Testing Java Code

With MICROEJ SDK 6

© MicroEJ 2024

MICROEJ®

MICROEJ®

DISCLAIMER

All rights reserved. Information, technical data and tutorials contained in this document are proprietary under copyright law of MicroEJ S.A. Without written permission from MicroEJ S.A., copying or sending parts of the document or the entire document by any means to third parties is not permitted. Granted authorizations for using parts of the document or the entire document do not mean MicroEJ S.A. gives public full access rights.

The information contained herein is not warranted to be error-free.

MicroEJ® and all relative logos are trademarks or registered trademarks of MicroEJ S.A. in France and other Countries.

Other trademarks are proprietary of their respective owners.

Java™ is Sun Microsystems' trademark for a technology for developing application software and deploying it in cross-platform, networked environments. When it is used in this site without adding the "™" symbol, it includes implementations of the technology by companies other than Sun. Java™, all Java-based marks and all related logos are trademarks or registered trademarks of Sun Microsystems Inc, in the United States and other Countries.

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

© MICROEJ 2024

JUnit Academic

© MICROEJ 2024

JUNIT DEFINITION

MICROEJ

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

© MICROEJ 2024

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”.

© MICROEJ 2024

ANNOTATIONS (2/2)

MICROEJ

- `@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

© MICROEJ 2024

JUNIT EXAMPLE

MICROEJ

```
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.

© MICROEJ 2024

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.

© MICROEJ 2024

GOOD PRACTICES

MICROEJ

- 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.

© MICROEJ 2024

Hand-On

Create and Run tests in an Add-On Library project

PREREQUISITES

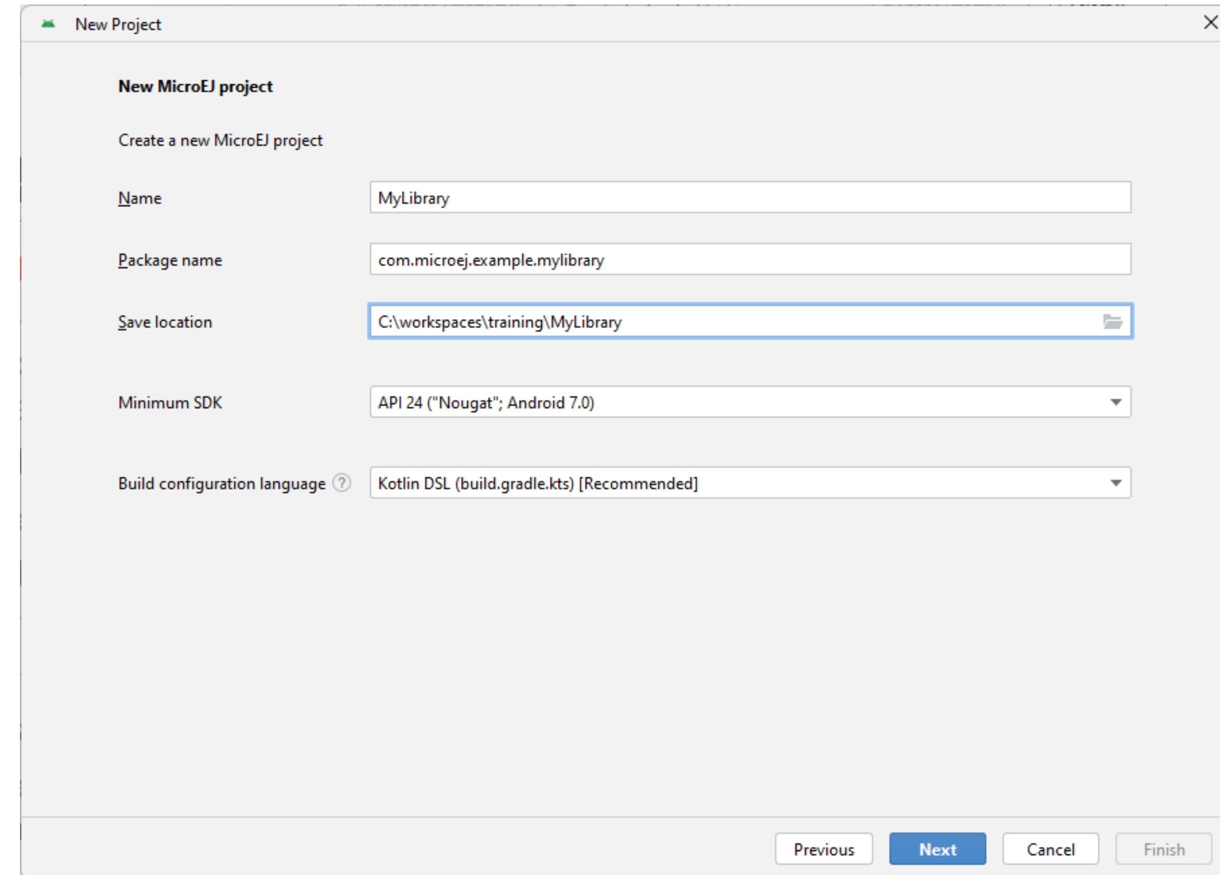
- MICROEJ SDK 6 (https://docs.microej.com/en/latest/SDK6UserGuide/install.html)
- NXP i.MX RT1170 VEE Port (https://github.com/MicroEJ/nxp-vee-imxrt1170-evk/tree/main)

- This training has been tested on:
  - Android Studio IDE with MICROEJ SDK plug-in 0.6.0.
  - NXP i.MX RT1170 VEE Port 2.2.0.

CREATE AN ADD-ON LIBRARY PROJECT (1/2)

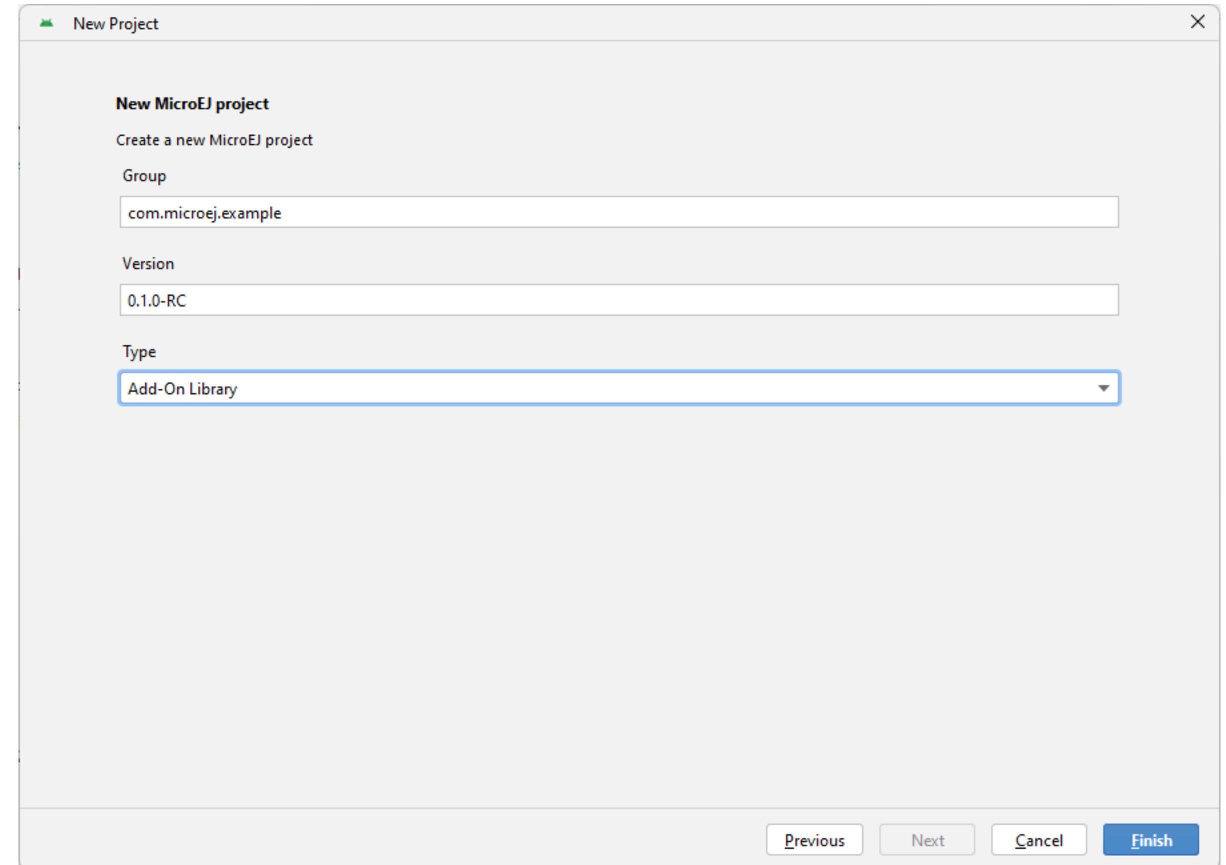
