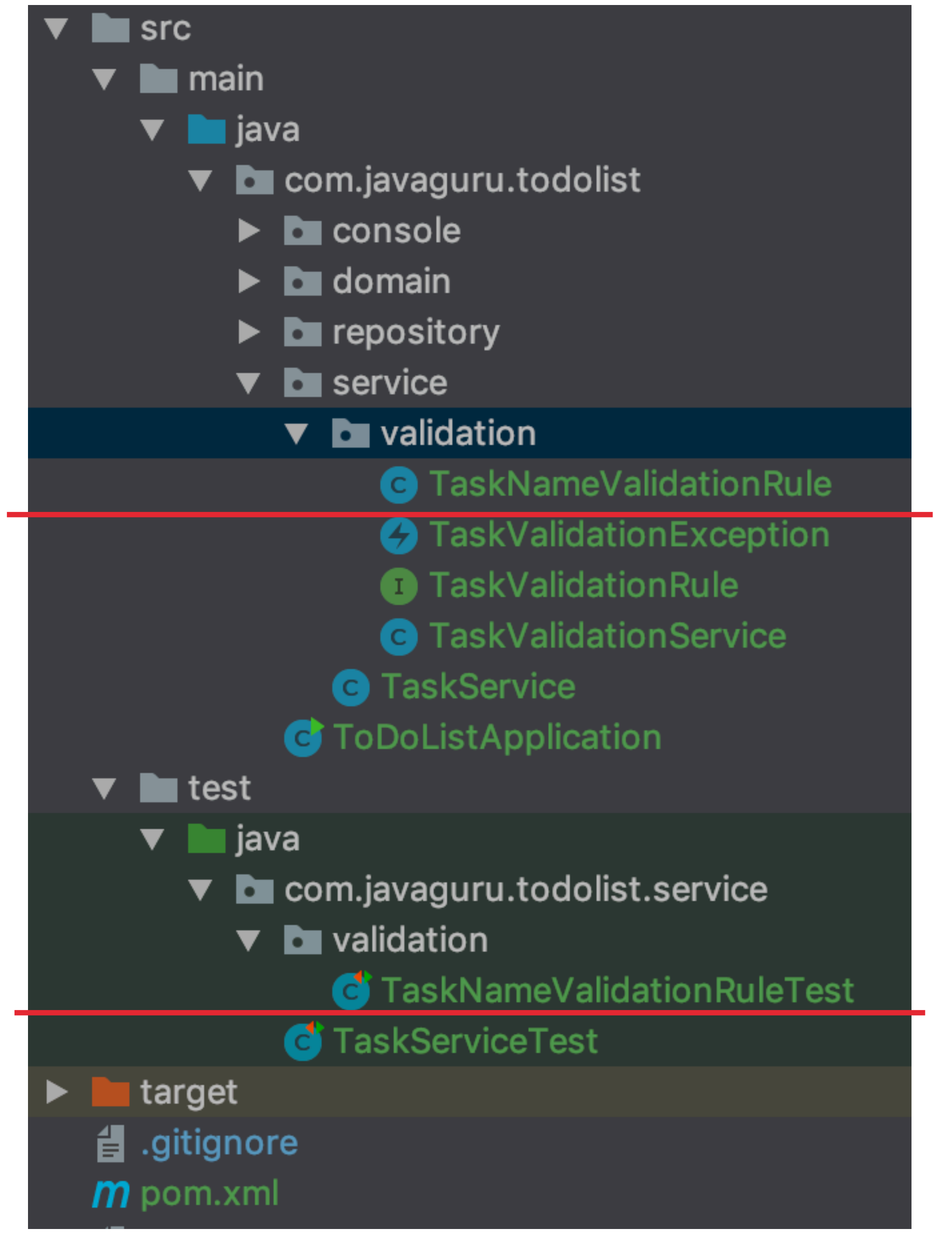
EXAMPLE 2

```java
public class TaskNameValidationRule implements TaskValidationRule {

    @Override
    public void validate(Task task) {
        checkNotNull(task);
        if (task.getName() == null) {
            throw new TaskValidationException("Task name must be not null.");
        }
    }
}
```



```java
import com.javaguru.todolist.domain.Task;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;

public class TaskNameValidationRuleTest {

    @Rule
    public final ExpectedException expectedException = ExpectedException.none();

    private TaskNameValidationRule victim = new TaskNameValidationRule();

    private Task input;

    @Test
    public void shouldThrowTaskValidationException() {
        input = task(null);

        expectedException.expect(TaskValidationException.class);
        expectedException.expectMessage("Task name must be not null.”);

        victim.validate(input);
    }

    private Task task(String name) {
        Task task = new Task();
        task.setName(name);
        return task;
    }

}
```

Check that your project structure is the same



```
▼ src
  ▼ main
    ▼ java
      ▼ com.javaguru.todolist
        ▶ console
        ▶ domain
        ▶ repository
        ▼ service
          ▼ validation
              TaskNameValidationRule
              TaskValidationException
              TaskValidationRule
              TaskValidationService
            TaskService
          ToDoListApplication
  ▼ test
    ▼ java
      ▼ com.javaguru.todolist.service
        ▼ validation
            TaskNameValidationRuleTest
          TaskServiceTest
▶ target
  .gitignore
  pom.xml
```

