Android-Development
using
Android Studio

A guided Tutorial with respect to Test-Driven Development

v15.03.2017

by
Patrick Radkohl & *Stephan Frühwirt*

Inhaltsverzeichnis

Android Studio

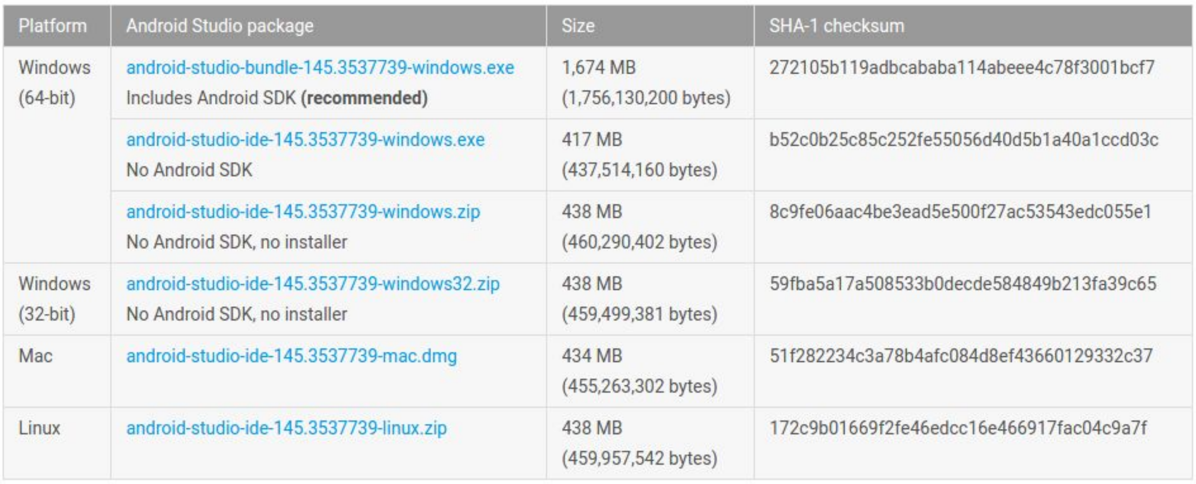
Android Studio is the most popular integrated development environment (IDE) for Android-Development. It’s based on the IntelliJ IDE, and is officially developed and supported by Google.

Installation Process

The current version of Android Studio can be downloaded at the following link:

https://developer.android.com/sdk/installing/studio.html

There are different version for multiple operating systems available:

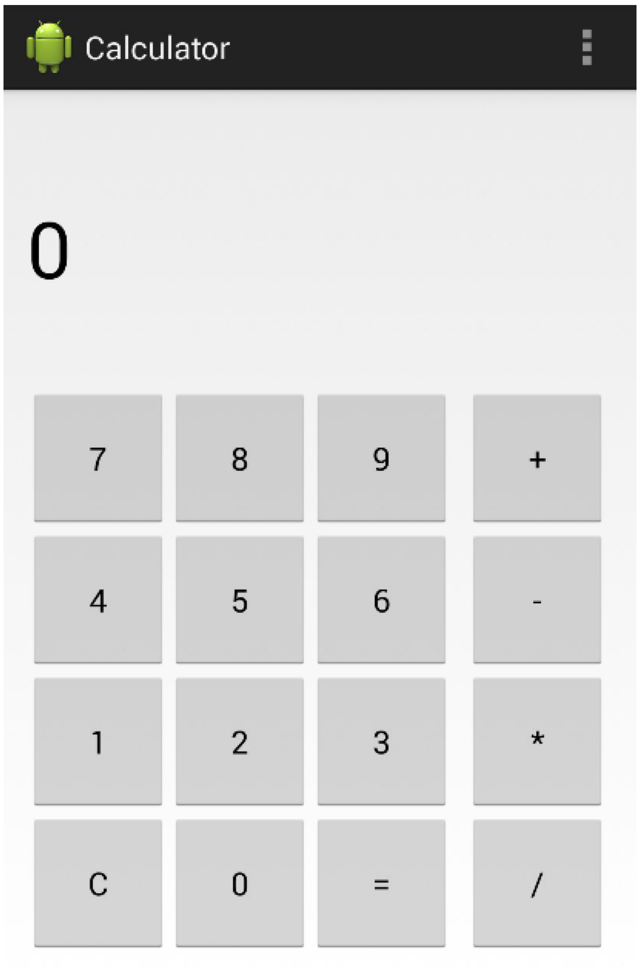


Platform	Android Studio package	Size	SHA-1 checksum
Windows (64-bit)	android-studio-bundle-145.3537739-windows.exe Includes Android SDK **(recommended)**	1,674 MB (1,756,130,200 bytes)	272105b119adbcababa114abeee4c78f3001bcf7
	android-studio-ide-145.3537739-windows.exe No Android SDK	417 MB (437,514,160 bytes)	b52c0b25c85c252fe55056d40d5b1a40a1ccd03c
	android-studio-ide-145.3537739-windows.zip No Android SDK, no installer	438 MB (460,290,402 bytes)	8c9fe06aac4be3ead5e500f27ac53543edc055e1
Windows (32-bit)	android-studio-ide-145.3537739-windows32.zip No Android SDK, no installer	438 MB (459,499,381 bytes)	59fba5a17a508533b0decde584849b213fa39c65
Mac	android-studio-ide-145.3537739-mac.dmg	434 MB (455,263,302 bytes)	51f282234c3a78b4afc084d8ef43660129332c37
Linux	android-studio-ide-145.3537739-linux.zip	438 MB (459,957,542 bytes)	172c9b01669f2fe46edcc16e466917fac04c9a7f

At the first startup it many take a while until all components are properly initialized.

