Testing Java Code

With MICROEJ SDK 6

DISCLAIMER

PREREQUISITES

Hardware required:

- NXP i.MX RT1170 Evaluation Kit (EVKB) + micro-USB cable + RK055HDMIPI4MA0 display panel

Environment Setup:

- Follow the NXP i.MX RT1170 Evaluation Kit Getting Started to setup your environment and run a demo application on the Virtual Device and on the i.MX RT1170 Evaluation Kit.
- Note: the next slides are using **IntelliJ IDEA** with **MicroEJ plugin for IntelliJ IDEA 1.1.0**. This training supports all other available IDEs (Android Studio, VS Code, …)

This training requires the Getting Started to be completed until the **Run an Application on the i.MX RT1170 Evaluation Kit** section (included).

⚠️ The path to the NXP i.MX RT1170 VEE Port sources should be as short as possible and contain **no whitespace** or **non-ASCII character.**

This training provides a step-by-step hands on to create a library from scratch.
Note that the source code of the library is available in a package provided with those slides.

WHAT YOU WILL LEARN

- JUnit Basics
- Create and configure unit tests on a Java project
- Run tests on the Simulator
- Generate a Code Coverage report
- Run tests on a Device
- Advanced Configurations

JUnit Academic

JUNIT DEFINITION

JUnit is a unit testing framework for the Java programming language.

JUnit provides:

- Annotations to structure your test case
- A set of assertion methods useful for writing tests
- Facilities to execute test suites
- Stats for each test:
  - Pass / Fail status
  - Execution time

ANNOTATIONS (1/2)

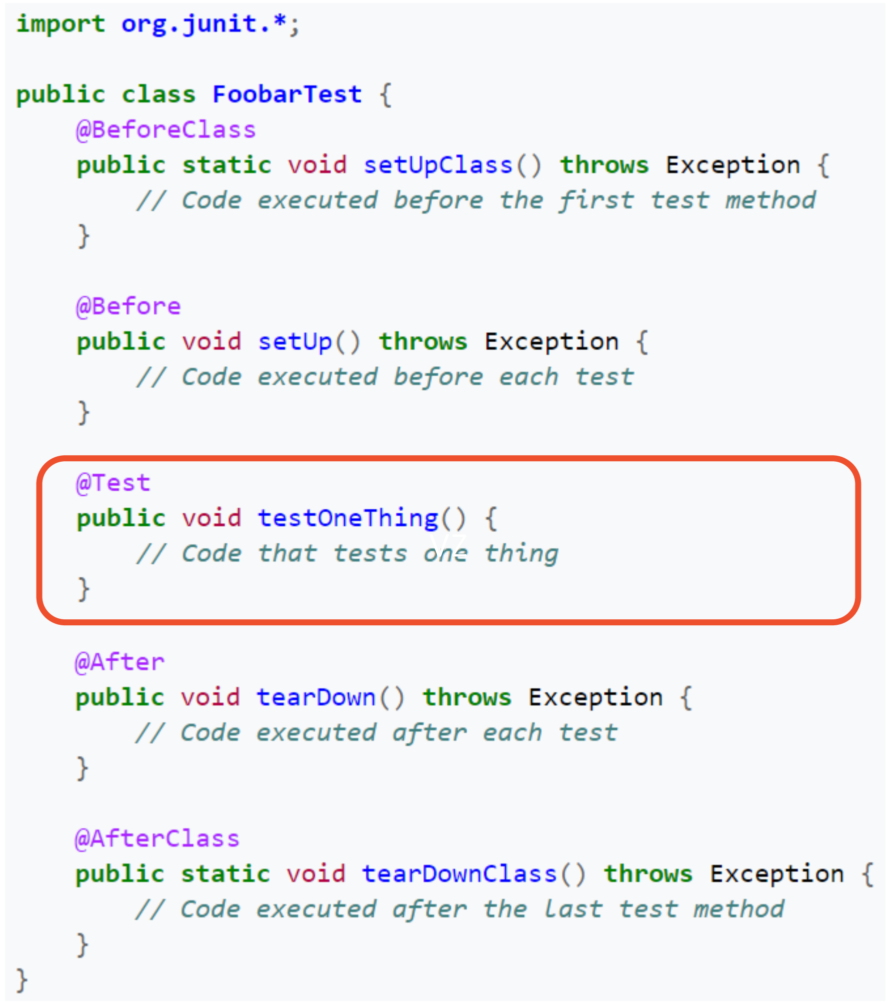
MicroEJ is compliant with a subset of JUnit version 4. MicroEJ JUnit processor supports the following annotations:

- **`@Before`**
  - Code executed before each test.
  - Name Convention : **`setUp()`**
- **`@After`**
  - Code executed after each test
  - Name Convention : **`tearDown()`**
- **`@Test`**
  - Indicates that the method is a test method that should be executed by the JUnit framework.
- **`@Test(expected = MyException.class)`**
  - Indicates that the method is a test method that is expected to throw a specific exception, in this case, “MyException”.

ANNOTATIONS (2/2)

- **`@BeforeClass`**
  - Code executed before the first test method
  - Name Convention : **`setUpBeforeClass()`**
- **`@AfterClass`**
  - Code executed after the last test method
  - Name Convention : **`tearDownClass()`**
- **`@Ignore`**
  - Code ignored by the test suite

JUNIT EXAMPLE



```java
import org.junit.*;

public class FoobarTest {
    @BeforeClass
    public static void setUpClass() throws Exception {
        // Code executed before the first test method
    }

    @Before
    public void setUp() throws Exception {
        // Code executed before each test
    }

    @Test
    public void testOneThing() {
        // Code that tests one thing
    }

    @After
    public void tearDown() throws Exception {
        // Code executed after each test
    }

    @AfterClass
    public static void tearDownClass() throws Exception {
        // Code executed after the last test method
    }
}
```

- Junit 4 – Annotations Based
- Each test case entry point must be declared using the **org.junit.Test** annotation (**@Test** before a method declaration).
- **Tests execution order is not guaranteed.** JUnit considers that tests are independent of one another. It is recommended to write tests that are independent of one another to avoid issues related to execution order.
- Refer to JUnit documentation to get details on the usage of other annotations.

ASSERTIONS

In each test, the function and the result are checked by assertion, here is a non-exhaustive list of the available assertions:

- **`assertEquals(a,b)`**
  - Return true if ‘a’ is equals with ‘b’ (‘a’ and ‘b’ should be primitive or object)
- **`assertTrue(a)`** and **`assertFalse(a)`**
  - Asserts that a condition is true.
- **`assertSame(a,b)`** and **`assertNotSame(a,b)`**
  - Check if ‘a’ or ’b’ referred to the same object
- **`assertNull(a)`** and **`assertNotNull(a)`**
  - Return true if ‘a’ is NULL or not. ‘a’ must be an Object.
- **`fail(message)`**
  - Stop the test and raise exception.

Check JUnit Javadoc for more information about available Assertions.

GOOD PRACTICES

- Prefer black-box tests (with a maximum coverage).
- Here is the test packages naming convention:
  - Suffix package with .test for black-box tests.
  - Use the same package for white-box tests (allow to use classes with package visibility).
- Run tests as often as possible, ideally after each code change. You can execute tests in CI every day.
- Write a test for any reported bugs, even if it’s fixed.
- Test each methods separately, JUnit stops at the first error.
- Be careful, private methods cannot be tested!
  - If you want to test a function but you don’t want to expose it, use the package visibility.

Hands-On

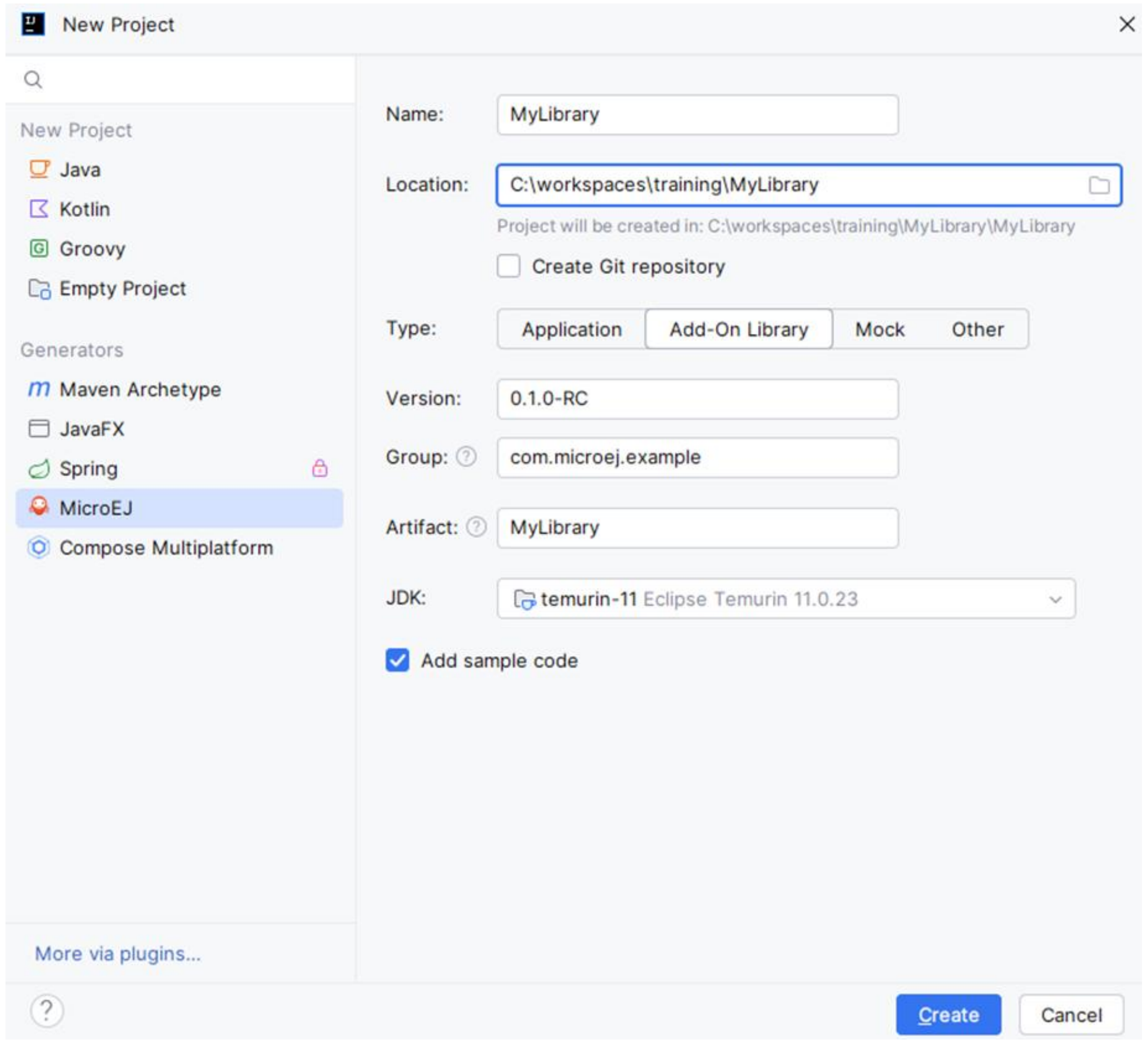
Create and configure unit tests on a Java project

CREATE AN ADD-ON LIBRARY PROJECT

The creation of a project with IntelliJ IDEA is done as follows:

- Click on **File > New > Project...**.
- Select **MicroEJ** in Generators list on the left panel.
- Fill the name of the project in the **Name** field.
- Select the location of the project in the **Location** field.
- Select the **Add-on Library** project type.
- Fill the version of the artifact to publish in the **Version** field.
- Fill the group of the artifact to publish in the **Group** field.
- Fill the name of the artifact to publish in the **Artifact** field.
- Select the JVM used by Gradle in the **JDK** combobox.

