




Free tutorial, donate to support



by Lars Vogel

AdChoices 

Google Play

One place to discover, enjoy and share your favorite entertainment.
play.google.com

Windows Server 2012

Windows Server 2012 Keeps Your Apps Up And Running. Learn More.
microsoft.com/ws2012

Top 5 Android Cell Phone

Best Models, Deals & Services Save Big on Android Smart Phone!

Android Development Tutorial

Based on Android 4.1

Lars Vogel

Version 10.7

Copyright © 2009, 2010, 2011, 2012 Lars Vogel

20.10.2012

Revision History

Revision 0.1	04.07.2009
Created	
Revision 0.2 - 10.7	07.07.2009 - 20.10.2012
bug fixing and enhancements	

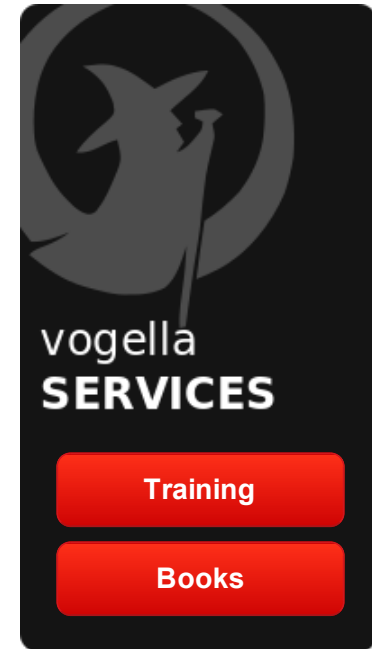
Development with Android and Eclipse

This tutorial describes how to create Android applications with Eclipse. It is based on Eclipse 4.2 (Juno), Java 1.6 and Android 4.1 (Jelly Bean).

Table of Contents

1. What is Android?

- 1.1. Android Operation System
- 1.2. Google Play (Android Market)
- 1.3. Security and permissions



[Shop Android Phones](#)

Find Great Deals on
Android Phones best
android device

AcumenAdvice.com

2. Basic Android User Interface components

- 2.1. Activity**
- 2.2. Fragments**
- 2.3. Views and ViewGroups**
- 2.4. Activities, Fragments and Views**

3. Other Android components

- 3.1. Intents**
- 3.2. Services**
- 3.3. ContentProvider**
- 3.4. BroadcastReceiver**
- 3.5. (HomeScreen) Widgets**
- 3.6. More Components**

4. Android Development Tools

- 4.1. Android SDK**
- 4.2. Android Development Tools**
- 4.3. Dalvik Virtual Machine**
- 4.4. How to develop Android Applications**
- 4.5. Resource editors**

5. Android Application Architecture

- 5.1. AndroidManifest.xml**
- 5.2. Activities and Lifecycle**
- 5.3. Configuration Change**
- 5.4. Context**

6. Resources

- 6.1. Resources**
- 6.2. Assets**

7. Using Resources

- 7.1. Reference to resources in code**
- 7.2. Reference to resources in XML files**
- 7.3. Activities and Layouts**

8. Installation

- 8.1. Eclipse**
- 8.2. Pre-requisites for using a 64bit Linux**
- 8.3. Install ADT Plug-ins and Android SDK**
- 8.4. Manual installation of the Android SDK**

- 8.5. Install a specific Android version
- 8.6. Install support library
- 8.7. Android Source Code
- 9. Android virtual device - Emulator
 - 9.1. What is the Android Emulator?
 - 9.2. Google vs. Android AVD
 - 9.3. Emulator Shortcuts
 - 9.4. Parameter
- 10. Tutorial: Create and run Android Virtual Device
- 11. Solving Android development problems
- 12. Conventions for the tutorials
 - 12.1. API version, package and application name
 - 12.2. Warning messages for Strings
- 13. Tutorial: Your first Android project
 - 13.1. Install the demo application
 - 13.2. Create Project
 - 13.3. Create attributes
 - 13.4. Add Views
 - 13.5. Edit View properties
 - 13.6. Change the Activity source code
 - 13.7. Start Project
- 14. Starting an installed application
- 15. Layout Manager and ViewGroups
 - 15.1. Available Layout Manager
 - 15.2. LinearLayout
 - 15.3. RelativeLayout
 - 15.4. GridLayout
 - 15.5. ScrollView
- 16. Tutorial: ScrollView
- 17. Fragments
 - 17.1. Fragments Overview
 - 17.2. When to use Fragments
 - 17.3. Configuration changes in Fragments
- 18. Fragments Tutorial
 - 18.1. Overview

- 18.2. Create Project**
- 18.3. Create standard layouts**
- 18.4. Activity**
- 18.5. Create Fragment classes**
- 18.6. Run**

19. Fragments Tutorial - layout for portrait mode

- 19.1. Create layouts for portrait mode**
- 19.2. DetailActivity**
- 19.3. Run**

20. OptionMenu and ActionBar

- 20.1. ActionBar**
- 20.2. OptionsMenu**
- 20.3. Creating the menu**
- 20.4. Changing the menu**
- 20.5. Reacting to menu entry selection**

21. Advanced ActionBar

- 21.1. Using the home icon**
- 21.2. Custom Views in the ActionBar**
- 21.3. Contextual action mode**
- 21.4. Context menus**

22. Tutorial: ActionBar

- 22.1. Project**
- 22.2. Add a menu XML resource**

23. Tutorial: Using the contextual action mode

24. ActionBar navigation with Fragments

25. DDMS perspective and important views

- 25.1. DDMS - Dalvik Debug Monitor Server**
- 25.2. LogCat View**
- 25.3. File explorer**

26. Shell

- 26.1. Android Debugging Bridge - Shell**
- 26.2. Uninstall an application via adb**
- 26.3. Emulator Console via telnet**

27. Deployment

- 27.1. Overview
- 27.2. Deployment via Eclipse
- 27.3. Export your application
- 27.4. Via external sources
- 27.5. Google Play (Market)

- 28. Thank you
- 29. Questions and Discussion
- 30. Links and Literature

- 30.1. Source Code
- 30.2. Android Resources
- 30.3. vogella Resources

1. What is Android?

1.1. Android Operation System

Android is an operating system based on Linux with a Java programming interface.

