Introduction

First steps

Contents

Part 1

Creating a new project

Create a new Android project with Android Studio. You can use the welcome window (Figure 1) of Android Studio. Select “Start a new Android Studio project”.

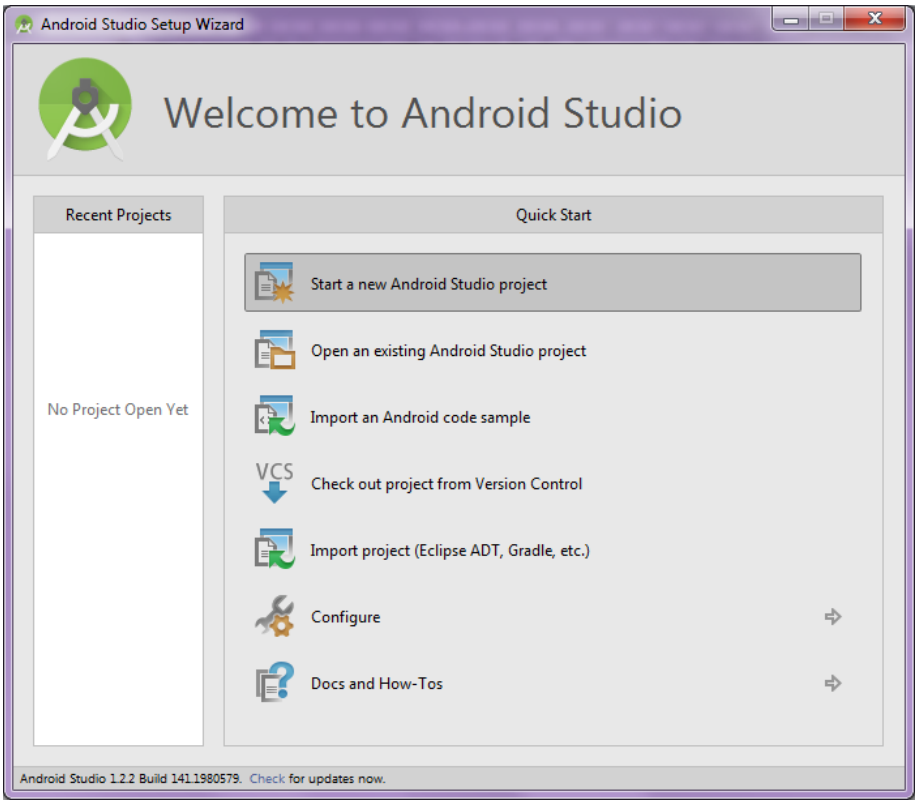


Figure 1

If this window is not visible, for creating a new project go to File -> New project.

We have to insert the application and package name (Figure 2); and the project location has to be selected.

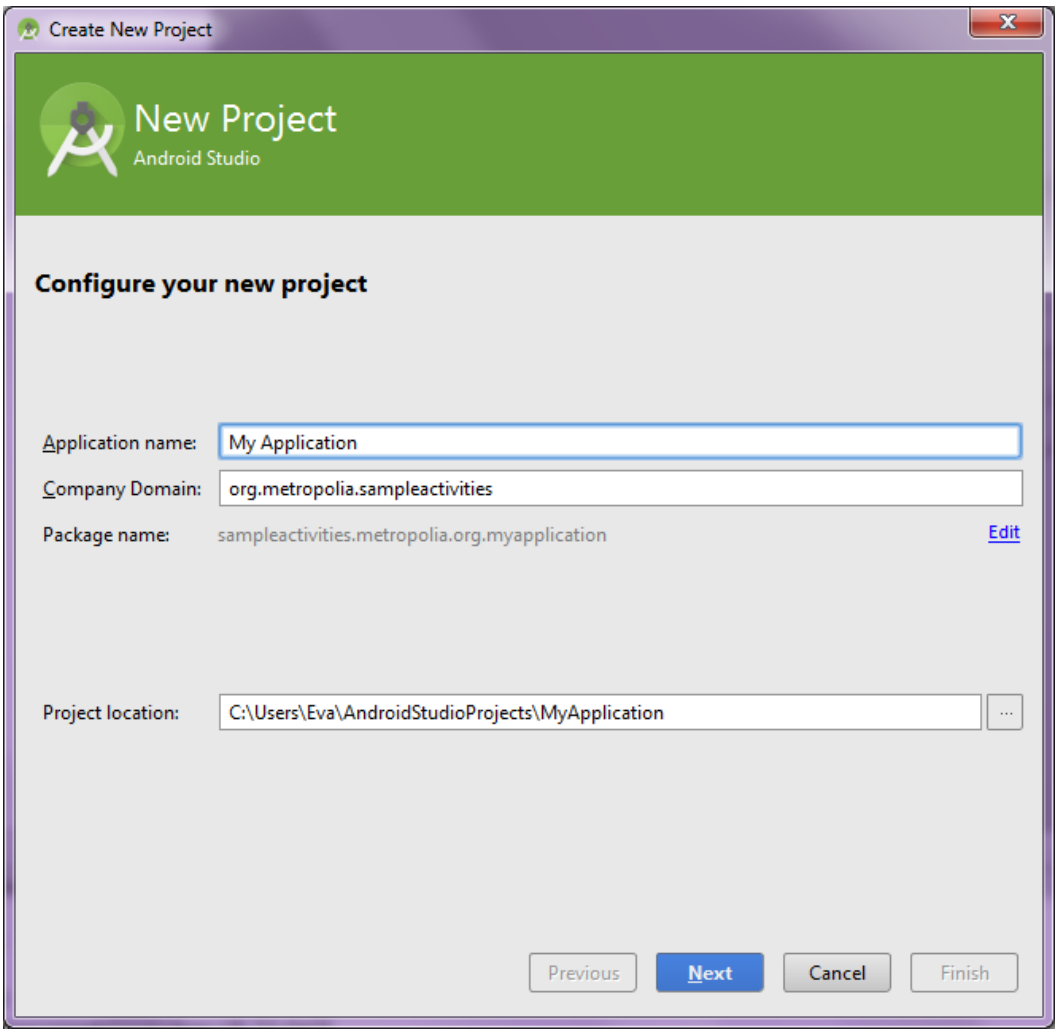


Figure 2

In the next screen we can choose the kind of application (Figure 3): phone and tablet, TV, wear, etc. In this window we can select the version of Android for which we want to compile the project.

