Programmers Documentation

**Android Math1App**

Table of Contents

[1. Introduction 3](#_Toc482179952)

[1.1 Screens 3](#_Toc482179953)

[1.2 Layouts 3](#_Toc482179954)

[activityMain.xml 3](#_Toc482179955)

[1.3 Classes 4](#_Toc482179956)

[Aufgabe 4](#_Toc482179957)

[MainActivity 4](#_Toc482179958)

[Pool 4](#_Toc482179959)

[Statistic 4](#_Toc482179960)

[TestAufgabe 4](#_Toc482179961)

[xmlParser 4](#_Toc482179962)

[1.4 Functions 4](#_Toc482179963)

[2. APK Installation 5](#_Toc482179964)

[2.1 Allow the device to install Unknown Sources Applications 5](#_Toc482179965)

[2.2 Installation 6](#_Toc482179966)

[2.3 Uninstall the App 7](#_Toc482179967)

[3. Mathe1App 8](#_Toc482179968)

[3.1 Introduction Screen 8](#_Toc482179969)

[Mainactivity.onCreate 8](#_Toc482179970)

[1 Displays the Logos 9](#_Toc482179971)

[2 Loads the introduction 9](#_Toc482179972)

[3 Goes to the Next corresponding Task 9](#_Toc482179973)

[4 Goes to the Statistic Screen 10](#_Toc482179974)

[5 Starts from the first task 10](#_Toc482179975)

[3.2 Task Screen 10](#_Toc482179976)

[TestAufgabe.onCreate 11](#_Toc482179977)

[1 Embedded Tasks in “activity\_test\_aufgabe.xml” 12](#_Toc482179978)

[2 Start the Task 12](#_Toc482179979)

[3 Back to the start 13](#_Toc482179980)

[3.3 Statistic Screen 13](#_Toc482179981)

[1. OnTouchScreen 14](#_Toc482179982)

[2. Next Graph 15](#_Toc482179983)

[3. Previous Graph 15](#_Toc482179984)

[4. To the Beginning 15](#_Toc482179985)

[Statistic.onCreate 15](#_Toc482179986)

[Statistic.getViewFwd 15](#_Toc482179987)

[Statistic.evalQtest 17](#_Toc482179988)

[Statistic.nextGraph 17](#_Toc482179989)

[Statistic.lastGraph 18](#_Toc482179990)

[Statistic.correctnessGraph 19](#_Toc482179991)

[Statistic.timeGraph 20](#_Toc482179992)

[Statistic.fellingGraph 21](#_Toc482179993)

[Statistic.back2Start 21](#_Toc482179994)

[Statistic.OnKeyDown 22](#_Toc482179995)

[Statistic.onTouch 22](#_Toc482179996)

[4. Github 22](#_Toc482179997)

[4.1 Install Git 22](#_Toc482179998)

[4.2 Create an account in GitHub 22](#_Toc482179999)

[4.3 Tell Android Studio that the current Project will work with Git 22](#_Toc482180000)

[4.4 Add Github account in Android Studio. 22](#_Toc482180001)

[4.5 Share project in GitHub 22](#_Toc482180002)

[4.6 Clone project 23](#_Toc482180003)

[4.7 Types of collaborative development models 23](#_Toc482180004)

[Fork & pull 23](#_Toc482180005)

[Shared repository model 23](#_Toc482180006)

[4.8 Create an Organization 23](#_Toc482180007)

[4.9 Commit and Push 24](#_Toc482180008)

[Adding a new resource file 25](#_Toc482180009)

[5. Language Control 25](#_Toc482180010)

[6. XML Files 26](#_Toc482180011)

[6.1 Procedure to read the .xml files 26](#_Toc482180012)

[6.1.1 Intro.xml 26](#_Toc482180013)

[6.1.2 Test01.xml 27](#_Toc482180014)

[6.1.3 Test02.xml and Test03.xml 28](#_Toc482180015)

[6.1.4 Pool01.xml and Pool02.xml 29](#_Toc482180016)

[6.2 Saving and reading the score file of the phone 29](#_Toc482180017)

[6.2.1 Saving 29](#_Toc482180018)

[6.2.2 Reading 31](#_Toc482180019)

[7. Change the name of an Android App 33](#_Toc482180020)

[8. ZOOM 36](#_Toc482180021)

# Introduction

1.1 Screens

The Mathe1App application Works with 3 main Screens: Introduction, Tasks and Statistic, the Task screen is divided in 5 different activities: Task Presentation, Task started & Time, Solution Comparison, Self Qualification and Correctness.  
These screens are going to be described later on in this paper, in the section “Mathe1App”.

There are some parameters that are needed for the functionality of this App, these parameters are stored in different XML files, that contains: The Introduction, the text, help and description of each test and the average result of each test.

1.2 Layouts

The Mathe1App works with 8 layouts:

activityMain.xml  
 Works with the Activity MainActivity.java  
  
activity\_statistic.xml  
Works with the Activity Statistic.java  
  
activity\_test\_aufgabe.xml   
Works with the Activity TestAufgabe.java  
  
einschaetzung.xml   
Embedded when needed, in the Activity TestAufgabe.java to display the self-evaluation.  
  
loesung.xml   
 Embedded when needed, in the Activity TestAufgabe.java to display the check box, to verify that the user has compared his answer.  
  
richtigkeit.xml   
Embedded when needed, in the Activity TestAufgabe.java to display the percentage of accuracy to be selected by the user.  
  
timer.xml   
Embedded when needed, in the Activity TestAufgabe.java to display the time to complete the Task.

1.3 Classes   
This App is generated by 6 Classes: Aufgabe, MainActivity, Pool, Statistic, TestAufgabe, XmlParser.

Aufgabe  
 This class contains the necessary fields and constructors to create an Aufgabe Object for each Task performed.

MainActivity   
This class has all the methods for the functionality of the first screen (Main Screen) of the App.

Pool   
This class contains all the necessary fields and constructors to create a Pool Object for each Pool performed.

Statistic   
This Class shows the screen with Statistics for each Task and its exercises, also including the pool Tasks.

TestAufgabe   
 This class contains all the methods for the functionality of the screen that show the different stages of each task (Including the pool Tasks) and their different exercises.

xmlParser   
This class contains all the required methods to parse the different “.xml” files that contains the parameters for desire functionality of the App.

1.4 Functions

The functions in this App where created under the following format:

**public void** FunctionName (Parameters) {  
 *// Coments about the Function* System.***out***.println(**"--> FunctionName"**);

**try**{

Function Body

} **catch** (Exception e) {  
 System.***out***.println(**"ERROR : ClassName.FunctionName --> "** + e);  
 }  
} *// FunctionName*

The first thing that a function does is write its Name in the System.out, so a map of functions used is displayed, so a programmer can see the logic of which function is executed in which time.

Every function has a Try Catch, were in case of an error, the name of the Class is described, followed by the name of the functions + the error. So in this way a programmer can see in the output of the App where exactly an error was created.

At the end of the Function is always commented the name of the function.

# APK Installation

First the file „app-release.apk“ should be download and save in a directory in the Android device.  
There are 3 important fact for the installation:  
  
1. Allow the device to install Unknown Sources Applications.

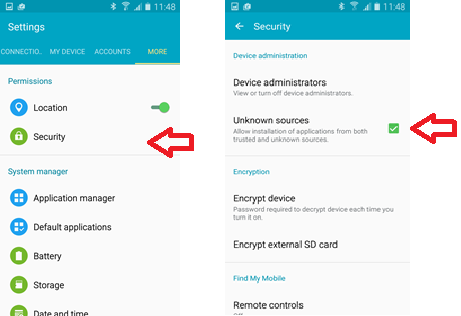
2. Installation.

3. Uninstalling the App.

## 2.1 Allow the device to install Unknown Sources Applications

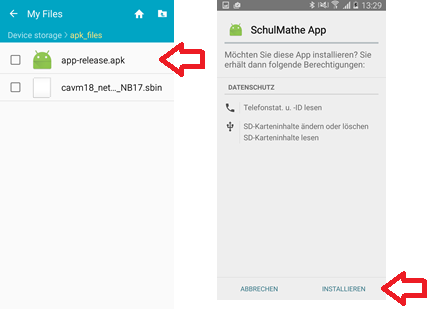
This step is necessary to install an App with an APK file.

* Go to Settings 🡪 Options 🡪 Security
* In the next screen check the option „Unknown Sources“.