Example: Calculator

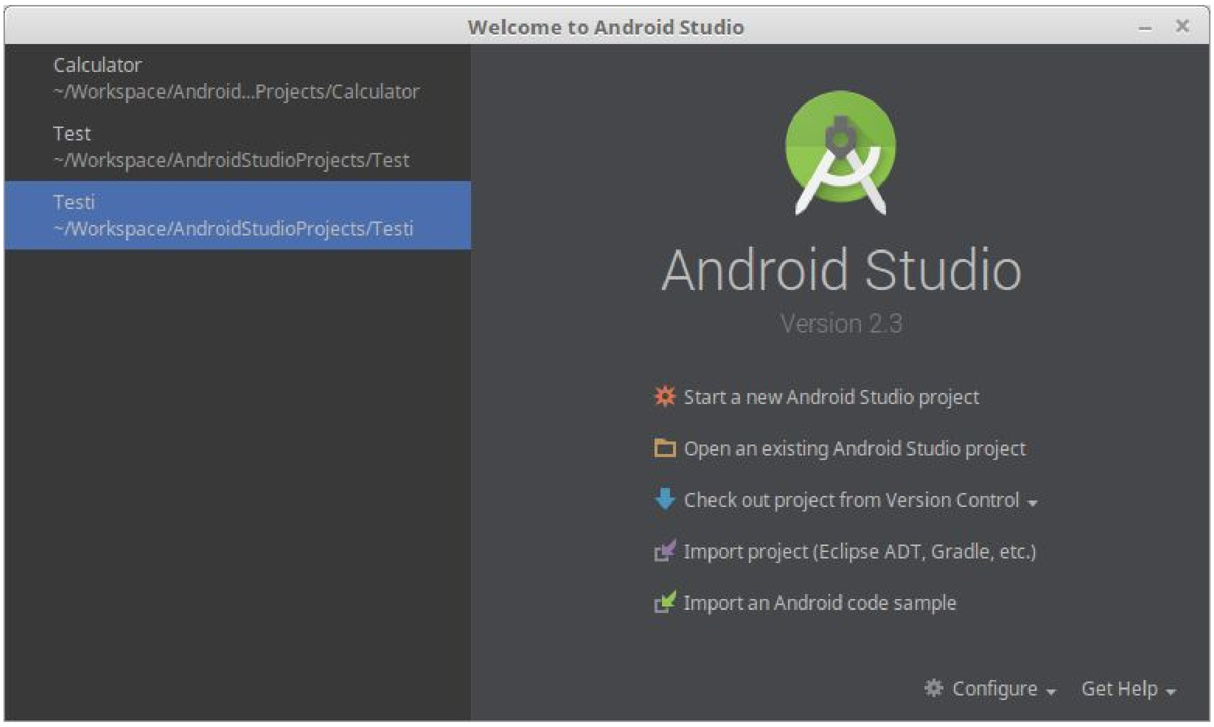
This chapter contains a step by step guide for a simple calculator app. The user interface consists of buttons for the numbers, operators and utility functions. Furthermore, there is a GUI element (TextView) to display the calculation results.



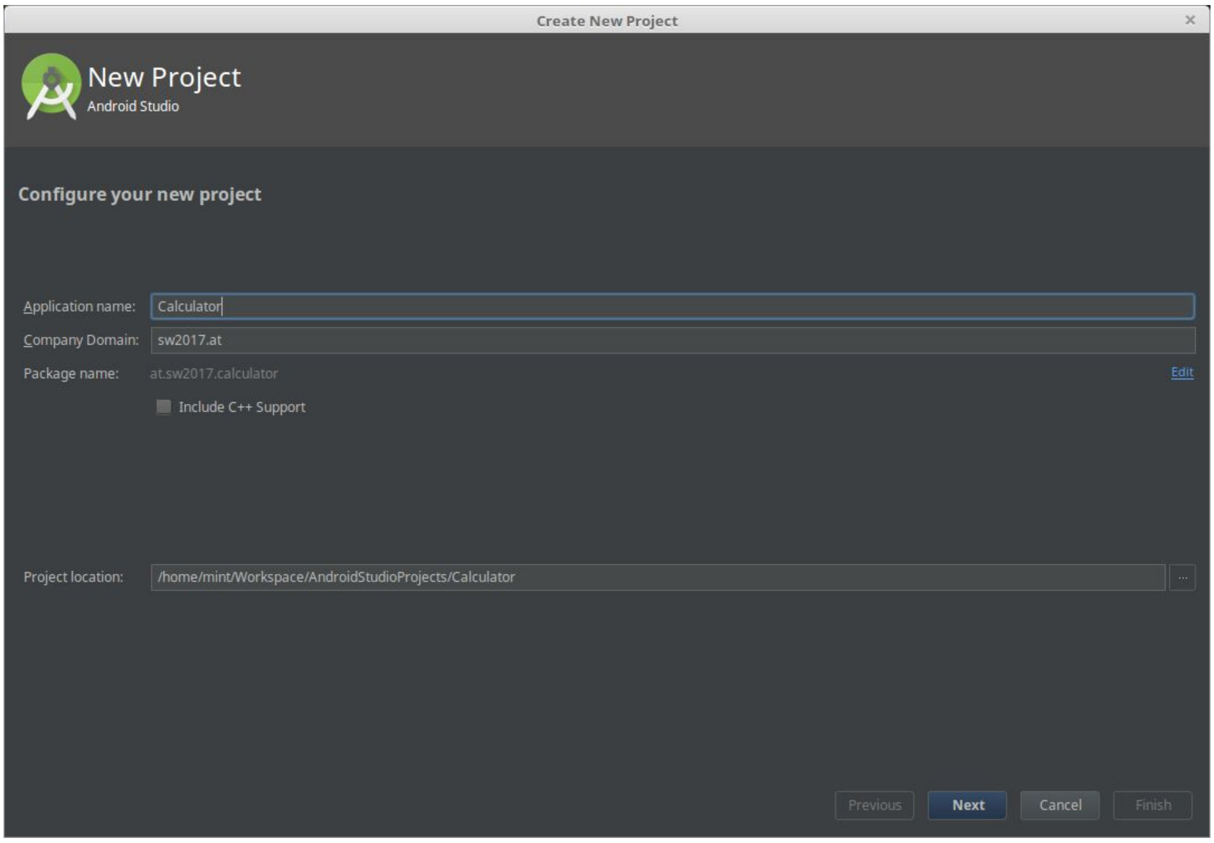
The software development methodology used in this tutorial is called Test-driven Development.

Solutions to widely known problems, and other useful informations are provided in the last chapter of this document.

After the first startup of Android Studio, the following window should appear:

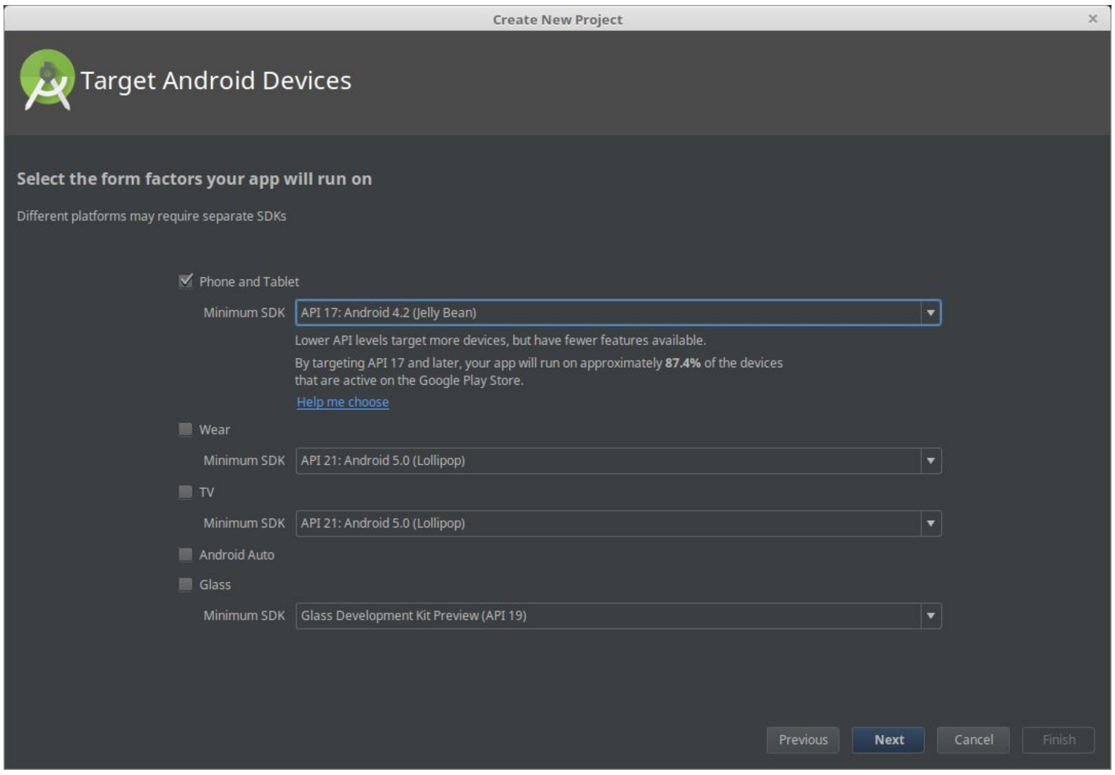

Welcome to Android Studio
Calculator
~/Workspace/Android...Projects/Calculator
Test
~/Workspace/AndroidStudioProjects/Test
Testi
~/Workspace/AndroidStudioProjects/Testi
Android Studio
Version 2.3
Start a new Android Studio project
Open an existing Android Studio project
Check out project from Version Control
Import project (Eclipse ADT, Gradle, etc.)
Import an Android code sample
Configure
Get Help

Select *New Project*, and enter the following:
for Application Name: *Calculator*
for Company Domain: *sw2017.at*
for the *Project location* choose an arbitrary location on the filesystem.



Application name: Calculator
Company Domain: sw2017.at
Package name: at.sw2017.calculator
Edit
Include C++ Support
Project location: /home/mint/Workspace/AndroidStudioProjects/Calculator

In the next step the minimal required API version of the project must be selected. It defines the minimal required Android version on which the app is runnable. For this demo it is sufficient to select API level 17.



Create New Project

Target Android Devices

Select the form factors your app will run on

Different platforms may require separate SDKs

☑ Phone and Tablet

Minimum SDK: API 17: Android 4.2 (Jelly Bean)

Lower API levels target more devices, but have fewer features available.

By targeting API 17 and later, your app will run on approximately **87.4%** of the devices that are active on the Google Play Store.

Help me choose

☐ Wear

Minimum SDK: API 21: Android 5.0 (Lollipop)

☐ TV

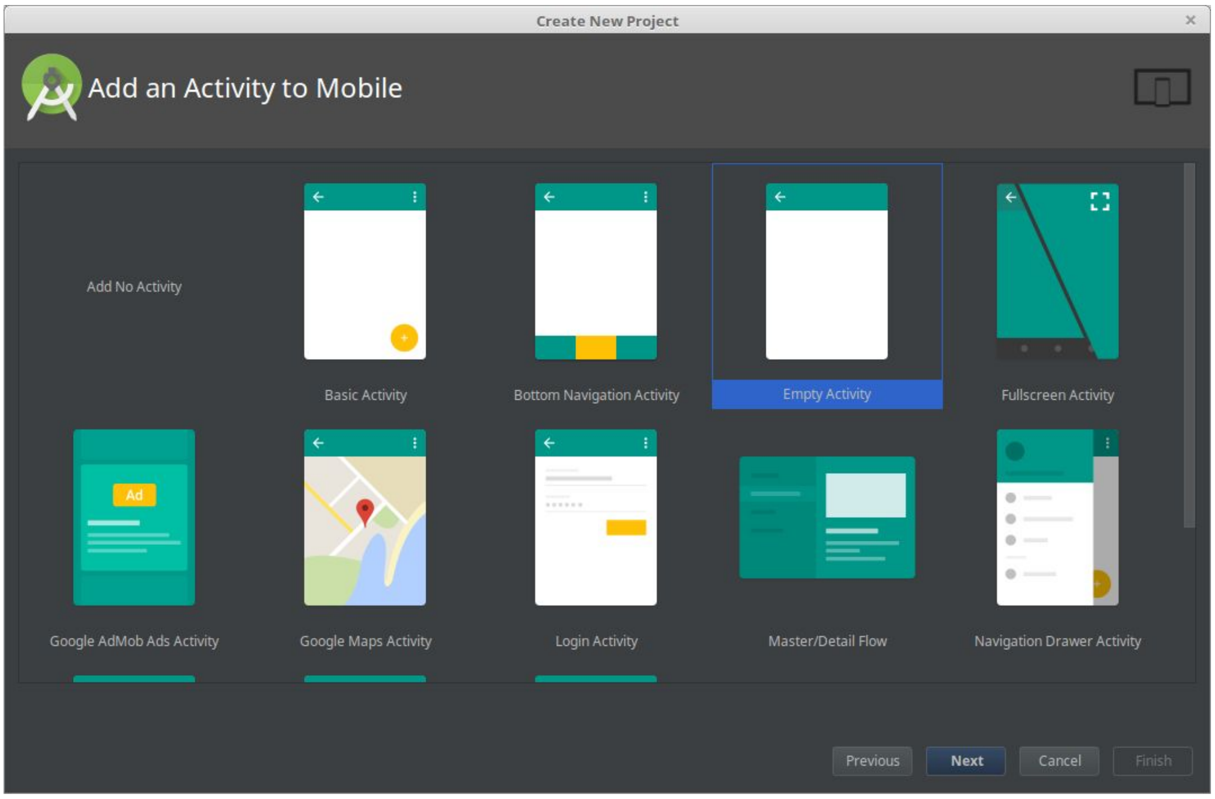
Minimum SDK: API 21: Android 5.0 (Lollipop)

☐ Android Auto

☐ Glass

Minimum SDK: Glass Development Kit Preview (API 19)

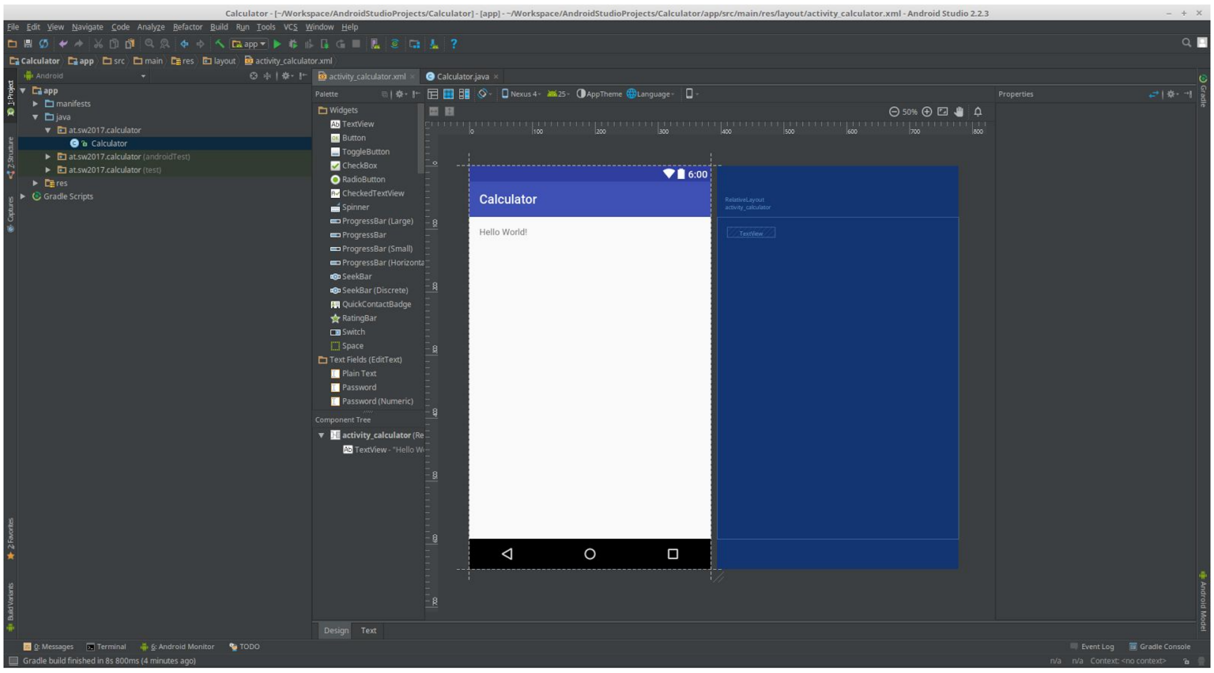
After that the *Empty Activity* should be selected as container for the user interface.