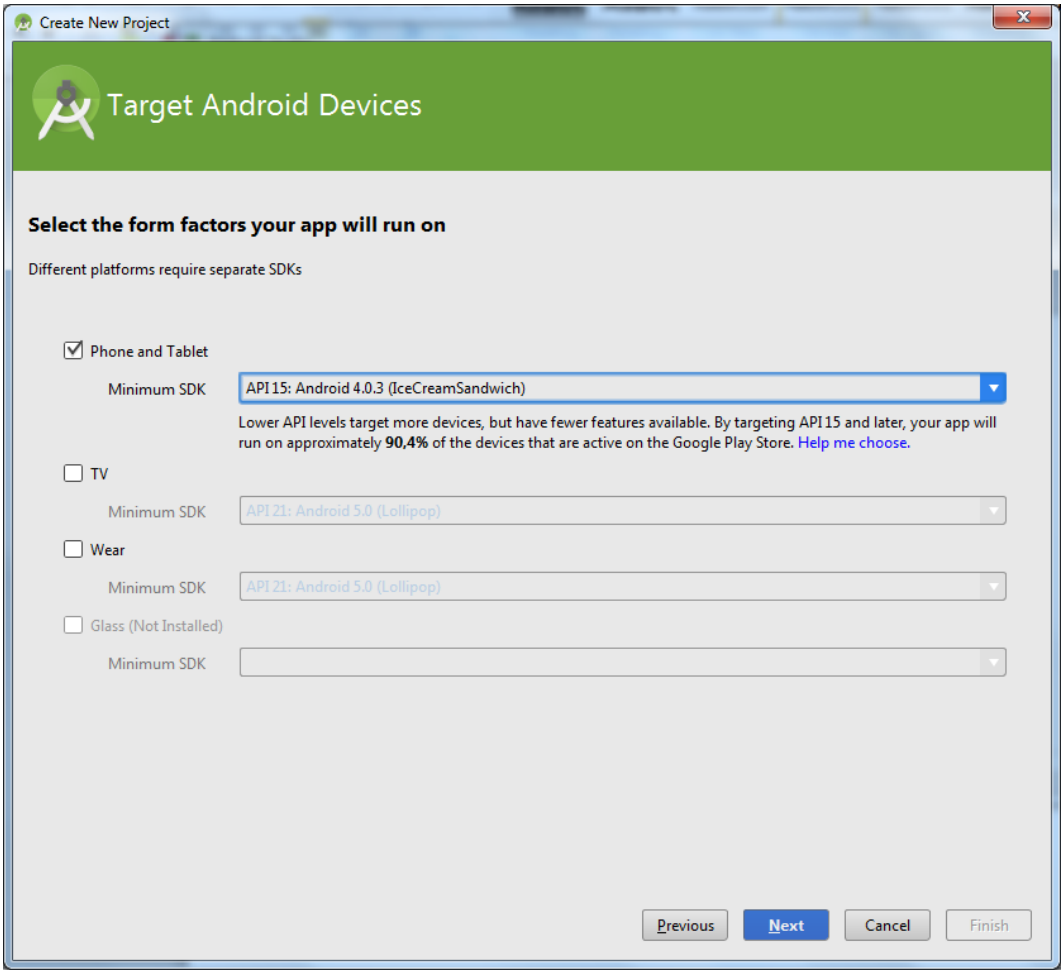

Create New Project
Target Android Devices
Select the form factors your app will run on
Different platforms require separate SDKs
Phone and Tablet
Minimum SDK
API 15: Android 4.0.3 (IceCreamSandwich)
Lower API levels target more devices, but have fewer features available. By targeting API 15 and later, your app will run on approximately 90,4% of the devices that are active on the Google Play Store. Help me choose.
TV
Minimum SDK
API 21: Android 5.0 (Lollipop)
Wear
Minimum SDK
API 21: Android 5.0 (Lollipop)
Glass (Not Installed)
Minimum SDK
Previous
Next
Cancel
Finish

Figure 3

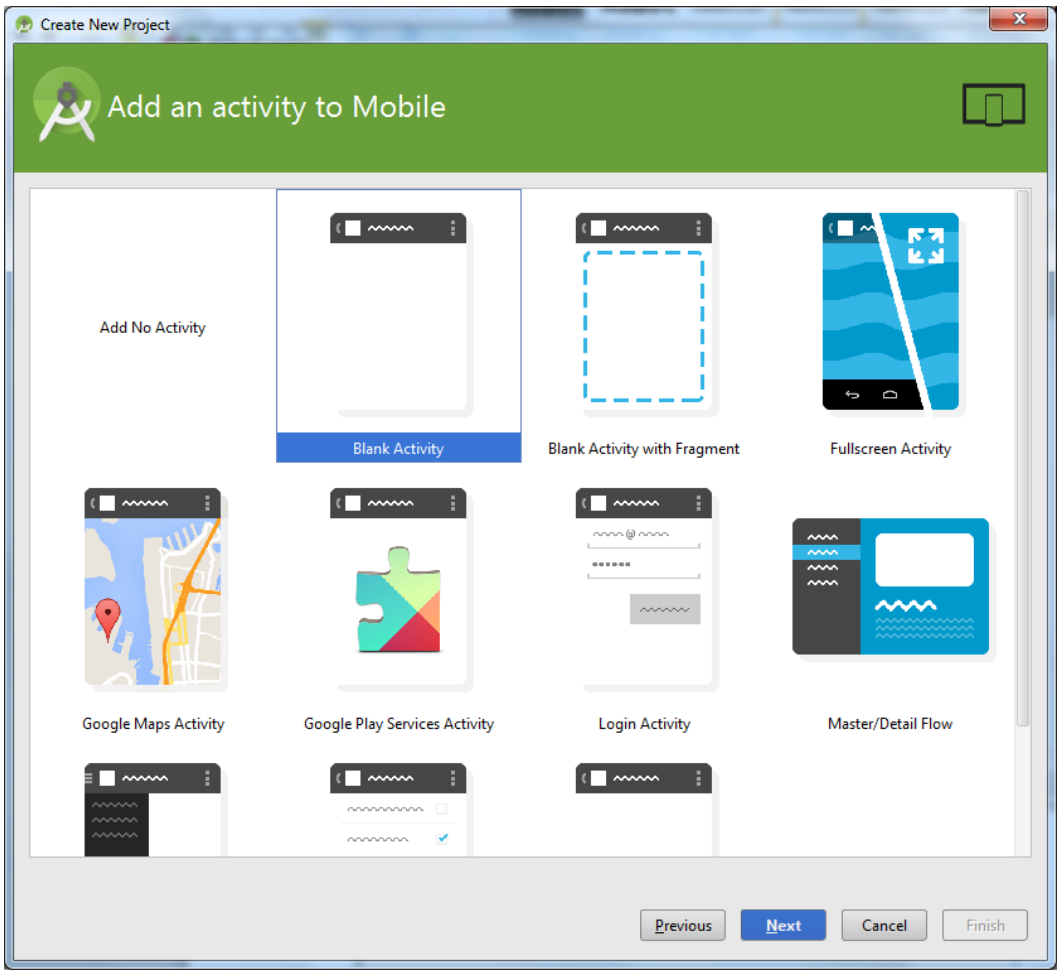


Figure 4

In the next screen (Figure 4) we can select the kind of the first activity that will be automatically created with the project. At this moment we select “Blank Activity”.