The Android Software Development Kit (Android SDK) provides all necessary tools to develop Android applications. This includes a compiler, debugger and a device emulator, as well as its own virtual machine to run Android programs.

Android is currently primarily developed by Google.

Android allows background processing, provides a rich user interface library, supports 2-D and 3-D graphics using the OpenGL libraries, access to the file system and provides an embedded SQLite database.

Android applications consist of different components and can re-use components of other applications. This leads to the concept of a *task* in Android; an application can re-use other Android components to archive a task. For example you can trigger from your application another application which has itself registered with the Android system to handle photos. In this other application you select a photo and return to your application to use the selected photo.

1.2. Google Play (Android Market)

Google offers the *Google Play* service in which programmers can offer their Android application to Android users. Google phones include the *Google Play* application which allows to install applications.

Google Play also offers an update service, e.g. if a programmer uploads a new version of his application to Google Play, this service will notify existing users that an update is available and allow to install it.

Google Play used to be called *Android Market*.

1.3. Security and permissions

During deployment on an Android device, the Android system will create a unique user and group ID for every Android application. Each application file is private to this generated user, e.g. other applications cannot access these files.

In addition each Android application will be started in its own process.

Therefore by means of the underlying Linux operating system, every Android application is isolated from other running applications.

If data should be shared, the application must do this explicitly, e.g. via a *Service* or a *ContentProvider*.

Android also contains a permission system. Android predefines permissions for certain tasks but every application can define additional permissions.

An Android application declare its required permissions in its *AndroidManifest.xml* configuration file. For example an application may declare that it requires access to the Internet.

Permissions have different levels. Some permissions are automatically granted by the Android system, some are automatically rejected.

In most cases the requested permissions will be presented to the user before installation of the application. The user needs to decide if these permissions are given to the application.

If the user denies a permission required by the application, this application cannot be installed. The check of the permission is only performed during installation, permissions cannot be denied or granted after the installation.

Not all users pay attention to the required permissions during installation. But some users do and they write negative reviews on Google Play.

2. Basic Android User Interface components

The following gives a short overview of the most important user interface components in Android.

2.1. Activity

An *Activity* represents the visual representation of an Android application. *Activities* use *Views* and *Fragments* to create the user interface and to interact with the user.

An Android application can have several *Activities*.

2.2. Fragments

Fragments are components which run in the context of an *Activity*. Fragment components encapsulate application code so that it is easier to reuse it and to support different sized devices.

Fragments are optional, you can use *Views* and *ViewGroups* directly in an *Activity* but in professional applications you always use them to allow the reuse of your user interface components on different sized devices.

2.3. Views and ViewGroups

Views are user interface widgets, e.g. buttons or text fields. The base class for all *Views* is the `android.view.View` class. *Views* have attributes which can be used to configure their appearance and behavior.

A *ViewGroup* is responsible for arranging other *Views*. *ViewGroups* is also called *layout managers*. The base class for these layout managers is the `android.view.ViewGroup` class which extends the `View` class.

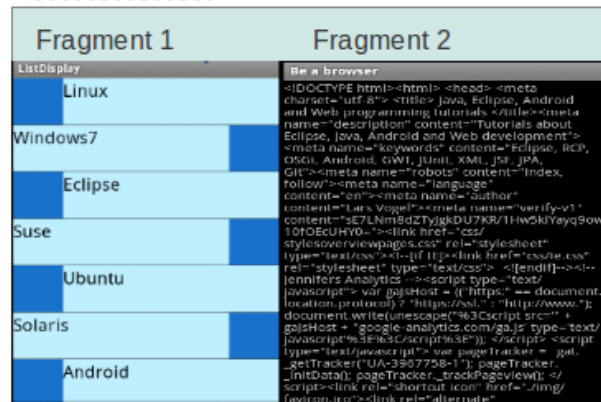
ViewGroups can be nested to create complex layouts. You should not nestle *ViewGroups* too deeply as this has a negative impact on the performance.

2.4. Activities, Fragments and Views

Activities are defined with different layouts. These layouts can be picked based on several different factoring including the size of the actual device.

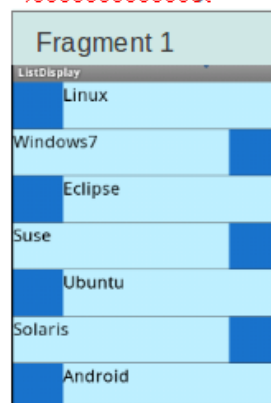
The following picture shows an *Activity* called *MainActivity*. On a wide screen it shows two *Fragments*. On a smaller screen it shows one *Fragment* and allows that the user navigate to another *Activity* called *SecondActivity* which displays the second *Fragment*.

MainActivity



Wide screen,
e.g. tablet

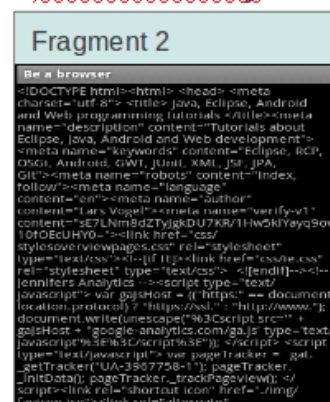
MainActivity



Start new
Activity



SecondActivity



Smaller
screen, e.g.
handheld

3. Other Android components

Android has several more components which can be used in your Android application.