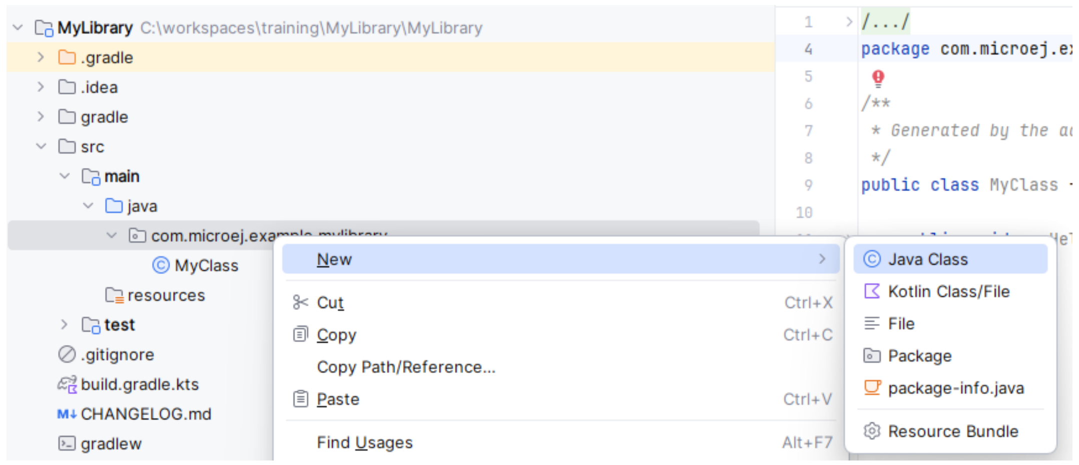
- Click on **Create** button.



Name: MyLibrary
Location: C:\workspaces\training\MyLibrary
Project will be created in: C:\workspaces\training\MyLibrary\MyLibrary
Create Git repository
Type: Add-On Library
Version: 0.1.0-RC
Group: com.microej.example
Artifact: MyLibrary
JDK: temurin-11 Eclipse Temurin 11.0.23
Add sample code

ADD CLASSES TO THE PROJECT (1/2)

- Right-Click on the **com.microej.example.mylibrary** package.
- Select **New > Java Class**:



```
/.../
package com.microej.e

/**
 * Generated by the a
 */
public class MyClass
```

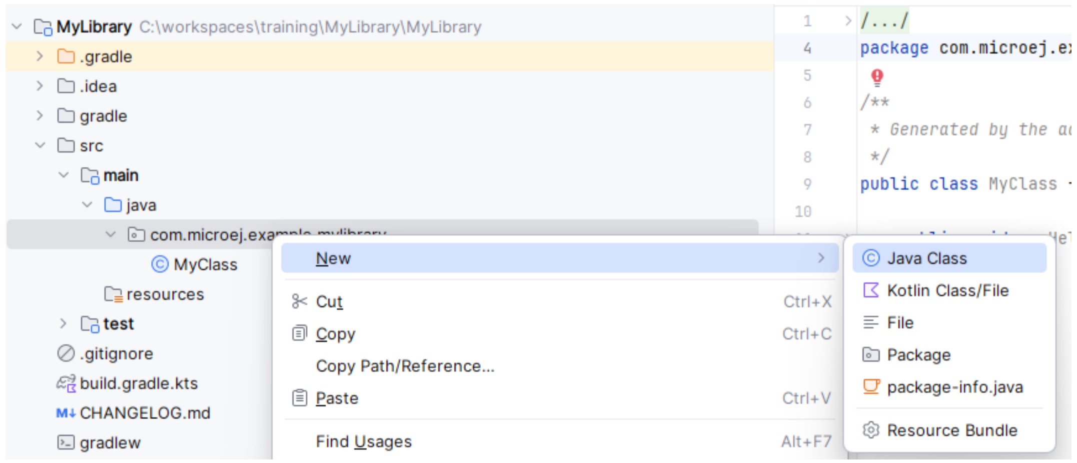
- Create the **Calculator** class.

- Add the following code:

```
public class Calculator {
   private final int a, b;

   /**
    * Calculator class providing methods to compute the sum and
    * the division of 2 parameters.
    * @param a the 1st parameter
    * @param b the 2nd parameter
    */
   public Calculator(int a, int b) {
      this.a = a; this.b = b;
   }

   /**
    * @return the sum of ‘a’ and ‘b’
    */
   public int sum() {
      return this.a + this.b;
   }

   /**
    * @return the division of ‘a’ and ‘b’
    */
   public int divide() {
      return this.a / this.b;
   }
}
```

ADD CLASSES TO THE PROJECT (2/2)

- Right-Click on the **com.microej.example.mylibrary** package.
- Select **New > Java Class**:



```
/.../
package com.microej.e

/**
 * Generated by the a
 */
public class MyClass
```

- Create the **Statistics** class.

- Add the following code:

```
public class Statistics {

    /* Prevent class initialization */
    private Statistics(){

    }

    /**
     * Computes the mean of an array.
     * @param numbers array of numbers
     * @return mean of the array
     */
    public static int mean(int[] numbers) {
        if (numbers.length == 0) {
            throw new IllegalArgumentException("Array cannot be empty");
        }
        int sum = 0;
        for (int num : numbers) {
            sum += num;
        }
        return sum / numbers.length;
    }
}
```

Running Tests On Simulator

TEST ON SIMULATOR

- Tests can be executed on the Simulator. They are run on a target VEE Port and generate a JUnit XML report.

- Executing tests on the Simulator allows to check the behavior of the code in an environment similar to the target device but without requiring the board.
  This solution is therefore less constraining and more portable than testing on the board.

TESTSUITE CONFIGURATION

The following testsuite configuration is defined inside the following block in the **build.gradle.kts** file:

```
testing {
   suites { // (1)
      val test by getting(JvmTestSuite::class) { // (2)
         microej.useMicroejTestEngine(this) // (3)

         dependencies { // (4)
            implementation(project())
            implementation("ej.api:edc:1.3.7")
            implementation("ej.library.test:junit:1.11.0")
            implementation("org.junit.platform:junit-platform-launcher:1.8.2")
         }
      }
   }
}
```

This piece of configuration is the minimum configuration required to define a testsuite on the Simulator:

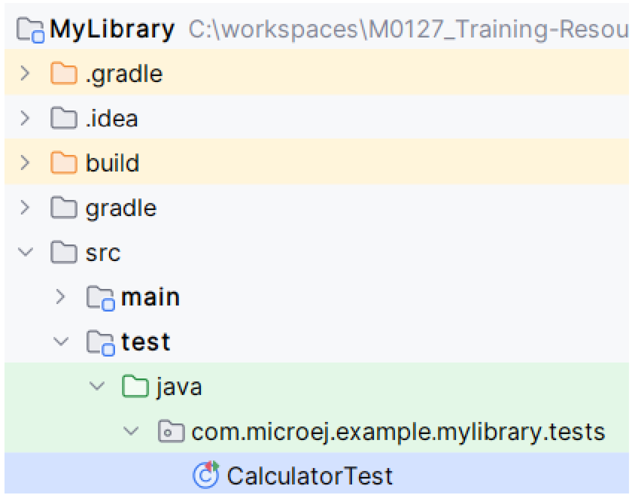
- (1): configures all the testsuites of the project.
- (2): configures the built-in test suite provided by Gradle. Use this testsuite to configure the tests on the Simulator.
- (3): declares that this testsuite uses the MicroEJ Testsuite Engine. By default, the MicroEJ Testsuite Engine executes the tests on the Simulator.
- (4): adds the dependencies required by the tests. The first line declares a dependency to the code of the project. The second line declares a dependency on the edc Library. The third line declares a dependency to the JUnit API used to annotate Java Test classes. Finally the fourth line declares a dependency to a required JUnit library.

Note: the testsuite is already configured when creating an Add-On library project.

ADD CALCULATOR TEST CLASS

CREATE THE TEST CLASS

- Right-Click on the **src/test/java** folder.
- Select **New > Package:**
  - Create the **com.microej.example.mylibrary.tests** package.
- Select **New > Java Class:**
  - Create the **CalculatorTest** class.