Create New Project

Add an Activity to Mobile

In the next window the project creation is finalised by choosing a name (e.g. *Calculator*) for the activity, and pressing *Finish*. All other settings on this window require no further modification. After that Android Studio should look like this:



Gradle build finished in 8s 800ms (4 minutes ago)

After the gradle build process and the IDEs internal code parsing and indexing is done we can start with our developing. Note that this can depending on your computer take some time. After this step we should be able to create boot up our emulator and start our *hello_world* application.

Automatically generated files

After the creation of the project a bunch of files will be automatically generated. Two of these are especially relevant in the current state of the project. The *Calculator.java* holds the program logic, and is located in *at.sw2017.calculator*.

The GUI of the application is defined by *activity_calculator.xml.* It is located in *res* > *layout* > *activity_calculator.xml*

The XML-File can be opened in two different views. The *Design* view shows the graphical representation of the user interface, and allows simple manipulations using Drag and Drop. The *Text* view allows a more sophisticated manipulation of the GUI, by editing the XML-Representation manually.

The *Calculator.java* contains the class Calculator, which extends the AppCompatActivity class. Therefore, it is possible to communicate with the user interface using specific methods. For example, the onCreate method is used for the initiation of the activity and the corresponding UI.

The class *AppCompatActivity* inherits from the plain *Activity* class and handles in this simple example the Titlebar of your Application. Changing the extended class from *AppCompatActivity* to *Activity* you will see that this Titlebar is gone and only the empty space with one lonely Hello World will be present anymore. For a more exact description of the different activity classes please look up the documentation.

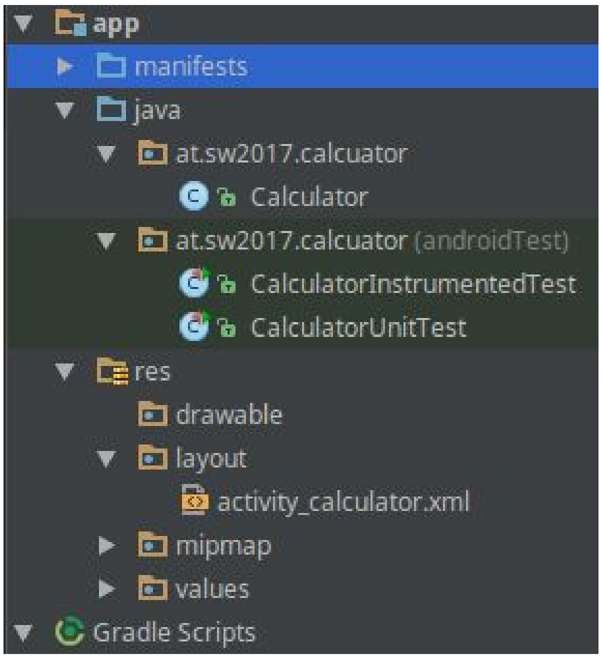
In this state the application is already executable, and displays the text “Hello World”. The app can either be executed on a physical android device, or a emulator (see chapter Emulator).

Test-driven Development

The software development methodology used for this project is called Test-driven Development. The usual way for software development is to write code first, and test it afterwards. For this example the opposite strategy is applied: Therefore, first a simple test case is written (which should fail), and afterwards the missing pieces are implemented to pass this test case. This two steps are repeated for each feature.
One benefit of this methodology is that each feature is tested in a separate test case, and therefore bound to the corresponding code. This way of testing leads to an implicit up-to-date documentation of the program.

Setup of the test environment

Basically there are two types of tests: Test cases for the user interface, and for the program logic. In this section the essential elements, and templates for both types of test cases are added to the project.
In the first step the sub-packages *test* and *uitest* are moved to the *androidTest* package. The autogenerated classes names should be adapted to match our application. After that the package hierarchy should look like one of the following figures:



```
app
  manifests
  java
    at.sw2017.calcuator
      Calculator
    at.sw2017.calcuator (androidTest)
      CalculatorInstrumentedTest
      CalculatorUnitTest
  res
    drawable
    layout
      activity_calculator.xml
    mipmap
    values
Gradle Scripts
```

Note: This is the android project view of Android Studio. If you change the view your folder structure will look different.

As already vaguely introduced there are two different kinds of test cases. Instrument tests and Unit test. Instrument tests are executed on our android device so their main purpose is to take care of our UI testing. For the Unit tests the execution doesn’t use the android device so we are much faster here. We will use this kind of testing for the backend functions.

Our first test case

Before we can actually start to implement test cases and code we have to get familiar with our working environment. To do this we have to read a basic overview about our IDE Android Studio. Please read the overview on this page.
https://developer.android.com/studio/intro/index.html
Only continue when you are done with it!

The first actual test case we will implement will test if there are all the buttons present. Since this fits in the UI category of testing we will place our new test case in the file *CalculatorInstromentFile.* When opening up the file it should look something like this.

```
package at.sw2017.calculator;

import android.content.Context;
import android.support.test.InstrumentationRegistry;
import android.support.test.runner.AndroidJUnit4;

import org.junit.Test;
import org.junit.runner.RunWith;

import static org.junit.Assert.assertEquals;

/**
* Instrumentation test, which will execute on an Android device.
*
* @see <a href="http://d.android.com/tools/testing">Testing documentation</a>
*/
@RunWith(AndroidJUnit4.class)
public class CalculatorInstrumentedTest {

   @Test
   public void useAppContext() throws Exception {
       // Context of the app under test.
       Context appContext = InstrumentationRegistry.getTargetContext();
       assertEquals("at.sw2017.calcuator", appContext.getPackageName());
   }
}
```

To understand how to create an actual test case we also need some information about the testing environment itself. We will use the Espresso Framework for testing. In the next step we want you to understand the framework a little bit better. To do this please read the page:
https://developer.android.com/training/testing/ui-testing/espresso-testing.html
Please only continue afterwards!