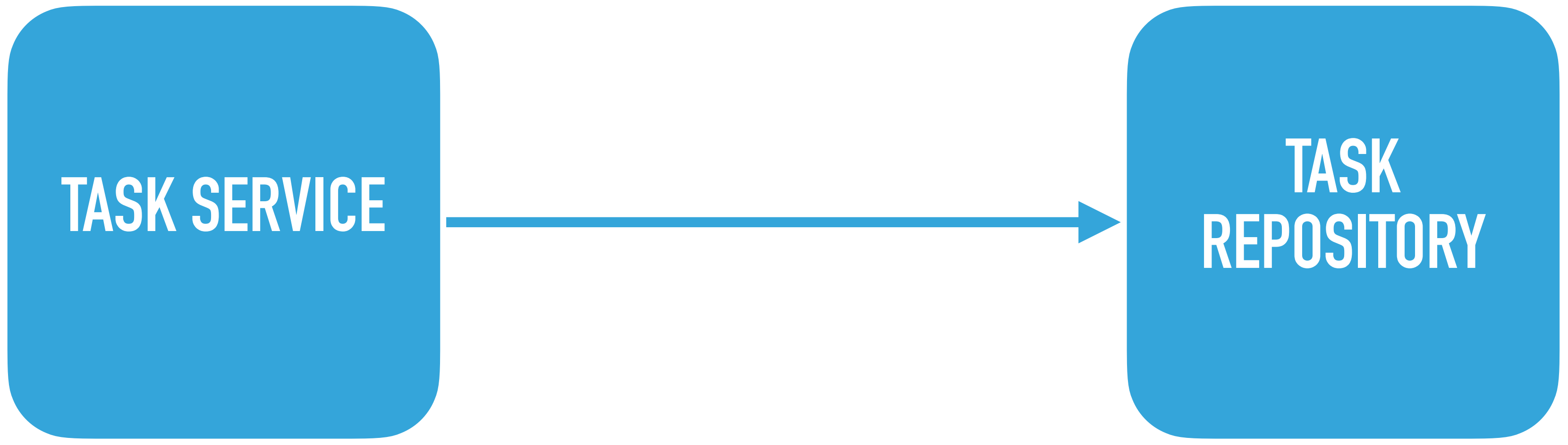
EXAMPLE 3

```java
public class TaskService {

    private TaskInMemoryRepository repository = new TaskInMemoryRepository();
    private TaskValidationService validationService = new TaskValidationService();

    public Long createTask(Task task) {
        validationService.validate(task);
        Task createdTask = repository.insert(task);
        return createdTask.getId();
    }

    public Task findTaskById(Long id) {
        return repository.findTaskById(id);
    }
}
```

DEPENDENCY

- TaskRepository is **dependency** for TaskService.
  - We can’t create TaskService without TaskRepository
- At this moment TaskRepository is **hard dependency** for TaskService.