## 2.2 Installation

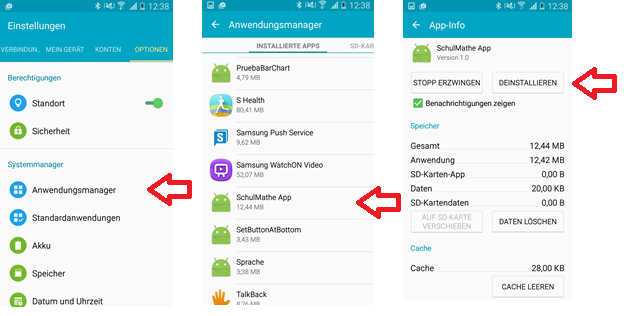
* Go to the path in your device where the APK file is stored and click on it.
* In the next screen press Install. (The App records some information in the device memory, that is why it ask for permissions before the installation).



The image shows the procedure for “SchulMathe App”, but it works in the same way for “Mathe1App”

## 2.3 Uninstall the App

Before installing an update, the last version of the App should be uninstall.  
  
- Go to Settings 🡪 Options 🡪 Application Manager  
- In the „Installed Apps“ list look for „Mathe1App“ and click on it.  
- In the next screen press „Uninstall“



The image shows the procedure for “SchulMathe App”, but it works in the same way for “Mathe1App”

# Mathe1App

This screen is created with the Layout “activity\_main.xml” and the Activity “MainAcitivity.java”.

As mentioned in the introduction, the MatheApp application Works with 3 main Screens: Introduction, Task and Statistic.

## 3.1 Introduction Screen

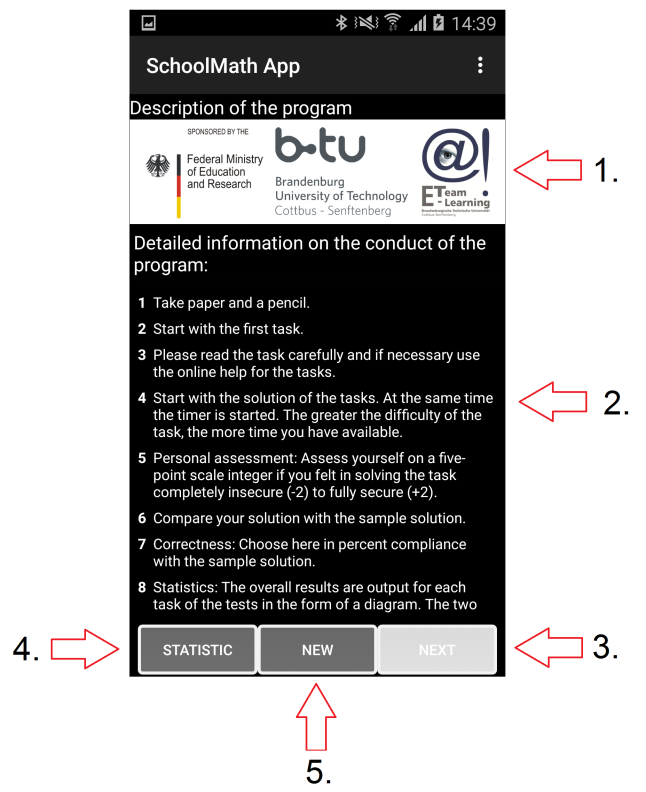


Figure 1. The arrows show the 4 different functionalities of the Main Screen. The image shows the procedure for “SchulMathe App”, but it works in the same way for “Mathe1App”

### Mainactivity.onCreate

MainActivity.onCreate --> MainActivity.activateBtnStatistic --> MainActivity.call2parseIntro

When the Test = 3 and n\_Aufgabe = qntTaskAuf\_3: it means that the last task from the 3rd Task was already evaluated, then the Next button should be set to unable.

MainActivity.ActivateBtnStatistic   
Opens the file “aufgabeValues.xml” and parse it in order to see if there are any values, if there are values it means that the Statistic could be generated when the Button “Statistic” is pressed.

MainActivity.Call2parseIntro  
 Calls the function "XmlParser.parseIntro" that reads the introduction information and place the returned values in the Vector "vecInstruction" then the info of the vector is displayed on the screen.

#### XmlParser.parseIntro

It receives the Context and the name of the file to be parse, the name of the file is declared in a constant in the MainActivity.java file.  
  
**static final int *xmlIntro*** = R.xml.***intro***;

Then it parses the information contained in this file and returns a vector.

### 1 Displays the Logos

The logos are displayed directly in the layout “activity\_main.xml” the name of the image is “logos.png”

### 2 Loads the introduction

Described above in the section “MainActivity.Call2parseIntro”.

### 3 Goes to the Next corresponding Task

In the layout “activity\_Main.xml” is described that when the button “Next” is pressed the function “CallFirstTask” should be executed.

MainActivity.CallFirstTask  
Shows the screen of the next task to be evaluated, for this it calls the intent “activity\_test\_aufgabe”.

MainActivity.CallFirstTask 🡪 MainActivity.parse 🡪 MainActivity.readXMLQntTest

MainAcitivity.parse(contextParse, xml01, n\_Test)

Parse the information from the XML that contains the different Tasks (Aufgaben)

receives the parameters Context, name of the file to parse, and the number of the test to parse. In this case the name of the file to parse is recorded in the constant xml01.

**static final int *xml01*** = R.xml.***test01***;  
  
MainActivity.readXMLQntTest

Reads the values saved in the memory for each task in a .xml file to prepare the quantity of values that the array will get:

aufgabeValues.xml --> saves the values for the normal tests

poolValues.xml --> saves the values for the pool

To read the values from a normal test is uses the function

XmlParser.parseXmlAufQntTest(fin): Receives the File name (fin) to parse and parse it.

And to read the values from a Pool it uses the Function XmlParser.parseXmlPoolQntTest(fin): Receives the File name (fin) to parse and parse it.

### 4 Goes to the Statistic Screen

In the layout “activity\_Main.xml” is described that when the button “Statistic” is pressed the function “ShowStatistic” should be executed.

MainActivity.ShowStatistic   
First it verifies if there is data to show the Statistic, for this it uses the functions (MainActivity.parse, readXMLQntTest) that were already explained above and the functions (MainActivity.parseNextMain, poolParseMain) that are explained below:

MainActivity.parseNextMain *Parse the information from the XML that contains the different Tasks in order to display the average TOP and BOTTOM score of each Statistic*

MainActivity.poolParseMain *Parse the information from the XML that contains the different Pool Tasks in order to display the average TOP and BOTTOM score of each Statistic*

### 5 Starts from the first task

In the layout “activity\_Main.xml” is described that when the button “New” is pressed the function “ClearFiles” should be executed.

mainActivity.clearFiles Clear all the files and variables and then calls the intent “activity\_test\_aufgabe” (The call of the intent is made exactly as the function “callFirstTask”.

## 3.2 Task Screen

This screen is created with the layout “activity\_test\_aufgabe.xml” and the Activity “TestAufgabe.java”.  
The Task screen is divided in 4 different activities: Task Presentation, Task started & Time, Solution Comparison, -Self Qualification, Correctness

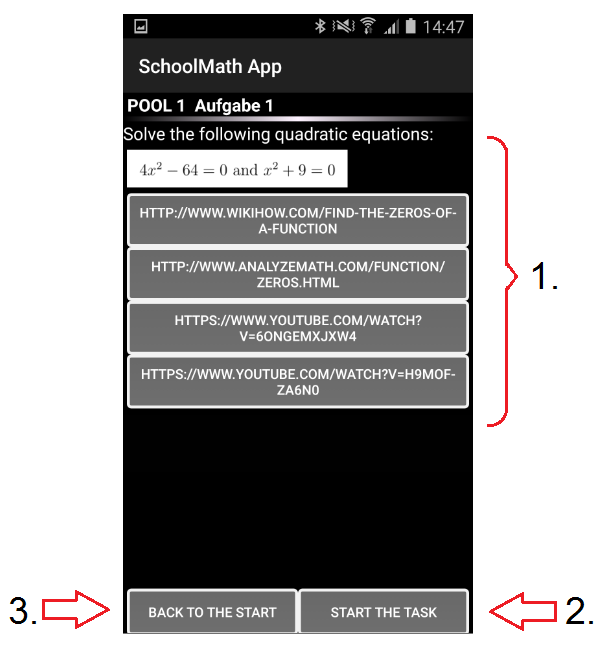


Figure 2. The arrows show the different functionalities from the Task Screen. The image shows the procedure for “SchulMathe App”, but it works in the same way for “Mathe1App”

### TestAufgabe.onCreate

TestAufgabe.onCreate 🡪 TestAufgabe.evaluateAufgabe 🡪 TestAufgabe.readXML 🡪 TestAufgabe.poolVerification

If the next Test to evaluate is a Pool 🡪 TestAufgabe.loadPoolScreen

Else 🡪 TestAufgabe.loadScreen

#### TestAufgabe.evaluateAufgabe

When the score is < than the value of MainActivity.score2pool, goes to the respective pool exercises, when is a new test to be evaluated, resets the variables and parse the respective xml.