After reading this it is clear that the first thing we need to do is to set up the *ActivityTestRule* for our Calculator class. When adding the rule (code below) we notice that there are some errors in the code. This comes from the missing includes in the program. We can click into one of the words marked with an error like *Rule* and press **Alt+Enter.** With this action we bring up the quick fix menu from the IDE which allows us to fix the imports with an click on *Import Class.*

```
@Rule
public ActivityTestRule<Calculator> mActivityRule = new
ActivityTestRule<>(Calculator.class);
```

The first one checks if all buttons are available, i.e. the buttons for the digits, for the operators, and utility functions. The following code snippet shows the test case, which checks the availability of all calculator buttons:

```
@Test
public void testButtons() throws Exception {

   for (int i = 0; i <= 9; i++) {
       onView(withText(Integer.toString(i))).perform(click());
   }
   onView(withText("+")).perform(click());
   onView(withText("-")).perform(click());
   onView(withText("*")).perform(click());
   onView(withText("/")).perform(click());

   onView(withText("=")).perform(click());
   onView(withText("C")).perform(click());
}
```

It is important that every test method is public. The name has the prefix *test* to indicate that this is a designated test method. In the end it is up to you how to name your test cases but we encourage you to start them with *test.* To start the test case it's sufficient to rightclick on the method and select Run. This test case will obviously fail, because we have not implemented buttons yet.

Creation of buttons

For this part we open the view layout designer of Android Studio. We recommend you to research about one of the newer features called Constraint Layouts. There are lots of different resources online which you can read. However, for this Tutorial it is sufficient to watch the following four minutes video on youtube: https://www.youtube.com/watch?v=XamMbnzI5vE.
Carry on if you are done watching!

All view elements can be found in the file *activity_calculator.xml* (*res* > *layout*). There are two different ways how you can change a View. We can switch between them in the bottom left of the IDE. Just click on the ***Design*** tab or the ***Text*** tap to switch between them. In the Design tab we now drag&drop the buttons into the screen. To adapt the size of each element, we go back to the Text-tab:

```
android:layout_width="80dip"
android:layout_height="80dip"
```

Moreover, we have to give the element an unique name and define the shown text/ symbol:

```
android:text="0"
android:id="@+id/buttonZero"
```

Finally, the XML-representation of one button should look like as follows:

```
<Button
   android:id="@+id/button0"
   android:layout_width="80dp"
   android:layout_height="80dp"
   android:text="@string/_0"
   app:layout_constraintBottom_toBottomOf="parent"
   android:layout_marginBottom="8dp"
   app:layout_constraintLeft_toRightOf="@+id/buttonC"
   android:layout_marginStart="8dp" />
```

Note: This can differ from yours depending on how you did the constraint layout!

Next, we open the Calculator-class. Here we create a private member variable for each button we have in our calculator. In the onCreate() method we create a “link” between the view-elements and the logic part in order to access the buttons:

```
buttonAdd = (Button) findViewById(R.id.buttonAdd);
```

The method findViewById(...) returns a view object. We can easily cast that value to Button because every element (Button, TextView, etc.) is derived from View. Furthermore, we can access the ID of each element (which we have defined in the XML-file before) using R.id.buttonAdd. R stands for the resource file, where all ids are stored.

Afterwards, we have to register an OnClickListener for each button in order to notice a pressed button:

```
buttonAdd.setOnClickListener(this);
```

To use the method above we have to change the header of the Calculator class:

```
public class Calculator extends Activity implements View.OnClickListener {
    ...
}
```

After we have done this, an onClick() method gets generated automatically. (red underlined code press **Alt+Enter** and press Implement Methods)

Because we do not want to do all the steps described above for every number button we define a method which does the linking using findViewById(...) and the registration of the OnClickListener. We can add every button to an ArrayList. Thus, we do not need to create ten additional member variables.

```
public void setUpNumberButtonListener() {
   for (int i = 0; i <= 9; i++) {
       String buttonName = "button" + i;

       int id = getResources().getIdentifier(buttonName, "id",
R.class.getPackage().getName());

       Button button = (Button) findViewById(id);
       button.setOnClickListener(this);

       numberButtons.add(button);
   }
}
```

Now we can run the test which we have written before. If you have named all buttons correctly, the test should pass.

Test: testInputField

Next, we want to test if the correct number is displayed in a textbox after we have pressed several number buttons:

```
@Test
public void testInputField(){
   for(int i = 9 ; i >= 0; i--){
       onView(withText(Integer.toString(i))).perform(click());
   }
   onView(withText("9876543210")).check(matches(isDisplayed()));
}
```

The line after the for loop looks for a given string on the screen as we already researched from the basic Espresso Tutorial (earlier link!). The test will pass if the string was found.

Code: testInputField

First, we need to add a text view to the XML-File, define another member variable of the type *TextView* and link it to the XML-object using *findViewById*:

```
numberView = (TextView) findViewById(R.id.textView);
```

Then, we switch to the onClick() method:
We have to find out, which button was clicked. This can be done by using a switch block.

```
@Override
public void onClick(View v) {
    Button clickedButton = (Button) v;

    switch (clickedButton.getId()) {
        case R.id.buttonAdd:
            break;
        case R.id.buttonSub:
            break;
        case R.id.buttonMul:
            break;
        case R.id.buttonDiv:
            break;
        case R.id.buttonEqual:
            break;
        case R.id.buttonClear:
            break;
        default:
            String recentNumber = numberView.getText().toString();
            if (recentNumber.equals("0")) {
                recentNumber = "";
            }
            recentNumber += clickedButton.getText().toString();
            numberView.setText(recentNumber);
    }
}
```

The first six case blocks represent the mathematical operators and additional buttons. In the default block we have to save the shown number. Then we check, if this number is 0. If this is the case, we have to empty the text view to avoid inputs like 0123. After that, we can concatenate the old number with the new number and write the result back to the text view.

Test: testClearButton

After pressing the clear-button the text view should be reset to “0”:

```
@Test
public void testClearButton(){
   onView(withText("3")).perform(click());
   onView(withText("C")).perform(click());

   onView(withId(R.id.textView)).check(matches(withText("0")));
}
```

For this check we directly search in the textview where our displayed number is located. If we would try the earlier approach where we search just after a string like `onView(withText("0"))` would not work this time because we have now two views with the text “0”. One zero is located in our textView and the second one is our button. Hence it is necessary to reduce our search space. To conclude, in this test case we don’t just search for the occurrence of the string but we search specifically the text in our textview.

Code: testClearButton

We introduce a new method called *clearTextView()*. This method simply sets the text of the text view to “0”.

```
private void clearTextView() {
    numberView.setText("0");
}
```

The method has to be called in the proper case block in our switch statement. We will have to extend this method later on.