The creation of a project with Android Studio is done as follows:

- Click on **File > New > Project....**
- Select **Generic > New MicroEJ project.**
- Click on the **Next** button.
- Fill the name of the project in the **Name** field.
- Fill the package name of the project in the **Package** name field.
- Select the location of the project in the **Save location** field.
- Keep the default Android SDK in the **Minimum SDK** field.
- Select **Kotlin** for the **Build configuration language** field.



CREATE AN ADD-ON LIBRARY PROJECT (1/2)

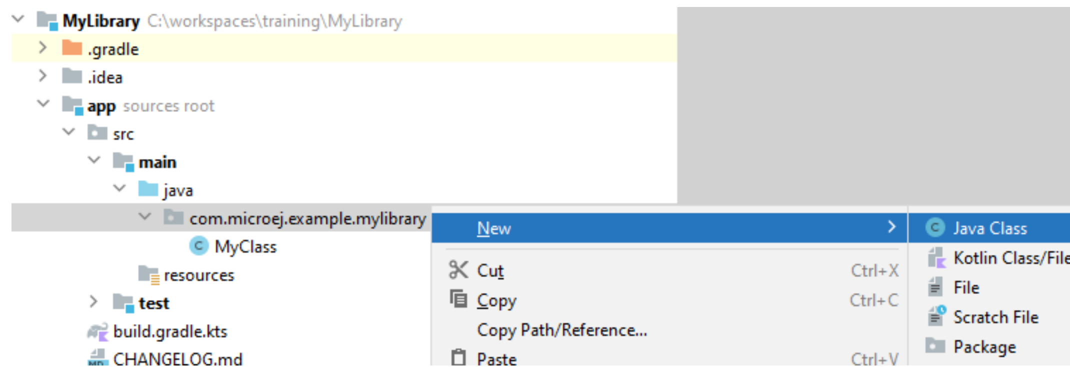
- Click on **Next** button.
- Fill the group of the artifact to publish in the **Group** field.
- Fill the version of the artifact to publish in the **Version** field.
- Select the **Addon-Library** module type among in the drop-down list.
- Click on **Finish** button.



© MICROEJ 2024

ADD CLASSES TO THE PROJECT (1/2)

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



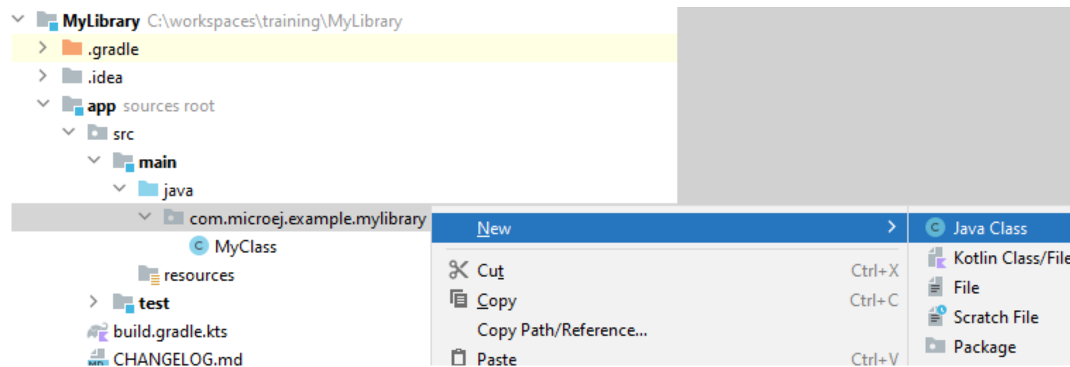
- Create the **Calculator** class.

- 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**:



- 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;
    }
}
```