DEPENDENCIES



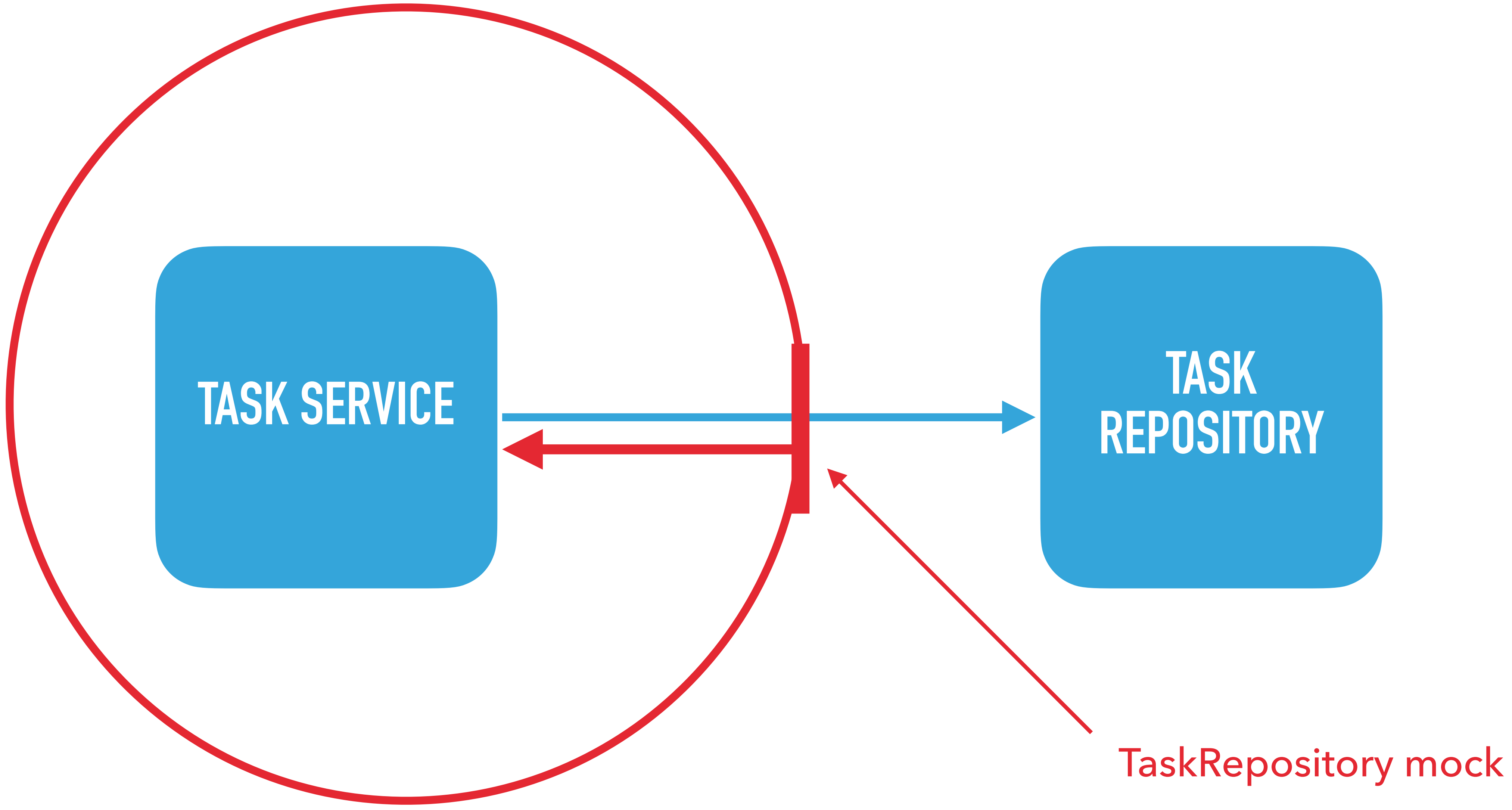
BACK TO UNIT TESTING

- Unit Testing - is a level of software testing where **individual** units/components of a software are tested.
- We need to test TaskService separately from real task repository. How we can do that?

WHAT IS MOCKING?

- Mocking is a way to test the functionality of a class in isolation. Mocking does not require a database connection or properties file read or file server read to test a functionality. Mock objects do the mocking of the real service. A mock object returns a dummy data corresponding to some dummy input passed to it.

MOCK



mockito

+1 dependency to pom.xml

```xml
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>2.23.4</version>
    <scope>test</scope>
</dependency>
```

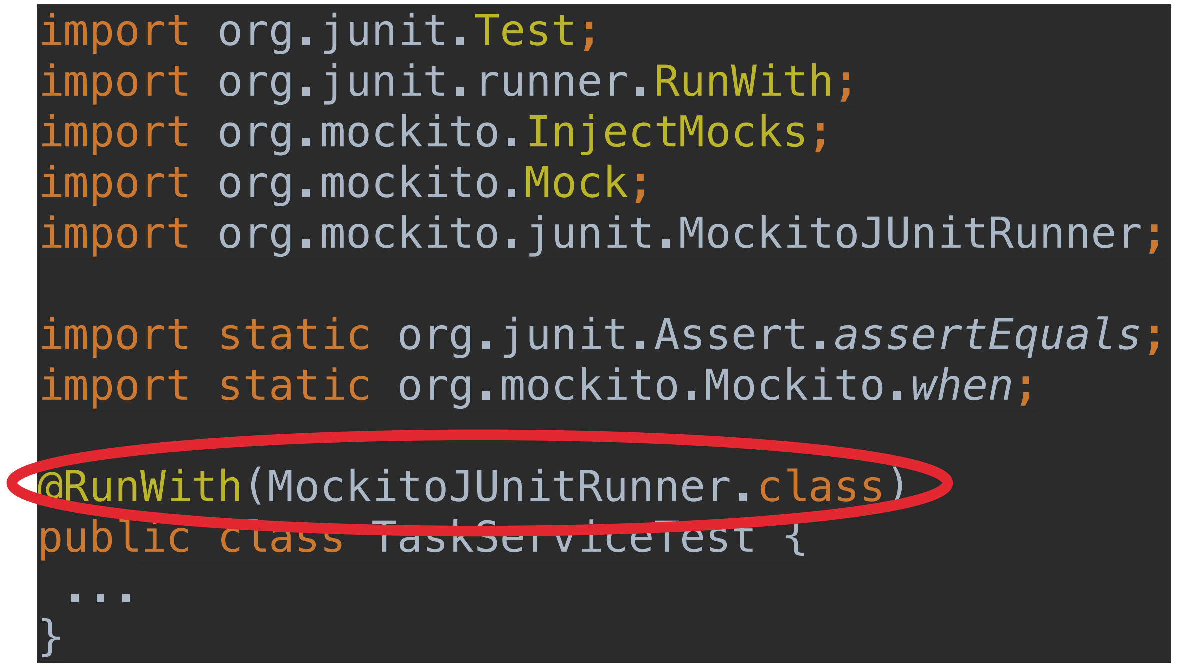
WHAT IS MOCKITO?

- Mockito facilitates creating mock objects seamlessly. It uses Java Reflection in order to create mock objects for a given interface. Mock objects are nothing but proxy for actual implementations.
- Consider a case of Task Repository which returns the details of a task. During development, the actual task repository cannot be used to get real-time data. So we need a dummy implementation of the task repository. Mockito can do the same very easily, as its name suggests.

BENEFITS

- **No Handwriting** – No need to write mock objects on your own.
- **Refactoring Safe** – Renaming interface method names or reordering parameters will not break the test code as Mocks are created at runtime.
- **Return value support** – Supports return values.
- **Exception support** – Supports exceptions.
- **Order check support** – Supports check on order of method calls.
- **Annotation support** – Supports creating mocks using annotation.