MyLibrary C:\workspaces\M0127_Training-Resou

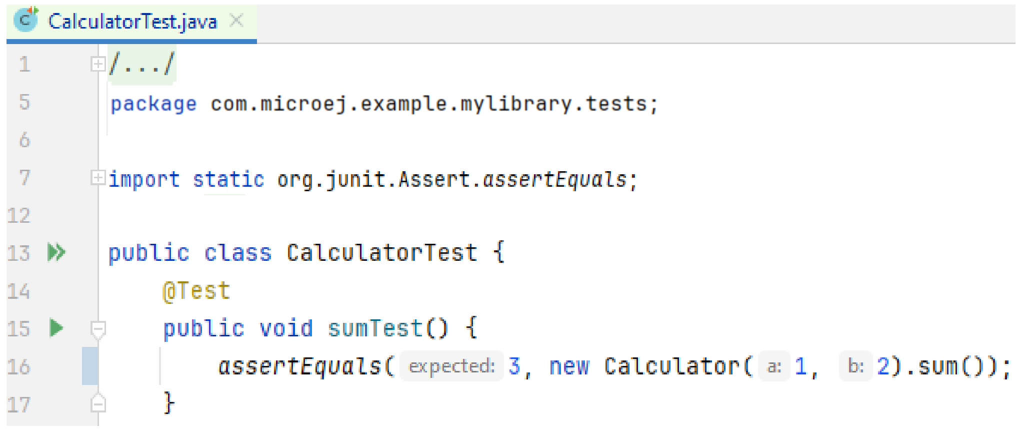
```
> .gradle
> .idea
> build
> gradle
v src
  > main
  v test
    v java
      v com.microej.example.mylibrary.tests
          CalculatorTest
```

CREATE A TEST CASE

- In the **CalculatorTest** editor, press **Alt + Insert**.
- Select **TestMethod > Junit 4**.
- Call it **sumTest**.
- Add the following code to test the **sum()** function of the **Calculator** class.

```
assertEquals(3, new Calculator(1, 2).sum());
```

- The **CalculatorTest** class should look like that:



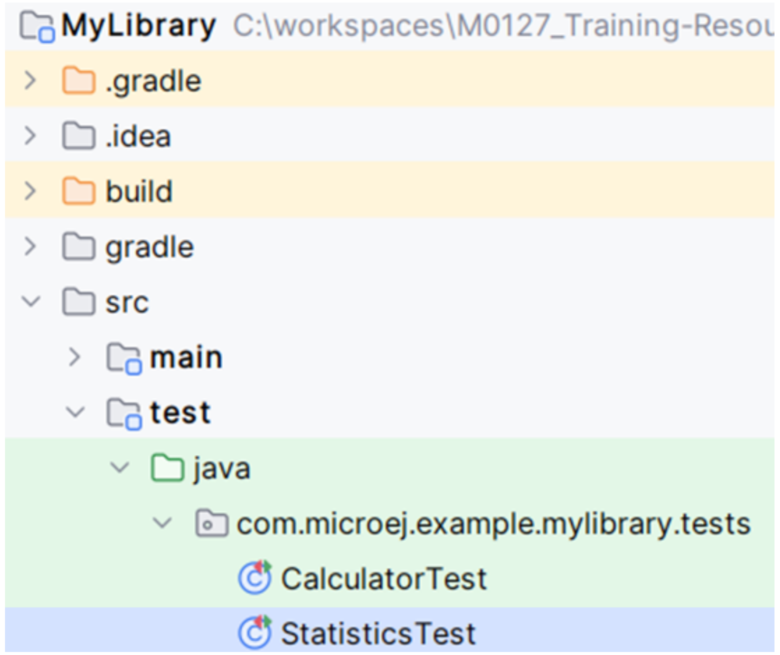
CalculatorTest.java

```
/.../
package com.microej.example.mylibrary.tests;

import static org.junit.Assert.assertEquals;

public class CalculatorTest {
    @Test
    public void sumTest() {
        assertEquals( expected: 3, new Calculator( a: 1, b: 2).sum());
    }
```

ADD STATISTICS TEST CLASS

CREATE THE TEST CLASS

- Right-Click on the **com.microej.example.mylibrary.tests** package.
- Select **New > Java Class**.
- Create the **StatisticsTest** class.


MyLibrary C:\workspaces\M0127_Training-Resou
.gradle
.idea
build
gradle
src
main
test
java
com.microej.example.mylibrary.tests
CalculatorTest
StatisticsTest

CREATE A TEST CASE

- In the **StatisticsTest** editor, press **Alt + Insert**.
- Select **TestMethod > Junit 4**.
- Call it **meanTest**.
- Add the following code to test the **meanTest()** function of the **Statistics** class:

```
/**
 * Tests the {@link Statistics#mean(int[])} method with a valid data set.
 * Asserts that the mean is not equal to 0 and checks that the calculated mean
 * is equal to the expected value of 6.
 */
@Test
public void meanTest() {
   int[] data = {10,5,5,10,2,4};
   assertNotEquals(0, Statistics.mean(data));
   assertEquals(6, Statistics.mean(data));
}
```

SETUP A VEE PORT

Before running tests, at least one target VEE Port must be configured.
If several VEE Ports are defined, the testsuite is executed on each of them.

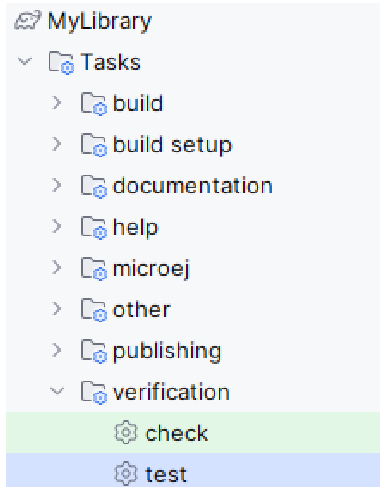
In **build.gradle.kts**, add the **NXP i.MX RT1170 VEE Port 3.0.0** in the **dependencies** section:

```
dependencies {
   implementation("ej.api:edc:1.3.7")


   //Uncomment the microejVee dependency to set the VEE Port or Kernel to use
   microejVee("com.nxp.vee.mimxrt1170:vee-port:3.0.0")
}
```

EXECUTE THE TESTS

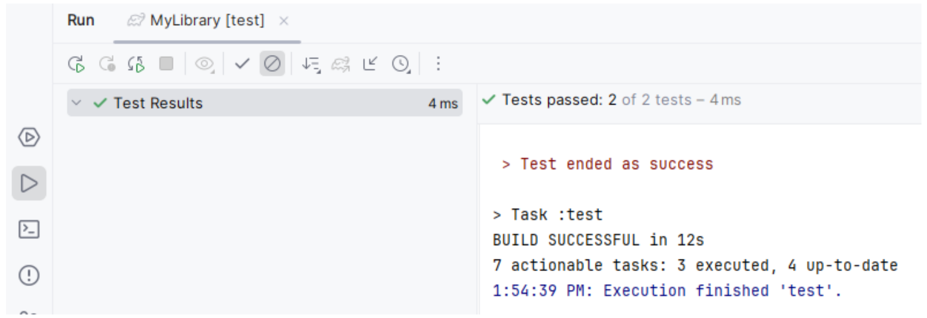
Once the testsuite is configured, it can be run thanks to the **test** Gradle task.
This task is bound to the **check** and the **build** Gradle lifecycle tasks, which means that the tests are also executed when launching one of these tasks.

To execute the tests, double-click on the **test** task in the Gradle tasks view:



```
MyLibrary
  Tasks
    build
    build setup
    documentation
    help
    microej
    other
    publishing
    verification
      check
      test
```

The Testsuite engine launches the available test cases on Simulator. The status can be checked in the console view:

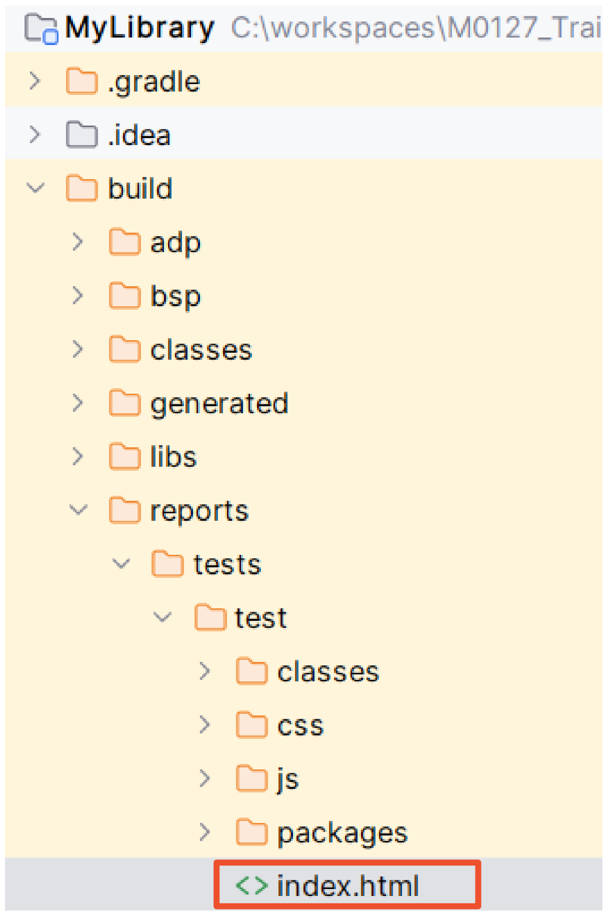


```
Tests passed: 2 of 2 tests – 4 ms

 > Test ended as success

> Task :test
BUILD SUCCESSFUL in 12s
7 actionable tasks: 3 executed, 4 up-to-date
1:54:39 PM: Execution finished 'test'.
```

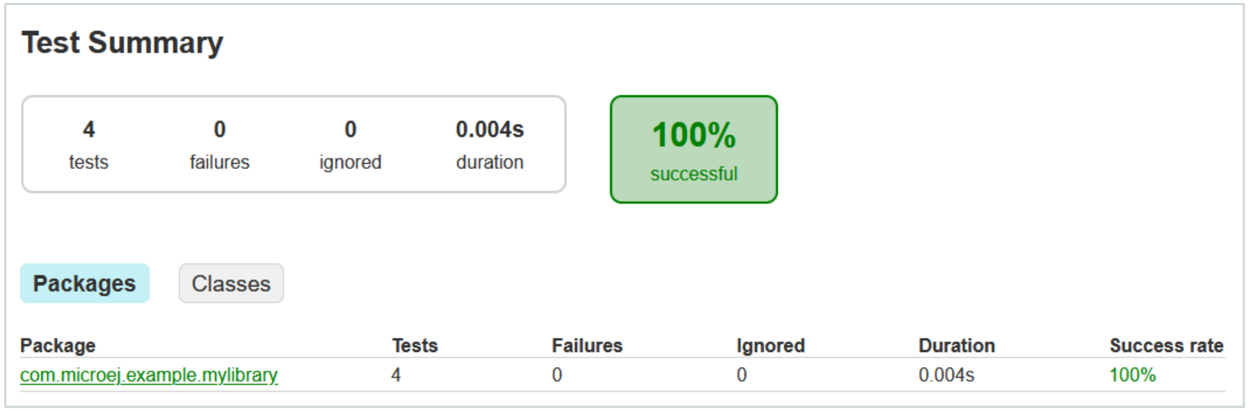
ℹ️ Running the **test** task again will not run the tests if the source code is unchanged

