Indhold

Test Case 2
Acceptance criteria 2
Create Maven Project 3
Add project name 4
The default POM file 5
Add dependencies 6
Rebuild Maven Project 7
Create CalculatorTest class 8
Create first test method 10
Run the test 11
Add Calculator instance 12
Fix Calculator class 12
Create NumberSource interface 14
Add method to NumberSource interface 14
Add multiply method to Calculator class 15
Update CalculatorTest class 16
Mock the NumberSource instance 17
Parameterized tests 19
Implement the remaining test methods 20

Test Case

Imagine, we're developing a headless (no UI) calculator.

It gets data from, different kind of number sources, which could be in formats like .csv, .xml, .json…

We'll use IntelliJ with the JUnit 5 and Mockito Mocks frameworks to test our solution.

We don't have a real number source implementation yet (so we'll use an interface and mock it out).

The goal is to implement the `multiply()` method, that gets it data from a number source.

Acceptance criteria

We want to assure that following criteria are tested and validated

- Positive * Positive = Positive (+*+ = +)
- Negative * Positive = Negative (-*+ = -)
- Positive * Negative = Negative (+*- = -)
- Negative * Negative = Positive (-*- = +)

We'll start by using the annotation

```
@Test
```

But before making all the test methods, we want to use parameterized tests, where each test can be tested for a set of values, e.g., 1L, 10L, 100L, and maximum long value.

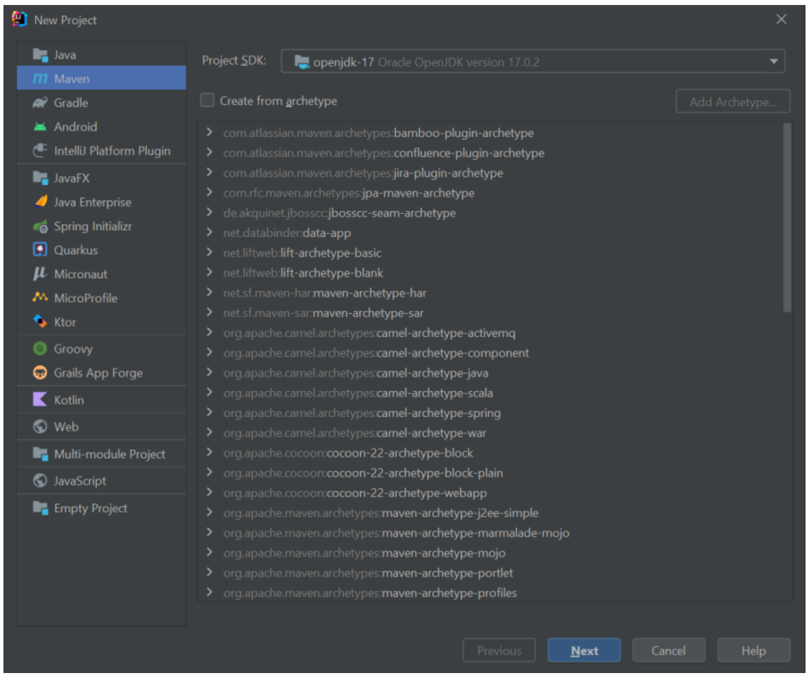
```
@ParameterizedTest
@ValueSource(longs = {1L, 10L, 100L, Long.MAX_VALUE})
```

Create Maven Project

We'll start by creating our Java project, via Maven.

In IntelliJ

- Go to **File > New… > Project**
- Select **Maven**
- You could select a pre-defined Maven archetype, but we'll just use a default Maven project.
- Click **Next**.


New Project
Java
Maven
Gradle
Android
IntelliJ Platform Plugin
JavaFX
Java Enterprise
Spring Initializr
Quarkus
Micronaut
MicroProfile
Ktor
Groovy
Grails App Forge
Kotlin
Web
Multi-module Project
JavaScript
Empty Project
Add Archetype...
com.atlassian.maven.archetypes:bamboo-plugin-archetype
com.atlassian.maven.archetypes:confluence-plugin-archetype
com.atlassian.maven.archetypes:jira-plugin-archetype
com.rfc.maven.archetypes:jpa-maven-archetype
de.akquinet.jbosscc:jbosscc-seam-archetype
net.databinder:data-app
net.liftweb:lift-archetype-basic
net.liftweb:lift-archetype-blank
net.sf.maven-har:maven-archetype-har
net.sf.maven-sar:maven-archetype-sar
org.apache.camel.archetypes:camel-archetype-activemq
org.apache.camel.archetypes:camel-archetype-component
org.apache.camel.archetypes:camel-archetype-java
org.apache.camel.archetypes:camel-archetype-scala
org.apache.camel.archetypes:camel-archetype-spring
org.apache.camel.archetypes:camel-archetype-war
org.apache.cocoon:cocoon-22-archetype-block
org.apache.cocoon:cocoon-22-archetype-block-plain
org.apache.cocoon:cocoon-22-archetype-webapp
org.apache.maven.archetypes:maven-archetype-j2ee-simple
org.apache.maven.archetypes:maven-archetype-marmalade-mojo
org.apache.maven.archetypes:maven-archetype-mojo
org.apache.maven.archetypes:maven-archetype-portlet
org.apache.maven.archetypes:maven-archetype-profiles
Previous
Next
Cancel
Help

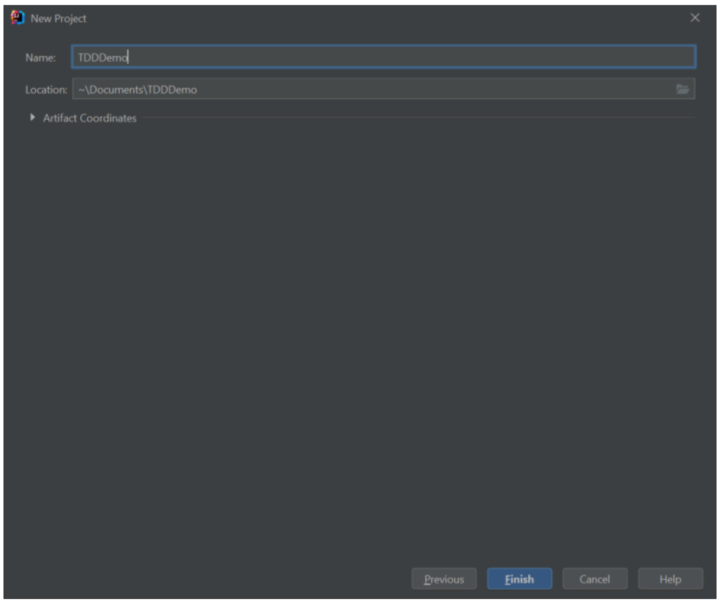
Project SDK: openjdk-17 Oracle OpenJDK version 17.0.2