Test: testAddition, testSubtraction, testMultiplication, testDivision

As already mentioned we can do UI-tests as well as JUnit-tests (used for testing the logic) for testing our application. Now we want to create some JUnit-tests where we test all mathematical operations. For doing this we use the auto generated class *CalculationsUnitTest* from earlier. For the *Calculations* class we plan a class with a private constructor and static methods which will make testing really simple. As soon as the class is created we can implement our tests. All tests are very similar. Therefore, all tests for the mathematical operations will have the same structure:

```
@Test
public void testDoAddition(){
   int result = Calculations.doAddition(2, 3);
   assertEquals(5, result);
}
```

Per convention, the assertEquals(...) receives the expected result as first parameter and the actual result as second parameter.
We have to consider one special case when we implement the test for the division: A division by 0 will throw an exception. Therefore, we have to define how our calculator should behave in this case. To keep this tutorial simple we just define that a division by 0 returns 0. However, this special case has to be tested too! We write two tests for the division and append the number 1 to the second tests name. Because of this naming, we instantly know that we test the same method and that there are more than one cases that need to be tested. The whole class looks as follows:

```
public class CalculatorUnitTest {

   @Test
   public void testDoAddition(){
       int result = Calculations.doAddition(2, 3);
       assertEquals(5, result);
   }

   @Test
   public void testDoSubtraction(){
       int result = Calculations.doSubtraction(7, 2);
       assertEquals(5, result);
   }

   @Test
   public void testDoMultiplication(){
       int result = Calculations.doMultiplication(2, 3);
       assertEquals(6, result);
   }

   @Test
   public void testDoDivision(){
       int result = Calculations.doDivision(8, 4);
       assertEquals(2, result);
   }

   @Test
   public void testDoDivision1(){
       int result = Calculations.doDivision(8, 0);
       assertEquals(0, result);
   }

   @Test
   public void testDoDivision2(){
       int result = Calculations.doDivision(11, 4);
       assertEquals(2, result);
   }

}
```

These tests can all be executed at the same time by right clicking on the class header and choosing *Run CalculatorUnitTest*.

There exist several assert methods that can be used for testing:

- assertTrue
- assertFalse
- assertEquals
- assertSame
- assertNotSame
- assertNull

- assertNotNull

Code: testAddition, testSubtraction, testMultiplication, testDivision

As described above, we create a new class *Calculations*, where we do all our calculations. This class has a private constructor (no objects can be created) and all methods are static methods which are called using Classname.method():

```
class Calculations {

   private Calculations() {
   }

   static int doAddition(int firstNumber, int secondNumber) {
       return firstNumber + secondNumber;
   }

   static int doSubtraction(int firstNumber, int secondNumber) {
       return firstNumber - secondNumber;
   }

   static int doMultiplication(int firstNumber, int secondNumber) {
       return firstNumber * secondNumber;
   }

   static int doDivision(int firstNumber, int secondNumber) {
       if (secondNumber == 0) {
           return 0;
       }
       return firstNumber / secondNumber;
   }
}
```

As you can see in this listing, we have already considered the special case of the division by 0. Therefore, we have implemented the necessary mathematical operations.

Now we have to think about how a calculation has to be executed:

1. Insert a number
2. Choose a mathematical operation
   - The content of the text view gets set empty
3. Insert another number
4. Press „=“

Additionally, we have to save the first number and remember, which mathematical operation should be applied in order to calculate the correct result. Therefore, we create a method which saves the first number in a member variable and clears the text view:

```
private void clearNumberView() {
    String tempString = numberView.getText().toString();
    if(!tempString.equals("")){
        firstNumber = Integer.valueOf(tempString);
    }
    numberView.setText("");
}
```

First, we save the actual string from the text view and check if it is empty (casting an empty string to integer and saving it to an integer variable would cause an exception). Then we save the value and clear the text view.
Furthermore, we have to define an enum with some states in order to remember the selected operation:

```
public enum State {
    ADD, SUB, MUL, DIV, INIT, NUM
}
```

The recent state has to be saved in a member variable. At the beginning we are in the INIT-state. This state has to be updated properly in every case block in our switch statement. In addition to that, we have to customize the default-block: At the beginning we are in the INIT-state, which means that the text view shows a 0. Therefore, we can now check if the INIT-state is set and change to the NUM-state. This is just an “in-between-state” which has to be different to all other states. After pressing the “=” button we return to the INIT-state. After applying these changes the *onClick* method should look like as follows:

```
@Override
public void onClick(View v) {
    Button clickedButton = (Button) v;

    switch (clickedButton.getId()) {
        case R.id.buttonPlus:
            clearNumberView();
            state = State.ADD;
            break;
        case R.id.buttonMinus:
            clearNumberView();
            state = State.SUB;
            break;
        case R.id.buttonMultiply:
            clearNumberView();
            state = State.MUL;
            break;
        case R.id.buttonDivide:
            clearNumberView();
            state = State.DIV;
            break;
        case R.id.buttonEqual:
            calculateResult();
            state = State.INIT;
            break;
        case R.id.buttonClear:
            clearTextView();
            break;
        default:
            String recentNumber = numberView.getText().toString();
            if (state == State.INIT) {
                recentNumber = "";
                state = State.NUM;
            }
            recentNumber += clickedButton.getText().toString();
            numberView.setText(recentNumber);
    }
}
```

It is also important to reset all values when we click on the clear button. Therefore, we adapt the *clearTextView()* method to set the variable of the first number to 0 and reset the state to the INIT-state:

```
private void clearTextView() {
    numberView.setText("0");
    firstNumber = 0;
    state = State.INIT;
}
```

The last thing we have to implement is the functionality of the “=” button. For doing this, we create another method *calculateResult()*. First, we have to save the shown number and check if it is an empty string (casting an empty string to integer and saving it to an integer variable would cause an exception). Then we add a switch statement where we check which operation should be executed (ADD, SUB, MUL, DIV). Depending on the state we call the appropriate method of our *Calculations* class and update the text view:

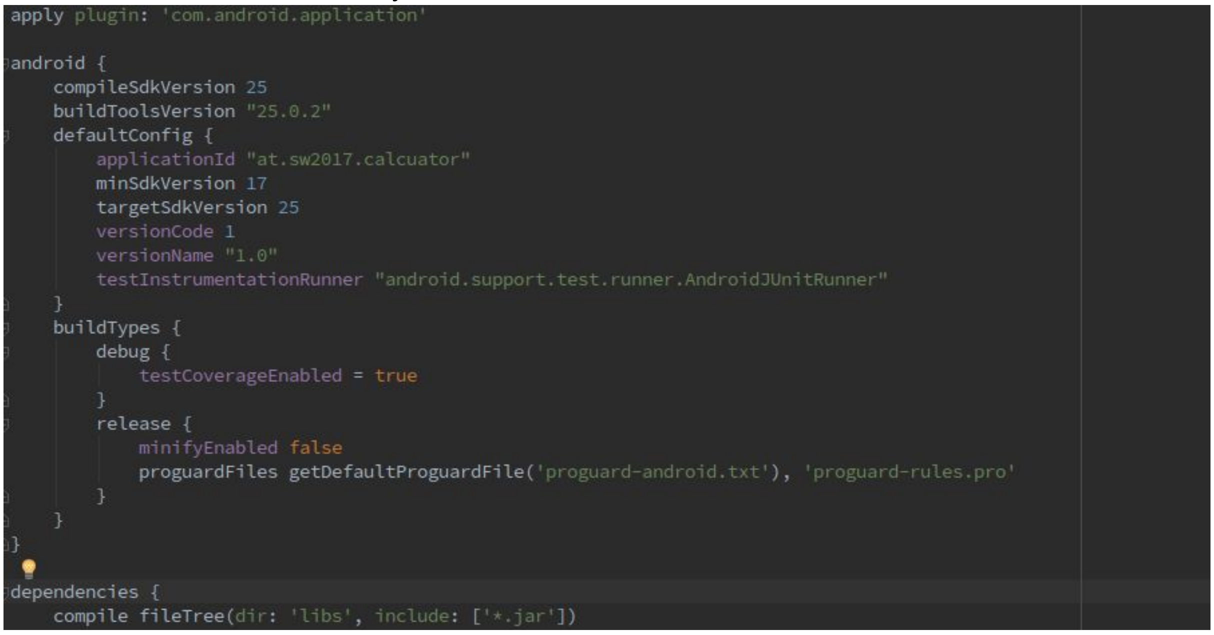
```
private void calculateResult() {
    int secondNumber = 0;

    String tempString = numberView.getText().toString();
    if(!tempString.equals("")){
        secondNumber = Integer.valueOf(tempString);
    }

    int result;
    switch(state){
        case ADD:
            result = Calculations.doAddition(firstNumber, secondNumber);
            break;
        case SUB:
            result = Calculations.doSubtraction(firstNumber, secondNumber);
            break;
        case MUL:
            result = Calculations.doMultiplication(firstNumber, secondNumber);
            break;
        case DIV:
            result = Calculations.doDivision(firstNumber, secondNumber);
            break;
        default:
            result = secondNumber;
    }
    numberView.setText(Integer.toString(result))
}
```