3.1. Intents

Intents are asynchronous messages which allow the application to request functionality from other components of the Android system, e.g. from *Services* or *Activities*. An application can call a component directly (*explicit Intent*) or ask the Android system to evaluate registered components based on the *Intent* data (*implicit Intents*). For example the application could implement sharing of data via an *Intent* and all components which allow sharing of data would be available for the user to select. Applications register themselves to an *Intent* via an *IntentFilter*.

Intents allow to combine loosely coupled components to perform certain tasks.

3.2. Services

Services perform background tasks without providing a user interface. They can notify the user via the notification framework in Android.

3.3. ContentProvider

A *ContentProvider* provides a structured interface to application data. Via a *ContentProvider* your application can share data with other applications. Android contains an SQLite database which is frequently used in conjunction with a *ContentProvider*. The SQLite database would store the data, which would be accessed via the *ContentProvider*.

3.4. BroadcastReceiver

BroadcastReceiver can be registered to receive system messages and *Intents*. A *BroadcastReceiver* will get notified by the Android system, if the specified situation happens. For example a *BroadcastReceiver* could get called once the Android system completed the boot process or if a phone call is received.

3.5. (HomeScreen) Widgets

Widgets are interactive components which are primarily used on the Android homescreen. They typically display some kind of data and allow the user to perform actions via them. For example a *Widget* could display a short summary of new emails and if the user selects an email, it could start the email application with the selected email.

3.6. More Components

Android provide many more components but the list above describes the most important ones. Other Android components are *Live Folders* and *Live Wallpapers* . *Live Folders* display data on the homescreen without launching the corresponding application while *Live Wallpapers* allow to create animated backgrounds.

4. Android Development Tools

4.1. Android SDK

The *Android Software Development Kit* (SDK) contains the necessary tools to create, compile and package Android application. Most of these tools are command line based.

The Android SDK also provides an Android device emulator, so that Android applications can be tested without a real Android phone. You can create *Android virtual devices* (AVD) via the Android SDK, which run in this emulator.

The Android SDK contains the *Android debug bridge* (adb) tool which allows to connect to an virtual or real Android device.

4.2. Android Development Tools

Google provides the *Android Development Tools* (ADT) to develop Android applications with Eclipse. ADT is a set of components (plug-ins) which extend the Eclipse IDE with Android development capabilities.

ADT contains all required functionalities to create, compile, debug and deploy Android applications from the Eclipse IDE. ADT also allows to create and start AVDs.

The Android Development Tools (ADT) provides specialized editors for resources files, e.g. layout files. These editors allow to switch between the XML representation of the file and a richer user interface via tabs on the bottom of the editor.

4.3. Dalvik Virtual Machine

The Android system uses a special virtual machine, i.e. the *Dalvik Virtual Machine* to run Java based applications. Dalvik uses an own bytecode format which is different from Java bytecode.

Therefore you cannot directly run Java class files on Android, they need to get converted in the Dalvik bytecode format.

4.4. How to develop Android Applications

Android applications are primarily written in the Java programming language. The Java source files are converted to Java class files by the Java compiler.

The Android SDK contains a tool called *dx* which converts Java class files into a *.dex* (Dalvik Executable) file. All class files of one application are placed in one compressed *.dex* file. During this conversion process redundant information in the class files are optimized in the *.dex* file. For example if the same String is found in different class files, the *.dex* file contains only once reference of this String.

These dex files are therefore much smaller in size than the corresponding class files.

The *.dex* file and the resources of an Android project, e.g. the images and XML files, are packed into an *.apk* (Android Package) file. The program *aapt* (Android Asset Packaging Tool) performs this packaging.

The resulting *.apk* file contains all necessary data to run the Android application and can be deployed to an Android device via the *adb* tool.