☐ Create from archetype

Add project name

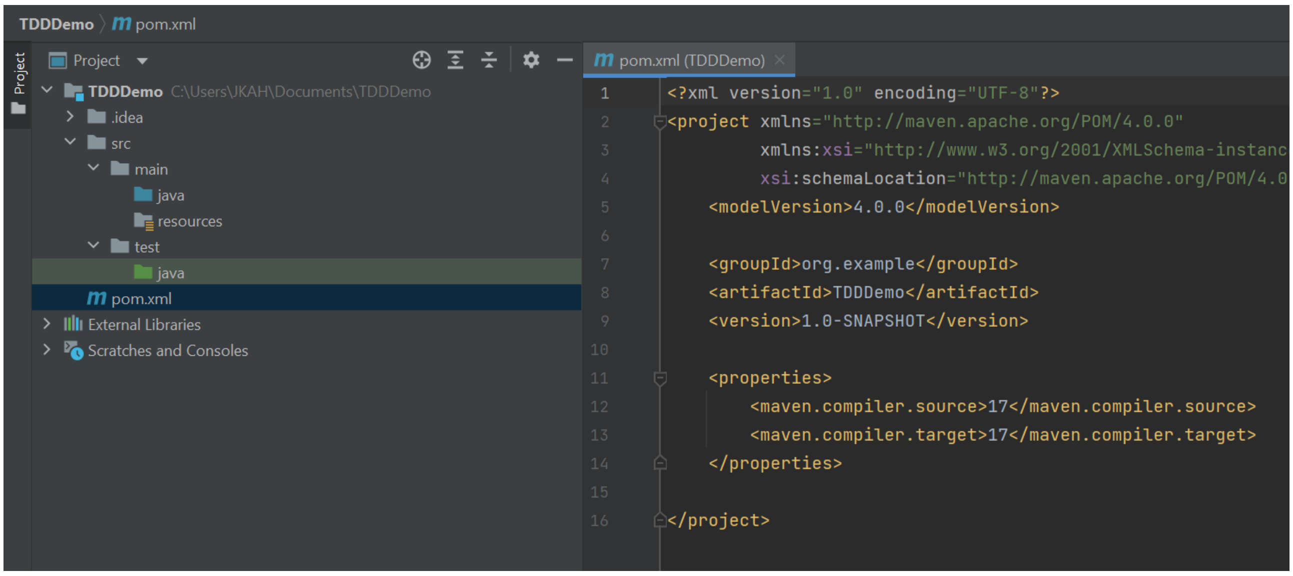
The project needs a name.

- Give the project the name: **TDDDemo**.
- Click **Finish**.



The default POM file

Let’s look at the Maven POM file.



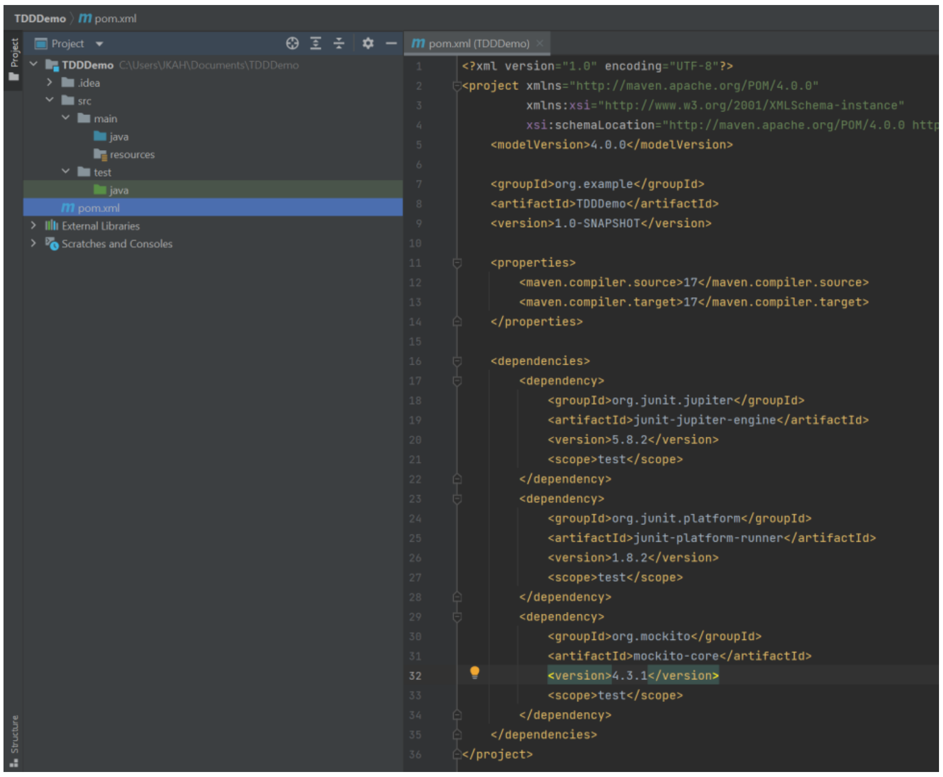
```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instanc
         xsi:schemaLocation="http://maven.apache.org/POM/4.0
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>TDDDemo</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
    </properties>

</project>
```

Add dependencies

Our project dependents on JUnit 5 and Mockito, which we must tell Maven, in the POM file.