Finally, the activity and layout names have to be inserted in the next screen (Figure 5). It is recommended to keep the name “MainActivity” for the first activity of our app.

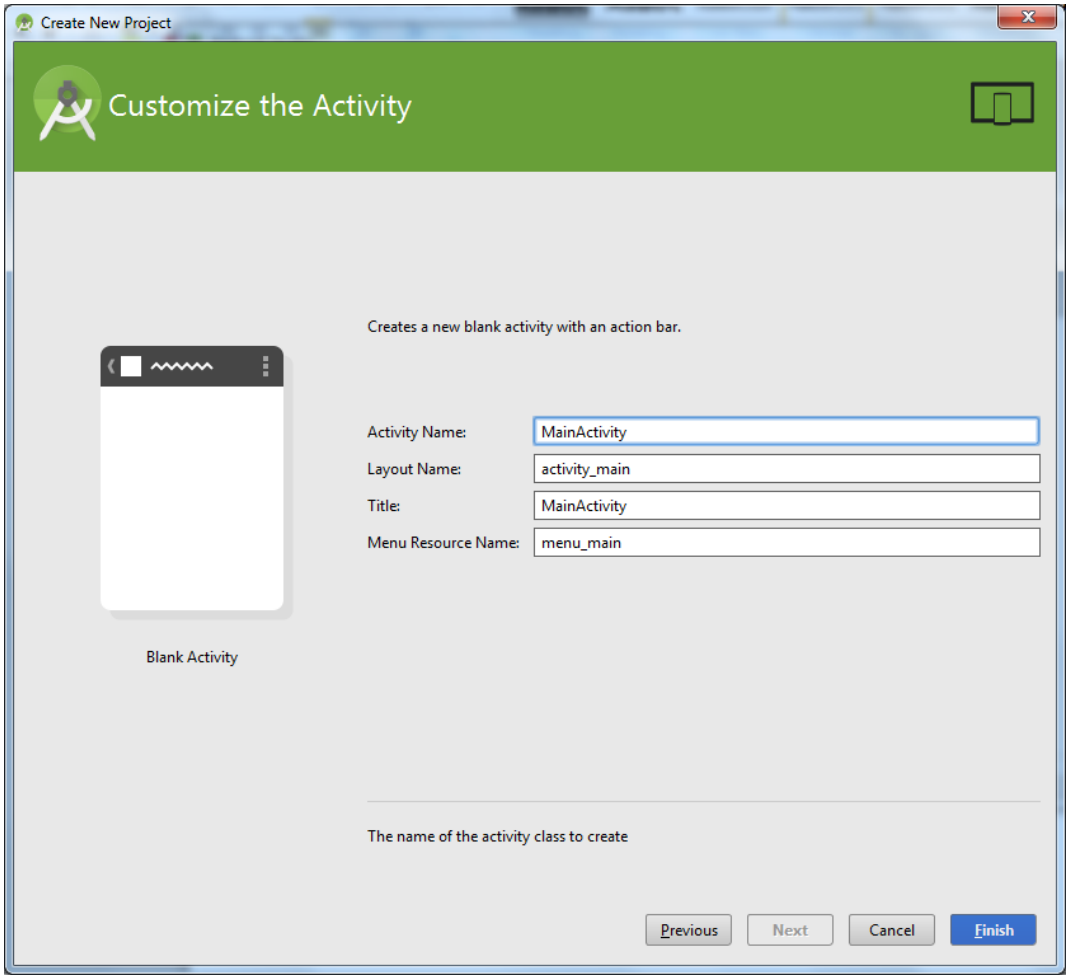


Figure 5

Our first screen

Editing the graphical interface file: activity_main.xml

We have to edit the graphical interface file *activity_main.xml*. It is located in *app -> res -> layout -> activity_main.xml* (Figure 6)

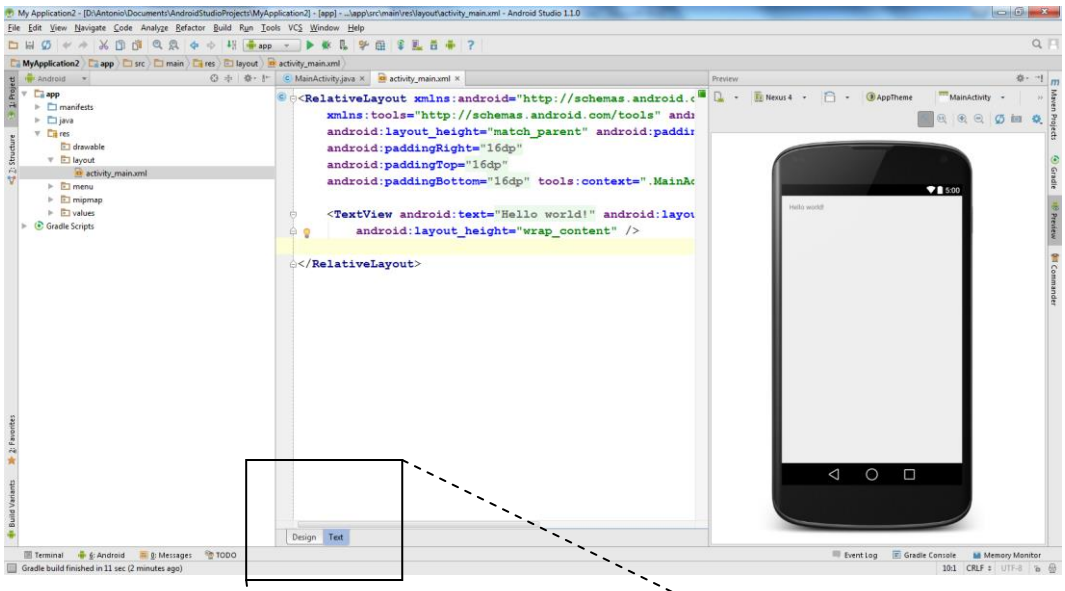


Figure 6

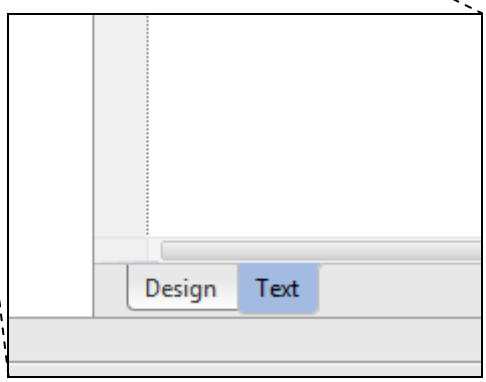


Figure 7

We will create the first screen of our application, that is, the screen that will appear when the user runs the app. For this, we will use a LinearLayout, which allows putting the elements (text fields, buttons, etc.) in a row or in a column (i.e., horizontally or vertically).