Quality control:

Now after this last step we have a functional calculator. At least we think everything is ok and works just fine. Since we have written every test in advance we specified exactly what functionality our project has to fulfil and we should have tested everything our program, right? Well yes, maybe. If only we had a chance to verify this to be sure that we really checked every single instruction and branch possible.
Well luckily there is such a possibility in Android Studio and it is called Java Code Coverage. Let's look into this and show you how it works.

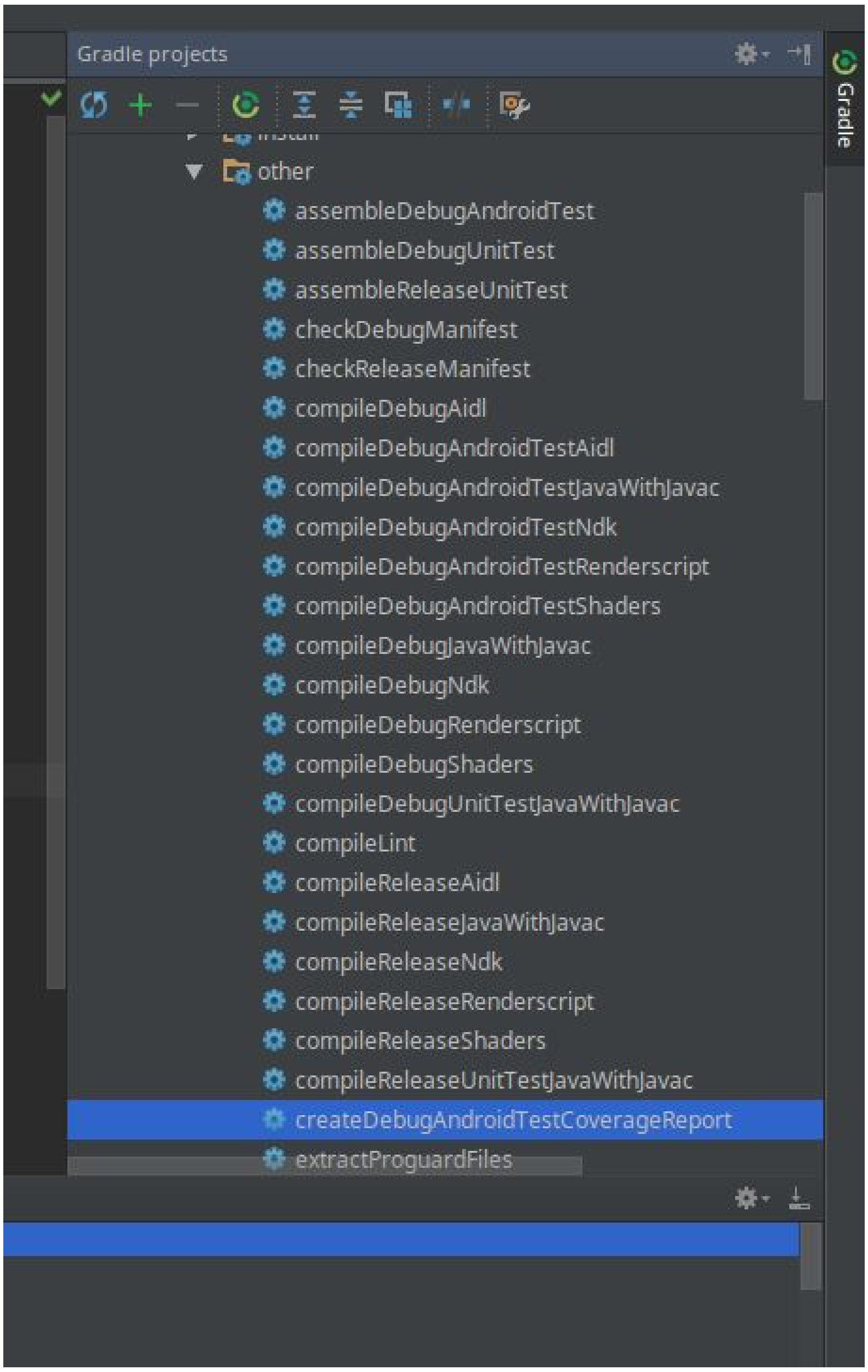
To activate it for your application we have to open up our *build.gradle (Module: app)* and add a new target to *buildTypes*. We will call this one *debug* and we add the variable *testCoverageEnabled* = *true* .

This should look like this after you done it.



```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 25
    buildToolsVersion "25.0.2"
    defaultConfig {
        applicationId "at.sw2017.calcuator"
        minSdkVersion 17
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        debug {
            testCoverageEnabled = true
        }
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
```