- Add the dependencies, below, as the **last thing** in the **project tag**, of the pom.xml file.

```
<dependencies>
   <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>5.8.2</version>
      <scope>test</scope>
   </dependency>
   <dependency>
      <groupId>org.junit.platform</groupId>
      <artifactId>junit-platform-runner</artifactId>
      <version>1.8.2</version>
      <scope>test</scope>
   </dependency>
   <dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-core</artifactId>
      <version>4.3.1</version>
      <scope>test</scope>
   </dependency>
</dependencies>
```



```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>TDDDemo</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>5.8.2</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.platform</groupId>
            <artifactId>junit-platform-runner</artifactId>
            <version>1.8.2</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.mockito</groupId>
            <artifactId>mockito-core</artifactId>
            <version>4.3.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

Note: Some of the dependency versions might have deprecated, since this tutorial was made, so you might have to update some of the versions.

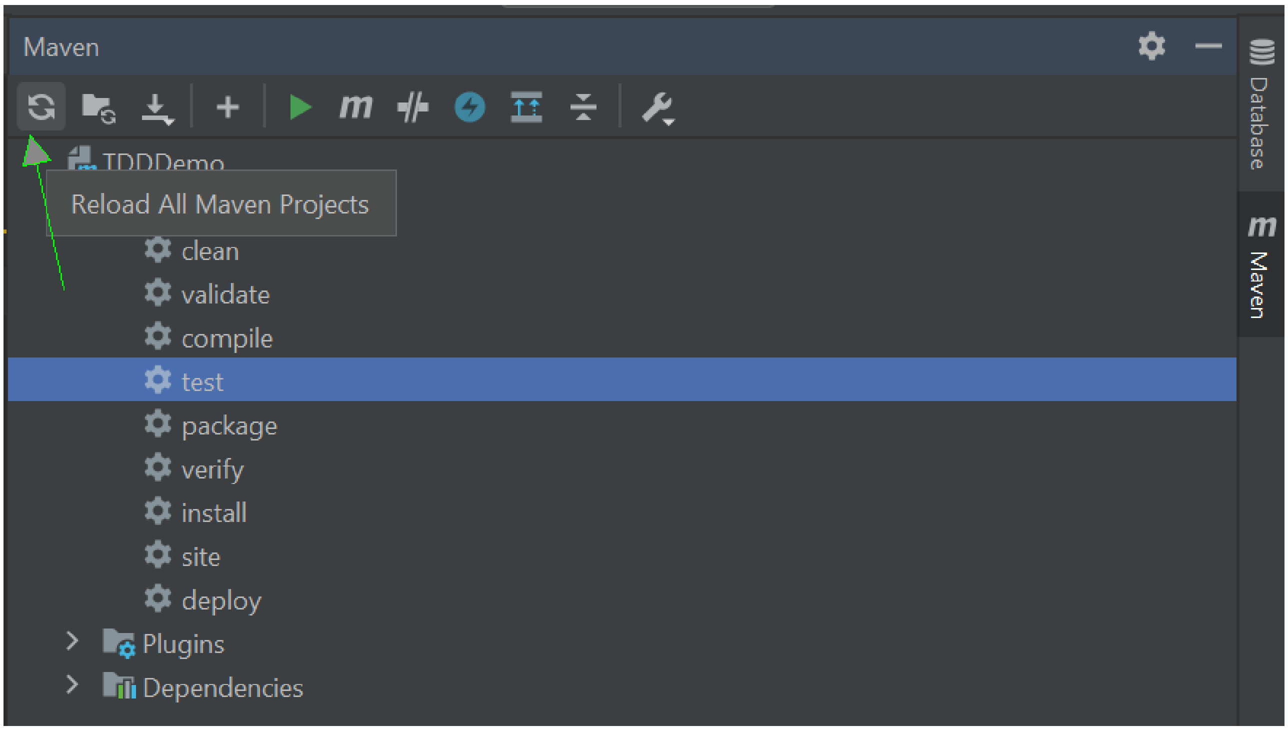
Rebuild Maven Project

Normally, Maven will discover the changes to the pom.xml file, and start to update and fetch its dependencies. But we can also force a rebuild manually.

Via the Maven Window, we can make Maven go fetch our newly configured dependencies.

Either

- **Right click** the **pom.xml** file, in the **Project Window** and select **Maven > Rebuild Project**
- or use the **Maven Window** toolbar icon.

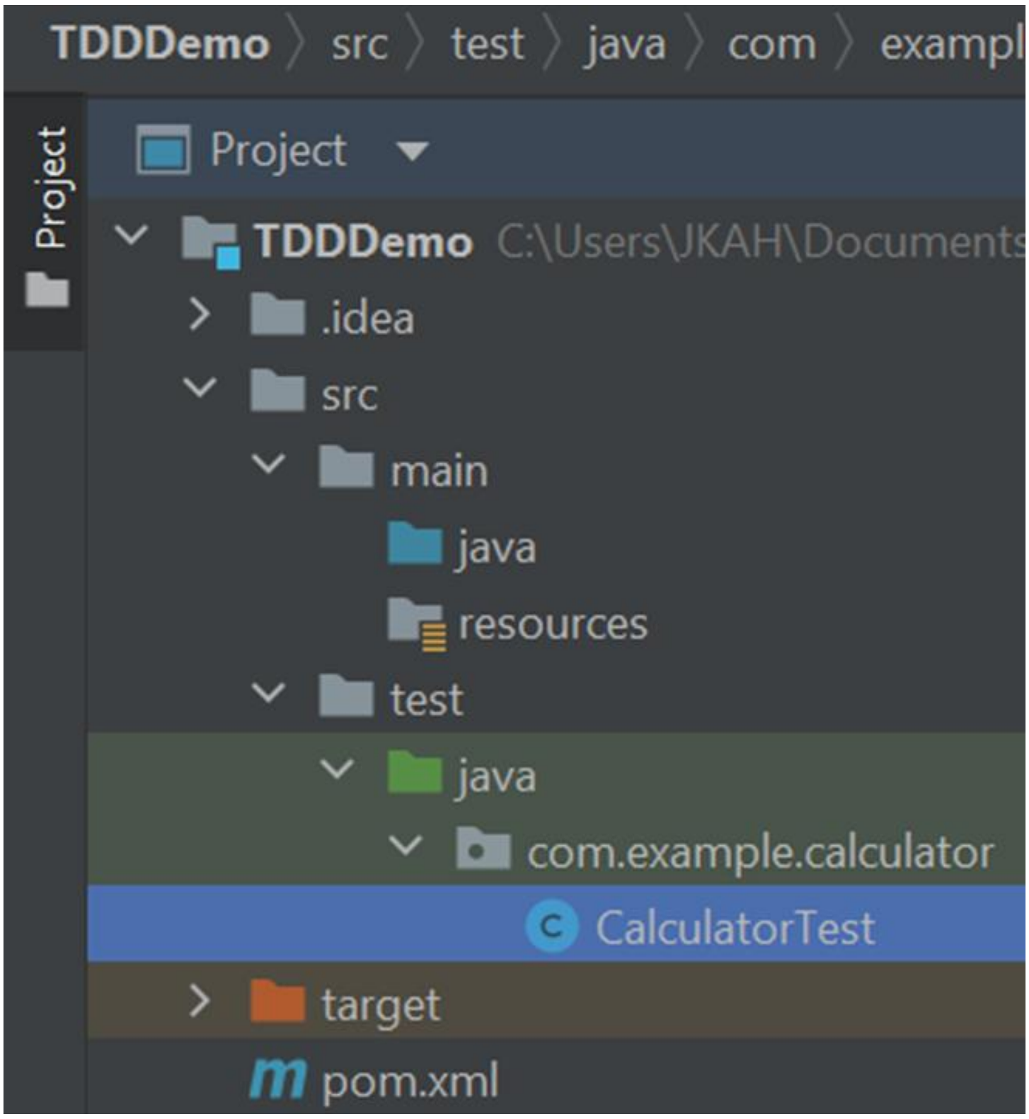


Create CalculatorTest class

Let’s create the test file. It has to be created in the green java test project folder.

In the **Project Window**

- **Right click** the green folder `src/test/java`.
- Select **New > Class** (CTRL+N).
- Input name: `CalculatorTest`
- Hit <**Enter**>.
- Move (F6) the class to package: `com.example.calculator`



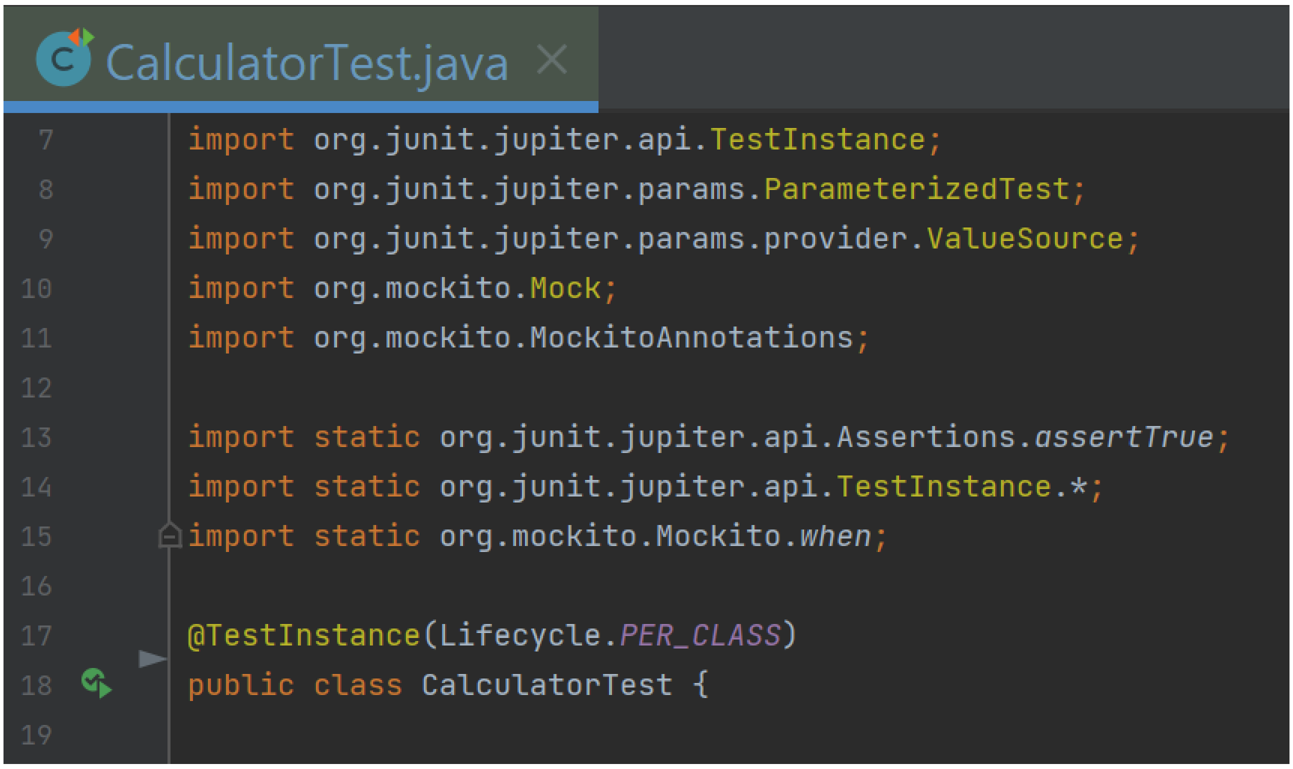
In the new Calculator class

- Add the annotation below, above the class definition.