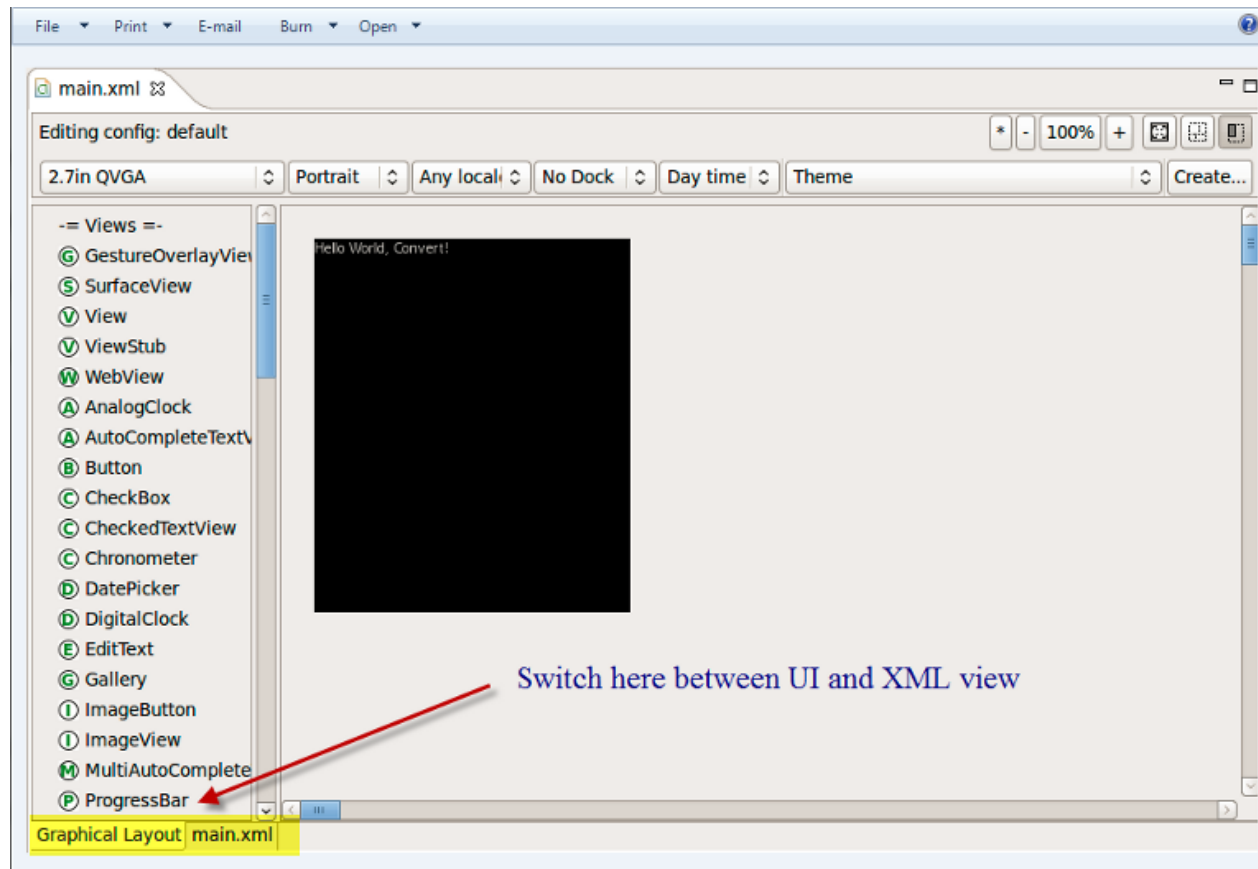
The Android Development Tools (ADT) performs these steps transparently to the user.

If you use the ADT tooling you press a button the whole Android application (*.apk* file) will be created and deployed.

4.5. Resource editors

The ADT allows the developer to define certain artifacts, e.g. Strings and layout files, in two ways: via a rich editor, and directly via XML. This is done via multi-page editors in Eclipse. In these editors you can switch between both representations by clicking on the tab on the lower part of the screen.

For example if you open the *res/layout/main.xml* file in the *Package Explorer* View of Eclipse, you can switch between the two representations as depicted in the following screenshot.



5. Android Application Architecture

5.1. AndroidManifest.xml

The components and settings of an Android application are described in the *AndroidManifest.xml* file. For example all *Activities* and *Services* of the application must be declared in this file.

It must also contain the required permissions for the application. For example if the application requires network access it must be specified here.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="de.vogella.android.temperature"
```

```

        android:versionCode="1"
        android:versionName="1.0">
        <application android:icon="@drawable/icon" android:label="@string/app_name">
            <activity android:name=".Convert"
                android:label="@string/app_name">
                <intent-filter>
                    <action android:name="android.intent.action.MAIN" />
                    <category android:name="android.intent.category.LAUNCHER" />
                </intent-filter>
            </activity>
        </application>
        <uses-sdk android:minSdkVersion="9" />

    </manifest>

```

The *package* attribute defines the base package for the Java objects referred to in this file. If a Java object lies within a different package, it must be declared with the full qualified package name.

Google Play requires that every Android application uses its own unique package. Therefore it is a good habit to use your reverse domain name as package name. This will avoid collisions with other Android applications.

android:versionName and *android:versionCode* specify the version of your application. *versionName* is what the user sees and can be any String.

versionCode must be an integer. The Android Market determine based on the *versionCode*, if it should perform an update of the applications for the existing installations. You typically start with "1" and increase this value by one, if you roll-out a new version of your application.

The *<activity>* tag defines an *Activity*, in this example pointing to the *Convert* class in the *de.vogella.android.temperature* package. An intent filter is registered for this class which defines that this *Activity* is started once the application starts (action *android:name="android.intent.action.MAIN"*). The category definition *category android:name="android.intent.category.LAUNCHER"* defines that this application is added to the application directory on the Android device.

The *@string/app_name* value refers to resource files which contain the actual value of the application name. The usage of resource file makes it easy to provide different resources, e.g. strings, colors, icons, for different devices and makes it easy to translate applications.

The *uses-sdk* part of the *AndroidManifest.xml* file defines the minimal SDK version for which your

application is valid. This will prevent your application being installed on unsupported devices.

5.2. Activities and Lifecycle

The Android system controls the lifecycle of your application. At any time the Android system may stop or destroy your application, e.g. because of an incoming call. The Android system defines a lifecycle for *Activities* via predefined methods. The most important methods are:

- `onSaveInstanceState()` - called after the Activity is stopped. Used to save data so that the Activity can restore its states if re-started
- `onPause()` - always called if the Activity ends, can be used to release resource or save data
- `onResume()` - called if the Activity is re-started, can be used to initialize fields

5.3. Configuration Change

An Activity will also be restarted, if a so called "configuration change" happens. A configuration change happens if an event is triggered which may be relevant for the application. For example if the user changes the orientation of the device (vertically or horizontally). Android assumes that an Activity might want to use different resources for these orientations and restarts the Activity.

In the emulator you can simulate the change of the orientation via **Ctrl+F11**.

You can avoid a restart of your application for certain configuration changes via the `configChanges` attribute on your Activity definition in your `AndroidManifest.xml`. The following Activity will not be restarted in case of orientation changes or position of the physical keyboard (hidden / visible).