In the "Design" tab (Figure 7) you can see a preview of the *activity_main.xml* screen, but for editing the screen it is recommended to use the other tab ("Text"). Thus, we can create the screens by using XML language.

Then we delete the default code and rewrite the next code:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical">

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />
    <TextView
```

```
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/text2" />
    <Button android:id="@+id/NewButton"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/textButton" />

</LinearLayout>
```

Some errors will be shown in *android:text*, but this is due to we have to add the values for the strings *"@string/…"*. In Android, it is recommended to create all strings of the application in a separate file. These kinds of files are resources that we have to create.

If we select the preview mode, we will see the current design.

Adding the needed strings

We have to open the *strings.xml* file (*res -> values -> strings.xml*), and write the following code:

```
<resources>
    <string name="app_name">FirstApp</string>
    <string name="hello">Hello, this is the first activity</string>
    <string name="text2">We are learning how to program for
Android</string>
    <string name="textButton">Accept</string>
</resources>
```

The Activity

For each XML file with graphical interface, we should create an Activity. An activity is a class (in Java language) where we can define the functionality of the screen (e.g., the functionality of a button).

By default, an Activity is created. We have to open it: *java -> sampleactivities.metropolia.org.myapplication -> MainActivity*.

This is the default code:

```
package sampleactivities.metropolia.org.myapplication;

import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;


public class MainActivity extends ActionBarActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
```

```
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is
present.
        getMenuInflater().inflate(R.menu.menu_main, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();

        //noinspection SimplifiableIfStatement
        if (id == R.id.action_settings) {
            return true;
        }