```
@TestInstance(Lifecycle.PER_CLASS)
```

- Use <**Alt+Enter**> to fix missing imports and make static imports.
- Or manually add the import line below

```
import static org.junit.jupiter.api.TestInstance.*;
```



CalculatorTest.java

```
import org.junit.jupiter.api.TestInstance;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.junit.jupiter.api.TestInstance.*;
import static org.mockito.Mockito.when;

@TestInstance(Lifecycle.PER_CLASS)
public class CalculatorTest {

```

Create first test method

This step has several sub-steps and starts with creating the first failing skeleton test method that will be a template for the rest of test methods.

First, we'll just make the method, that fails purposely.

- Make sure the new class is in package: `com.example.calculator`
- Add code

```
@Test
public void calculator_Multiply_PositiveAndPositive_ReturnsPositive() {

       fail("Not Implemented");
}
```

- Use <**Alt+Enter**> to fix missing imports and make static imports.
- Or manually add the import line below

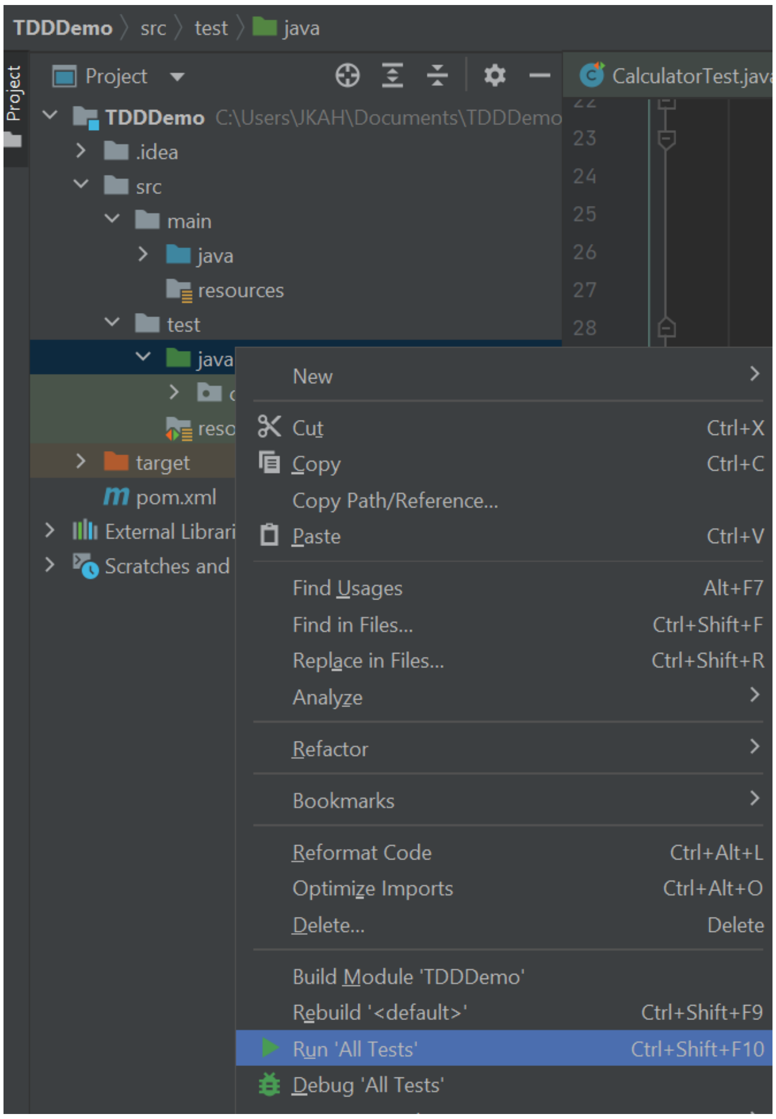
```
import static org.junit.jupiter.api.Assertions.fail;
```

Run the test

Now let’s run the test and see what happens.

In the **Project Window**

- **Right click** the green java folder.
- Select **Run ‘All Tests’** (CTRL-SHIFT F10)



The test should fail, with the message: **Not Implemented**.

Great!

Add Calculator instance

Now, we'll create the **Calculator** class and a **NumberSource** interface.

In method **calculator_Multiply_PositiveAndPositive_ReturnsPositive()**

- Remove code