In order for these annotations to be enabled, we'll need to **annotate the JUnit test with a runner** – *MockitoJUnitRunner* as in the following example:



```java
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.MockitoJUnitRunner;

import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.when;

@RunWith(MockitoJUnitRunner.class)
public class TaskServiceTest {
 ...
}
```

MOCKITO JUNIT RUNNER

- **MockitoJUnitRunner** gives you automatic validation of framework usage, as well as an automatic **initMocks()**
- The automatic validation of framework usage is actually worth having. It gives you better reporting if you make one of these mistakes:
    - You call the static **when** method, but don't complete the stubbing with a matching **thenReturn**, **thenThrow** or then.
    - You call verify on a mock, but forget to provide the method call that you are trying to **verify**.
    - You call the when method after **doReturn**, **doThrow** or **doAnswer** and pass a mock, but forget to provide the method that you are trying to stub

@Mock annotation

The most used widely used annotation in Mockito is *@Mock*. We can use *@Mock* to create and inject mocked instances

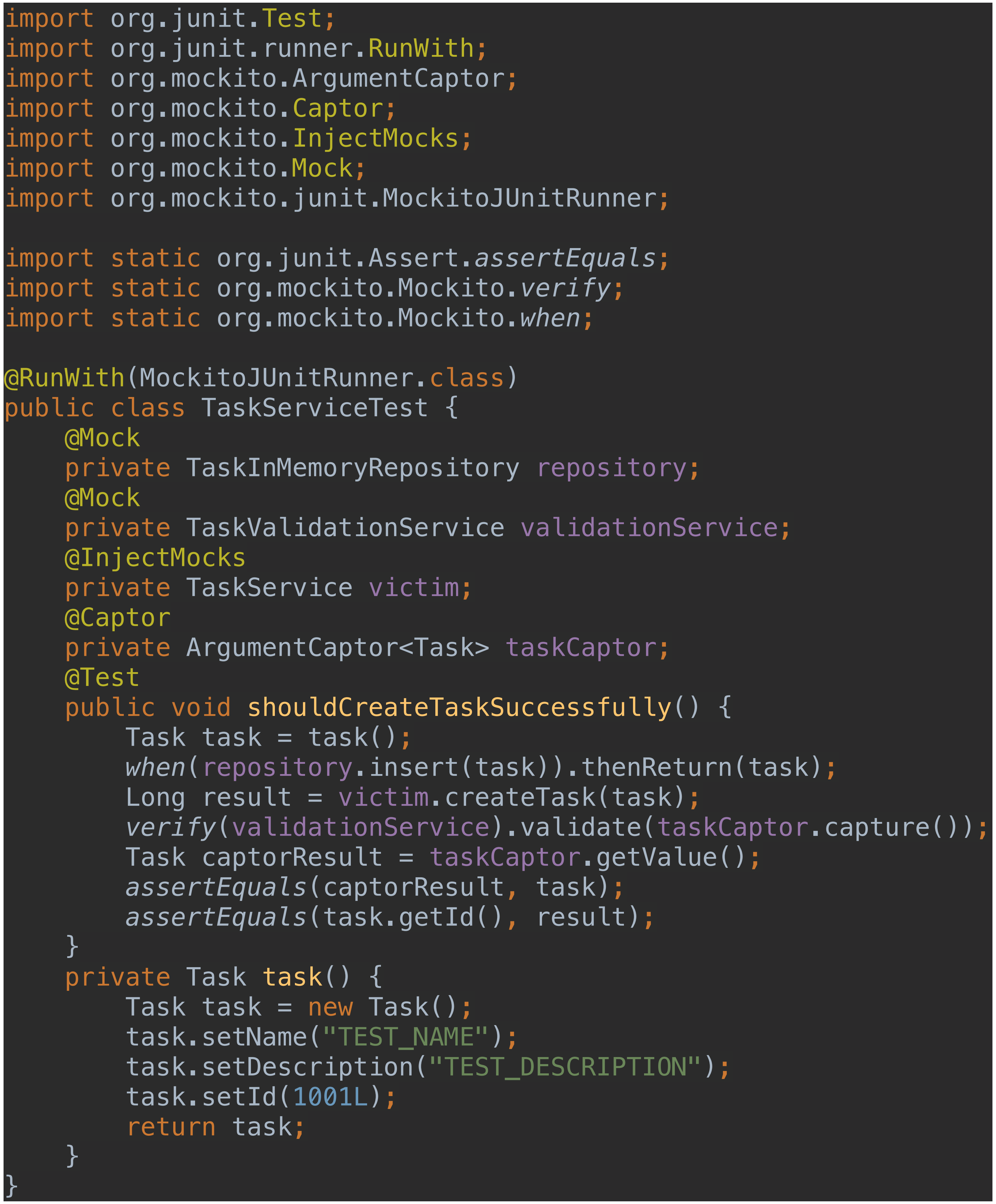
```
@Mock
private TaskInMemoryRepository repository;
```

@InjectMocks

Use *@InjectMocks* annotation – to inject mock fields into the tested object automatically.

```
@Mock
private TaskInMemoryRepository repository;

@InjectMocks
private TaskService victim;
```