© MICROEJ 2024

Running Tests On Simulator

© MICROEJ 2024

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.

© MICROEJ 2024

TESTSUITE CONFIGURATION

MICROEJ

The configuration of the testsuite of a project must be 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:1.3.5")
        implementation("ej.library.test:junit:1.10.0")
        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.

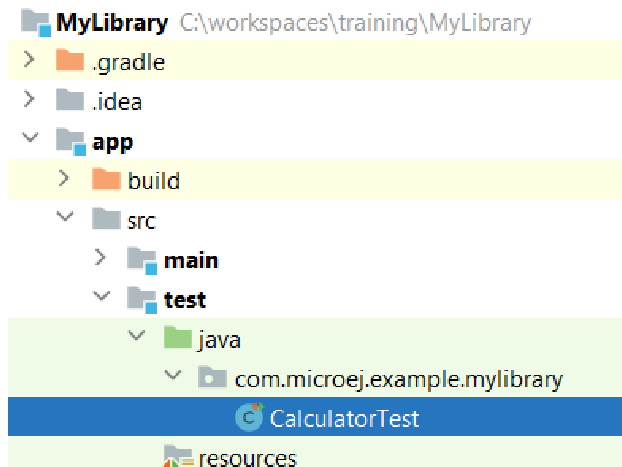
Note: the testsuite is already configured when creating an Add-On library project.

ADD CALCULATOR TEST CLASS

MICROEJ

CREATE THE TEST CLASS

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



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());
```

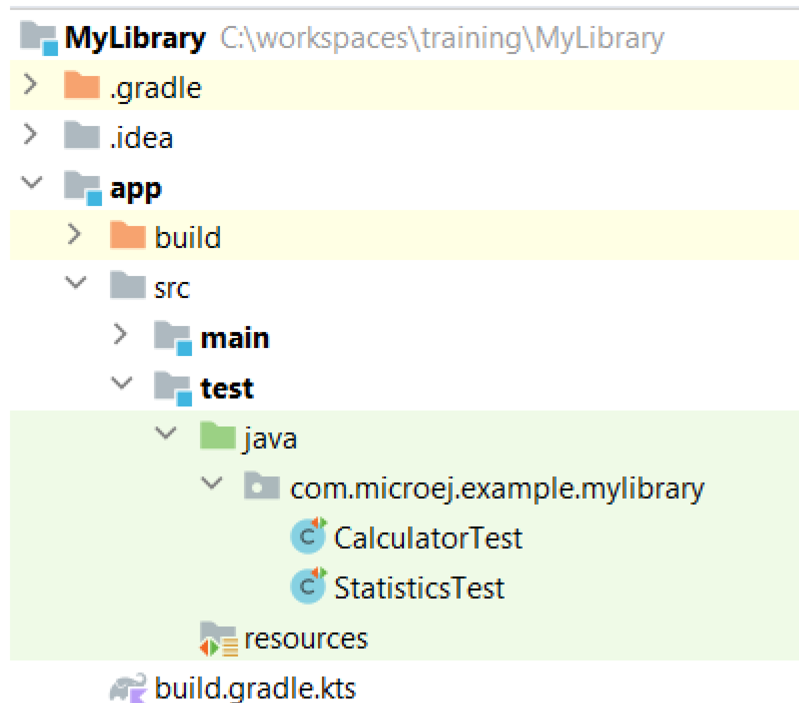
- The **CalculatorTest** class should look like that:

```
CalculatorTest.java
1   /.../
5   package com.microej.example.mylibrary;
6
7   import ...
12
13  public class CalculatorTest {
14      @Test
15      public void sumTest() {
16          assertEquals( expected: 3, new Calculator( a: 1,  b: 2).sum());
17      }
```

ADD STATISTICS TEST CLASS

CREATE THE TEST CLASS

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



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.

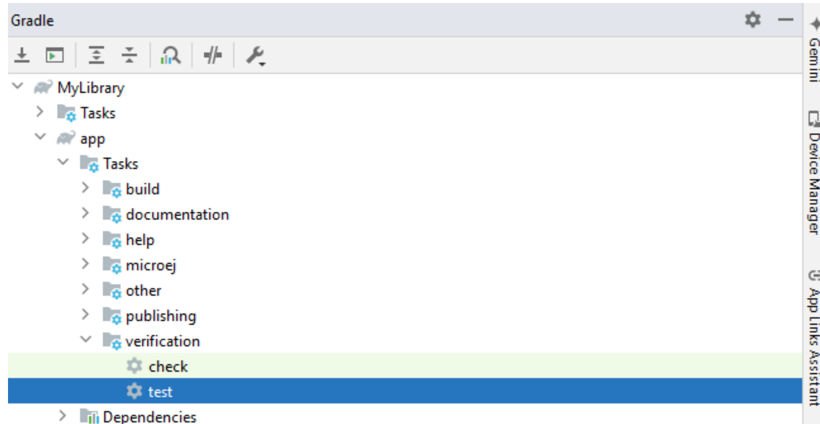
In **build.gradle.kts**, add the **NXP i.MX RT1170 VEE Port 2.2.0** (or later) in the **dependencies** section:

```
dependencies {
   implementation("ej.api:edc:1.3.5")
   implementation("ej.library.test:junit:1.10.0")

   //Uncomment the microejVee dependency to set the VEE Port or Kernel to use
   microejVee("com.nxp.vee.mimxrt1170:evk_platform:2.2.0")
}
```

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:



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

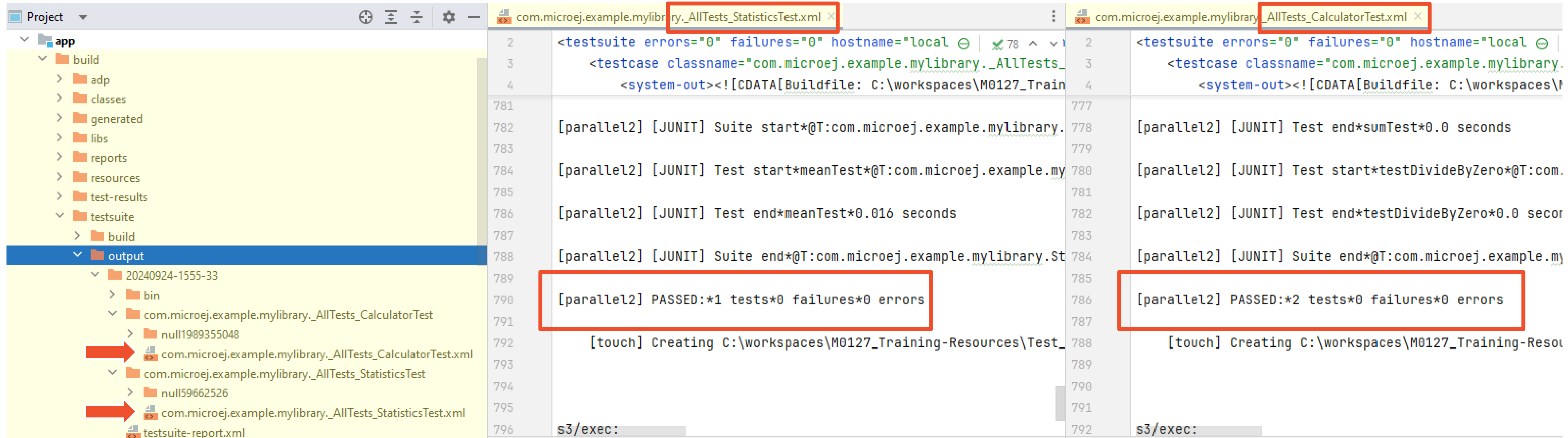
```
✔ Tests passed: 1 of 1 test – 3 sec 769 ms
C:\workspaces\training\MyLibrary\app\build\testsuite\output\20240813-1230-51\testsuite-report.xml
[I3] - No custom property file found for this test:
Running test (1/1): C:\workspaces\training\MyLibrary\app\build\classes\java\test\CalculatorTest.class
Test classname: _AllTests_CalculatorTest
Test output name: _AllTests_CalculatorTest
 > Test ended as success