```
fail("Not Implemented");
```

- Add code

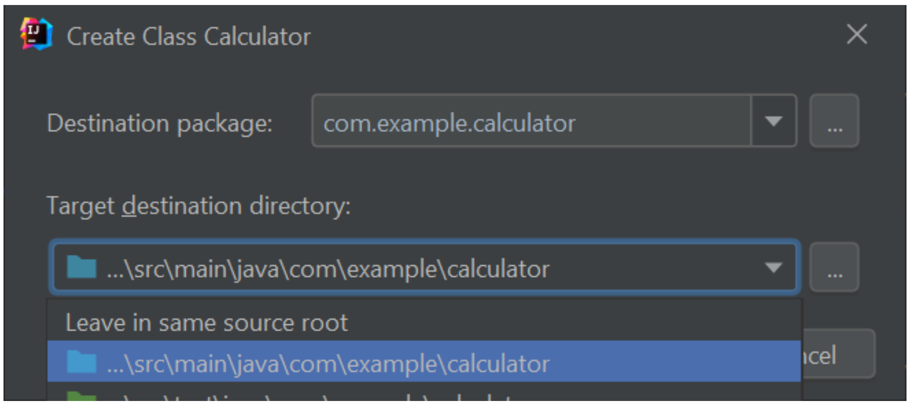
```
Calculator calculator = new Calculator(numberSource);
assertTrue( calculator.multiply() > 0);
```

Now let's start fixing the current errors.

Fix Calculator class

We need to create the class and handle the number source interface.

- Use <**Alt+Enter**>
    - Select **Create class 'Calculator'**.
- Make sure to change and select the **src/main/java/com/example/calculator** folder.

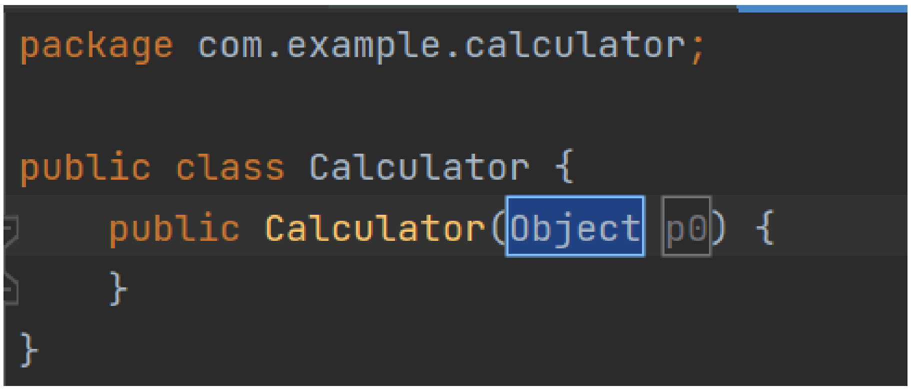


Create Class Calculator

Destination package: com.example.calculator

Target destination directory: ...\src\main\java\com\example\calculator

Leave in same source root
...\src\main\java\com\example\calculator
...



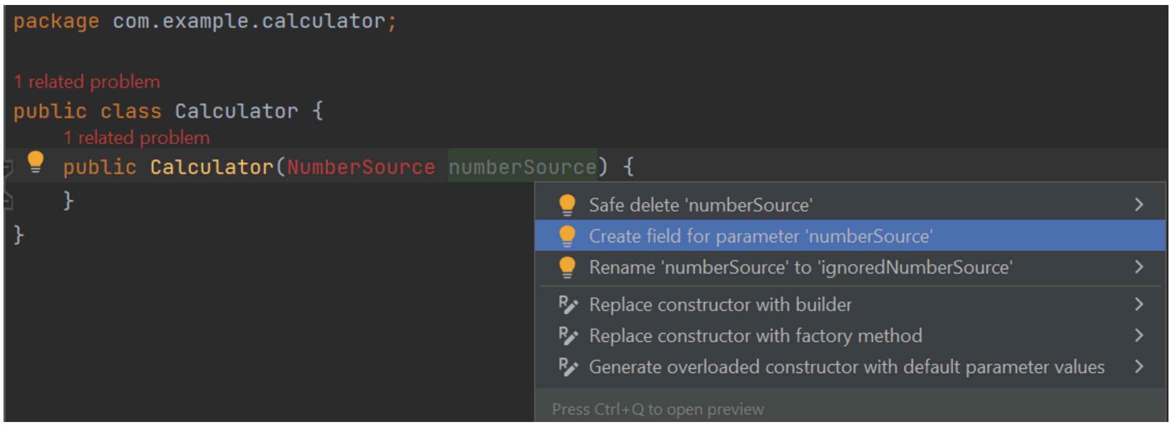
```
package com.example.calculator;

public class Calculator {
    public Calculator(Object p0) {
    }
}
```

We need to change the type and name of the constructor parameter.

- Change type to **NumberSource**.
- Change name to **numberSource**.

Place the cursor on the parameter name, and

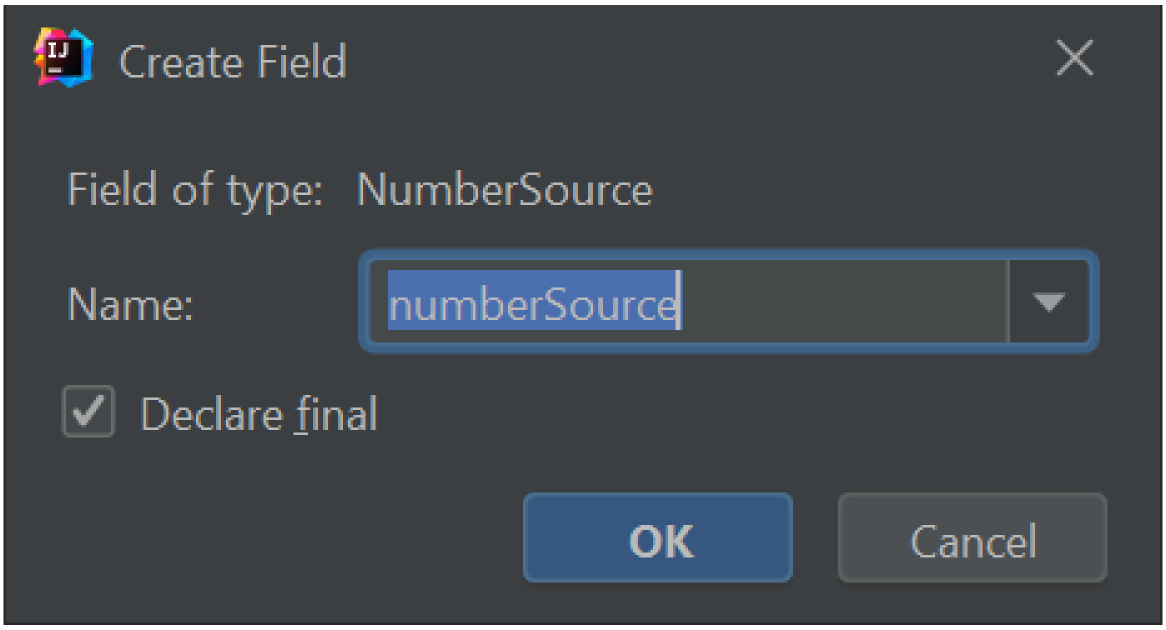
- Press <**Alt+Enter**>
- Select **Create field for parameter 'numberSource'**.



```
package com.example.calculator;

1 related problem
public class Calculator {
    1 related problem
    public Calculator(NumberSource numberSource) {
    }
}
```

Safe delete 'numberSource'
Create field for parameter 'numberSource'
Rename 'numberSource' to 'ignoredNumberSource'
Replace constructor with builder
Replace constructor with factory method
Generate overloaded constructor with default parameter values
Press Ctrl+Q to open preview

- Check the **Declare final**.



Create Field

Field of type: NumberSource

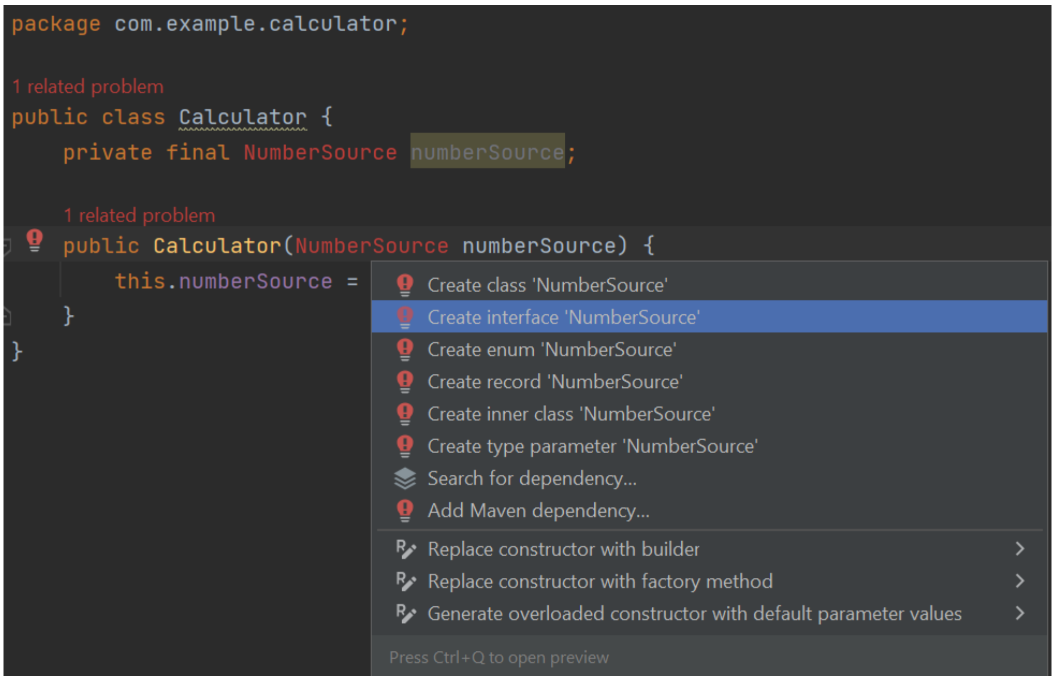
Name: numberSource

[✓] Declare final

Create NumberSource interface

Now, let’s create the **NumberSource** interface.

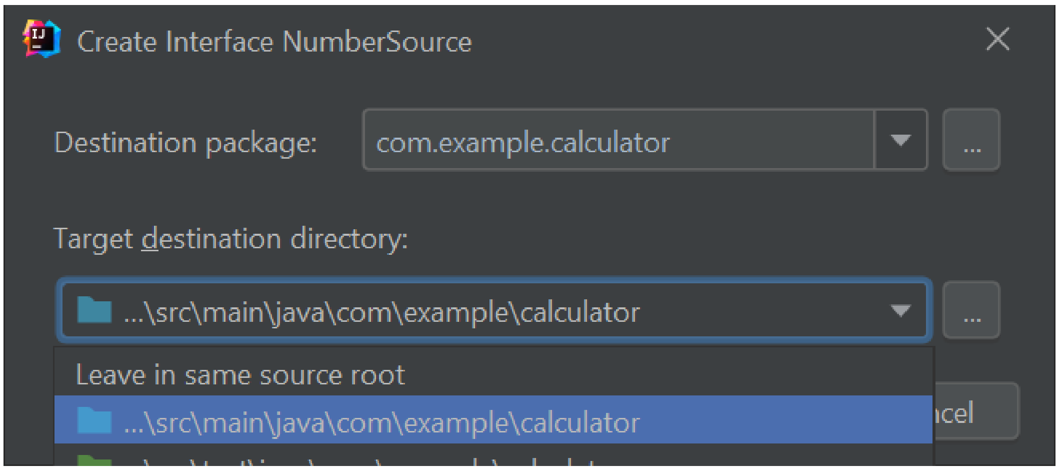
- Place the cursor on the **parameter** type, **NumberSource**, in the **constructor** method declaration.
- Press <**Alt+Enter**>.
- Select **Create interface ‘NumberSource’**.