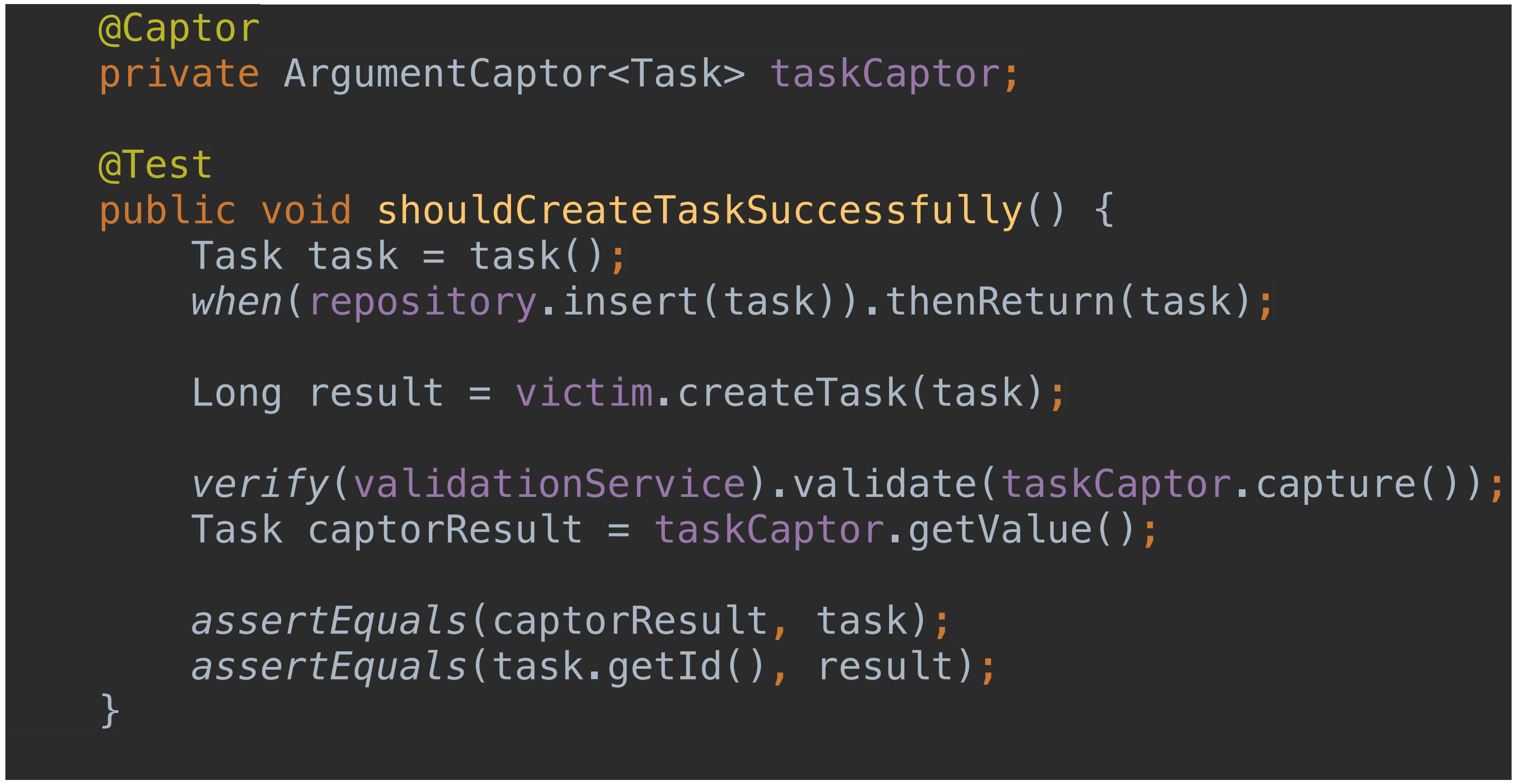
```java
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.MockitoJUnitRunner;

import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.when;

@RunWith(MockitoJUnitRunner.class)
public class TaskServiceTest {

    @Mock
    private TaskInMemoryRepository repository;

    @InjectMocks
    private TaskService victim;

    @Test
    public void shouldFindTask() {
        when(repository.findTaskById(1001L)).thenReturn(task());

        Task result = victim.findTaskById(1001L);

        assertEquals(task(), result);
    }

    private Task task() {
        Task task = new Task();
        task.setName("TEST_NAME");
        task.setDescription("TEST_DESCRIPTION");
        task.setId(1001L);
        return task;
    }
}
```



```java
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.ArgumentCaptor;
import org.mockito.Captor;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.MockitoJUnitRunner;

import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;

@RunWith(MockitoJUnitRunner.class)
public class TaskServiceTest {
    @Mock
    private TaskInMemoryRepository repository;
    @Mock
    private TaskValidationService validationService;
    @InjectMocks
    private TaskService victim;
    @Captor
    private ArgumentCaptor<Task> taskCaptor;
    @Test
    public void shouldCreateTaskSuccessfully() {
        Task task = task();
        when(repository.insert(task)).thenReturn(task);
        Long result = victim.createTask(task);
        verify(validationService).validate(taskCaptor.capture());
        Task captorResult = taskCaptor.getValue();
        assertEquals(captorResult, task);
        assertEquals(task.getId(), result);
    }
    private Task task() {
        Task task = new Task();
        task.setName("TEST_NAME");
        task.setDescription("TEST_DESCRIPTION");
        task.setId(1001L);
        return task;
    }
}
```