        return super.onOptionsItemSelected(item);
    }
}
```

We have to delete the "onCreateOptionsMenu" and "onOptionsItemSelected" methods because they will not be used now, and will be explained later. Therefore, *MainActivity* will only contain the "onCreate" method. To avoid errors while compiling, the *res -> menu -> menu_main.xml* file also has to be deleted.

The "onCreate" method is part of the life cycle of an Activity, and it is called when an activity is created. There are more methods in the life cycle (onStart, onResume, etc.), but they will be explained later.

The line *"setContentView(R.layout.activity_main);"* indicates the graphical interface of this activity. It means that when the activity is created, the "*activity_main.*xml" is displayed.

Running the example

Before being able to run this sample project in the emulator, we have to create an emulator. We have to click on the green arrow  (Figure 8), which is at the top of the screen.

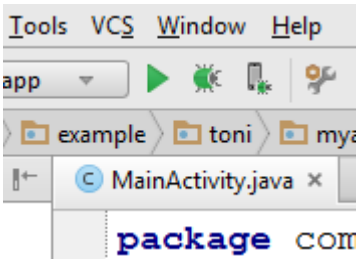


Figure 8

In this screen we can select if we want to execute the app on an emulator or a real device (Figure 9).

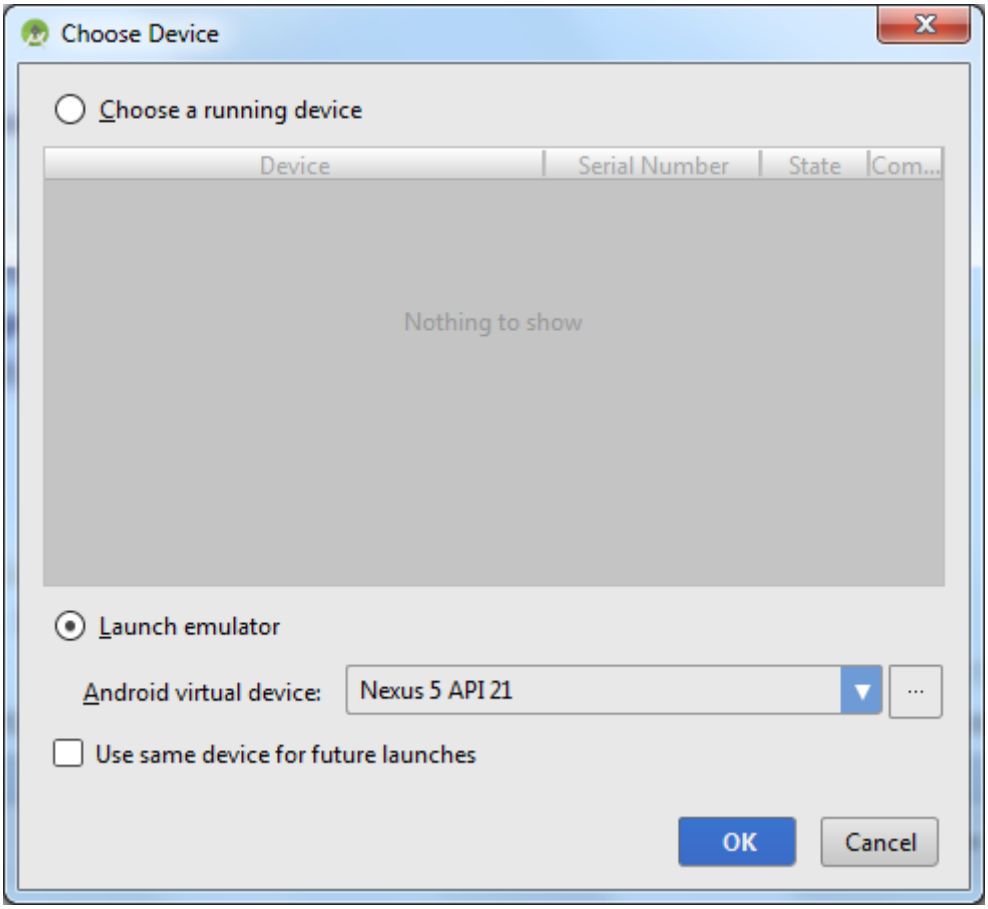


Figure 9

For creating a new emulator, click on the "…" button (Figure 9) and select a virtual device from the list or create a new one with the "Create Virtual Device…" option (Figure 10).

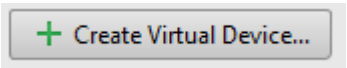


Figure 10

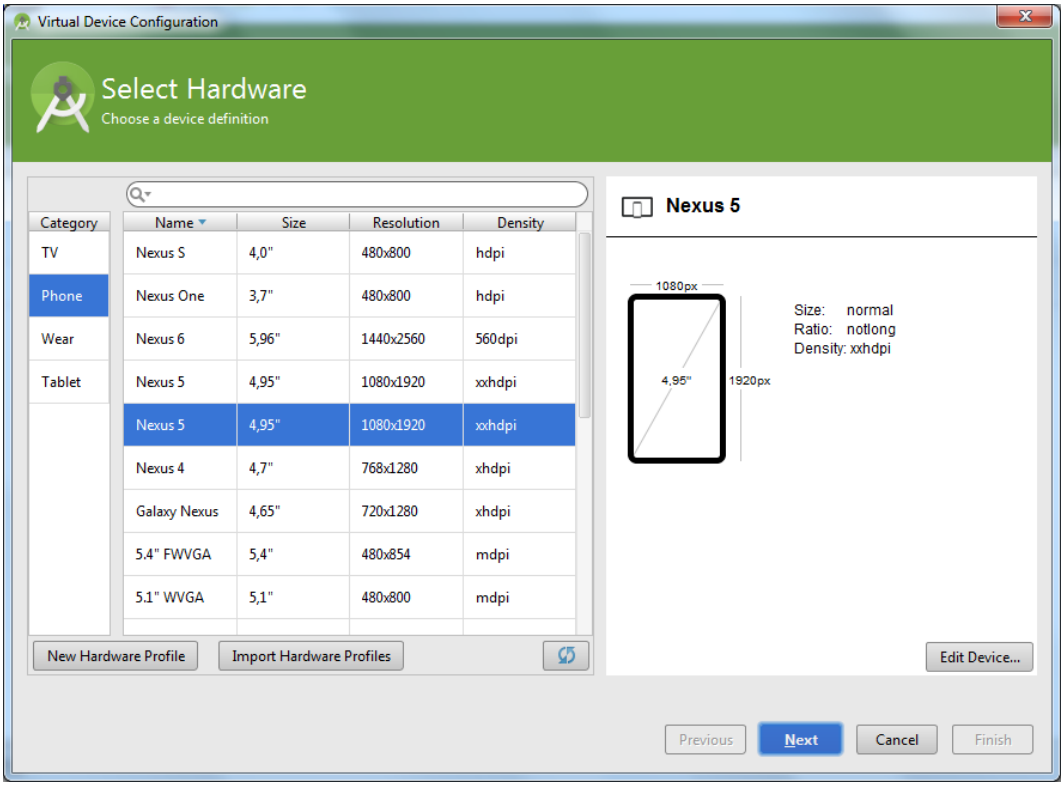


Figure 11

In this screen (Figure 11) we can select different kinds of emulators (screen sizes, resolutions, etc.). A good and recommended choice is Nexus 5. Click “Next” to continue.

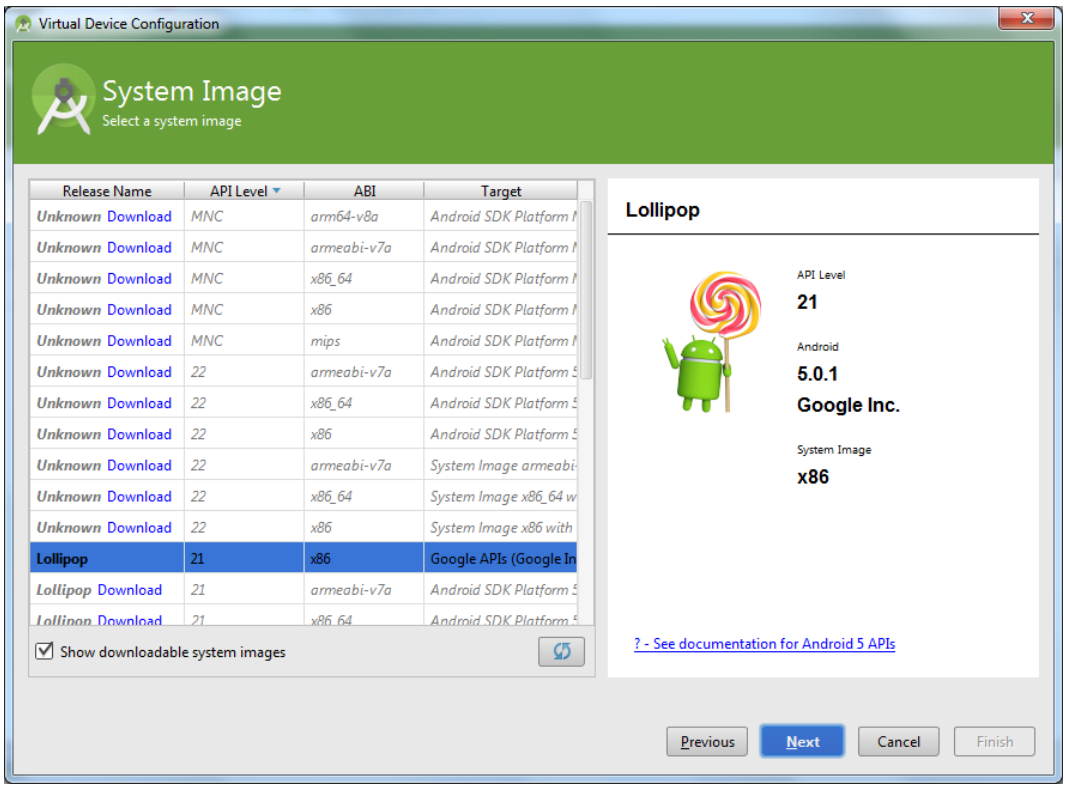


Figure 12

In this screen (Figure 12) we can select the android version of our emulator (it is important to select a version compatible with the android version of our project). Click “Next”.

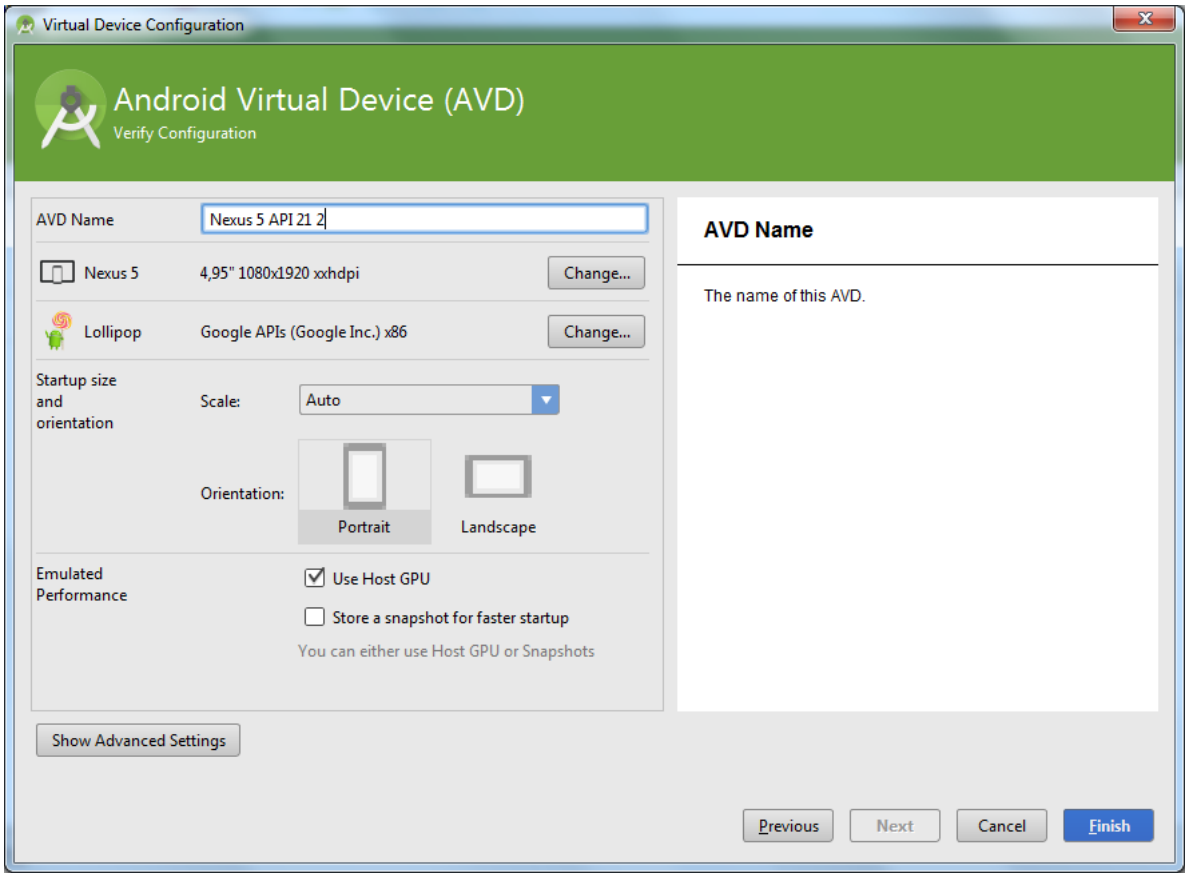


Figure 13

In the next screen (Figure 13) we have to insert the emulator name, orientation, etc. It is recommended to select the “Store a snapshot for faster startup” option for quickly launching the emulator, because deploying an application in an emulator could be a slow process.

Once the emulator has been created, it can be started using the green arrow option (Figure 14). Finally, the emulator will start running (it may take several minutes).

Figure 14

Adding functionality to the Button

Inserting the code that will be executed when the button is clicked

At this moment, the functionality of the button is empty. We will add some code to this button that will display a new screen when the button is clicked.

For doing this, in the main activity, within the *"onCreate"* method we have to insert the following code, after the line "*setContentView*":