If the next Test is a Pool 🡪 TestAufgabe.poolParse

If the next Test is not a Pool 🡪 TestAufgabe.parseNext

If there are no more Tests it goes to the Statistic, calling the intent “Statistic.class”

TestAufgabe.poolParse Parse the information from the XML that contains the different parameters to load the Pool Tasks

TestAufgabe.parseNext Parse the information from the XML that contains the different parameters to load the different tasks from each Test

#### TestAufgabe.readXML

Reads the values saved in the memory (Virtual) in a .xml file

aufgabeValues.xml --> contains the values for the evaluated tests  
poolValues.xml --> contains the values for the evaluated pools

To parse the Tests 🡪 XmlParser.parseXmlAufgabe

To parse the Pool 🡪 XmlParser.parseXmlPool

XmlParser.parseXmlAufgabe Parse the information from the XML that contains the different SAVED Tasks (Aufgaben), so the application can start from the last point

XmlParser.parseXmlPool Parse the information from the XML that contains the different SAVED Tasks (Aufgaben), so the application can start from the last point

#### TestAufgabe.poolVerification

evaluates if at the end of a Test, it should go to the pool or not

#### TestAufgabe.loadScreen

This method loads all the information necessary on screen, to start a new Aufgabe

#### TestAufgabe.loadPoolScreen

This method loads all the information necessary on screen, to start a new Aufgabe

### 1 Embedded Tasks in “activity\_test\_aufgabe.xml”

The Task screen is divided in 4 different activities: Task Presentation, Task started & Time, Solution Comparison, Self Qualification, Correctness

Task Presentation 🡪 The image and the help are loaded in the Screen, according to the values loaded into the Aufgabe class by the function “MainActivity.parse”

The rest of the activities shown in this screen are explain below.

### 2 Start the Task

In the activity\_test\_aufgabe.xml the button “Start the task” is recognized by the name btnAkt2 This button doesn’t has a constant string to display string on the string, so it´s display value is changed in each activity.

The first value is set by the functions loadScreen or LoadPoolScreen, and the value given by both functions is the same:

btnAkt2.setText(R.string.***Lösung\_beginnen***); = “Start the task”

The different values that this button can be found in the function “onClickBtnAkt”.

onClickBtnAct

Displays on the screen the necessary values according to the state (variable ActualZustand) of the current Aufgabe. The states can be:  
  
Task Started (Case 0):

btnAkt2 🡪 “Solution is completed”

Adds the timer to the screen 🡪 R.Layout.timer

Task Performed (Case 1):

btnAkt2 🡪 “To the solution”

Add the personal evaluation to the screen 🡪 R.Layout.einschaetzung

Personal Evaluated (Case 2)

Change the image to show the solution of the Test

btnAkt2 🡪 “Next”

Add the CheckBox to the Screen 🡪 R.layout.Loesung

Answer Compared (Case 3)

btnAkt2 🡪 “Next”

Add the qualifications to the screen 🡪 R.layout.richtigkeit

Evaluated (Case 4)

The next Test or Pool is going to be displayed

### 3 Back to the start

The App returns to the Main Screen of the App.

## 3.3 Statistic Screen

This screen is created with the layout “activity\_statistic.xml” and the Activity “Statistic.java”.

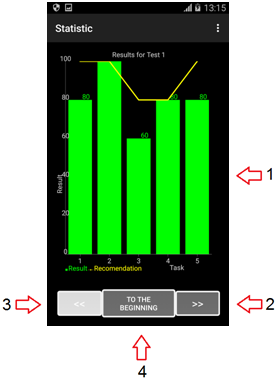


Figure 3. The arrows show the different functionalities from the Statistic Screen.

### OnTouchScreen

When the Correctness Graph is pressed it changes and shows the Time Graph, when the time Graph is pressed it shows the Feeling Graph and When the Feeling Graph is pressed it goes back to the Correctness Graph. (Figure 4)  
  
The Correctness Graph is shown by the function 🡪 Statistic.correctnessGraph

The Time Graph is shown by the function 🡪 Statistic.timeGraph

The Feeling Graph is shown by the function 🡪 Statistic.FellingGraph

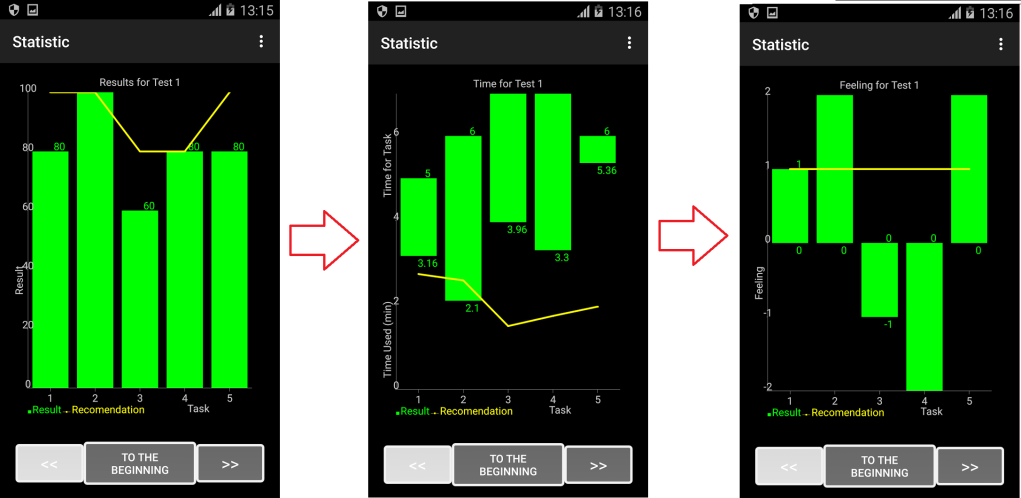


Figure 4: Every time the screen is touched it shows different Statistics. At the beginning it shows the correctness graph (left), when the screen is touched it shows the time graph (middle), when the screen is touched again it shows the feeling graph (right) and if the screen is touched again it goes back to the correctness graph.

### Next Graph

When this button is pressed the graph from the next Test or Pool is displayed, this is done by the function “Statistic.nextGraph”.

### Previous Graph

When this button is pressed the graph from the previous Test or Pool is displayed, this is done by the function “Statistic.lastGraph”.

### To the Beginning

When this button is pressed the App returns to the Main Screen.

### Statistic.onCreate

1. Obtains the first Task to evaluate

*// vecTest contains information like: Test1, Test2, pool2, Test3***nextAufgabe** = MainActivity.*vecTest*.get(**indx\_graph**).toString();

1. Sets the variable statisticData = true to indicate that the statistic was already generated and the button “Statistic” in the main screen can be displayed.

MainActivity.*statisticData* = **true**;

1. displays the first values of the chart on the screen calling the function Statistic.getViewFwd.

**layout** = (LinearLayout) findViewById(R.id.***BarGraph***);  
**gView** = getViewFwd(**this**);  
**layout**.addView(**gView**);

### Statistic.getViewFwd

1. Evaluates how many tasks has the Test to be displayed

**sizeY** = *evalQtest*(**nextAufgabe**);

1. Obtains the Qualification values from the Vector MainActivity.VecQualifikation and display them on the screen.

**while** (**indx\_graph** < **sizeY**) {  
 **j** = MainActivity.*vecQualifikation*.get(**indx\_graph**).toString();  
 **y\_Q\_max**[**indx\_graph**] = Integer.*parseInt*(**j**);  
 **y\_Q\_min**[**indx\_graph**] = 0;  
  