Mockito.verify(MockedObject).someMethodOnTheObject(someParametersToTheMethod);

verifies that the methods you called on your mocked object are indeed called. If they weren't called, or called with the wrong parameters, or called the wrong number of times, they would fail your test.



```java
@Captor
private ArgumentCaptor<Task> taskCaptor;

@Test
public void shouldCreateTaskSuccessfully() {
    Task task = task();
    when(repository.insert(task)).thenReturn(task);

    Long result = victim.createTask(task);

    verify(validationService).validate(taskCaptor.capture());
    Task captorResult = taskCaptor.getValue();

    assertEquals(captorResult, task);
    assertEquals(task.getId(), result);
}
```

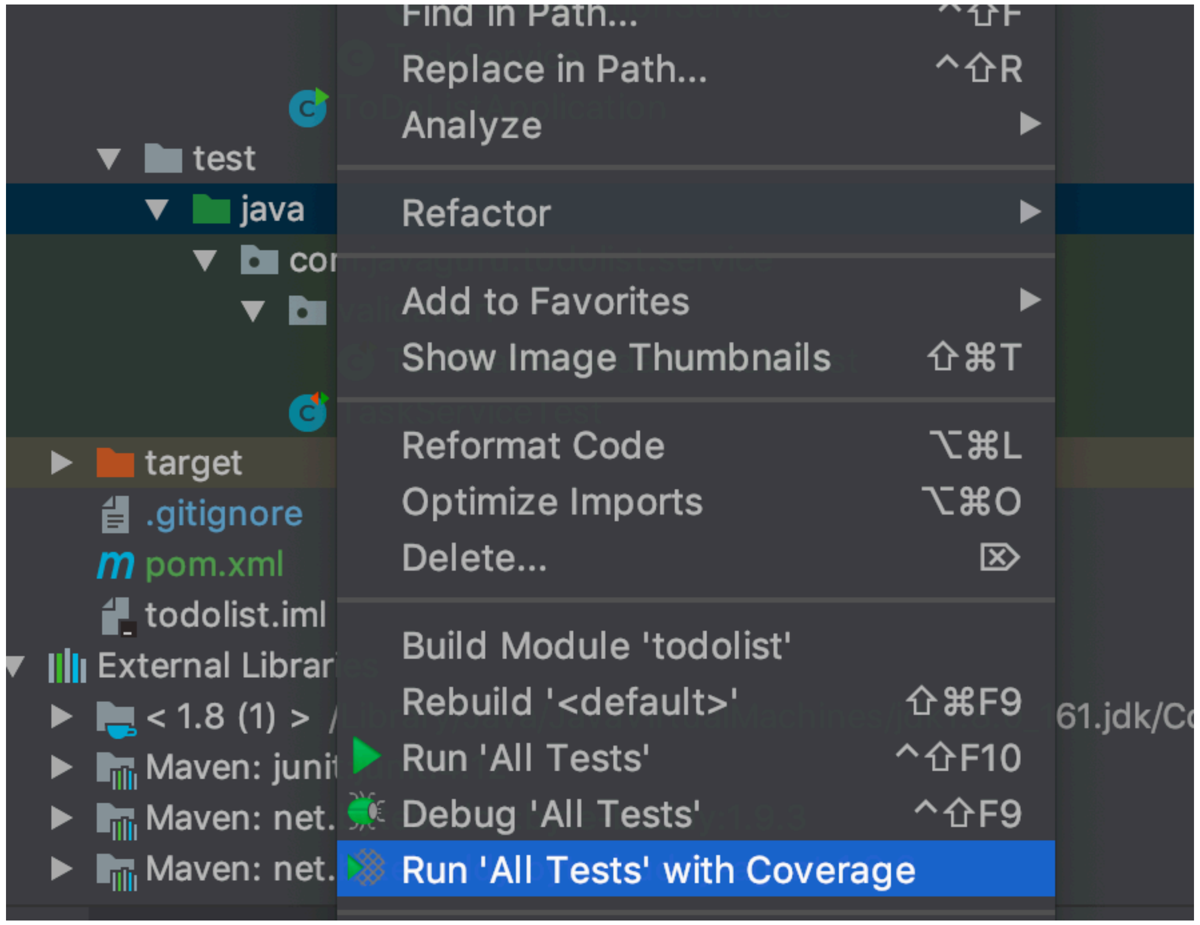
Use ArgumentCaptor to capture argument values for further assertions.

Mockito verifies argument values in natural java style: by using an equals() method. This is also the recommended way of matching arguments because it makes tests clean & simple. In some situations though, it is helpful to assert on certain arguments after the actual verification