```
<activity android:name=".ProgressTestActivity"
    android:label="@string/app_name"
    android:configChanges="orientation|keyboardHidden|keyboard">
</activity>
```

5.4. Context

The class `android.content.Context` provides the connection to the Android system and the resources of the project. It is the interface to global information about the application environment.

The *Context* also provides access to Android *Services*, e.g. the Location Service.

Activities and *Services* extend the `Context` class.

6. Resources

6.1. Resources

Android supports that resources, like images and certain XML configuration files, can be keep separate from the source code.

These resources must be defined in the *res* directory in a special folder dependent on their purpose. You can also append additional qualifiers to the folder name to indicate that the related resources should be used for special configurations, e.g. you can specify that a resource is only valid for a certain screen size.

The following table give an overview of the supported resources and their standard folder prefix.

Table 1. Resources

Resource	Folder	Description
Simple Values	/res/values	Used to define strings, colors, dimensions, styles and static arrays of strings or integers. By convention each type is stored in a separate file, e.g. strings are defined in the <i>res/values/strings.xml</i> file.
Layouts	/res/values	XML file with layout description files used to define the user interface for <i>Activities</i> and <i>Fragments</i> .
Styles and Themes	/res/values	Files which define the appearance of your Android application.
Animations	/res/anim	Define animations in XML for the property animation API which allows to animate arbitrary properties of objects over time.
Menus	/res/menu	Define the properties of entries for a menu.

The *gen* directory in an Android project contains generated values. *R.java* is a generated class which

contains references to certain resources of the project.

If you create a new resource, the corresponding reference is automatically created in *R.java* via the Eclipse ADT tools. These references are static integer values and define IDs for the resources.

The Android system provides methods to access the corresponding resource via these IDs.

For example to access a String with the *R.string.yourString* ID, you would use the `getString(R.string.yourString)` method.

R.java is automatically created by the Eclipse development environment, manual changes are not necessary and will be overridden by the tooling.

6.2. Assets

While the *res* directory contains structured values which are known to the Android platform, the *assets* directory can be used to store any kind of data. You access this data via the *AssetManager* which you can access the `getAssets()` method.

AssetManager allows to read an assets as *InputStream* with the `open()` method.

```
// Get the AssetManager
AssetManager manager = getAssets();

// Read a Bitmap from Assets
try {
    InputStream open = manager.open("logo.png");
    Bitmap bitmap = BitmapFactory.decodeStream(open);
    // Assign the bitmap to an ImageView in this layout
    ImageView view = (ImageView) findViewById(R.id.imageView1);
    view.setImageBitmap(bitmap);
} catch (IOException e) {
    e.printStackTrace();
}
```

7. Using Resources

7.1. Reference to resources in code

The *Resources* class allows to access individual resources. An instance of *Resources* can get access

via the `getResources()` method of the `Context` class.

`Resources` is also used by other Android classes, for example the following code shows how to create a `Bitmap` file from a reference ID.

```
BitmapFactory.decodeResource(getResources(), R.drawable.ic_action_search);
```

7.2. Reference to resources in XML files

In your XML files, for example your layout files, you can refer to other resources via the `@` sign.

For example, if you want to refer to a color which is defined in a XML resource, you can refer to it via `@color/your_id`. Or if you defined a "hello" string in an XML resource, you could access it via `@string/hello`.

7.3. Activities and Layouts

The user interface for *Activities* is defined via layouts. The layout defines the included Views (widgets) and their properties.

A layout can be defined via Java code or via XML. In most cases the layout is defined as an XML file.

XML based layouts are defined via a resource file in the `/res/layout` folder. This file specifies the ViewGroups, Views, their relationship and their attributes for this specific layout.

If a View needs to be accessed via Java code, you have to give the View a unique ID via the `android:id` attribute. To assign a new ID to a View use `@+id/yourvalue`. The following shows an example in which a Button gets the `button1` ID assigned.

```
<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Show Preferences" >
</Button>
```

By conversion this will create and assign a new `yourvalue` ID to the corresponding View. In your Java code you can later access a View via the method `findViewById(R.id.yourvalue)`.

Defining layouts via XML is usually the preferred way as this separates the programming logic from the layout definition. It also allows the definition of different layouts for different devices. You can also mix both approaches.

8. Installation

8.1. Eclipse

The following assumes that you have already Java and Eclipse installed and know how to use Eclipse.

8.2. Pre-requisites for using a 64bit Linux

The Android SDK is 32bit, therefore on a 64bit Linux system you need to have the package `ia32-libs` installed. For Ubuntu you can do this via the following command.