 **indx\_graph**++;  
} *// while*

1. Obtains the Average Qualifications from the Vector MainActivity.vecAvgUc and display them on the screen.

**while** (i2 < **sizeY**) {  
 **j** = MainActivity.*vecAvgUc*.get(i2).toString();  
 **y\_E**[i2] = Integer.*parseInt*(**j**);  
 **x\_E**[i2] = i2+1;  
  
 i2++;  
}

1. Sets the font size according to the definition (Dots per inch “Dpi”)of the device

DisplayMetrics metrics = **new** DisplayMetrics();  
getWindowManager().getDefaultDisplay().getMetrics(metrics);  
**switch**(metrics.**densityDpi**)  
{  
 **case** 640:  
 **mRenderer**.setChartTitleTextSize(50);  
 **mRenderer**.setAxisTitleTextSize(50);  
 **mRenderer**.setLabelsTextSize(50);  
 **mRenderer**.setMargins(**new int**[]{80, 80, 80, 80});  
 **mRenderer**.setLegendTextSize(50); *// Trujillo 07\_03\_2016* **renderer\_E**.setLineWidth(8);  
 **renderer\_Q**.setChartValuesTextSize(50);  
 **timeBlankSpace** = **" "**;  
 **timeBlankSpaceFront** = **" "**;;  
 **break**;  
 **case** 480: *// --> Samsung Galaxy, S4* **mRenderer**.setChartTitleTextSize(40);  
 **mRenderer**.setAxisTitleTextSize(40);  
 **mRenderer**.setLabelsTextSize(40);  
 **mRenderer**.setMargins(**new int**[]{65, 60, 60, 60});  
 **mRenderer**.setLegendTextSize(40); *// Trujillo 07\_03\_2016* **renderer\_E**.setLineWidth(7);  
 **renderer\_Q**.setChartValuesTextSize(40);  
 **timeBlankSpace** = **" "**;  
 **timeBlankSpaceFront** = **" "**;  
 **break**;  
 **case** 320:  
 **mRenderer**.setChartTitleTextSize(30);  
 **mRenderer**.setAxisTitleTextSize(30);  
 **mRenderer**.setLabelsTextSize(30);  
 **mRenderer**.setMargins(**new int**[]{50, 50, 50, 50});  
 **mRenderer**.setLegendTextSize(30); *// Trujillo 07\_03\_2016* **renderer\_E**.setLineWidth(6);  
 **renderer\_Q**.setChartValuesTextSize(30);  
 **timeBlankSpace** = **" "**;  
 **timeBlankSpaceFront** = **" "**;  
 **break**;  
 **case** 240:  
 **mRenderer**.setChartTitleTextSize(20);  
 **mRenderer**.setAxisTitleTextSize(20);  
 **mRenderer**.setLabelsTextSize(20);  
 **mRenderer**.setMargins(**new int**[]{40, 40, 40, 40});  
 **mRenderer**.setLegendTextSize(20); *// Trujillo 07\_03\_2016* **renderer\_E**.setLineWidth(5);  
 **renderer\_Q**.setChartValuesTextSize(20);  
 **timeBlankSpace** = **" "**;  
 **timeBlankSpaceFront** = **" "**;  
 **break**;  
 **case** 213:  
 **mRenderer**.setChartTitleTextSize(10);  
 **mRenderer**.setAxisTitleTextSize(10);  
 **mRenderer**.setLabelsTextSize(10);  
 **mRenderer**.setMargins(**new int**[]{30, 30, 30, 30});  
 **mRenderer**.setLegendTextSize(10); *// Trujillo 07\_03\_2016* **renderer\_E**.setLineWidth(4);  
 **renderer\_Q**.setChartValuesTextSize(10);  
 **timeBlankSpace** = **" "**;  
 **timeBlankSpaceFront** = **" "**;  
 **break**;  
 **case** 160: *//MDPI --> Samsung GT-p5210* **mRenderer**.setChartTitleTextSize(10);  
 **mRenderer**.setAxisTitleTextSize(10);  
 **mRenderer**.setLabelsTextSize(10);  
 **mRenderer**.setMargins(**new int**[]{20, 20, 20, 20});  
 **mRenderer**.setLegendTextSize(10); *// Trujillo 07\_03\_2016* **renderer\_E**.setLineWidth(3);  
 **renderer\_Q**.setChartValuesTextSize(10);  
 **timeBlankSpace** = **" "**;  
 **timeBlankSpaceFront** = **" "**;  
 **break**;  
 **case** 120: *//LDPI --> Samsung galaxz pocket s5310* **mRenderer**.setMargins(**new int**[]{ 20, 10, 20, 10 });  
 **mRenderer**.setLegendTextSize(10); *// Trujillo 07\_03\_2016* **renderer\_E**.setLineWidth(2);  
 **renderer\_Q**.setChartValuesTextSize(10);  
 **timeBlankSpace** = **" "**;  
 **timeBlankSpaceFront** = **" "**;  
 **break**;  
}

1. Sets the TimeGraph\_boolean = true, so the next time the screen is touched it will display the Time Graph, and the Resultgraph\_boolean and feelinggraph\_boolean are set to false to indicate that is not the time to show those screens when a touch screen is made.

**timeGraph\_boolean** = **true**;  
**ResultGraph\_boolean** = **false**;  
**feelingGraph\_boolean** = **false**;

1. Returns the Chart factory with a multiple series type, This type contains the values for the Bar graph and the yellow line to be display.

**types** = **new** String[]{RangeBarChart.***TYPE***, LineChart.***TYPE***};  
**return** ChartFactory.*getCombinedXYChartView*(var\_context, **dataset**, **mRenderer**, **types**);

### Statistic.evalQtest

The vector “vecTest” has a value for each task performed, so, if 5 tasks where made for the Test1, 2 tasks for the Pool1 and 5 tasks for Test2; the vector vecTest would have the following values:  
  
vecTest={Test1, Test1, Test1, Test1, Test1, Pool1, Pool1, Test2, Test2, Test2, Test2, Test2}

So to evaluate the number of tasks that an exercise has, the next loop is performed:  
**while**(vEnumTest.hasMoreElements()) {  
 j = vEnumTest.nextElement().toString();  
  
 **if** (j.equals(h)){  
 count++;  
 }  
} *// while*

### Statistic.nextGraph

This function generates the graph for the next Test or Pool.

1. Evaluates which Test or Pool is going to be displayed next and the quantity of tasks that it contains.

**nextAufgabe** = MainActivity.*vecTest*.get(**indx\_graph**).toString();  
**sizeY** = *evalQtest*(**nextAufgabe**);

1. Obtains the values of the vector vecQualifikation so the correctness graph from the next test could be displayed.

**while** (h < **sizeY**) {  
 **j** = MainActivity.*vecQualifikation*.get(**indx\_graph**).toString();  
 y\_Qfwd\_max[h] = Integer.*parseInt*(**j**);  
 y\_Qfwd\_min[h] = 0;  
  
 **indx\_graph**++;  
 h++;  
} *// while*

1. Obtains the values of the vector vecAVgUc to display the yellow average correctness line.

**while** (h2 < **sizeY**) {  
 **j** = MainActivity.*vecAvgUc*.get(i2).toString();  
 y\_Efwd[h2] = Integer.*parseInt*(**j**);  
 x\_Efwd[h2] = i2+1;  
  
 i2++;  
 h2++;  
} *// while*

1. Sets the TimeGraph\_boolean = true, so the next time the screen is touched it will display the Time Graph, and the Resultgraph\_boolean and feelingGraph\_boolean are set to false to indicate that is not the time to show those screens when a touch screen is made.

**timeGraph\_boolean** = **true**;  
**ResultGraph\_boolean** = **false**;  
**feelingGraph\_boolean** = **false**;

1. Repaints the GraphView, so the new values of the graph can be displayed.

gView.repaint();

### Statistic.lastGraph

This function generates the graph for the previous Test or Pool.

1. Evaluates which Test or Pool was displayed before and the quantity of tasks that it contains.

**nextAufgabe** = **sTests**[**indx\_sTests**];  
**sizeY** = *evalQtest*(**nextAufgabe**);

1. Obtains the values of the vector vecQualifikation so the correctness graph from the next test could be displayed.

**while** (h < **sizeY**) {  
 **j** = MainActivity.*vecQualifikation*.get(**indx\_graph**).toString();y\_Qrew\_max[h] = Integer.*parseInt*(**j**);y\_Qrew\_min[h] = 0;  
  
 **indx\_graph**++;  
 h++;  
} *// while*

1. Obtains the values of the vector vecAVgUc to display the yellow average correctness line.

**while** (h2 < **sizeY**) {  
 *//j = MainActivity.vecEmpfindung.get(i2).toString();* **j** = MainActivity.*vecAvgUc*.get(i2).toString();  
 y\_Erew[h2] = Integer.*parseInt*(**j**);  
 x\_Erew[h2] = i2 + 1;  
  
 i2++;  
 h2++;  
} *// while*

1. Repaints the GraphView, so the new values of the graph can be displayed.

gView.repaint();

1. Sets the TimeGraph\_boolean = true, so the next time the screen is touched it will display the Time Graph, and the Reusltgraph\_boolean and fellinggraph\_boolean are set to false to indicate that is not the time to show those screens when a touch screen is made.

**timeGraph\_boolean** = **true**;  
**ResultGraph\_boolean** = **false**;  
**feelingGraph\_boolean** = **false**;

### Statistic.correctnessGraph

This function returns to the first graph (correctnessGraph) when the screen of the feelingGraph is touched.