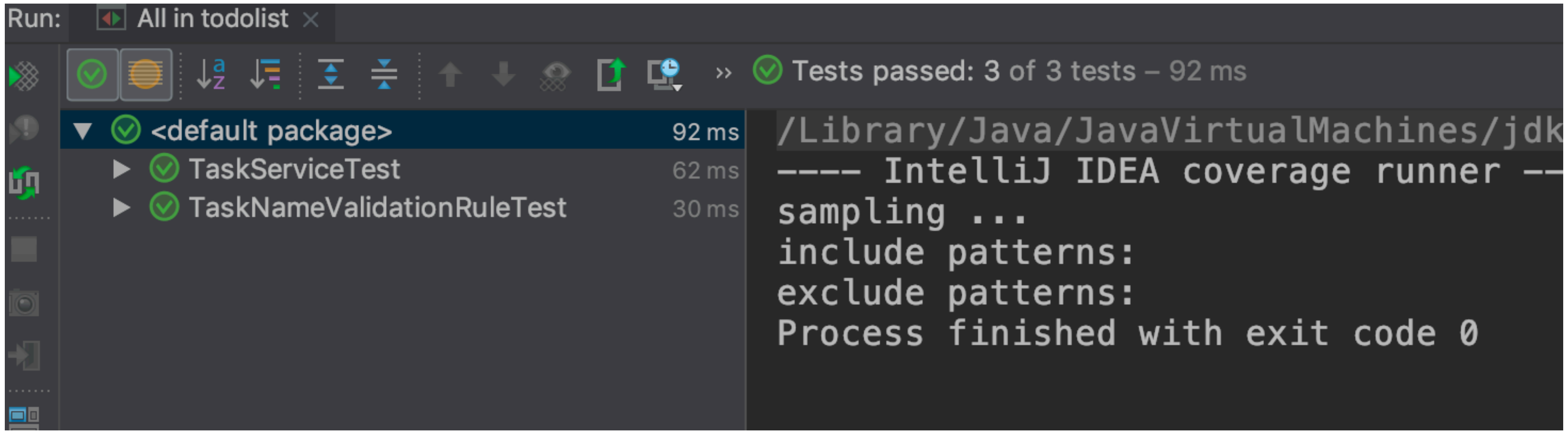
CODE COVERAGE

Step 1. Right click on test directory (java)

Step 2. Choose "Run All Tests with Coverage"

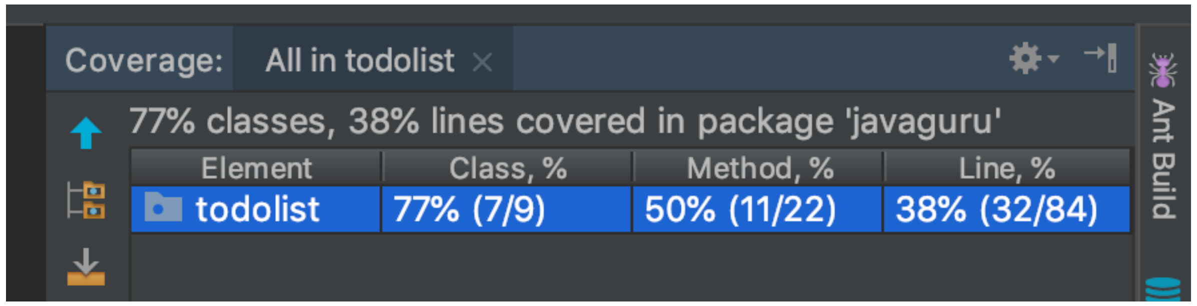


Check that your tests are successfully passed



```
Tests passed: 3 of 3 tests – 92 ms

<default package>               92 ms
  TaskServiceTest               62 ms
  TaskNameValidationRuleTest    30 ms
```

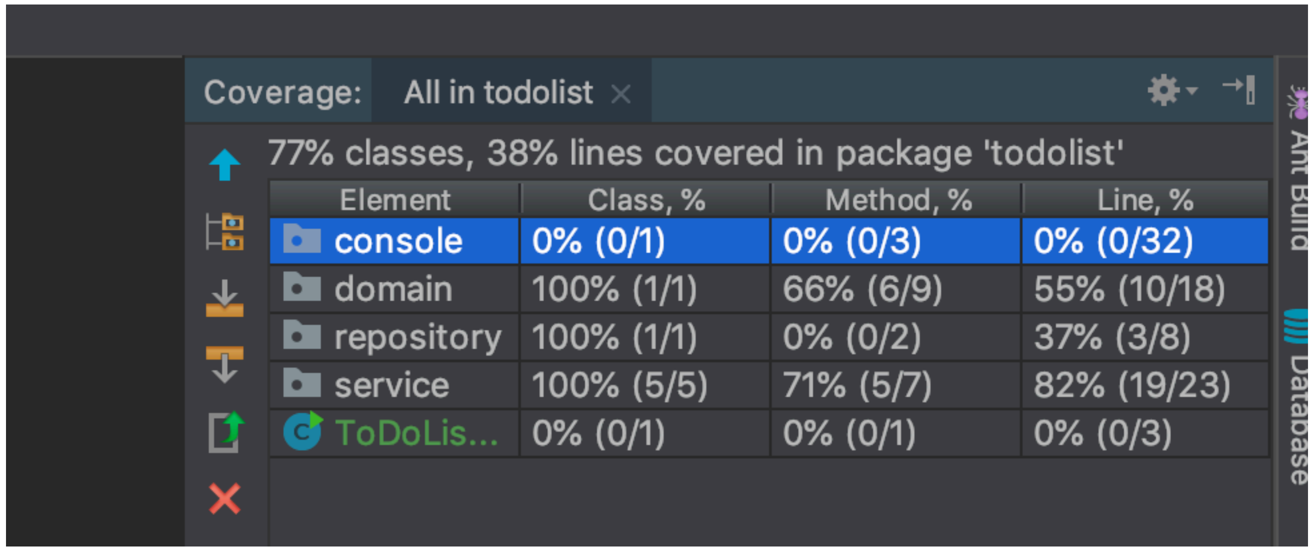
```
/Library/Java/JavaVirtualMachines/jdk
---- IntelliJ IDEA coverage runner --
sampling ...
include patterns:
exclude patterns:
Process finished with exit code 0
```



77% classes, 38% lines covered in package 'javaguru'

Element	Class, %	Method, %	Line, %
todolist	77% (7/9)	50% (11/22)	38% (32/84)