```
package com.example.calculator;

1 related problem
public class Calculator {
    private final NumberSource numberSource;

    1 related problem
    public Calculator(NumberSource numberSource) {
        this.numberSource =
    }
}
```

Create class 'NumberSource'
Create interface 'NumberSource'
Create enum 'NumberSource'
Create record 'NumberSource'
Create inner class 'NumberSource'
Create type parameter 'NumberSource'
Search for dependency...
Add Maven dependency...
Replace constructor with builder
Replace constructor with factory method
Generate overloaded constructor with default parameter values
Press Ctrl+Q to open preview

- Remember to select the **src/main/java/com/example/calculator** production code folder.



Create Interface NumberSource
Destination package: com.example.calculator
Target destination directory:
...\src\main\java\com\example\calculator
Leave in same source root
...\src\main\java\com\example\calculator
ncel

Add method to NumberSource interface

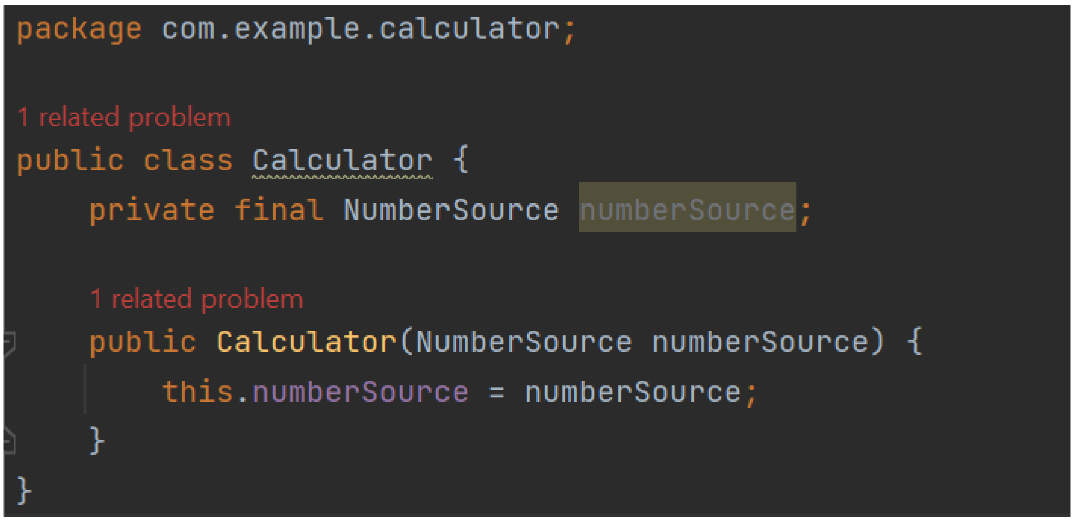
The NumberSource interface, needs a method called next(), that will supply the next number.

- Add interface method: `long next();`
- Save the file.

Add multiply method to Calculator class

Now we have defined the **NumberSource** interface, let's go back to the Calculator class and implement the multiply method.

Notice that the class and method mention **1 related problem**.

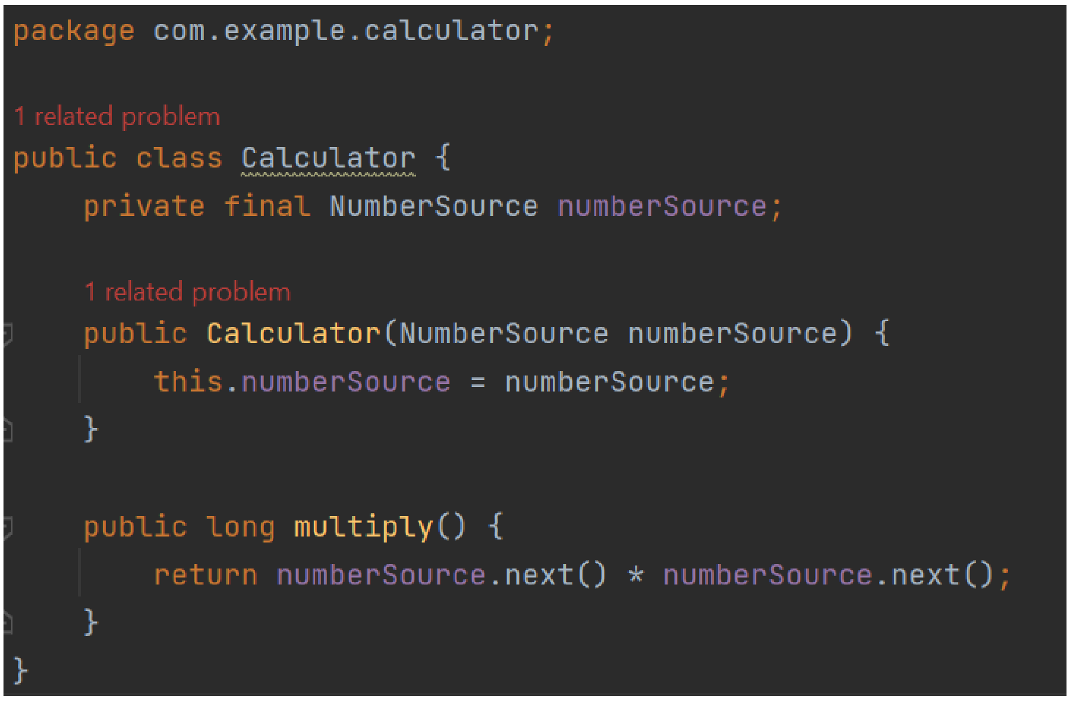