1. Gathers the information from vecqualifikation to show the correctness bars.

**while** (h < sizeY\_temp) {  
 **j** = MainActivity.*vecQualifikation*.get(indx\_temp).toString();  
 y\_Corr\_max[h] = Integer.*parseInt*(**j**);  
 *//y\_Corr\_min[h] = h + 1;* y\_Corr\_min[h] = 0;  
  
 indx\_temp++;  
 h++;  
} *// while*

1. Gathers the information from vecAvgUc to show the average correctness with a yellow line.

**while** (h2 < **sizeY**) {  
 **j** = MainActivity.*vecAvgUc*.get(i2).toString();  
 y\_Lim[h2] = Integer.*parseInt*(**j**);  
 x\_Lim[h2] = h2 + 1;  
  
 i2++;  
 h2++;  
} *// while*

1. Repaints the screen with the new information gathered.

gView.repaint();

1. Sets the timeGraph\_boolean to True, so the next time the screen is pressed it will show the time graph.

**timeGraph\_boolean** = **true**;  
**ResultGraph\_boolean** = **false**;  
**feelingGraph\_boolean** = **false**;

### Statistic.timeGraph

When the screen of the correctnessGraph is touched, this function displays on the screen the information of the timeGraph for the current Test or Pool

1. Gathers the information from the vector vecTime and vecTimeTask, and transform it into a valid time so it can be shown on the screen.

**while** (h < sizeY\_temp) {  
 **j** = MainActivity.*vecTime*.get(indx\_temp).toString();  
 j\_temp = Double.*parseDouble*(MainActivity.*vecTimeTask*.get(indx\_temp).toString());posMin = **j**.lastIndexOf(**":"**);  
 sMinutes = **j**.substring(0,posMin);sSeconds = **j**.substring(**j**.lastIndexOf(**":"**) + 1);  
 seconds = ((Integer.*parseInt*(sSeconds))\*100)/60;  
 y\_Time\_Temp = Double.*parseDouble*(sMinutes + **"."** + seconds); *// Trujillo 06\_03\_2016* **if** (y\_Time\_Temp > j\_temp){  
 y\_Time\_Temp = j\_temp - 0.01;  
 }  
 y\_Time\_max[h] = j\_temp; *// Trujillo 06\_03\_2016  
 //x\_Time[h] = h + 1;* y\_Time\_min[h] = y\_Time\_Temp;  
  
 indx\_temp++;  
 h++;  
} *// while*

1. Gathers the information from vecTime4taskUt to show the average time for each exercise with a yellow line.

**while** (h2 < **sizeY**) {  
 indx\_temp = **indx\_graph** - sizeY\_temp;  
 **j** = MainActivity.*vecTime4taskUt*.get(i2).toString();  
 j\_temp = Float.*parseFloat*(MainActivity.*vecTimeTask*.get(indx\_temp).toString());**if** (**j**.equals(**""**)){  
 **j** = **"0.0"**;  
 }  
 y\_Lim[h2] = j\_temp - Double.*parseDouble*(**j**);  
 x\_Lim[h2] = h2 + 1;  
  
 i2++;  
 h2++;  
} *// while*

1. Repaints the screen with the new information gathered.

gView.repaint();

1. Sets the feelingGraph\_boolean to True, so the next time the screen is pressed it will show the feeling graph.

**timeGraph\_boolean** = **false**;  
**ResultGraph\_boolean** = **false**;  
**feelingGraph\_boolean** = **true**;

### Statistic.fellingGraph

When the screen of the timeGraph is touched, this function displays in the screen the information of the feelingGraph for the current Test or Pool

1. Gathers the information from the vector vecEmpfindung to show the feeling graph in the screen.

**while** (h < sizeY\_temp) {  
 **j** = MainActivity.*vecEmpfindung*.get(indx\_temp).toString();y\_Feel\_temp = Integer.*parseInt*(**j**);  
  
 **if** (y\_Feel\_temp > 0) {  
 y\_Feel\_max[h] = y\_Feel\_temp;y\_Feel\_min[h] = 0;  
 } **else** {  
 y\_Feel\_max[h] = 0;  
 y\_Feel\_min[h] = y\_Feel\_temp;  
 }  
  
 **if** (y\_Feel\_temp < 0) {  
 **if** ((y\_Feel\_temp == -1) && (**mRenderer**.getYAxisMin() == 0)) {  
 **mRenderer**.setYAxisMin(-1);  
 } **else if** (y\_Feel\_temp == -2) {  
 **mRenderer**.setYAxisMin(-2);  
 }  
 }  
  
 indx\_temp++;  
 h++;  
} *// while*

1. Gathers the information from vecTime4taskUt to show the average time for this exercise with a yellow line.

**while** (h2 < **sizeY**) {  
 **j** = MainActivity.*vecEmpUf*.get(h2).toString();  
  
 y\_Lim[h2] = Integer.*parseInt*(**j**);  
 x\_Lim[h2] = h2 + 1;  
  
 i2++;  
 h2++;  
} *// while*

1. Repaints the screen with the new information gathered.

gView.repaint();

1. Sets the ResultGraph\_boolean to True, so the next time the screen is pressed it will show the correctness graph.

**timeGraph\_boolean** = **true**;  
**ResultGraph\_boolean** = **true**;  
**feelingGraph\_boolean** = **false**;

### Statistic.back2Start

Returns to the mains screen.

System.*exit*(0);

### Statistic.OnKeyDown

Goes back to the first screen when the android custom back button is pressed. This method is necessary, so the “Back to start” button from our application works in the same way as the custom “back button” from android.

Intent intent = **new** Intent(**this**, MainActivity.**class**);  
**int** xmlScreen = R.layout.***activity\_main***;  
intent.putExtra(***EXTRA\_MESSAGE***, xmlScreen);  
startActivity(intent);  
finish();

### Statistic.onTouch

This method is the one that calls to the necessary function when the screen is touched.

**case** MotionEvent.***ACTION\_DOWN***:  
 **if** (**timeGraph\_boolean**) {  
 **mRenderer**.setYAxisMin(0);timeGraph();  
 } **else if**(**ResultGraph\_boolean**) {  
 **mRenderer**.setYAxisMin(0);  
correctnessGraph();  
 } **else if** (**feelingGraph\_boolean**) {  
 **mRenderer**.setYAxisMin(0);  
fellingGraph();  
 }**break**;

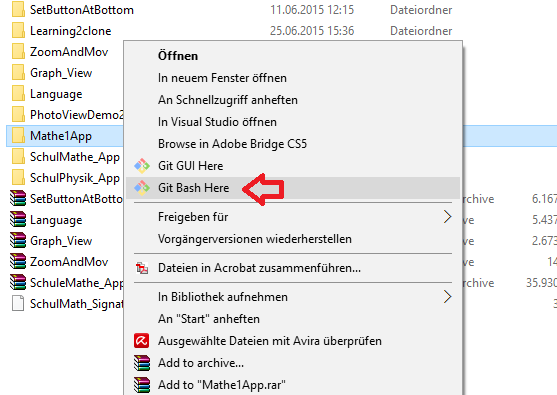
# Github

4.1 Install Git  
The installation file could be found in: git-scm.com

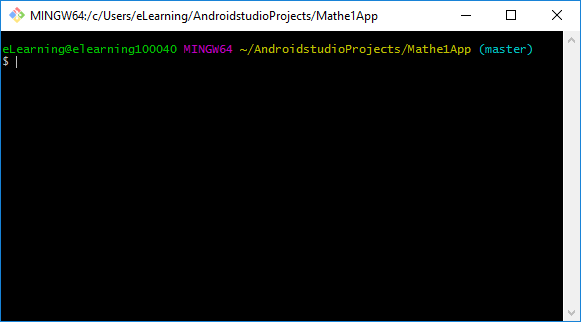
4.2 Create an account in GitHub  
Go to www.github.com and sign up.

4.3 Tell Android Studio that the current Project will work with Git  
VCS 🡪 Enable Version Control Integration

4.4 Add Github account in Android Studio.  
Now use Windows Explorer and navigate to the root of your projects folder. Right click and select ***Git Bash*** (If you do not see this option, then first install [Git for Windows](http://git-scm.com/download/win)).



The next screen will open:



In the Git bash screen type:

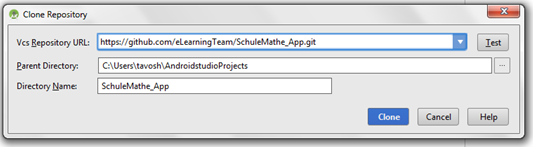
git remote add origin <https://github.com/eLearningTeam/Mathe1App>

Then in Android Studio, go to:

File 🡪 Settings 🡪 Version Control 🡪 GitHub  
  
- Add your Login and Account (The one that was created in www.github.com) in the corresponding fields.  
*NOTE: It is recommended to increase the time “Connection timeout”.*

4.5 Share project in GitHub  
VCS 🡪 Import into Version Control 🡪 Share Project on GitHub  
- Then a Master Password should be added (If it was not added before)  
- In the pop up screen add the name of the repository and a description (optional)  
- Press Share.