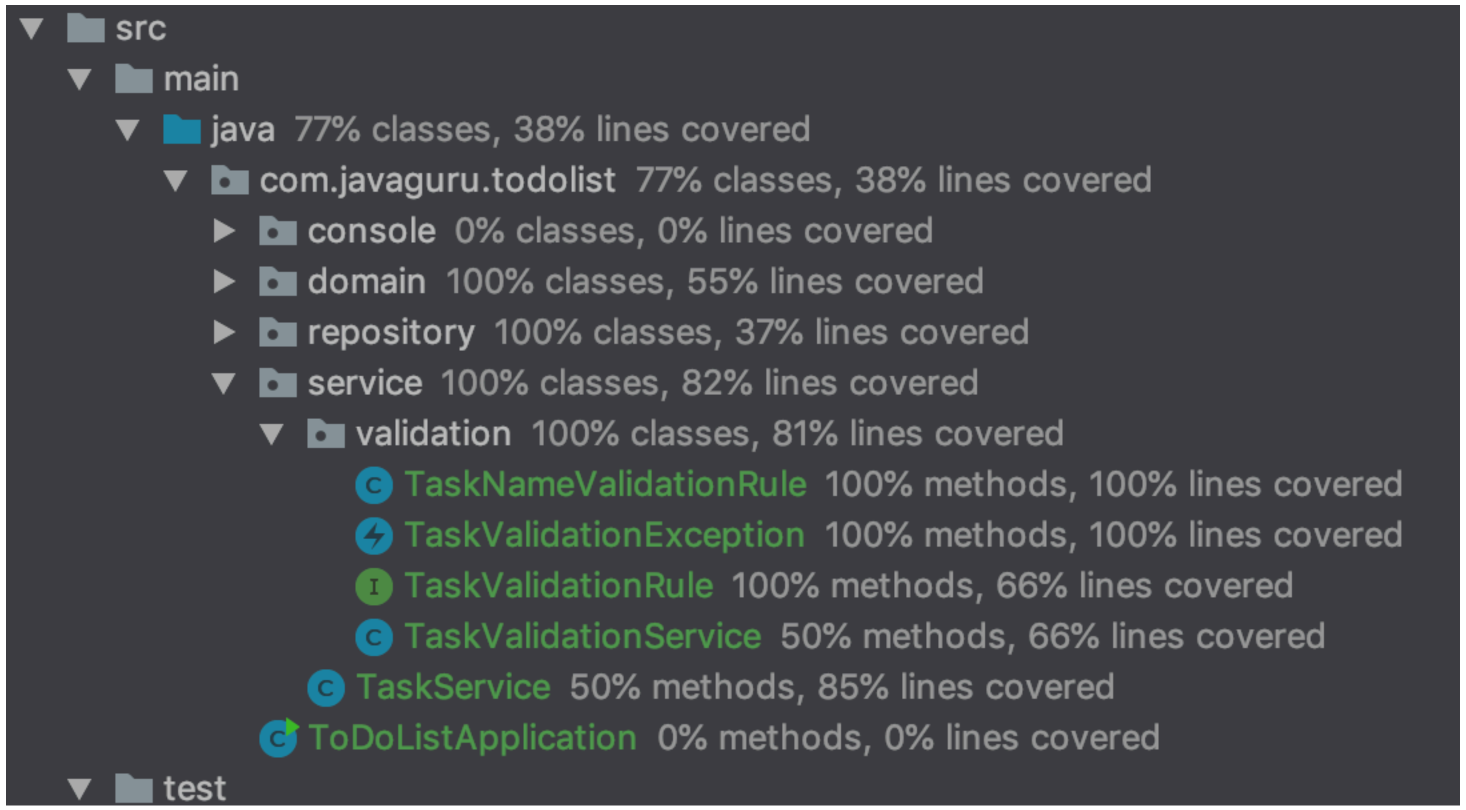
Check your code coverage



77% classes, 38% lines covered in package 'todolist'

Element	Class, %	Method, %	Line, %
console	0% (0/1)	0% (0/3)	0% (0/32)
domain	100% (1/1)	66% (6/9)	55% (10/18)
repository	100% (1/1)	0% (0/2)	37% (3/8)
service	100% (5/5)	71% (5/7)	82% (19/23)
ToDoLis...	0% (0/1)	0% (0/1)	0% (0/3)

Also IntelliJ idea can show code coverage in your project structure



```
▼ src
  ▼ main
    ▼ java  77% classes, 38% lines covered
      ▼ com.javaguru.todolist  77% classes, 38% lines covered
        ▶ console  0% classes, 0% lines covered
        ▶ domain  100% classes, 55% lines covered
        ▶ repository  100% classes, 37% lines covered
        ▼ service  100% classes, 82% lines covered
          ▼ validation  100% classes, 81% lines covered
              TaskNameValidationRule  100% methods, 100% lines covered
              TaskValidationException  100% methods, 100% lines covered
              TaskValidationRule  100% methods, 66% lines covered
              TaskValidationService  50% methods, 66% lines covered
            TaskService  50% methods, 85% lines covered
          ToDoListApplication  0% methods, 0% lines covered
  ▼ test
```

REFERENCES

- https://www.baeldung.com/mockito-series
- https://www.baeldung.com/mockito-behavior
- https://www.tutorialspoint.com/mockito/index.htm
- https://site.mockito.org/
- https://junit.org/junit4/
- https://maven.apache.org/