```
Button button = (Button) findViewById(R.id.NewButton);
button.setOnClickListener(new OnClickListener() {

       @Override
       public void onClick(View v) {
       Intent intent = new Intent();
       intent.setClass(getBaseContext(), Activity2.class);
       startActivity(intent);
       }
});
```

With the above code, we have set a listener to the button, so when the button is clicked, this functionality will be executed. An error will be shown over *Activity2.class* because this class is not created yet.

Important note: for referencing the button, we use the ID created in the attribute *<Button android:id="@+id/NewButton"* of "*activity_main.xml*". The classes that are not recognized have to be imported (Button, View, Intent).

Activity2.class will be the second screen (Activity) that we have to create.

Creating the second screen

With the right mouse button on the source package, we select *New -> Java Class* and we indicate the following:

Name: Activity2

Kind: Class

Click "Ok".

The error in the first Activity will disappear. Then we have to extend from Activity and create the "*onCreate*" method in the second activity. The code is as follows:

```
public class Activity2 extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.layout2);
      }
}
```

Import the unrecognized classes (Activity, Bundle). *layout2* will be marked with an error because it is not created yet.

Creating layout2 for the second screen

With the right mouse button on *res* -> *layout* we have to select *New* -> *Layout resource file*, and then we insert a name, in this case *"layout2"*.

Now we will use another kind of Layout: a RelativeLayout. In this Layout the elements (components) are placed with relative positions to other elements.

We have to write the following code in the new layout:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
    <TextView
        android:id="@+id/textView1"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello2" />

</RelativeLayout>
```

We have to insert the new string for this second screen. In this case we need the string "*hello2"*. For doing this, open the *"strings.xml"* file and add the following code:

```
<string name="hello2">Second activity</string>
```

Finally, before testing the example again, we have to edit the *"AndroidManifest.xml"* file (in the "*manifests*" folder). When a new activity is created, it has to be declared in the manifest file.

This file contains important information about our app, for example:

- Initial activity
- Application name
- Application version

We have to add the declaration of *"Activity2"* on the same level as the other activity:

```
<activity android:name=".Activity2"></activity>
```

Once this is done, we can test again the application on the emulator.

Changing the application

Finishing the first activity

At this moment, when the second activity is created, the first activity is stopped, and if we click on the back button of the phone the second activity is closed and the first one is shown.

There is one way for destroying the first activity when the second one is created, that is, when the second activity is shown and the back button is clicked, the application closes. For doing this we have to add the following line of code:

```
finish();
```

This line would be within the "*onCreate*" method of the first activity, after the "startActivity" line, which starts the second activity. Thus, once the second activity is shown, the first activity is destroyed. Compile and run the app again to see the difference.

Practical exercise 1

Create two new activities (Activity3 and Activity4) with different layouts and views (student's choice). In the first activity (MainActivity) include a new button that shows Activity3 when it is clicked. In Activity3 create another button for showing Activity4. Finally, when Activity4 is closed, show directly the first activity (MainActivity).