4.6 Clone projectVCS 🡪 Checkout from Version Control 🡪 GitHub

* Write the Vcs Repository URL and press Clone
* 

## 4.7 Types of collaborative development models

There are two popular models of collaborative development on GitHub: Fork & pull and shared repository.

Fork & pull  
The fork & pull model let anyone fork an existing repository and push changes to their personal fork without requiring access be granted to the source repository. The changes must then be pulled into the source repository by the project maintainer.

Shared repository model   
The **shared repository model** is more prevalent with small teams and organizations collaborating on private projects. Everyone is granted push access to a single shared repository and topic branches are used to isolate changes.  
  
The “Shared repository model” will be used, so the project can be developed in a team without having someone accepting “the pull requests” from the team.

#### **4.8 Create an Organization** to implement the “Shared repository model”, an organization must be creted:

#### Create a organization:

#### Set up the organization:

#### Invite the Team Members:

## 4.9 Commit and Push

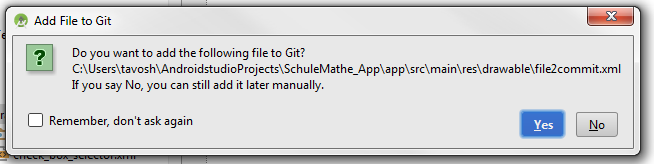
VCS 🡪 Commit Changes…  
- Add a comment in the Commit Message display (Very important to inform the other members of the team about the new changes)

- Press “Commit and Push”: the changes will be commited and then pushed to GitHub.

- (2 steps) Press “Commit”: The changes will be commited, but not pushed to GitHub. To push the changes, go to:  
VCS 🡪 Commit Changes… And this time push the changes.

### Adding a new resource file

For Committing and Pushing a new resource it should be added to Git. When creating a new resource a popup window will ask if the File should be added to Git:



If “Yes” is pressed, the file will be added to Git, but if “No” is pressed, the file can be added later on by:  
Right click on the new resource 🡪 Git 🡪 + Add

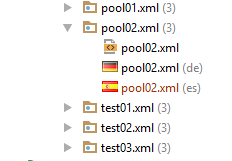
# Language Control

Android identifies the language of the App and it changes the language automatically and reads the values from the File with the information from the other language. The default language is English.

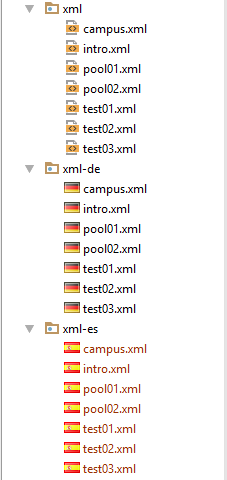
The files that required different language (English or German) in this App are:

* images 🡪 res.drawable
* values 🡪 res.values.strings
* xml files with special parameters for the App 🡪 res.xml

In the next figure, we can see an example of the pool02 and the 3 .xml files that it contents, the three files has the same name, they are just different in the file where Android Studio storage each one:



And in the next figure we can see the way that Android Studio storage these files:



# XML Files

These files are included in res.xml and contain special parameters for the App.

Intro.xml 🡪 Contains the Parameters to display in the Main Screen: Tittle, instructions, Impressum and Contact.

Pool01.xml and Pool02.xml 🡪 Contain the Parameters to display on each Task from the Pool 1 and Pool 2.

Test01.xml, test02.xml and test03.xml 🡪 Contain the Parameters to display on each Task from the Test 01, Test 02 and Test 03.

## Procedure to read the .xml files

### 6.1.1 Intro.xml

In the class **MainActivity.java** in the method onCreate, the function **call2parseIntro** is called.

This function displays on the screen what is written in the file **intro.xml**.  
 For example:

*1 Take paper and pencil.*

In the Layout **activity\_main.xml** the textView **Step1Title** displays the number “1” mentioned above and the textView **step1** displays the text “Take paper and pencil” and there are to text view for the other steps.  
  
Then to know the name of the Step Title “1” we use the variable **strsStep** and know the text that goes according this number, we use the variable **strStepTitle**, this variables are inside a loop (while), so they get values from each ID in the **intro.xml**.

**while**(vEnum.hasMoreElements()) {  
 strStep = **"step"** + cont;  
 strSepTitle = **"step"** + cont + **"Title"**;

This collects the number of the ID and creates the name of the variable, then knowing the name we can look for this variable en the layout **activity\_main.xml**:

nextStep = getResources().getIdentifier(strStep , **"id"**, getPackageName());  
 nextStepTitle = getResources().getIdentifier(strStepTitle , **"id"**, getPackageName());

and now that we have found the variable that we want to feel, we feel it with the next code:

textTitle.setText(vEnumTitle.nextElement().toString() + **" "**);  
 textStep.setText(vEnum.nextElement().toString());

after this we increment the variable cont and the loop begins again, so it can display the next step:

*2 Start with the first task.*

### 6.1.2 Test01.xml

This file contains the information of the first Task (Exercises, time for the tasks and text to display), when the **Continue** Button is pressed, the function **CallFirstTask** is called. First it initializes the Test to 1 and the Task to 0, to be sure that it will begin from the first task. The same code is also inside the function **ClearFiles**, which is called when the button new is pressed, first it cleans all the variables and then it proceeds also with the next code:

*n\_Aufgbe* = 0; *//the first Aufgabe is going to be evaluated  
 n\_Test* = 1; *// the first test is going to be evaluated*

The constant **xml01** is declared, so if the name of the package is changed, then it just has to be change in this line:

**static final int *xml01*** = R.xml.***test01***;

The file **xml01** is read calling the function **parse**

Context contextParse = **this**;  
 parse(contextParse, ***xml01***, *n\_Test*);

#### MainActivity.Parse

This function receives as parameters: The context, the name of the file that is going to be read and the number of the Test that is going to be parsed. This method reads the file until it reach the end of document, and every time it finds a start tag, for example <**time**> it records the content of this tag into the corresponding variable:

**case** XmlPullParser.***START\_TAG***:  
 *currentTag* = parser.getName();  
  
 **if** (*currentTag*.equalsIgnoreCase(***AUFGABE***)) {  
 *currentAufgabe* = **new** Aufgabe();  
 aufgabeNumber++;  
 *currentAufgabe*.setAufgabe(aufgabeNumber);  
  
 } **else if** (*currentAufgabe* != **null**) {  
 **if** (*currentTag*.equalsIgnoreCase(***BILD***)) {  
 *currentAufgabe*.setImageAufgabe(parser.nextText());  
  
 } **else if** (*currentTag*.equalsIgnoreCase(***HILFE***)) {  
 *helpOn* = **true**;  
 eventType = parser.next();  
 *currentTag* = parser.getName();  
  
 *// Loop to read the different values that the tag HILFE could have* **while** (*currentTag*.equalsIgnoreCase(***VALUE***)) {  
 *helpValue* = parser.nextText();  
  
 **if** (*helpValue*.length() > 0){  
 *currentAufgabe*.setHilfe(*helpValue*);  
 }  
  
 eventType = parser.next();  
 *currentTag* = parser.getName();  
 } *// end while* } **else if** (*currentTag*.equalsIgnoreCase(***LOESUNG***)) {  
 *currentAufgabe*.setImageLoesung(parser.nextText());  
 } **else if** (*currentTag*.equalsIgnoreCase(***TEXT***)) {  
 *currentAufgabe*.setText(parser.nextText());  
 } **else if** (*currentTag*.equalsIgnoreCase(***TIME***)) {  
 **timeTemp** = parser.nextText(); *//Trujillo 06\_03\_2016  
 currentAufgabe*.setTime(**timeTemp**); *//Trujillo 06\_03\_2016  
 vecTimeTask*.add(**timeTemp**); *//Trujillo 06\_03\_2016* } **else if** (*currentTag*.equalsIgnoreCase(***AVGUC***)) {  
 *vecAvgUc*.add(parser.nextText());  
 } **else if** (*currentTag*.equalsIgnoreCase(***AVGDC***)) {  
 } **else if** (*currentTag*.equalsIgnoreCase(***AVGUF***)) {  
 *vecEmpUf*.add(parser.nextText());  
 } **else if** (*currentTag*.equalsIgnoreCase(***AVGDF***)) {  
 } **else if** (*currentTag*.equalsIgnoreCase(***AVGUT***)) {  
 *vecTime4taskUt*.add(parser.nextText());  
 } **else if** (*currentTag*.equalsIgnoreCase(***AVGDT***)) {  
 } *// if* } *// if* **break**;

### 6.1.3 Test02.xml and Test03.xml

This both test are called in the same way. This files are also declared with a constant name, so it will be easier to manipulate in the code:

**static final int *xml02*** = R.xml.***test02***;

**static final int *xml03*** = R.xml.***test03***;

They are read in the class **TestAufgabe.java** in the function **poolVerification\_pre** and **evaluateAufgabe**, they are analyzed in these two functions because **Test02** could occur after **pool01** or after **Test01**. Also **Test03** could occur after **Pool02** or after **Test02**. To parser these file the function **parseNext** is called:

parseNext(MainActivity.***xml02***, 2);

or for Test03

parseNext(MainActivity.***xml03***, 3);

#### TestAufgabe.parseNext()

This function works exactly like the function **MainActivy.parse**, the same function could not be used, because one of the parameters that it requires is the Context and it gave a lot of complications when the Context is send to another Intend.