BUILD SUCCESSFUL in 39s
6 actionable tasks: 6 executed
```

© MICROEJ 2024

TESTSUITE REPORT (1/2)

By default, each test class is executed in a dedicated instance of the Simulator.

A dedicated testsuite report is generated per test class (XML format):



Those reports are aggregated in a single testsuite report available in XML format in the following folder:

build/testsuite/output/YYYYMMDD-HHMM-SS/testsuite-report.xml

TESTSUITE REPORT (2/2)

MICROEJ

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

```
     [echo] =============== [ Launching on Simulator ] ===============

s3/exec/impl:

sleep.onWinXP:

mainMock/initDebug:

mainMock:
[parallel2] KO: sumTest Assertion failed: expected:<3> but was:<4>
[parallel2] Exception in thread "main" @T:java.lang.AssertionError@: expected:<3> but was:<4>
[parallel2] *at java.lang.Throwable.fillInStackTrace(Throwable.java:82)
[parallel2] *at java.lang.Throwable.<init>(Throwable.java:37)
[parallel2] *at java.lang.Error.<init>(Error.java:18)
```

© MICROEJ 2024

EXCEPTION HANDLING IN A TEST

MICROEJ

- 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:

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.

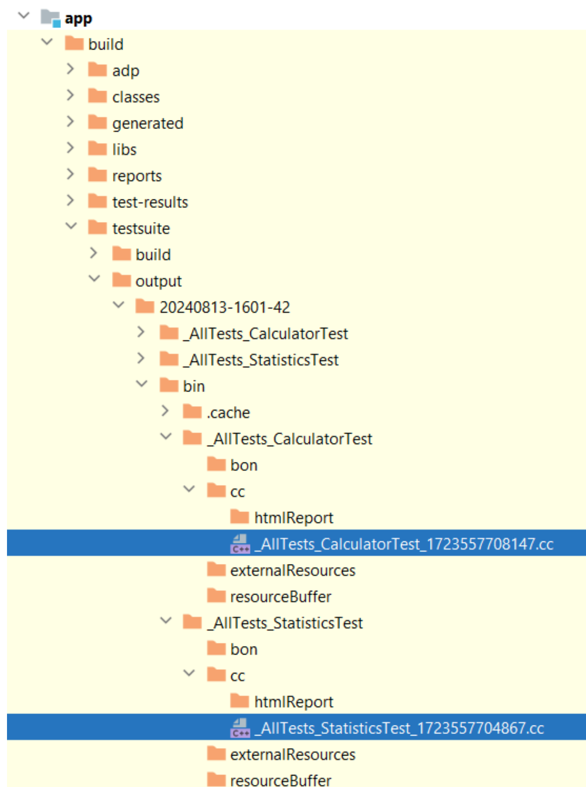
To generate the Code Coverage files (**.cc**) for each test, update the **build.gradle.kts** file as follows:

```
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")
            }
        }
    }
}
```

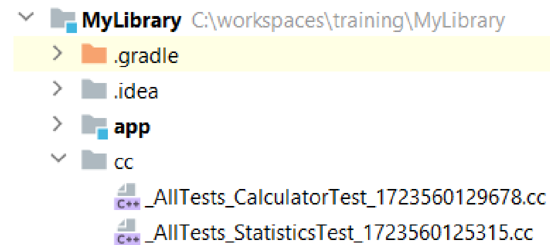
© MICROEJ 2024

GENERATE THE CODE COVERAGE REPORT (1/2)

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



2. Create a **cc/** folder in the **MyLibrary/** folder.
3. Copy all the .cc files inside:



4. Open a Command Prompt console in the **MyLibrary** folder, run the following command:

```
.\gradlew.bat execTool ^
  --name=codeCoverageAnalyzer ^
  --toolProperty="cc.dir=C:\PATH_TO_PROJECT\MyLibrary\cc" ^
  --toolProperty="cc.includes=com.microej.example.*" ^
  --toolProperty="cc.excludes=" ^
  --toolProperty="cc.src.folders=C:\PATH_TO_PROJECT\MyLibrary\app\src\main\java" ^
  --toolProperty="cc.html.dir=C:\PATH_TO_PROJECT\MyLibrary\cc\htmlReport"
```

NOTE: this process will be automatized in the next releases of MICROEJ SDK 6.

GENERATE THE CODE COVERAGE REPORT (2/2)

Console output after running the Code Coverage report generation:

```
C:\workspaces\training\MyLibrary>.\gradlew execTool --name=codeCoverageAnalyzer --toolProperty="cc.dir=C:\workspaces\tra
ining\MyLibrary\cc" --toolProperty="cc.includes=com.microej.example.*" --toolProperty="cc.excludes=" --toolProperty="cc.
src.folders=C:\workspaces\training\MyLibrary\app\src\main\java" --toolProperty="cc.html.dir=C:\workspaces\training\MyLib
rary\cc\html"
Loading files
2 *.cc files !
Path entry not found: C:\C
Path entry not found: C:\workspaces\training\MyLibrary\app\build\resources\test
Path entry not found: C:\workspaces\training\MyLibrary\app\build\resources\main
..
4 methods have been loaded from file(s)
Analyzing data
...........
Writing reports
...............................

BUILD SUCCESSFUL in 3s
```

The report is generated in the **MyLibrary/cc/html/** folder.

CODE COVERAGE REPORT ANALYSIS

MICROEJ

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 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"
                     )
                  ...
```

© MICROEJ 2024

RUN THE TESTS ON DEVICE

Start the Serial to Socket Transmitter tool to redirect the execution traces:

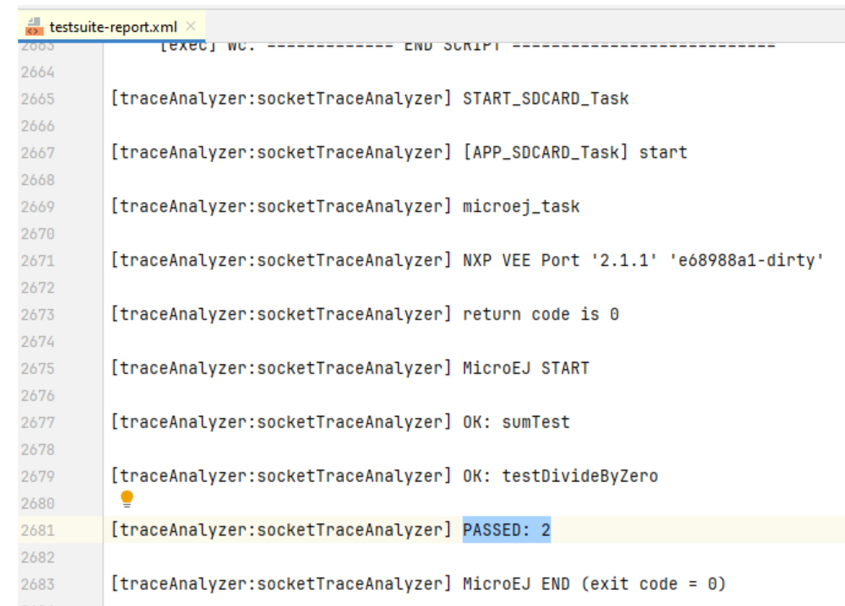
- Open a Command Prompt in the **MyLibrary** folder.
- Run the following command, edit the `comm.port` property according to your VEE Port COM port:

```
.\gradlew.bat execTool --name=serialToSocketTransmitter ^
--toolProperty="serail.to.socket.comm.port=COM7" ^
--toolProperty="serail.to.socket.comm.baudrate=115200" ^
--toolProperty="serail.to.socket.server.port=5555" ^
--console plain
```