TESTSUITE REPORT (1/2)

Once a testsuite is completed, the JUnit HTML report is generated in the module project location **build/reports/tests/<testsuite>/index.html**:



```
MyLibrary C:\workspaces\M0127_Trai
> .gradle
> .idea
v build
  > adp
  > bsp
  > classes
  > generated
  > libs
  v reports
    v tests
      v test
        > classes
        > css
        > js
        > packages
          index.html
```

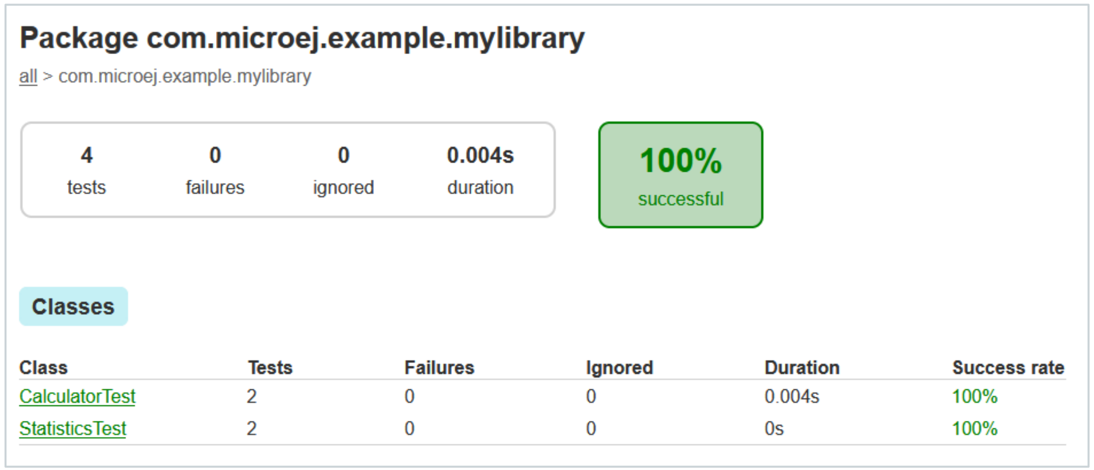


Test Summary

4	0	0	0.004s	100%
tests	failures	ignored	duration	successful

Packages | Classes

Package	Tests	Failures	Ignored	Duration	Success rate
com.microej.example.mylibrary	4	0	0	0.004s	100%



Package com.microej.example.mylibrary

all > com.microej.example.mylibrary

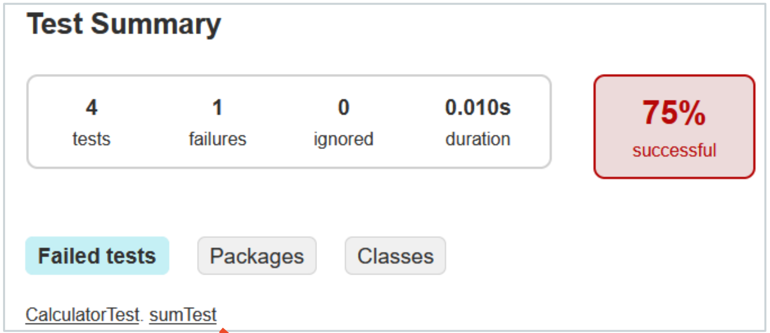
4	0	0	0.004s	100%
tests	failures	ignored	duration	successful

Classes

Class	Tests	Failures	Ignored	Duration	Success rate
CalculatorTest	2	0	0	0.004s	100%
StatisticsTest	2	0	0	0s	100%

TESTSUITE REPORT (2/2)

In case a failing test, the exception trace can be seen in the report:

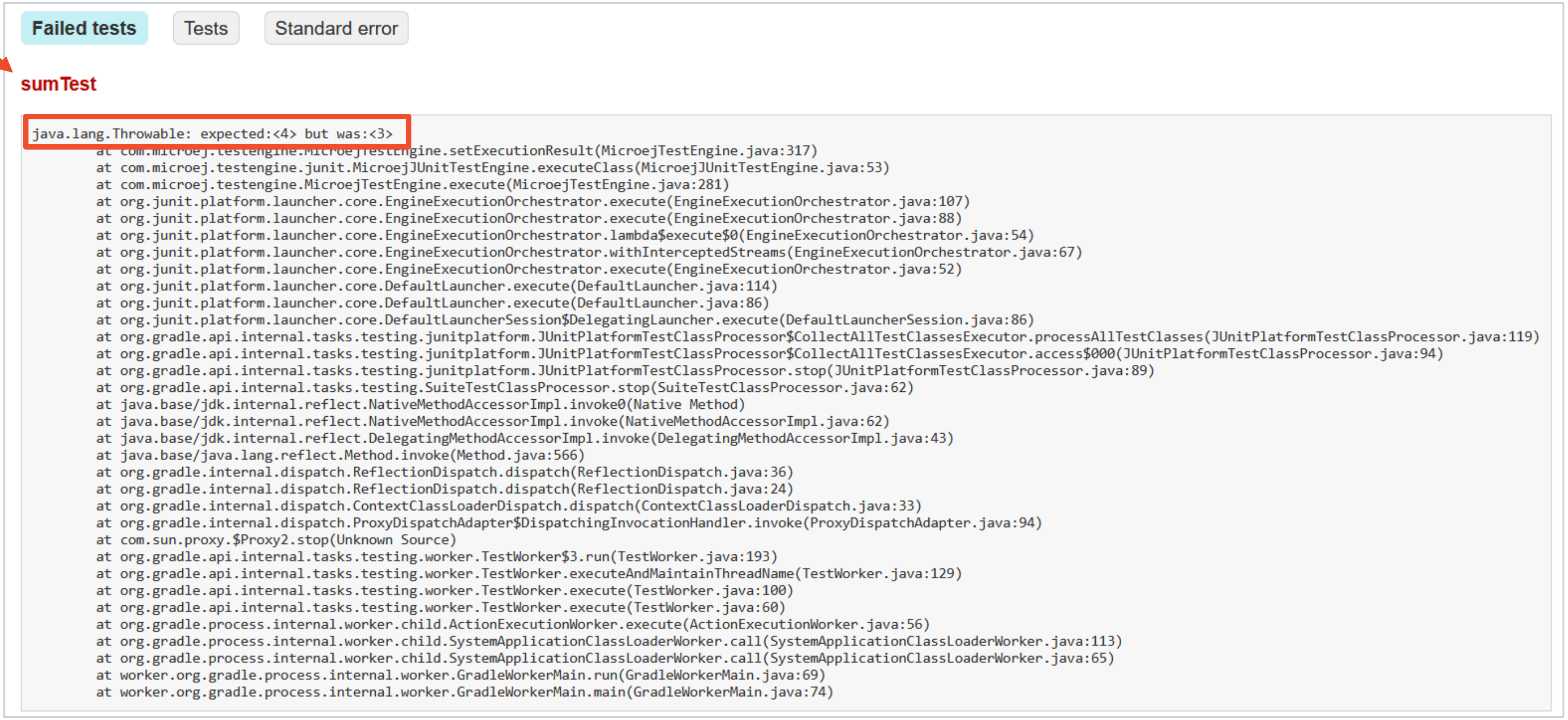


Test Summary

4	1	0	0.010s	75%
tests	failures	ignored	duration	successful

Failed tests | Packages | Classes

CalculatorTest. sumTest



Failed tests | Tests | Standard error

sumTest

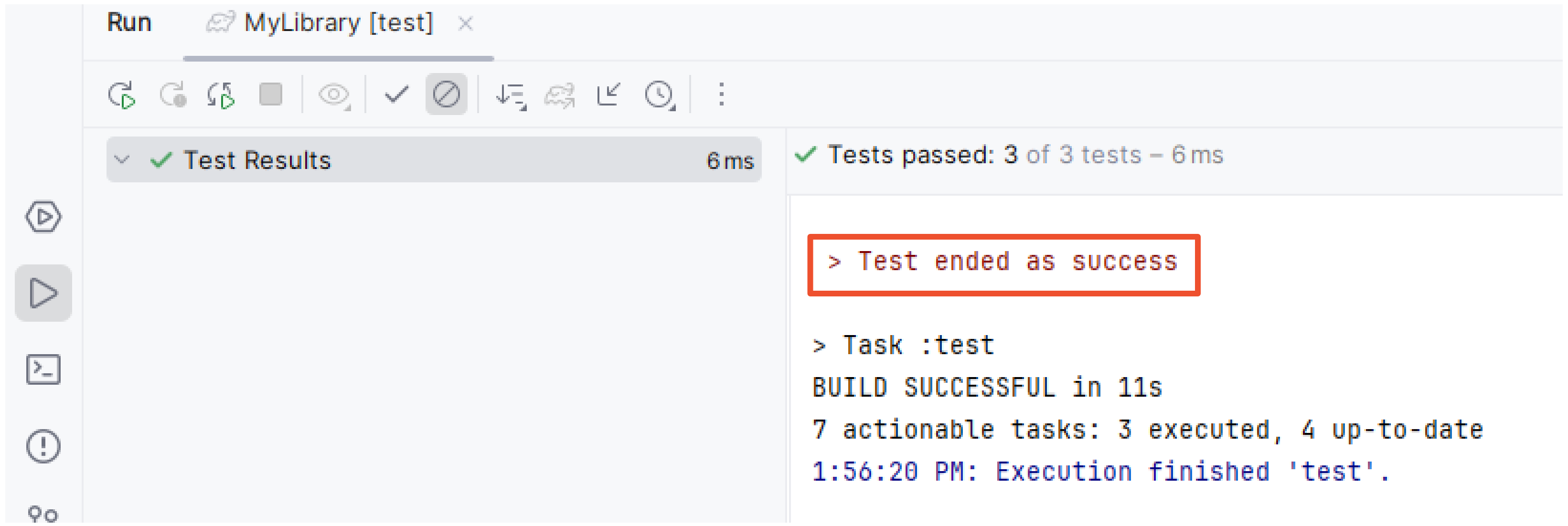
```
java.lang.Throwable: expected:<4> but was:<3>
        at com.microej.testengine.MicroejTestEngine.setExecutionResult(MicroejTestEngine.java:317)
        at com.microej.testengine.junit.MicroejJUnitTestEngine.executeClass(MicroejJUnitTestEngine.java:53)
        at com.microej.testengine.MicroejTestEngine.execute(MicroejTestEngine.java:281)
        at org.junit.platform.launcher.core.EngineExecutionOrchestrator.execute(EngineExecutionOrchestrator.java:107)
        at org.junit.platform.launcher.core.EngineExecutionOrchestrator.execute(EngineExecutionOrchestrator.java:88)
        at org.junit.platform.launcher.core.EngineExecutionOrchestrator.lambda$execute$0(EngineExecutionOrchestrator.java:54)
        at org.junit.platform.launcher.core.EngineExecutionOrchestrator.withInterceptedStreams(EngineExecutionOrchestrator.java:67)
        at org.junit.platform.launcher.core.EngineExecutionOrchestrator.execute(EngineExecutionOrchestrator.java:52)
        at org.junit.platform.launcher.core.DefaultLauncher.execute(DefaultLauncher.java:114)
        at org.junit.platform.launcher.core.DefaultLauncher.execute(DefaultLauncher.java:86)
        at org.junit.platform.launcher.core.DefaultLauncherSession$DelegatingLauncher.execute(DefaultLauncherSession.java:86)
        at org.gradle.api.internal.tasks.testing.junitplatform.JUnitPlatformTestClassProcessor$CollectAllTestClassesExecutor.processAllTestClasses(JUnitPlatformTestClassProcessor.java:119)
        at org.gradle.api.internal.tasks.testing.junitplatform.JUnitPlatformTestClassProcessor$CollectAllTestClassesExecutor.access$000(JUnitPlatformTestClassProcessor.java:94)
        at org.gradle.api.internal.tasks.testing.junitplatform.JUnitPlatformTestClassProcessor.stop(JUnitPlatformTestClassProcessor.java:89)
        at org.gradle.api.internal.tasks.testing.SuiteTestClassProcessor.stop(SuiteTestClassProcessor.java:62)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.base/java.lang.reflect.Method.invoke(Method.java:566)
        at org.gradle.internal.dispatch.ReflectionDispatch.dispatch(ReflectionDispatch.java:36)
        at org.gradle.internal.dispatch.ReflectionDispatch.dispatch(ReflectionDispatch.java:24)
        at org.gradle.internal.dispatch.ContextClassLoaderDispatch.dispatch(ContextClassLoaderDispatch.java:33)
        at org.gradle.internal.dispatch.ProxyDispatchAdapter$DispatchingInvocationHandler.invoke(ProxyDispatchAdapter.java:94)
        at com.sun.proxy.$Proxy2.stop(Unknown Source)
        at org.gradle.api.internal.tasks.testing.worker.TestWorker$3.run(TestWorker.java:193)
        at org.gradle.api.internal.tasks.testing.worker.TestWorker.executeAndMaintainThreadName(TestWorker.java:129)
        at org.gradle.api.internal.tasks.testing.worker.TestWorker.execute(TestWorker.java:100)
        at org.gradle.api.internal.tasks.testing.worker.TestWorker.execute(TestWorker.java:60)
        at org.gradle.process.internal.worker.child.ActionExecutionWorker.execute(ActionExecutionWorker.java:56)
        at org.gradle.process.internal.worker.child.SystemApplicationClassLoaderWorker.call(SystemApplicationClassLoaderWorker.java:113)
        at org.gradle.process.internal.worker.child.SystemApplicationClassLoaderWorker.call(SystemApplicationClassLoaderWorker.java:65)
        at worker.org.gradle.process.internal.worker.GradleWorkerMain.run(GradleWorkerMain.java:69)
        at worker.org.gradle.process.internal.worker.GradleWorkerMain.main(GradleWorkerMain.java:74)
```

EXCEPTION HANDLING IN A TEST

- Add a test that should throw an exception, e.g. a divide by zero:

```
@Test(expected = ArithmeticException.class)
public void testDivideByZero() {
   assertEquals(new Calculator(1, 0).divide(), 3); // 1/0 is invalid
}
```

- Run the **test** task.
- The test ran successfully:



```
> Test ended as success

> Task :test
BUILD SUCCESSFUL in 11s
7 actionable tasks: 3 executed, 4 up-to-date
1:56:20 PM: Execution finished 'test'.
```

Generating the Code Coverage Report

ENABLE CODE COVERAGE ANALYSIS

The Code Coverage analysis allows to:

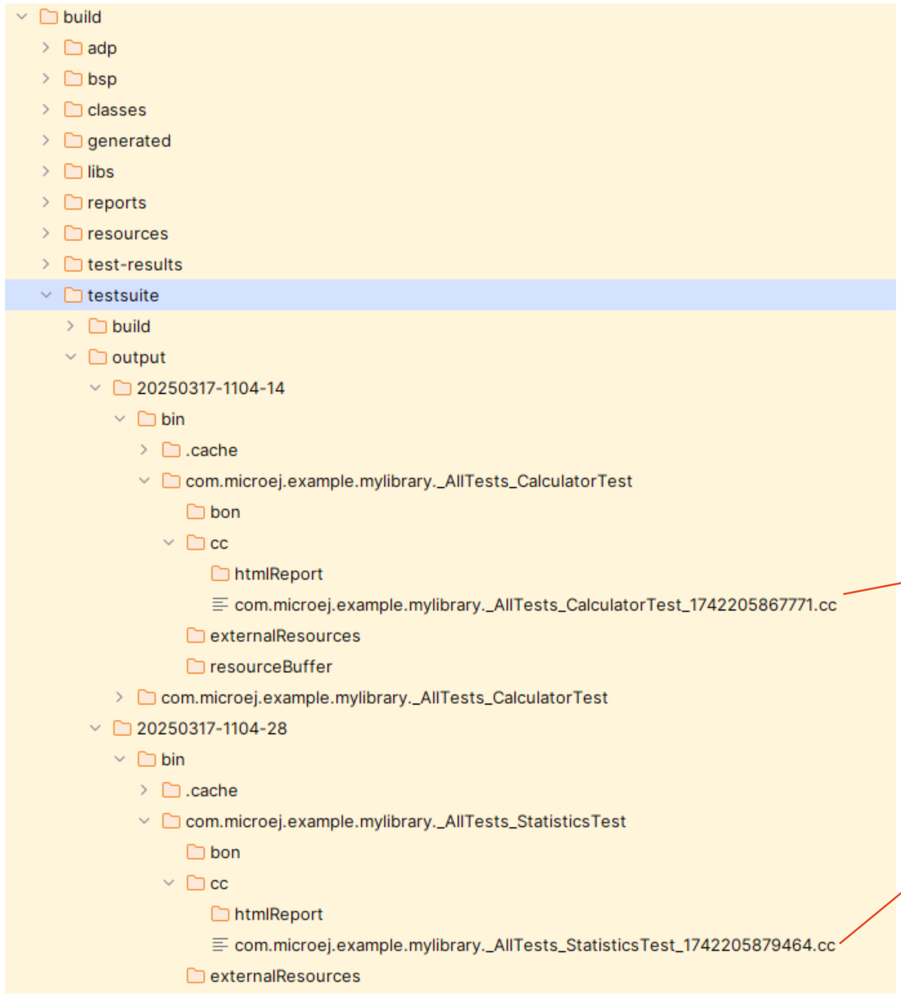
- List used and unused source code.
- Find untested or dead code.
- HTML report generation.

To generate the Code Coverage files (**.cc**) for each test, update the **build.gradle.kts** file as follows:

```
testing {
  suites {
    val test by getting(JvmTestSuite::class) {
      microej.useMicroejTestEngine(this)

      targets {
        all {
          testTask.configure {
            doFirst {
              systemProperties["microej.testsuite.properties.s3.cc.activated"] = "true"
              systemProperties["microej.testsuite.properties.s3.cc.thread.period"] = "15"
            }
          }
        }
      }

      dependencies {
        implementation(project())
        implementation("ej.api:edc:1.3.5")
        implementation("ej.library.test:junit:1.10.0")
        implementation("org.junit.platform:junit-platform-launcher:1.8.2")
      }
    }
  }
}
```

GENERATE THE CODE COVERAGE FILES

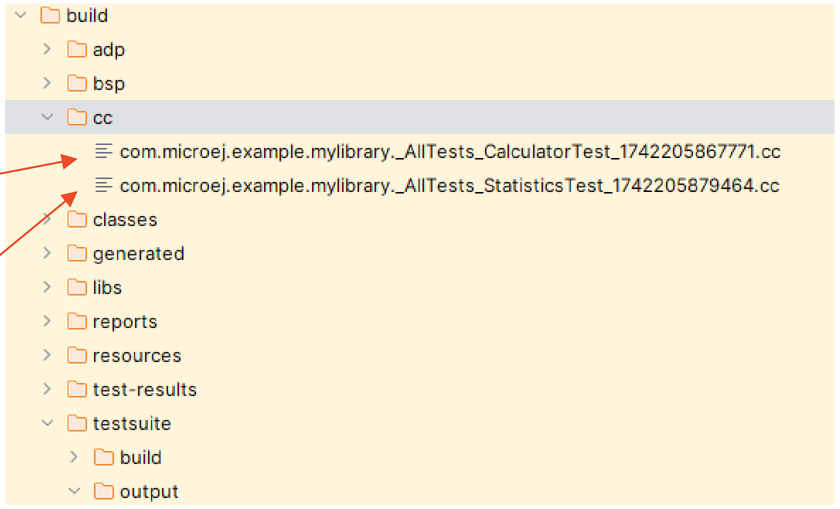
Run the **test** task.
The testsuite engine generates Code Coverage files (.cc) for each test class.



```
build
  adp
  bsp
  classes
  generated
  libs
  reports
  resources
  test-results
  testsuite
    build
    output
      20250317-1104-14
        bin
          .cache
          com.microej.example.mylibrary._AllTests_CalculatorTest
            bon
            cc
              htmlReport
              com.microej.example.mylibrary._AllTests_CalculatorTest_1742205867771.cc
            externalResources
            resourceBuffer
        com.microej.example.mylibrary._AllTests_CalculatorTest
      20250317-1104-28
        bin
          .cache
          com.microej.example.mylibrary._AllTests_StatisticsTest
            bon
            cc
              htmlReport
              com.microej.example.mylibrary._AllTests_StatisticsTest_1742205879464.cc
            externalResources
```

To generate the HTML Code Coverage Report:

- Create a new **cc/** folder in the **build/** folder,
- Copy each **.cc** file in the **build/cc/** folder.