### 6.1.4 Pool01.xml and Pool02.xml

Also declared as constants:

**static final int *xmlPool01*** = R.xml.***pool01***;  
 **static final int *xmlPool02*** = R.xml.***pool02***;

These two files are read in with the function **poolVerification\_pre()** which analyze if there is a pool to display and which pool should be displayed. To call to this method we use:

poolParse(MainActivity.***xmlPool01***, 1);

and for the pool02

poolParse(MainActivity.***xmlPool02***, 2);

#### TestAufgabe.poolParse()

This function works with the same parameters as the method **TestAufgabe.parseNext()** described above, it also uses the same parameters.

## 6.2 Saving and reading the score file of the phone

The next constants are declared for a better management of the files during the run process.

**public static** String *aufgabeValues* = **"math1\_aufgabeValues.xml"**;  
**public static** String *poolValues* = **"math1\_poolValues.xml"**;

### 6.2.1 Saving

The saving is done in the method **TestAufgabe.onClickBtnAkt()**. When the score is from the pool, it stores the score in the vector **aufPoolLoad** and if it is a Test it is recorded in the vector **aufLoad**.

*// Set the qualification of the current task/Pool and saves the info in the XML* **if** (MainActivity.*poolActivated*) {  
 MainActivity.*aufPoolLoad*.setQualifikation(*intZahl1*);  
 writePoolXML();  
 } **else** {  
 MainActivity.*aufLoad*.setQualifikation(*intZahl1*);  
 writeAufgabeXML();  
 }

As we can see in this code, when the pool is activated, it calls the function **WritePoolXML()** which records the score in the .XML that is saved in our phone, the same behavior has the method **writeAufgabeXML** for the Tests.

#### writePoolXML()

The name of the file where the scores are saved for the Pool is **math1\_poolValues.xml** it is created in the code with the next lines:

FileOutputStream fOutCntxt = **null**;  
fOutCntxt = context.openFileOutput(**"poolValues.xml"**, Context.***MODE\_PRIVATE***);  
  
File myFile = **new** File(**"/sdcard/poolValues.xml"**); *// Trujillo 07\_03\_2016*myFile.createNewFile();  
FileOutputStream fOut = **new** FileOutputStream(myFile);

Then the head of the xml file is created:

myOutWriter.append(**"<?xml version=\"1.0\" encoding=\"utf-8\"?>\n"**);  
myOutWriter.append(**"<Poolaufgaben>\n"**);

Then the xml file is filled with a for cycle that runs until it reach the quantity of Task to evaluate:

**for**(**int** x=0; x < MainActivity.*poolAfgb2Eval*.size(); x++) {  
 **if** (x <= MainActivity.*n\_poolQntAufEvl*) {  
 Pool poolTemp = MainActivity.*poolAfgb2Eval*.get(x);  
 **int** test = poolTemp.getPoolTest();  
 **int** aufgabe = poolTemp.getPoolAufgb();  
 String timeRequired = poolTemp.getTimeRequired();  
 **int** zustand = poolTemp.getZustand();  
 **int** qualifikation = poolTemp.getQualifikation();  
 **int** empfindung = poolTemp.getEmpfindung();  
  
 myOutWriter.append(**"\t<test id=\""** + test + **"\">\n"**);  
 myOutWriter.append(**"\t\t<aufgabe>"** + aufgabe + **"</aufgabe>\n"**);  
 myOutWriter.append(**"\t\t<timeRequired>"** + timeRequired + **"</timeRequired>\n"**);  
 myOutWriter.append(**"\t\t<zustang>"** + zustand + **"</zustang>\n"**);  
 myOutWriter.append(**"\t\t<qualifikation>"** + qualifikation + **"</qualifikation>\n"**);  
 myOutWriter.append(**"\t\t<empfindung>"** + empfindung + **"</empfindung>\n"**);  
 myOutWriter.append(**"\t</test>\n"**);  
 }  
} *// for*

and at the end, the last line to complete the xml format should be added:

myOutWriter.append(**"</Poolaufgaben>\n"**);

#### writeAufgabeXML()

It works the same as the method **writePoolXML()** but it writes it’s content in the file **math1\_aufgabeValues.xml**:

FileOutputStream fOutCntxt = **null**;  
fOutCntxt = context.openFileOutput(**"aufgabeValues.xml"**, Context.***MODE\_PRIVATE***);  
OutputStreamWriter myOutWriter = **new** OutputStreamWriter(fOutCntxt);  
  
File myFile = **new** File(**"/sdcard/aufgabeValues.xml"**);  
myFile.createNewFile();  
FileOutputStream fOut = **new** FileOutputStream(myFile);  
  
myOutWriter.append(**"<?xml version=\"1.0\" encoding=\"utf-8\"?>\n"**);  
myOutWriter.append(**"<aufgaben>\n"**);  
  
  
**for**(**int** x=0; x < MainActivity.*aufgb2Eval*.size(); x++) {  
 **if** (x <= MainActivity.*n\_QntityAufEval*) {  
 Aufgabe aufTemp = MainActivity.*aufgb2Eval*.get(x);  
 **int** test = aufTemp.getTest();  
 **int** aufgabe = aufTemp.getAufgabe();  
 String timeRequired = aufTemp.getTimeRequired();  
 **int** zustand = aufTemp.getZustand();  
 **int** qualifikation = aufTemp.getQualifikation();  
 **int** empfindung = aufTemp.getEmpfindung();  
  
 myOutWriter.append(**"\t<test id=\""** + test + **"\">\n"**);  
 myOutWriter.append(**"\t\t<aufgabe>"** + aufgabe + **"</aufgabe>\n"**);  
 myOutWriter.append(**"\t\t<timeRequired>"** + timeRequired + **"</timeRequired>\n"**);  
 myOutWriter.append(**"\t\t<zustang>"** + zustand + **"</zustang>\n"**);  
 myOutWriter.append(**"\t\t<qualifikation>"** + qualifikation + **"</qualifikation>\n"**);  
 myOutWriter.append(**"\t\t<empfindung>"** + empfindung + **"</empfindung>\n"**);  
 myOutWriter.append(**"\t</test>\n"**);  
 }  
} *// for*

myOutWriter.append(**"</aufgaben>\n"**);

### 6.2.2 Reading

When the button Intent **TestAufgabe.java** is loaded, it verifies if it should start from the beginning or if there are saved data to display. This is done by the function **poolVerifcation** with the following code:  
  
**try** {  
 **if** (MainActivity.*startsFromSavedInfo*) {

When it starts from saved info, it verifies which Tests and pools have been already performed and after it calls the function **readXML()**, which calls to the specific parser (Test or Pool) to load the save data into the according variables:

**if** (MainActivity.*startsFromSavedInfo*) {  
 fin = openFileInput(**"aufgabeValues.xml"**);  
 XmlParser.*parseXmlAufgabe*(fin);  
 fin.close();  
}  
  
**if** (MainActivity.*startsFromSavedPoolInfo*) {  
 fin = openFileInput(**"poolValues.xml"**);  
 XmlParser.*parseXmlPool*(fin);  
 fin.close();  
}

#### Function XmlParser.parseXmlAufgabe()

This function works just as the other parsers works, it detects when a start tag begins and depending on the tag it records the necessary information in the corresponding variables.