```
package com.example.calculator;

1 related problem
public class Calculator {
    private final NumberSource numberSource;

    1 related problem
    public Calculator(NumberSource numberSource) {
        this.numberSource = numberSource;
    }
}
```

We'll take care of it, in a minute, after adding a multiply method to our calculator.

- Add code

```
public long multiply() {

       return numberSource.next() * numberSource.next();

}
```



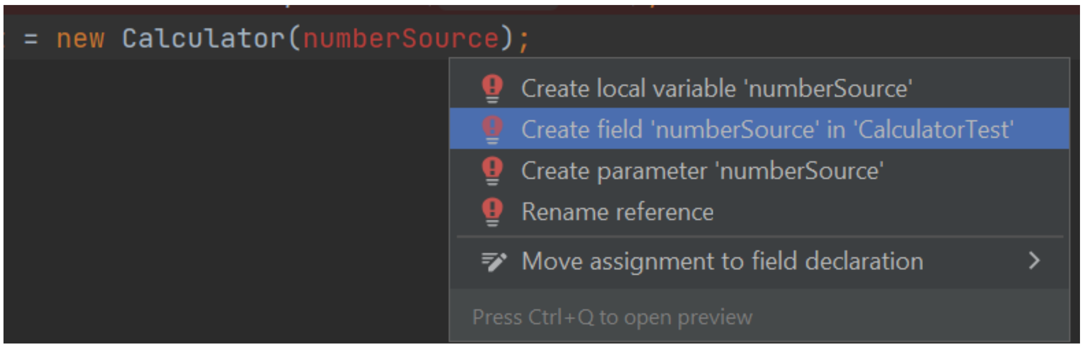
```
package com.example.calculator;

1 related problem
public class Calculator {
    private final NumberSource numberSource;

    1 related problem
    public Calculator(NumberSource numberSource) {
        this.numberSource = numberSource;
    }

    public long multiply() {
        return numberSource.next() * numberSource.next();
    }
}
```

Now let's go fix the problems in **CalculatorTest** class.

Update CalculatorTest class

We need to declare the **numberSource** variable passed to the **Calculator constructor**.

- Place the cursor on the **numberSource** variable.
- Press <**Alt+Enter**>.
- Select **Create field 'numberSource' in 'CalculatorTest'**.



```
= new Calculator(numberSource);
```

Create local variable 'numberSource'
Create field 'numberSource' in 'CalculatorTest'
Create parameter 'numberSource'
Rename reference
Move assignment to field declaration
Press Ctrl+Q to open preview

In the **Project Window**

- **Right click** the green java folder.
- Select **Run 'All Tests'** (CTRL-SHIFT F10)

The test should fail. So, we're not done, yet.

The problem is

- Cannot invoke "**com.example.calculator.NumberSource.next()**" because "**this.numberSource**" is null

Fair enough, we haven't instantiated an implementation of a **NumberSource**.

We don't have a **NumberSource** implementation, so we'll mock (stub) it, via the Mockito framework, and thereby create our own proxy supplying data, that we have 100% control over.

This will also avoid referencing an implementation, which is often much more complex to control than an interface.

Mock the NumberSource instance

We need two methods from the Mockito framework.

- MockitoAnnotations.openMocks(this);
- Mockito.when(numberSource.next()).thenReturn(10L, 10L);

MockitoAnnotations.openMocks(this)

Initializes the mocking framework, for this instance.

Mockito.when(numberSource.next()).thenReturn(10L, 10L);

This simulates the **NumberSource** to contain two number, returning one at a time, via a call to the **next()** method of the **numberSource** instance.

Ex. first call to **numberSource.next()** returns the first 10L, and next call will return the next value, which, in this case is also 10L.

To start with, we want to mock out the numberSource field.

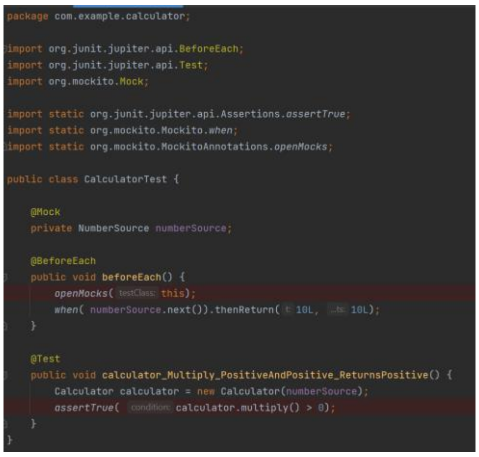
- Add **@Mock**, above the **numberSource** class field.

We want to run the mock methods for each test method, that by time will be added to this test class, because the calculator will get more methods, as time goes by.

So, we'll use the **@BeforeEach** annotation on a new method, that we also call **beforeEach**.

- Add code

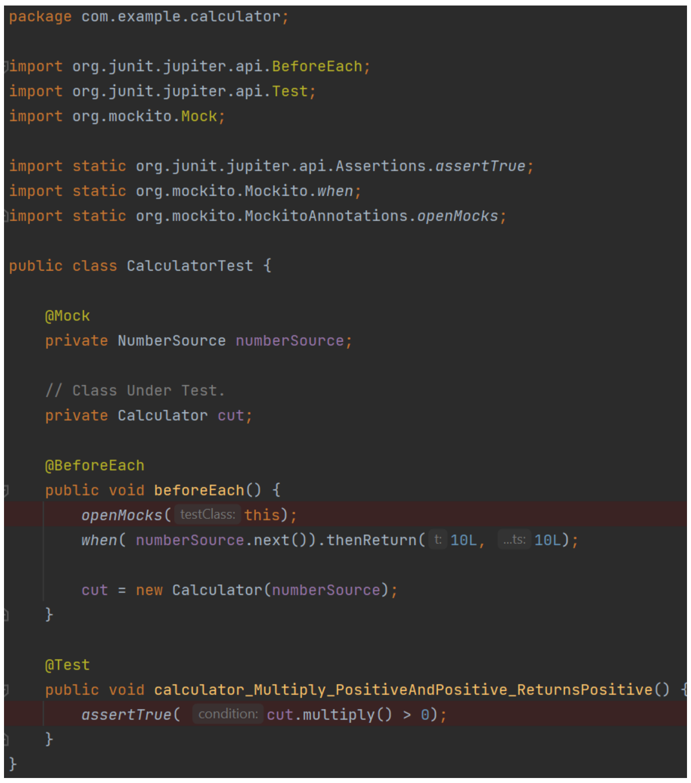
```
@BeforeEach

public void beforeEach() {

        MockitoAnnotations.openMocks(this);

        Mockito.when(numberSource.next()).thenReturn( 10L, 10L);

}
```



```
package com.example.calculator;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.Mock;

import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.mockito.Mockito.when;
import static org.mockito.MockitoAnnotations.openMocks;

public class CalculatorTest {

    @Mock
    private NumberSource numberSource;

    @BeforeEach
    public void beforeEach() {
        openMocks( testClass: this);
        when( numberSource.next()).thenReturn( t: 10L, ...ts: 10L);
    }

    @Test
    public void calculator_Multiply_PositiveAndPositive_ReturnsPositive() {
        Calculator calculator = new Calculator(numberSource);
        assertTrue( condition: calculator.multiply() > 0);
    }
}
```

Let’s also move the instantiation of the calculator.

- Change and move the calculator instantiation to be an **instance field**.
- Instantiate the calculator in the **beforeEach** method.
- Refactor the calculator field and name it **cut or sut** (ClassUnderTest or SubjectUnderTest)