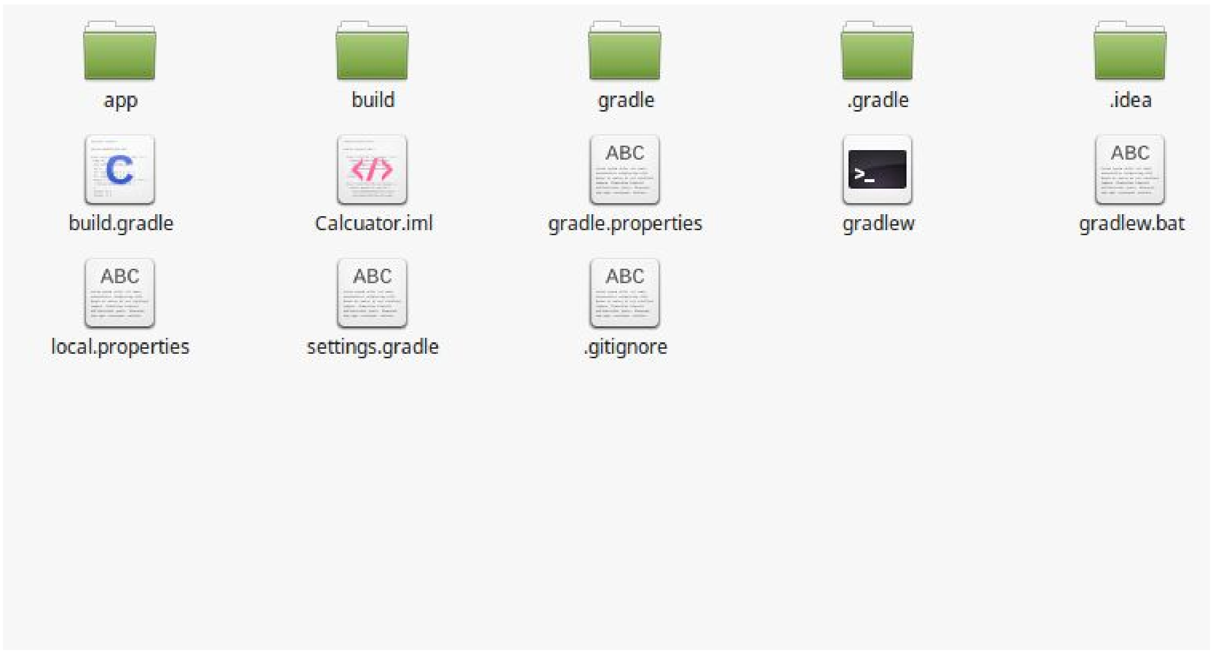
Now we have to resynchronize our project do the Gradle file changes. After the sync we got a new gradle job ready under the category *"other"* and it is called *createDebugAndroidTestCoverReport (seen in the picture below)*. If you don't have the side menu where you can toggle the gradle job visibility you can show and hide it with the really small Mousepad Symbol at the left-bottom corner of your IDE. Just give it a click. If you don't find it try google or ask us.



```
▼ other
    assembleDebugAndroidTest
    assembleDebugUnitTest
    assembleReleaseUnitTest
    checkDebugManifest
    checkReleaseManifest
    compileDebugAidl
    compileDebugAndroidTestAidl
    compileDebugAndroidTestJavaWithJavac
    compileDebugAndroidTestNdk
    compileDebugAndroidTestRenderscript
    compileDebugAndroidTestShaders
    compileDebugJavaWithJavac
    compileDebugNdk
    compileDebugRenderscript
    compileDebugShaders
    compileDebugUnitTestJavaWithJavac
    compileLint
    compileReleaseAidl
    compileReleaseJavaWithJavac
    compileReleaseNdk
    compileReleaseRenderscript
    compileReleaseShaders
    compileReleaseUnitTestJavaWithJavac
    createDebugAndroidTestCoverageReport
    extractProguardFiles
```

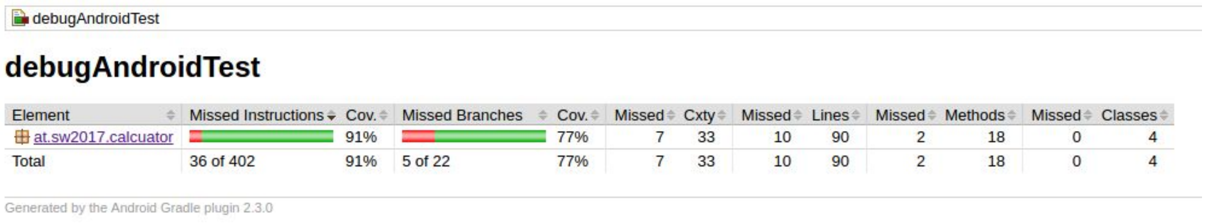
This was the complicated stuff already. Now we need to start up our emulator or plug in our device wherever we want to execute our test cases (We recommend the emulator because we will test it over this method with an emulator).

After it is finished you can check out your generated code coverage report. For that you open up your Project on your hard disk. You should find something like this:



The folder of the report is under *app/build/reports/coverage/debug/index.html*

After opening the file you will see something like this:



debugAndroidTest

debugAndroidTest

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
at.sw2017.calcuator		91%		77%	7	33	10	90	2	18	0	4
Total	36 of 402	91%	5 of 22	77%	7	33	10	90	2	18	0	4

Generated by the Android Gradle plugin 2.3.0

So we can see our testing was good - but not all we could do. There are still some branches in our code execution untested with our current test set. Also as a result there are also a bunch of instructions untested. Since we do test driven development we should in a perfect world have 100% coverage so your **last task in this tutorial** is to show what you have learned so far and **fix your test suite to get a perfect score**.

When you’re finished or you have questions visit us! We will try to give you the needed support! Happy coding and huge success.

Appendix

Configuration of an emulator

In order to execute and test your code you will need whether an Android-smartphone or you can use an emulator. If there is no smartphone available, you have to use an emulator. However, Android-emulators are very very slow. The start procedure as well as the execution take much more time than on a real device. Therefore, make sure that you DO NOT close your emulator during the programming session. Otherwise you will have to wait up to 5 minutes until it is ready for action again.
You can find the virtual device manager under *Tools* > *Android* > *AVD Manager*. This opens a new window where you create a new emulator, edit or delete it. In addition, you can start an emulator using the AVD Manager.
You should talk to your team members about the configuration of the emulator since you have to specify the supported API-version.

Installation of the most important packages

The first crucial step for creating an emulator is the installation of all appropriate packages for your supported API-version. This can easily be done using the SDK-manager which can be found in *Tools* > *Android* > *SDK Manager*.

Speedup

In order to make your emulator faster we recommend the installation of the package *Intel x86 Atom System Image*. Additionally, you should go to *Extras* and install the *Intel x86 Emulator Accelerator*. Eventually, you have to manually install HAXM[1] to get a speedup. The installation file can be found in SDK-folder after downloading the *Intel x86 Emulator Accelerator* package.
Now you can create a new emulator and choose *Intel Atom (x86)* as CPU and check *Use host GPU*. Now your emulator should be much faster than before.

[1] https://software.intel.com/en-us/android/articles/intel-hardware-accelerated-execution-manager

Useful information + Shortcuts

Zeilennummerierung einschalten
Preferences – Editor – Appearance – Show line numbers

Quickfix-Menü
ALT+ENTER
This can be used e.g. for creating a new test case or for automatically including a class

Autocompletion
CTRL+Leertaste

Reformat code
CTRL + ALT + L (Win)
OPTION + CMD + L (Mac)

Generating a new method
ALT + Insert (Win)
CMD + N (Mac)

Build
CTRL + F9 (Win)
CMD + F9 (Mac)

Build and Run
SHIFT + F10 (Win)
CTRL + R (Mac)

Complete list of shortcuts
Mac: http://www.jetbrains.com/idea/docs/IntelliJIDEA_ReferenceCard_Mac.pdf
Win/Linux. http://www.jetbrains.com/idea/docs/IntelliJIDEA_ReferenceCard.pdf