Parameters between activities

In some situations sending a parameter (a string, an integer, etc.) to a second activity could be interesting. For doing this we will use an Intent. In this case we will send a string from the first activity to the second one. We have to modify the code inside the button's listener:

```
Intent intent = new Intent();
intent.setClass(getBaseContext(), Activity2.class);
//Sending a parameter to the second activity
String aux = "String created in MainActivity";
intent.putExtra("parameter", aux);
startActivity(intent);
```

With this code we are sending the parameter *aux* to the second activity (it is associated to the *parameter* key). Now we have to retrieve this string in the second activity.

For showing this data in the second activity we have to create a new TextView in Activity2, by adding the following lines of code in *layout2.xml*:

```
<TextView
        android:id="@+id/parameterView"
        android:layout_width="fill_parent"
```

```
        android:layout_height="wrap_content"
        android:layout_below="@id/textView1" />
```

Then we have to recover the parameter in Activity2. The parameter will be shown in the TextView created. In the *"onCreate"* method of Activity2 we have to add the following code:

```
Bundle extras = getIntent().getExtras();
String dataParameter = extras.getString("parameter");
TextView textData = (TextView) findViewById(R.id.parameterView);
textData.setText(dataParameter);
```

Finally we should run again the example for checking this new functionality.

Practical exercise 2

Create two new EditTexts in the first activity (MainActivity) and show their contents in the two created activities in Practical exercise 1 (Activity3 and Activity4).

Changing the graphical interface

Changes in activity_main.xml

- Firstly, we select the button added in *activity_main.xml*, and change its attribute *layout width* to "wrap_content". We can now observe the differences by running the application or in the "Preview" tab.

```
android:layout_width="wrap_content"
```

If another button is added, how could it be aligned with the existing button?

- We create a second button and we observe it is placed below the first one.

```
<Button android:id="@+id/Button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/textButton"/>
```

- The solution is adding a horizontal LinearLayout and including both buttons inside as follows:

```
<LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal">

          <Button android:id="@+id/NewButton"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
```

```
            android:text="@string/textButton"/>
        <Button android:id="@+id/Button2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/textButton"/>

</LinearLayout>
```

Adjusting the positions of buttons:

- We will separate the buttons of the text with a *layout margin top* = 20px

```
android:layout_marginTop="20px"
```

- Centering the buttons

```
android:layout_gravity="center_horizontal"
```

Changes in layout2.xml

We will add two buttons at the bottom of the screen. For doing this, we have to add the two buttons in *layout2.xml* inside a LinearLayout.

```
<LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:layout_marginTop="20px"
        android:layout_centerHorizontal="true"
        android:layout_alignParentBottom="true">

          <Button android:id="@+id/NewButton"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="@string/textButton" />
          <Button android:id="@+id/Button2"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="@string/textButton" />

</LinearLayout>
```

With the lines *layout_centerHorizontal="true"* and *layout_alignParentBottom="true"*, the buttons will be horizontally centered and at the bottom of the screen. In this case the *layout_gravity* attribute is not allowed because the LinearLayout is inside a RelativeLayout and its behavior is different.

Practical exercise 3

Create a new layout with the buttons of a numeric keyboard (0 to 9) (e.g., a calculator) and follow this distribution:


7
8
9
4
5
6
1
2
3
0

Part 2

Options menu

We will create an options menu that will be shown when the "Menu" hard button is clicked.

Firstly, we have to create a XML file that will have the different options of the menu. The folder for creating this file is *res -> menu*. With the right mouse button on this folder, we select *New -> Menu resource file* and write the name *"main_menu.xml"*. Click "Ok".

Now we have to add the items to the menu with the following XML code:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/option1"
        android:title="@string/op1" />
    <item android:id="@+id/option2"
        android:title="@string/op2" />
</menu>
```

We have to add now the missing strings, it means, the text that will be shown in the different options of the menu. We add the following lines in file *strings.xml*.

```
<string name="op1">Option 1</string>
<string name="op2">Option 2</string>
```

Now we will create the next method in the main activity (*MainActivity*). This will allow associating the *main_menu.xml* to the Activity.

```
@Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main_menu, menu);
        return true;
    }
```

If we run the app, we will see the menu but nothing happens, so we will write some functionality to these options. To do this, add the following method to *MainActivity*.

```
@Override
public boolean onOptionsItemSelected(MenuItem item)
{
       int option = item.getItemId();
       switch (option)
       {
              case R.id.option1:
                 Intent intent = new Intent();
                 intent.setClass(getBaseContext(), Activity2.class);
                 //Sending a parameter to the second activity
                 String aux = "String created in MainActivity";
                 intent.putExtra("parameter", aux);
```

```
                startActivity(intent);
                break;
            case R.id.option2:
                break;
        }
        return false;
}
```

In this manner, when option 1 is clicked, the app will show the second activity. It remains writing functionality for the button of option 2.

An icon could be shown in the different options in this kind of menus, by simply adding the following line of code in each menu item in the *"main_menu.xml"* file.

```
android:icon="@mipmap/ic_launcher"
```

In this example we are using the same icon as the application icon. You can get more icons from the Android's Icon Library at https://www.google.com/design/icons/. If you run the app now, you will check the icons in the options menu. Note: icons are only showed in a menu when running versions older than Android 3.0.

Multilanguage

At this moment, all texts of the application have been saved in the *"strings.xml"* file, inside the *res/values* folder. Now we will add Multilanguage to our app. For doing this, we have to create a new folder inside *"res"* called *"values-es"*. This folder will contain all texts in Spanish language. We have to add at the end the two letters of the new language following the ISO-639-1 (more information at https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes). The *"values"* folder contains the strings in the default language (in this case, English). Besides this, our application will support Spanish language, it means, if a user uses his device with Spanish language the application automatically will be shown in Spanish.

Inside this new folder we have to copy the *"strings.xml"* file and we will change the title of the application for "Spanish title".

Finally, we could change the language of the emulator/device and check how the language of the application also changes. This process should be done when the development process has already been finished, when all strings are in the same file for avoiding duplicities.

Practical exercise 4

Translate the *strings.xml* file into Finnish and French languages (or choose other different). You may use an online translator such as Google Translator. Change the language of the emulator for checking the results.

Toasts and Dialogs

Some important components in Android applications are Toasts and Dialogs, which allow showing messages to the user or asking for some action.

Toast

A Toast is a pop-up message that is shown on the screen only for a few seconds.

We will show a Toast when the user selects Option 2 in the options menu. For doing this, we have to add the following code inside the listener in Option 2:

```
Toast.makeText(this,R.string.click2,Toast.LENGTH_LONG).show();
```

We also have to add the string to the *"strings.xml"* file.