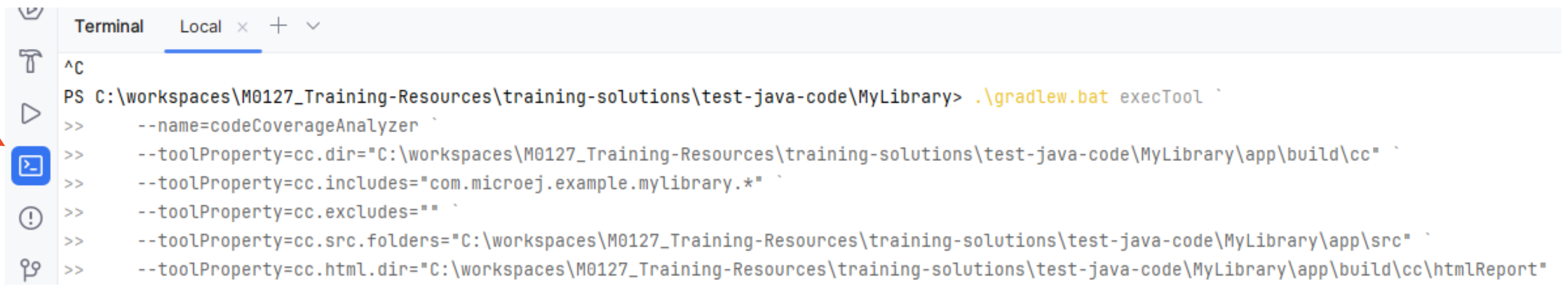


```
build
  adp
  bsp
  cc
    com.microej.example.mylibrary._AllTests_CalculatorTest_1742205867771.cc
    com.microej.example.mylibrary._AllTests_StatisticsTest_1742205879464.cc
  classes
  generated
  libs
  reports
  resources
  test-results
  testsuite
    build
    output
```

GENERATE THE CODE COVERAGE REPORT

Run the following command in the Terminal of the IDE (PowerShell):

```
.\gradlew.bat execTool `
    --name=codeCoverageAnalyzer `
    --toolProperty=cc.dir="C:\PATH_TO_PROJECT\MyLibrary\build\cc" `
    --toolProperty=cc.includes="com.microej.example.mylibrary.*" `
    --toolProperty=cc.excludes="com.microej.example.mylibrary.tests.*" `
    --toolProperty=cc.src.folders="C:\PATH_TO_PROJECT\MyLibrary\src" `
    --toolProperty=cc.html.dir="C:\PATH_TO_PROJEC\MyLibrary\build\cc\htmlReport"
```

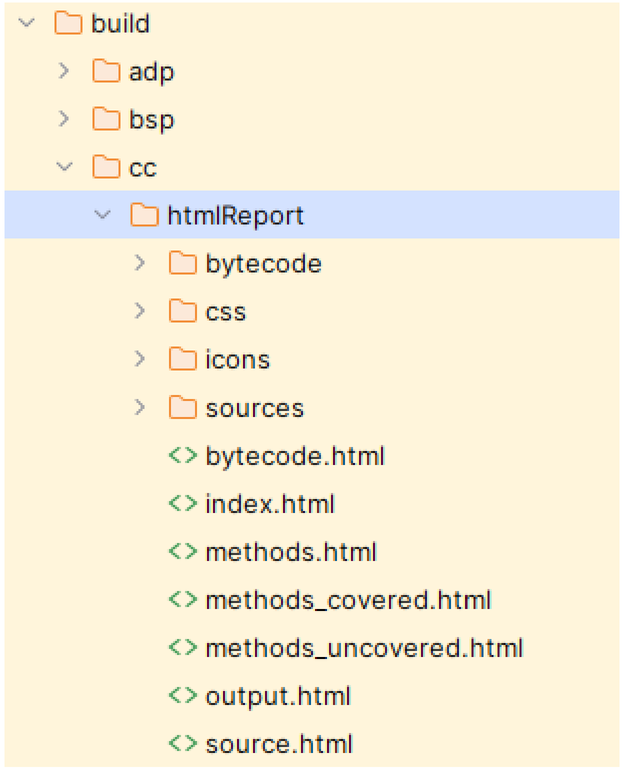


```
^C
PS C:\workspaces\M0127_Training-Resources\training-solutions\test-java-code\MyLibrary> .\gradlew.bat execTool `
>>     --name=codeCoverageAnalyzer `
>>     --toolProperty=cc.dir="C:\workspaces\M0127_Training-Resources\training-solutions\test-java-code\MyLibrary\app\build\cc" `
>>     --toolProperty=cc.includes="com.microej.example.mylibrary.*" `
>>     --toolProperty=cc.excludes="" `
>>     --toolProperty=cc.src.folders="C:\workspaces\M0127_Training-Resources\training-solutions\test-java-code\MyLibrary\app\src" `
>>     --toolProperty=cc.html.dir="C:\workspaces\M0127_Training-Resources\training-solutions\test-java-code\MyLibrary\app\build\cc\htmlReport"
```

Refer to the Code Coverage Analyzer documentation for more information about this tool.

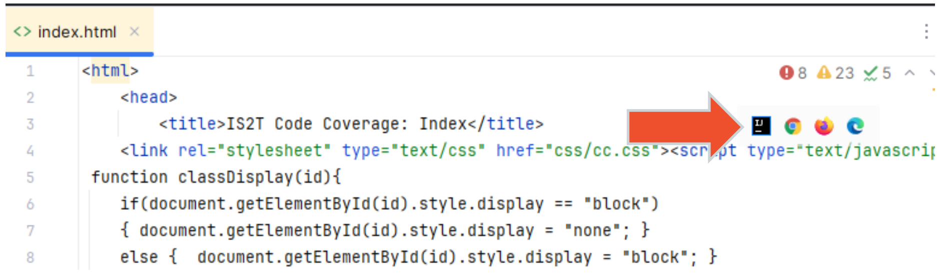
VISUALIZE THE CODE COVERAGE REPORT

- The HTML report is available in the **build/cc/htmlReport** folder:



```
build
  adp
  bsp
  cc
    htmlReport
      bytecode
      css
      icons
      sources
      bytecode.html
      index.html
      methods.html
      methods_covered.html
      methods_uncovered.html
      output.html
      source.html
```

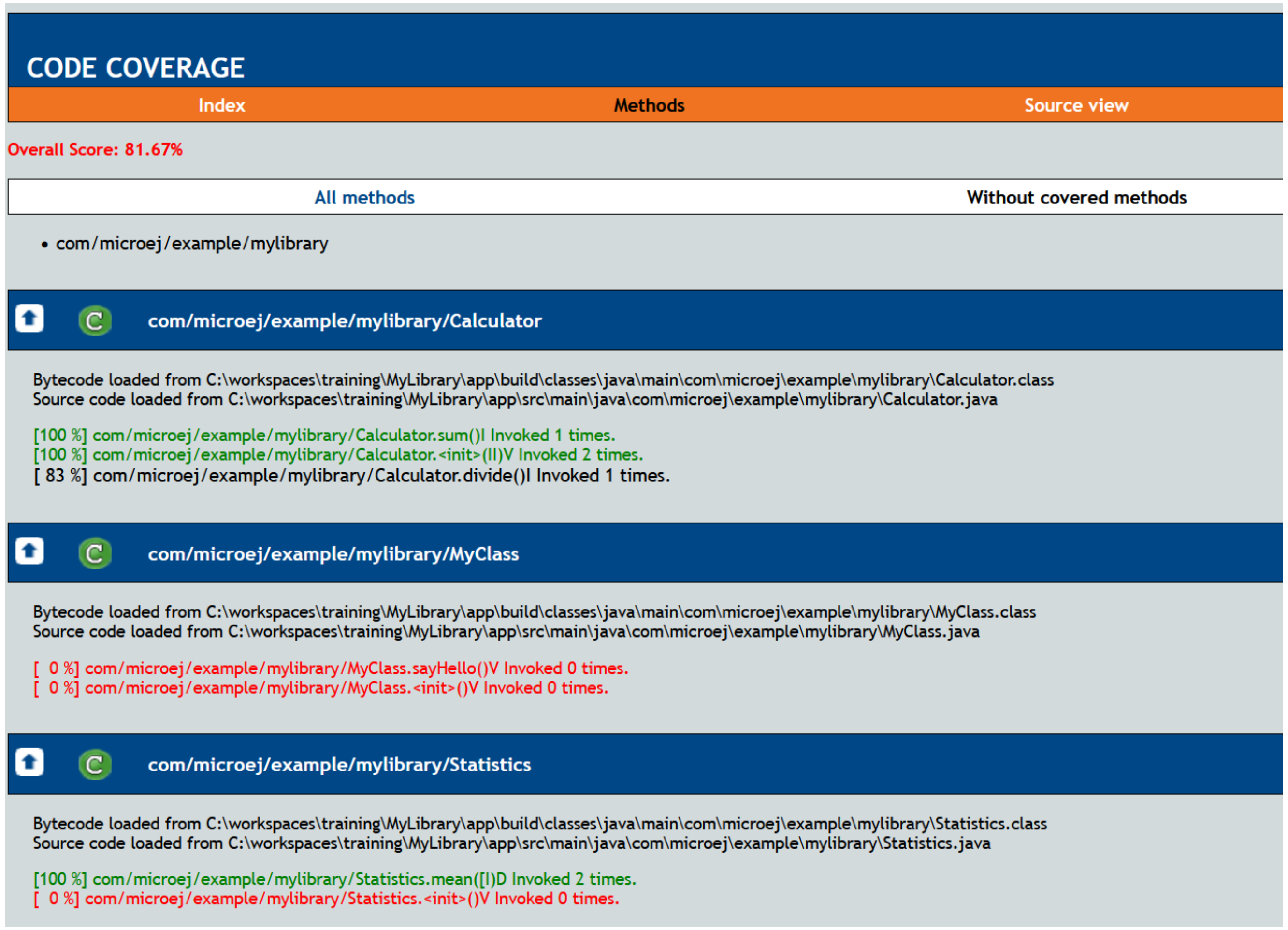
- Open the **index.html** file from your IDE
- Select the desired web browser to open the report:



```
<html>
    <head>
        <title>IS2T Code Coverage: Index</title>
    <link rel="stylesheet" type="text/css" href="css/cc.css"><scr pt type="text/javascrip
 function classDisplay(id){
    if(document.getElementById(id).style.display == "block")
    { document.getElementById(id).style.display = "none"; }
    else {  document.getElementById(id).style.display = "block"; }
```