```
apt-get install ia32-libs
```

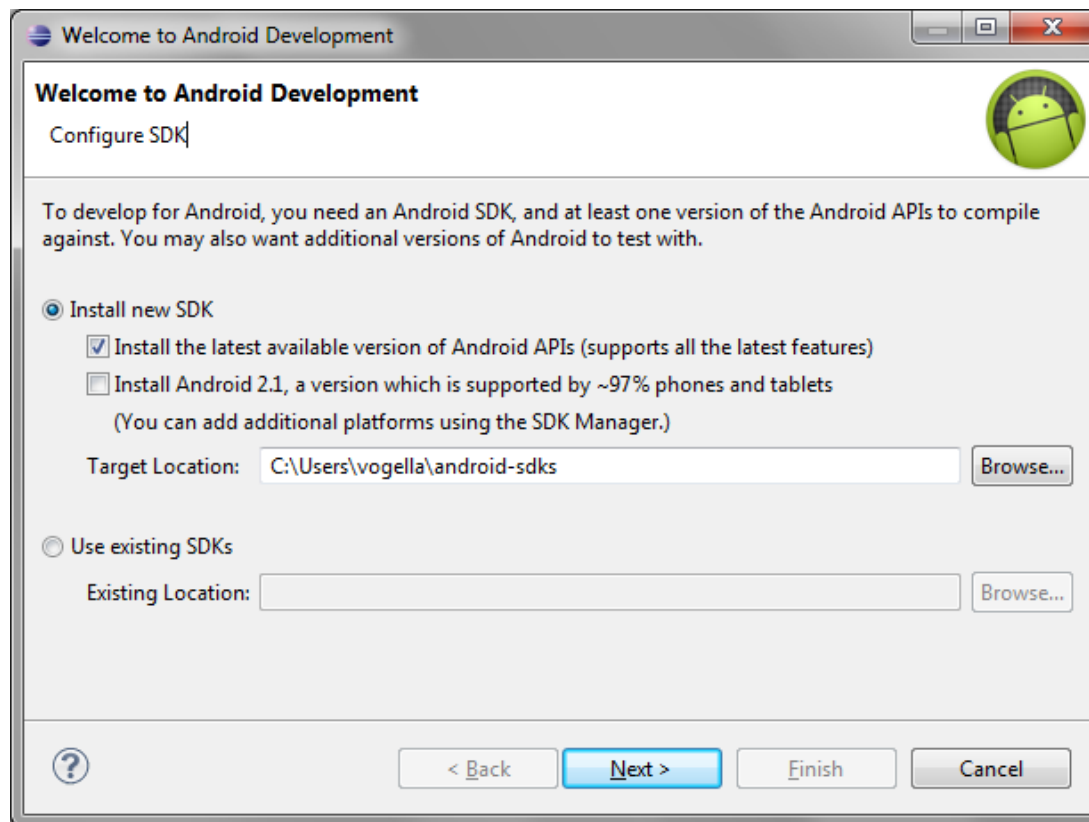
Please check your distribution documentation, if you are using a different flavor of Linux.

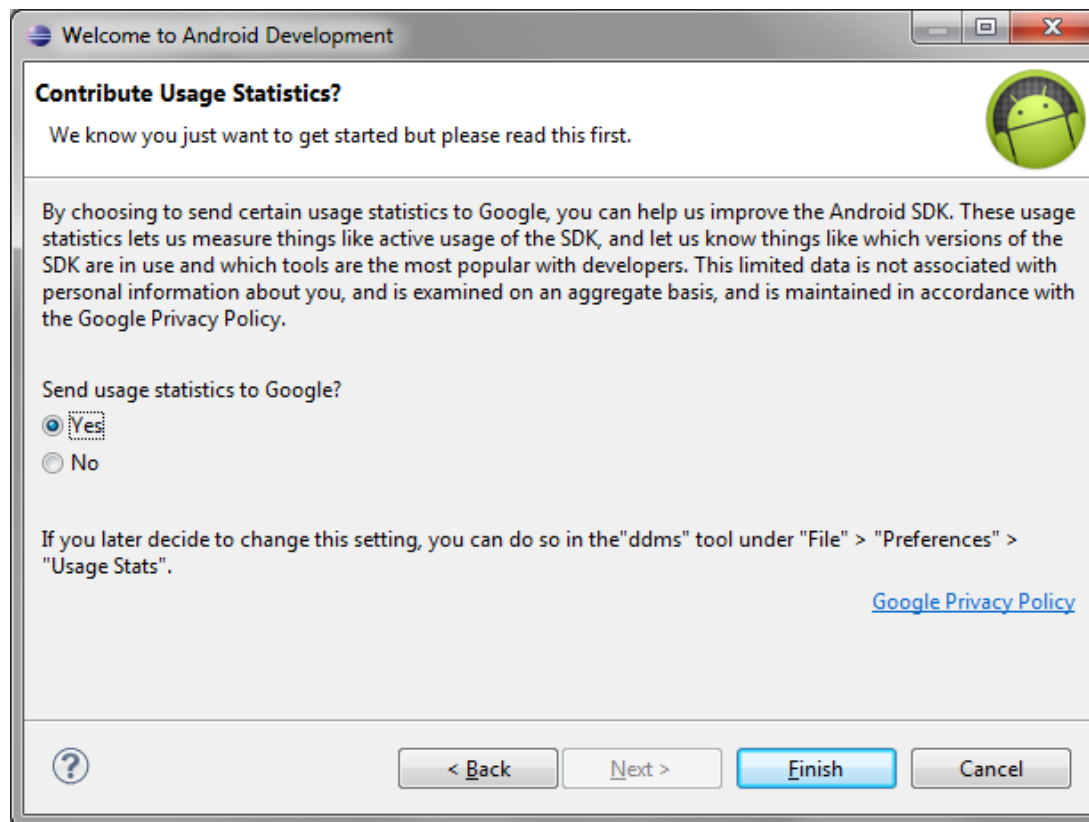
8.3. Install ADT Plug-ins and Android SDK

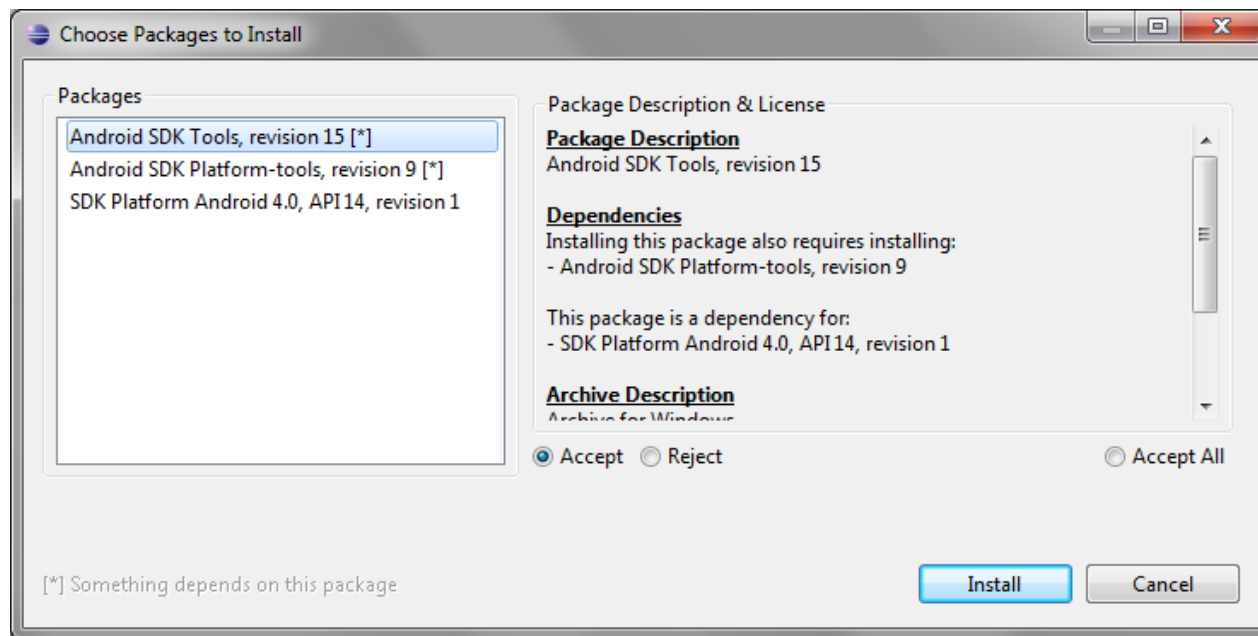
Use the Eclipse update manager via *Help* → *Install new software* to install all available components for the Android Development Tools (ADT) from the following URL:

```
https://dl-ssl.google.com/android/eclipse/
```

After the new Android development components are installed, you will be prompted to install the Android SDK. You can use the following wizard or go to the next section to learn how to do it manually.







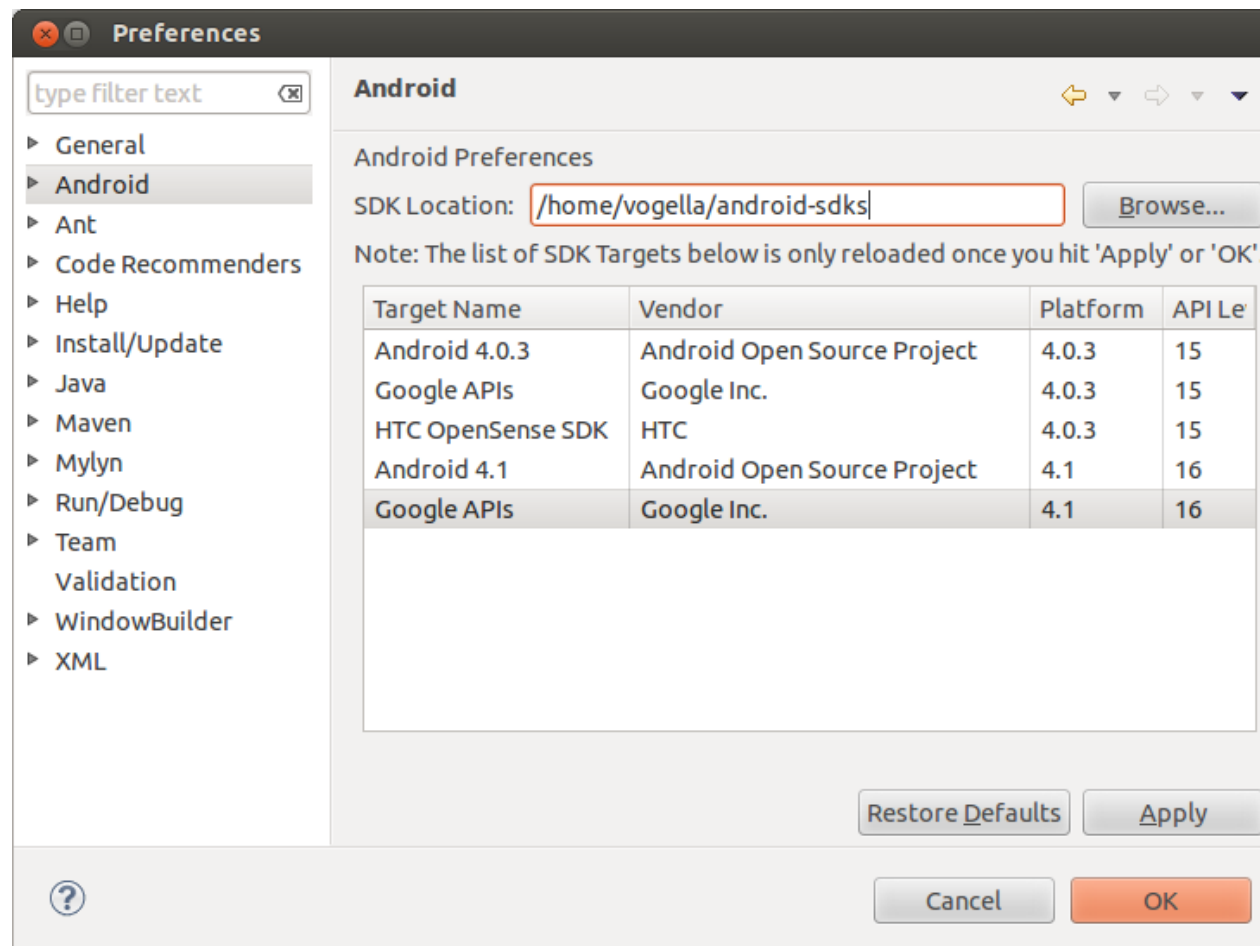
8.4. Manual installation of the Android SDK

After the installation of the ADT the Eclipse tooling allows to download the Android SDK automatically. Alternatively you can also manually download the Android SDK from the Android SDK download page.

<http://developer.android.com/sdk/index.html>

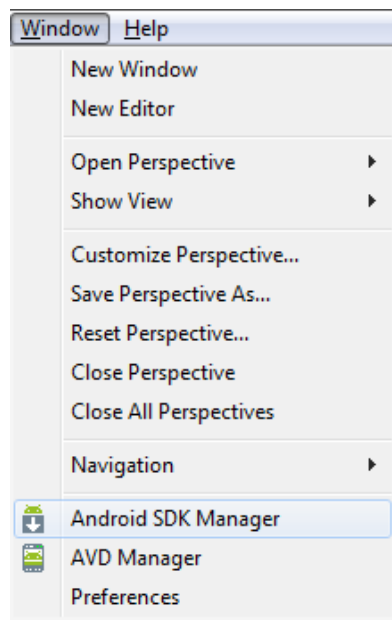