**case** XmlPullParser.***START\_TAG***:  
 *currentTag* = parser.getName();  
  
 **if** (*currentTag*.equalsIgnoreCase(***TEST***)) {  
 testNum = Integer.*parseInt*(parser.getAttributeValue(0));  
 *currentAufgabe* = MainActivity.*aufgb2Eval*.get(counter); *// Trujillo 03/05/2016  
 currentAufgabe*.setTest(testNum);  
  
 } **else if** (*currentTag* != **null**) {  
 **if** (*currentTag*.equalsIgnoreCase(***AUFGABE***)) {  
 aufNum = Integer.*parseInt*(parser.nextText());  
 *currentAufgabe*.setAufgabe(aufNum);  
 *//\_/1 MainActivity.n\_Aufgbe = aufNum;  
 //MainActivity.n\_QntityAufEval++;  
  
 // Data to create the statistic Graph* MainActivity.*vecTest*.add(**"Test "** + testNum);  
 } **else if** (*currentTag*.equalsIgnoreCase(***TIMEREQUIRED***)) {  
 timeAuf = parser.nextText();  
 *currentAufgabe*.setTimeRequired(timeAuf);  
 *//Data for the Statistic* MainActivity.*vecTime*.add(timeAuf);  
 } **else if** (*currentTag*.equalsIgnoreCase(***ZUSTANG***)) {  
 *currentAufgabe*.setZustand(Integer.*parseInt*(parser.nextText()));  
 } **else if** (*currentTag*.equalsIgnoreCase(***QUALIFIKATION***)) {  
 **int** intQual = Integer.*parseInt*(parser.nextText());  
 *currentAufgabe*.setQualifikation(intQual);  
 MainActivity.*sumQualfktion* = MainActivity.*sumQualfktion* + intQual;  
 *//Data for the Statistic* MainActivity.*vecQualifikation*.add(intQual);  
 } **else if** (*currentTag*.equalsIgnoreCase(***EMPFINDUNG***)) {  
 empAuf = Integer.*parseInt*(parser.nextText());  
 *currentAufgabe*.setEmpfindung(empAuf);  
 *//Data for the Statistic* MainActivity.*vecEmpfindung*.add(empAuf);  
 } *// if* } *// if* **break**;

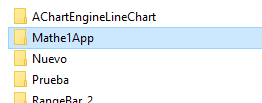
#### XmlParser.parseXmlPool()

It works the same as the **XmlParser.parseXmlAufgabe()** but this function reads the values of the pools:

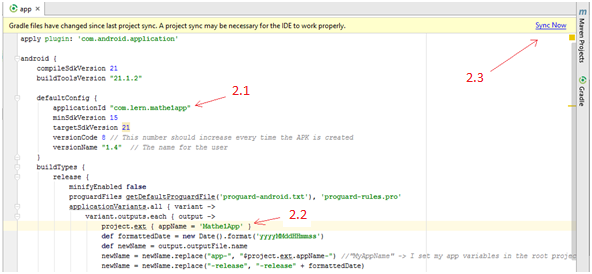
**case** XmlPullParser.***START\_TAG***:  
 *currentTag* = parser.getName();  
  
 **if** (*currentTag*.equalsIgnoreCase(***TEST***)) {  
 testNum = Integer.*parseInt*(parser.getAttributeValue(0));  
 *currentPool* = MainActivity.*poolAfgb2Eval*.get(counter); *// Trujillo 03/05/2016  
 currentPool*.setPoolTest(testNum);  
 } **else if** (*currentTag* != **null**) {  
 **if** (*currentTag*.equalsIgnoreCase(***AUFGABE***)) {  
 poolNum = Integer.*parseInt*(parser.nextText());  
 *currentPool*.setPoolAufgb(poolNum);  
 *// Data to create the Graph* MainActivity.*vecTest*.add(**"Pool "** + testNum);  
 } **else if** (*currentTag*.equalsIgnoreCase(***TIMEREQUIRED***)) {  
 timeAuf = parser.nextText();  
 *currentPool*.setTimeRequired(timeAuf);  
 *//Data for the Statistic* MainActivity.*vecTime*.add(timeAuf);  
 } **else if** (*currentTag*.equalsIgnoreCase(***ZUSTANG***)) {  
 *currentPool*.setZustand(Integer.*parseInt*(parser.nextText()));  
 } **else if** (*currentTag*.equalsIgnoreCase(***QUALIFIKATION***)) {  
 **int** intQual = Integer.*parseInt*(parser.nextText());  
 *currentPool*.setQualifikation(intQual);  
 *//Data for the Statistic* MainActivity.*vecQualifikation*.add(intQual);  
 } **else if** (*currentTag*.equalsIgnoreCase(***EMPFINDUNG***)) {  
 empAuf = Integer.*parseInt*(parser.nextText());  
 *currentAufgabe*.setEmpfindung(empAuf);  
 *//Data for the Statistic* MainActivity.*vecEmpfindung*.add(empAuf);  
 } *// if* } *// if* **break**;

# Change the name of an Android App

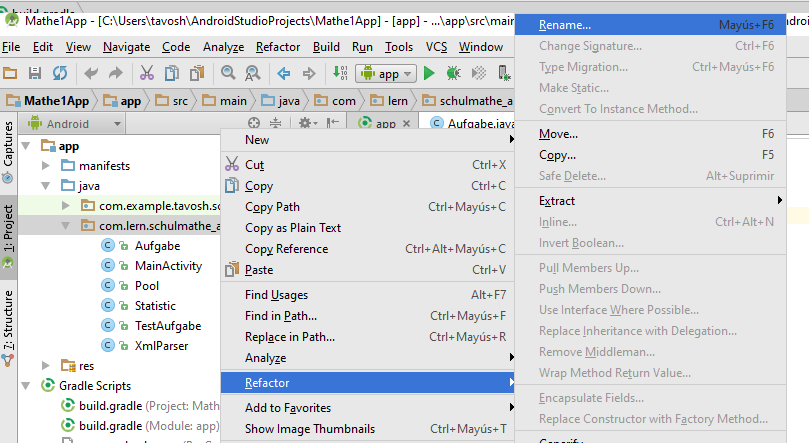
1. Change the name of the file, in this case the new name will be “Mathe1App”



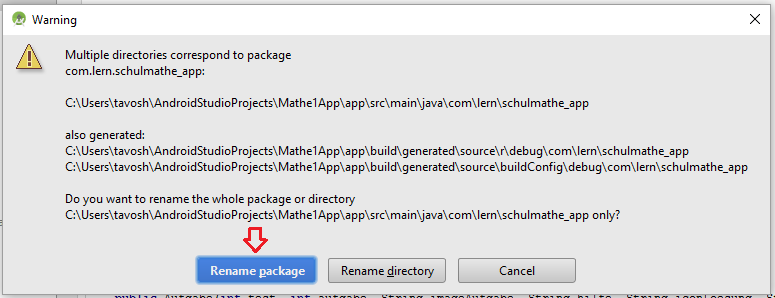
1. in the build.grade(Module: app) File:
   1. In build.gradle change the applicationId to the new name of the App. (everything in lowercase)
   2. Also in the build.gradle the project.ext should be change. (this is the name that the APK package will have).
   3. Sync Now



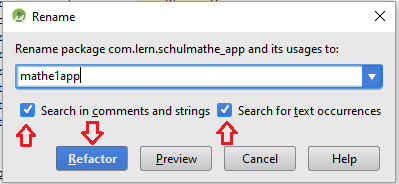
1. In the file that contains all the classes press right click: Refactor 🡪 Rename



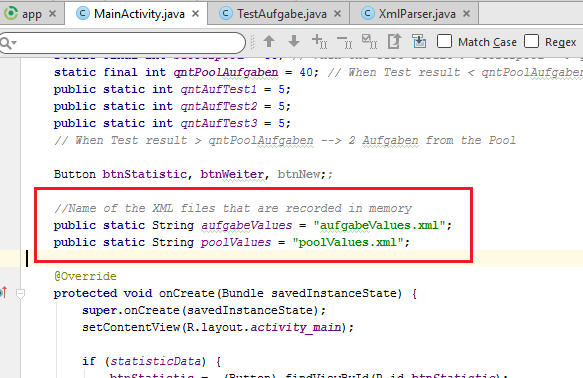
* 1. In the next window select Rename Package



3.2 In the next window, change the name of the App, then check the two checkboxes that are in the window, and then press Refactor.



1. In MainActivity.java change the Values of the Strings aufgabeValues and poolValues



1. Change the name of the app in the Strings file. (For German and for English)

# ZOOM

To get the Zoom functionality we used the library located in “**com.github.chrisbanes:PhotoView:1.2.6**” in order to make it work we configure it in the file build.gradle(Project:Mathe1App), adding the following code in the repositories:

maven {  
 url **"https://jitpack.io"** }

And in the file build.gradle(Module:app) The following code should be declared in the dependencies area:

compile **'com.github.chrisbanes:PhotoView:1.2.6'**

Having this we can use the Zoom functionality, which is needed in the file **TestAufgabe.java**, where we have to add the following code:

PhotoViewAttacher photoView = **new** PhotoViewAttacher(**image**);  
 photoView.update();

In the next methods:

* At the end of OnCreate.
* loadPoolScreen, after changing the image to display (also after change the image in the exception).
* At the end of onClickBtnAkt.