```
<string name="click2">Click in Option 2</string>
```

We can change the duration of the Toast by using Toast.LENGTH_LONG or Toast.LENGTH_SHORT.

Dialogs

Other important aspects are dialogs. Dialogs allow asking for user confirmation about some actions, for example, deleting a file or closing the application.

For showing a dialog, firstly we have to insert this code inside the event of Option 2 of the above example:

```
AlertDialog.Builder alert = new AlertDialog.Builder(this);

alert.setTitle(R.string.titleDialog);
alert.setMessage(R.string.messageDialog);

alert.setPositiveButton(R.string.ok, new
DialogInterface.OnClickListener() {
	public void onClick(DialogInterface dialog, int whichButton) {

	}
	});

alert.setNegativeButton(R.string.cancel, new
DialogInterface.OnClickListener() {
	public void onClick(DialogInterface dialog, int whichButton) {

	}
	});

alert.show();
```

Then we have to add the missing strings in the *"strings.xml"* file.

```
<string name="titleDialog">Dialog</string>
```

```
<string name="messageDialog">Message</string>
<string name="ok">Accept</string>
<string name="cancel">Cancel</string>
```

If we run the example we will see the dialog.

"Back" button

In Android there is a special button called "Back" button. The main functionality of this button is closing an activity or closing the application. In some situations it could be interesting to override this functionality and using the button for something else.

By overriding the *"onKeyDown"* method we can manage the behavior of this button (and some other controls).

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {

       if ((keyCode == KeyEvent.KEYCODE_BACK)) {

Toast.makeText(this,R.string.backPressed,Toast.LENGTH_LONG).show();
     }
       return false;
}
```

Now we have to add the missing string.

```
<string name="backPressed">The back button has been pressed</string>
```

Run and test the example.

Practical exercise 5

In the previous practical exercises include dialog messages before closing an activity and before closing the application. The user should confirm these operations.

Also, show a new message (a Toast is enough) when a new activity is opened.

Life cycle of an Activity

There are four states of an Activity:

- **Running**: The activity is shown foreground in the screen and the user can interact with it. It is at the top of the activities stack.
- **Paused**: The activity is shown on the screen but it is in background, behind other activity. The user cannot interact with the activity. When the activity is completely covered, it goes to Stopped state.

- **Stopped**: The activity is not shown on the screen. It is recommended to save its state for recovering it when it will be shown again.
- **Destroyed**: The activity has been finished, or destroyed. In this state the activity is out of the activities stack.

There are different events for managing these states:

- **onCreate**(): It is invoked when the activity is created.
- **onRestart**(): Called after the activity has been stopped.
- **onStart**(): It is invoked when the activity is visible for the user on the screen.
- **onResume**(): It is invoked when the activity interacts with the user.
- **onPause**(): It is invoked when other activity is partially shown in foreground.
- **onStop**(): It is invoked when the activity changes its state to Stopped. It is not shown on the screen.
- **onDestroy**(): It is invoked when the activity is destroyed.

With the following example, we will test the functionality of the aforementioned methods:

```
@Override
    public void onRestart()
    {
       super.onRestart();
       Log.d("MainActivity", "onRestart");
    }

    @Override
    public void onStart()
    {
       super.onStart();
       Log.d("MainActivity", "onStart");
    }

    @Override
    public void onResume()
    {
       super.onResume();
       Log.d("MainActivity", "onResume");
    }

    @Override
    public void onPause()
    {
       super.onPause();
       Log.d("MainActivity", "onPause");
    }

    @Override
    public void onStop()
    {
```

```
        super.onStop();
        Log.d("MainActivity", "onStop");
    }

    @Override
    public void onDestroy()
    {
        super.onDestroy();
        Log.d("MainActivity", "onDestroy");
    }
```

We also have to add the following line in the *"onCreate"* method:

```
Log.d("MainActivity", "onCreate");
```

By doing this, we are writing a log message in the Log system when an event is called, and we will see the life cycle of the activity. For more information about the Android activity's lifecycle, please check http://developer.android.com/reference/android/app/Activity.html

Practical exercise 6

Check (see the *Logcat* tab) the life cycle of an activity when:

- A new activity is opened without closing the first one.
- A new activity is opened destroying the previous activity.
- The application closes.
- The application minimizes using the "*Home*" button of the device.

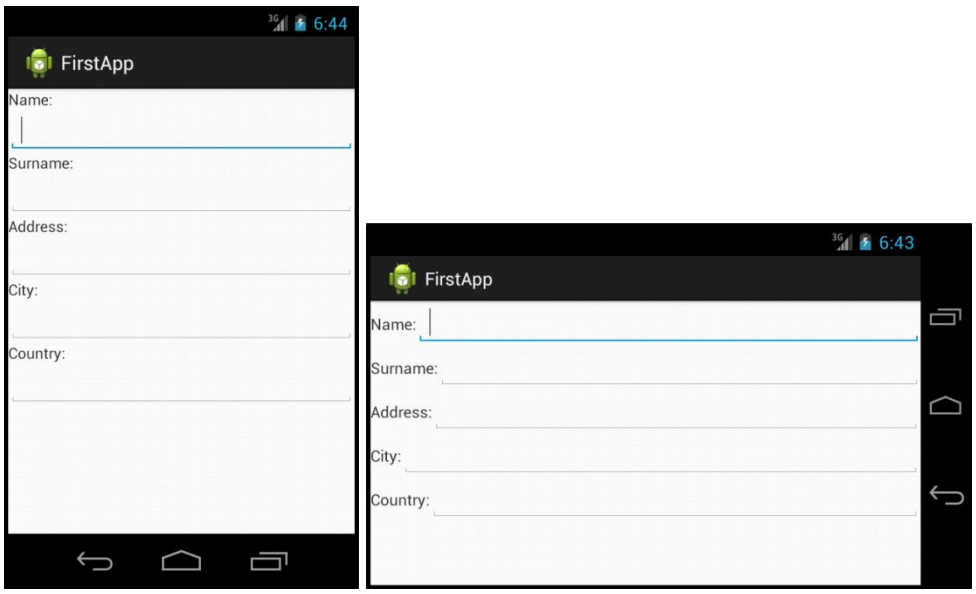
Practical exercise 7

Change the application icon by adding new image resources in different resolutions.

Tip: http://developer.android.com/guide/practices/screens_support.html

Practical exercise 8

Create a new activity with a layout showing at least the following fields (TextViews + EditTexts): Name, Surname, Address, City and Country. This layout should be adaptable to landscape (horizontal) or portrait (vertical) orientations of the device. Example:



Tip: http://developer.android.com/training/basics/supporting-devices/screens.html