CODE COVERAGE REPORT ANALYSIS

RESULTS



CODE COVERAGE

Index | Methods | Source view

Overall Score: 81.67%

All methods | Without covered methods

- com/microej/example/mylibrary

com/microej/example/mylibrary/Calculator

```
Bytecode loaded from C:\workspaces\training\MyLibrary\app\build\classes\java\main\com\microej\example\mylibrary\Calculator.class
Source code loaded from C:\workspaces\training\MyLibrary\app\src\main\java\com\microej\example\mylibrary\Calculator.java

[100 %] com/microej/example/mylibrary/Calculator.sum()I Invoked 1 times.
[100 %] com/microej/example/mylibrary/Calculator.<init>(II)V Invoked 2 times.
[ 83 %] com/microej/example/mylibrary/Calculator.divide()I Invoked 1 times.
```

com/microej/example/mylibrary/MyClass

```
Bytecode loaded from C:\workspaces\training\MyLibrary\app\build\classes\java\main\com\microej\example\mylibrary\MyClass.class
Source code loaded from C:\workspaces\training\MyLibrary\app\src\main\java\com\microej\example\mylibrary\MyClass.java

[  0 %] com/microej/example/mylibrary/MyClass.sayHello()V Invoked 0 times.
[  0 %] com/microej/example/mylibrary/MyClass.<init>()V Invoked 0 times.
```

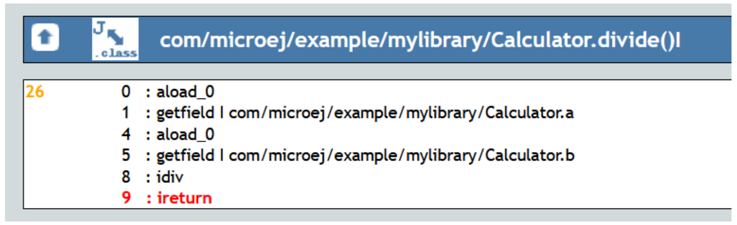
com/microej/example/mylibrary/Statistics

```
Bytecode loaded from C:\workspaces\training\MyLibrary\app\build\classes\java\main\com\microej\example\mylibrary\Statistics.class
Source code loaded from C:\workspaces\training\MyLibrary\app\src\main\java\com\microej\example\mylibrary\Statistics.java

[100 %] com/microej/example/mylibrary/Statistics.mean([I)D Invoked 2 times.
[  0 %] com/microej/example/mylibrary/Statistics.<init>()V Invoked 0 times.
```

ANALYSIS

Poor code coverage can be seen in the following cases:

- **Calculator** class:
  - The **divide()** method didn't return properly during the test (it threw an exception) Implement an other test to fully cover this method. See **ByteCode** view:



com/microej/example/mylibrary/Calculator.divide()I

```
26    0  : aload_0
      1  : getfield I com/microej/example/mylibrary/Calculator.a
      4  : aload_0
      5  : getfield I com/microej/example/mylibrary/Calculator.b
      8  : idiv
      9  : ireturn
```

- **MyClass** class: no tests have been implemented for this class. Either implement the tests or exclude the class from the report generation.
- **Statistics** class: private constructors can't be excluded from the code coverage analysis.

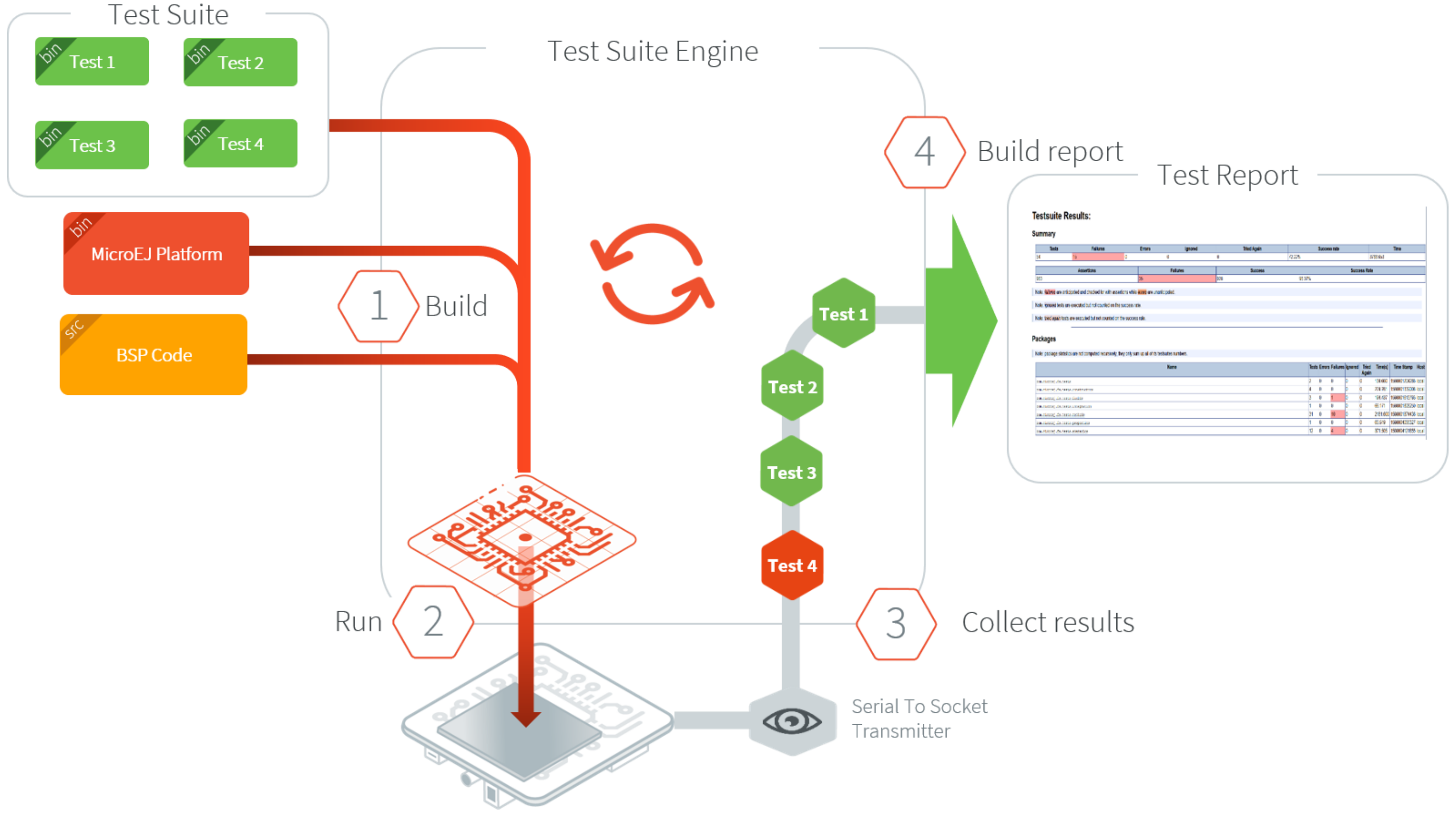
Running Tests on Device

TESTSUITE ON DEVICE OVERVIEW

The SDK allows to execute a testsuite on a device. This requires to:

- Have a VEE Port which implements the BSP Connection.
- Have a device connected to your workstation both for programming the Executable and getting the output traces. Consult your VEE Port specific documentation for setup.
- Start the Serial to Socket Transmitter tool if the VEE Port does not redirect execution traces.

The execution of the tests on the target is handled automatically by the Testsuite Engine (see schematic).



Testsuite on Device Overview

TESTSUITE CONFIGURATION (1/2)

The configuration is similar to the one used to execute a testsuite on the Simulator.
Update the configuration as follows in **build.gradle.kts:**

- Replace the line:
  microej.useMicroejTestEngine(this) by **microej.useMicroejTestEngine(this, TestTarget.EMB)**
- Add the **import** statement at the beginning of the file:
  - **import com.microej.gradle.plugins.TestTarget**

TESTSUITE CONFIGURATION (2/2)

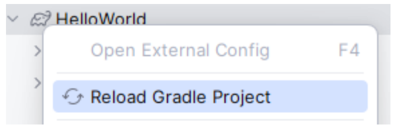
- Add the required properties as follows:

```
testing {
   suites {
      val test by getting(JvmTestSuite::class) {
         microej.useMicroejTestEngine(this, TestTarget.EMB)

         targets {
            all {
               testTask.configure {
                  doFirst {
                     systemProperties["microej.testsuite.properties.s3.cc.activated"] = "true"
                     systemProperties["microej.testsuite.properties.s3.cc.thread.period"] = "15"

                     systemProperties = mapOf(
                        // Enable the build of the Executable
                        "microej.testsuite.properties.deploy.bsp.microejscript" to "true",
                        "microej.testsuite.properties.microejtool.deploy.name" to "deployToolBSPRun",
                        // Tell the testsuite engine that the VEE Port Run script redirects execution traces
                        // Configure the TCP/IP address and port if the VEE Port Run script
                  does not redirect execution traces
                        "microej.testsuite.properties.testsuite.trace.ip" to "localhost",
                        "microej.testsuite.properties.testsuite.trace.port" to "5555"
                     )
                  …
```

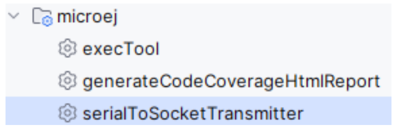
SERIAL TO SOCKET TRANSMITTER TOOL

The Serial to Socket Transmitter tool needs to be setup to redirect the execution traces:

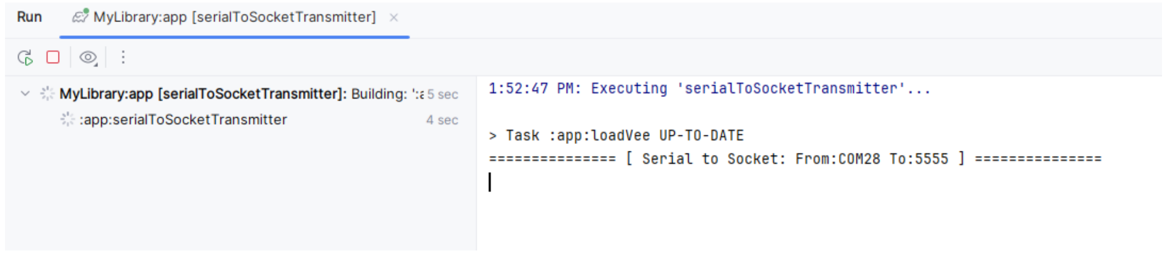
- Copy/Paste the following code in the **build.gradle.kts** file of your **MyLibrary** project.
    - Note: to ease the copy/paste, the code is also available in the training package provided with those slides
- Click on Reload Gradle Project:



- Once done, the tool appears in the **microej** tasks view, run it:



The tool starts in the console:


Run
MyLibrary:app [serialToSocketTransmitter]
MyLibrary:app [serialToSocketTransmitter]: Building: ': 5 sec
:app:serialToSocketTransmitter 4 sec

```
1:52:47 PM: Executing 'serialToSocketTransmitter'...

> Task :app:loadVee UP-TO-DATE
=============== [ Serial to Socket: From:COM28 To:5555 ] ===============
```

```
import com.microej.gradle.tasks.ExecToolTask
import com.microej.gradle.tasks.LoadVeeTask

val loadVee = tasks.withType(LoadVeeTask::class).named("loadTestVee")

tasks.register<ExecToolTask>("serialToSocketTransmitter") {
    group = "microej"
    applicationEntryPoint.set(microej.applicationEntryPoint)
    veeDir.set(loadVee.get().loadedVeeDir)
    toolName = "serialToSocketTransmitter"
    toolProperties.putAll(mapOf(
        "serail.to.socket.comm.port" to "COM28",
        "serail.to.socket.comm.baudrate" to "115200",
        "serail.to.socket.server.port" to "5555"
    ))
}
```

Add on top of the build.gradle.kts file

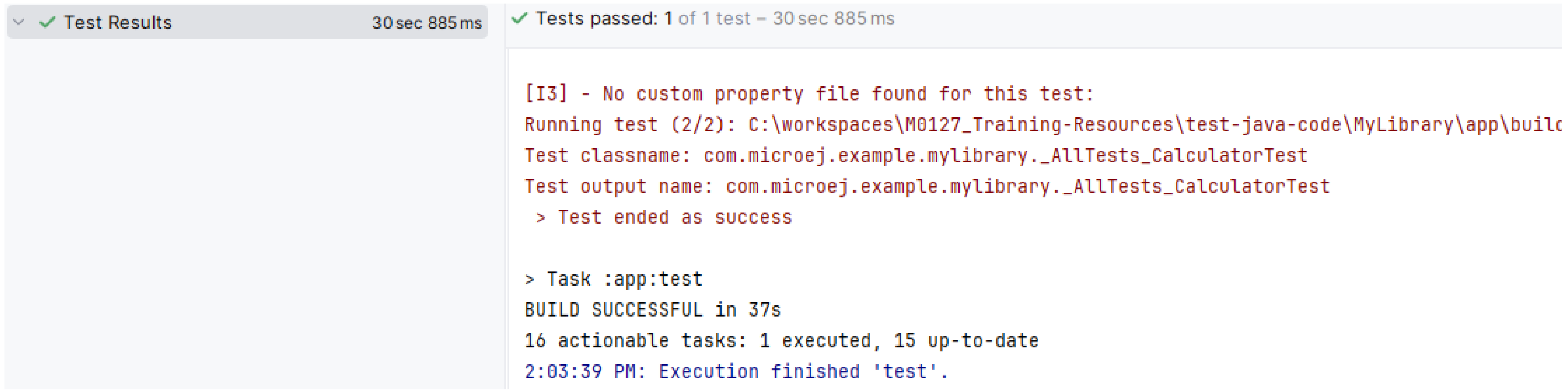
Select the COM Port of your device

Refer to the Serial To Socket Transmitter documentation to run the tool from command line.

RUN THE TESTS ON DEVICE

Run the tests on device:

- Run the **test** task.
- Tests are executed on the target.



Test Results 30 sec 885 ms
Tests passed: 1 of 1 test – 30 sec 885 ms

```
[I3] - No custom property file found for this test:
Running test (2/2): C:\workspaces\M0127_Training-Resources\test-java-code\MyLibrary\app\builc
Test classname: com.microej.example.mylibrary._AllTests_CalculatorTest
Test output name: com.microej.example.mylibrary._AllTests_CalculatorTest
 > Test ended as success

> Task :app:test
BUILD SUCCESSFUL in 37s
16 actionable tasks: 1 executed, 15 up-to-date
2:03:39 PM: Execution finished 'test'.
```

Test results can be checked in **build/reports/tests/test/index.html.**

Advanced Configurations

FILTER THE TESTS (1/2)

Gradle automatically executes all the tests located in the test source folder. If you want to execute only a subset of these tests, Gradle provides 2 solutions:

- Filtering configuration in the build script file.
- Filtering option in the command line.

To filter the tests in **build.gradle.kts**, add the following code

```
targets {
  all {
    testTask.configure {
      doFirst {
        systemProperties["microej.testsuite.properties.s3.cc.activated"] = "true"
        systemProperties["microej.testsuite.properties.s3.cc.thread.period"] = "15"

        filter {
          includeTestsMatching("StatisticsTest")
        }
```

In that case, only the **StatisticsTest** class will be executed.

Wildcard can be used to select a subset of tests (e.g. **com.microej.example.***)
Other methods are available for test filtering, such as **excludeTestsMatching** to exclude tests.

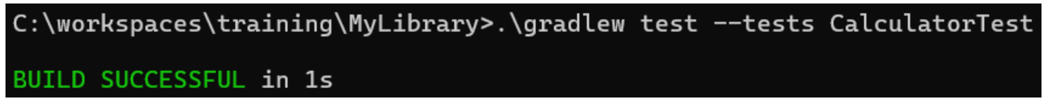
Refer to the Filter the Tests documentation for more information.

FILTER THE TESTS (2/2)

Gradle allows to filter the tests from the command line directly, thanks to the **--tests** option.

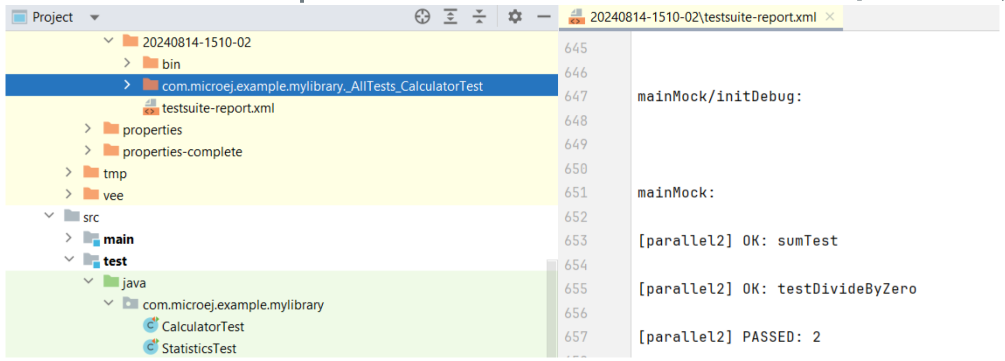
This can be convenient to quickly execute one test for example, without requiring a change in the build script file:

- Open a Command Prompt in the **MyLibrary** folder.
- Run the following command to run the **CalculatorTest**:
  - `.\gradlew.bat test --tests CalculatorTest`



```
C:\workspaces\training\MyLibrary>.\gradlew test --tests CalculatorTest

BUILD SUCCESSFUL in 1s
```

- The testsuite report is available in the **build/** folder, only **CalculatorTest** has been executed:



```
mainMock/initDebug:

mainMock:
[parallel2] OK: sumTest
[parallel2] OK: testDivideByZero
[parallel2] PASSED: 2
```

Note: the test class must not be excluded in the build script file, otherwise the test will fail.

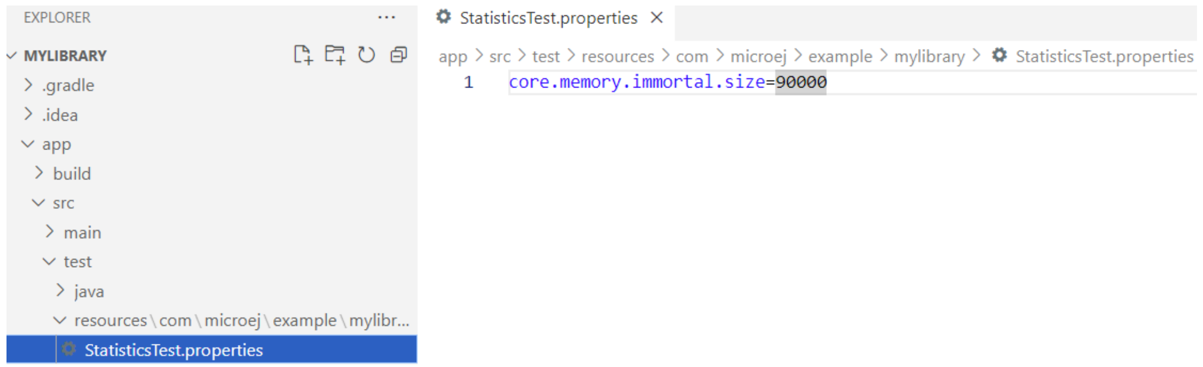
INJECT APPLICATION OPTIONS

Standalone Application Options can be defined to configure the Application or Library being tested. They can be defined globally, to be applied on all tests, or specifically to a test.

- Inject Application Options Globally: it must be prefixed by **microej.testsuite.properties.** and passed as a System Property, either in the command line or in the build script file.
  For example, to inject the property **core.memory.immortal.size**:
  - In the command line with **-D**:
    `.\gradlew.bat test -Dmicroej.testsuite.properties.core.memory.immortal.size=8192`
  - In the build script file:

```
targets {
    all {
        testTask.configure {
            ...

            doFirst {
                systemProperties = mapOf(
                    "microej.testsuite.properties.core.memory.immortal.size" to "8192"
                )
            }
        }
    }
}
```

- Inject Application Options For a Specific Test:
  - Add a **.properties** file in the **src/test/resources** folder with the same name as the generated test case file and within the same package than the test file.



```
EXPLORER
MYLIBRARY
  > .gradle
  > .idea
  v app
    > build
    v src
      > main
      v test
        > java
        v resources \ com \ microej \ example \ mylibr...
            StatisticsTest.properties
```

StatisticsTest.properties

app > src > test > resources > com > microej > example > mylibrary > StatisticsTest.properties

```
core.memory.immortal.size=90000
```

GOING FURTHER...

Visit the Test a Project documentation to learn more about:

- Running tests on a J2SE VM (useful when the usage of mock libraries like Mockito is needed).
- Mixing tests:
  - Mixing tests on the Simulator and on a device.
  - Mixing tests on the Simulator and on a J2SE VM.
- Advanced Configuration for the Testsuite engine.

THANK YOU

for your attention !

MICROEJ®