```java
package com.example.calculator;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.Mock;

import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.mockito.Mockito.when;
import static org.mockito.MockitoAnnotations.openMocks;

public class CalculatorTest {

    @Mock
    private NumberSource numberSource;

    // Class Under Test.
    private Calculator cut;

    @BeforeEach
    public void beforeEach() {
        openMocks( testClass: this);
        when( numberSource.next()).thenReturn( t: 10L,  ...ts: 10L);

        cut = new Calculator(numberSource);
    }

    @Test
    public void calculator_Multiply_PositiveAndPositive_ReturnsPositive() {
        assertTrue( condition: cut.multiply() > 0);
    }
}
```

In the **Project Window**

- **Right click** the green java folder.
- Select **Run ‘All Tests’** (CTRL-SHIFT F10)

The test should pass. Great work!

Now that’s not all, we still have to

- Support parameterized test, with different test values declared via the @ValueSource(…) annotation.
- Implement the last three test methods.

Parameterized tests

With parameterized tests you can trigger the test method x times, with different test values, without having to code it. By using the **@ValueSource** annotation, you can pass a list of values. If you declare five values, the test method will be called five time, passing the value in a needed method parameter of the same type, as declared in the **@ValueSource** annotation.

We need to use these annotations,

```
@ParameterizedTest

@ValueSource(longs = {1L, 10L, 100L, Long.MAX_VALUE})
```

and we need to define the dependency, in the pom.xml file, to fetch another jar file.

```
<dependency>

        <groupId>org.junit.jupiter</groupId>

        <artifactId>junit-jupiter-params</artifactId>

        <version>5.8.2</version>

        <scope>test</scope>

</dependency>
```

For each test, we want to pass in the values from the **@ValueSource** annotation, so we need to add a new long value parameter to the test method, and the value will be passed in automatically.

And we also want to use the **Mockito.when(…)** inside the test method(s).

So, above the test method

- Change the **@Test** annotation to **@ParameterizedTest**.

```
~~@Test~~
@ParameterizedTest
```

- Add annotation

```
@ValueSource(longs = {1L, 10L, 100L, Long.MAX_VALUE})
```

- Add a parameter (**long value**) to the test method:

```
public void calendar_Myltiply_PositiveAndPositive_ReturnsPositive(long value)
```

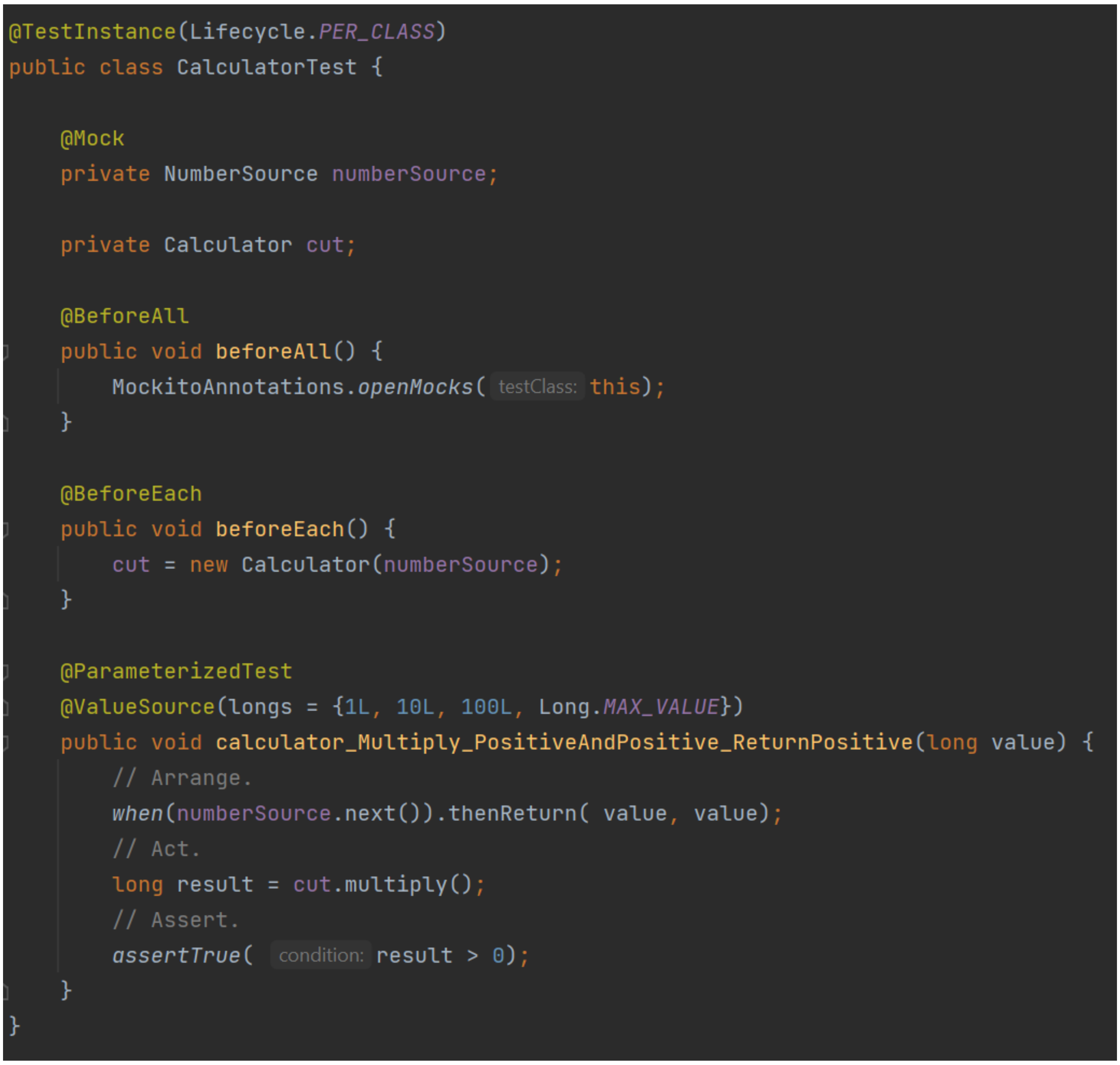
- Move use of **Mockito.when()** from **beforeEach()** to the test method.

```
when(numberSource.next()).thenReturn(value, value);
```

Implement the remaining test methods

To fulfill the acceptance criteria, we need to make the remaining methods.

```
calculator_Multiply_PositiveAndNegative_ReturnsNegative(long value)

calculator_Multiply_NegativeAndPositive_ReturnsNegative(long value)

calculator_Multiply_NegativeAndNegative_ReturnsPositive(long value)
```



```java
@TestInstance(Lifecycle.PER_CLASS)
public class CalculatorTest {

    @Mock
    private NumberSource numberSource;

    private Calculator cut;

    @BeforeAll
    public void beforeAll() {
        MockitoAnnotations.openMocks( testClass: this);
    }

    @BeforeEach
    public void beforeEach() {
        cut = new Calculator(numberSource);
    }

    @ParameterizedTest
    @ValueSource(longs = {1L, 10L, 100L, Long.MAX_VALUE})
    public void calculator_Multiply_PositiveAndPositive_ReturnPositive(long value) {
        // Arrange.
        when(numberSource.next()).thenReturn( value, value);
        // Act.
        long result = cut.multiply();
        // Assert.
        assertTrue( condition: result > 0);
    }
}
```