Run the tests on device:

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

Test results can be checked in **testsuite-report.xml:**



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)

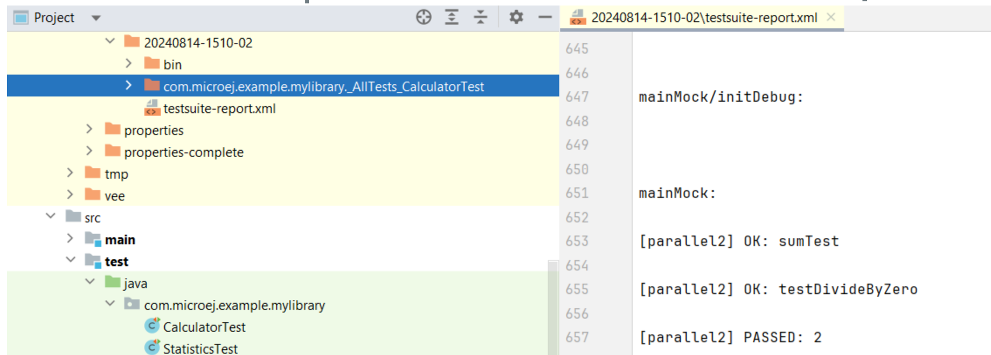
Gradle allows to filter the tests from the command line directly, thanks to the **--tests** option.

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:



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

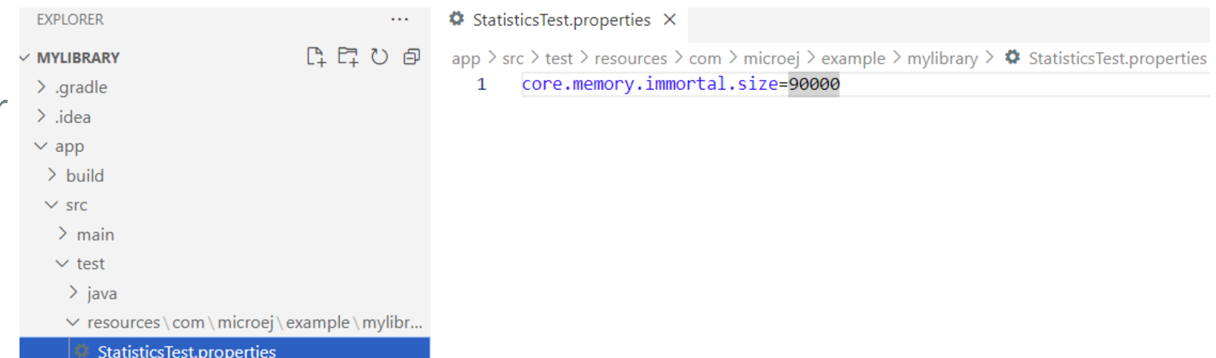
INJECT APPLICATION OPTIONS

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.



© MICROEJ 2024

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.

© MICROEJ 2024

THANK YOU

for your attention !

MICROEJ®