The download contains a zip file, which you can extract to any place in your file system, e.g. on my Linux system I placed it into the `/home/vogella/android-sdks` folder. Avoid using spaces in the path name, otherwise you may experience problems with the usage of the Android SDK.

You also have to define the location of the Android SDK in the Eclipse Preferences. In Eclipse open the Preferences dialog via the menu *Windows* → *Preferences*. Select Android and enter the installation path of the Android SDK.



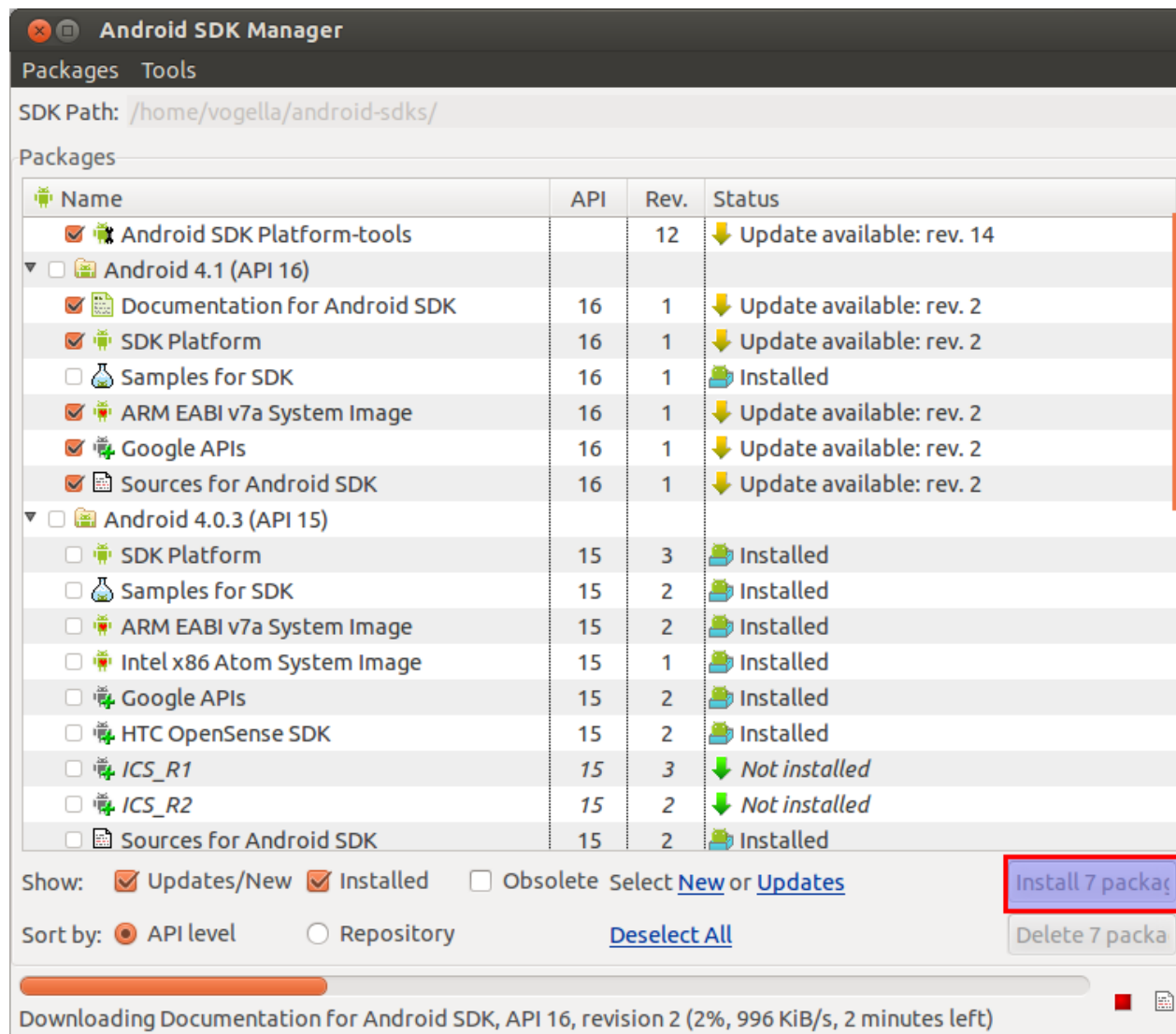
8.5. Install a specific Android version

The *Android SDK Manager* allows you to install specific versions of Android. Select *Window* → *Android SDK Manager* from the Eclipse menu.



The dialog allows you to install and delete packages.

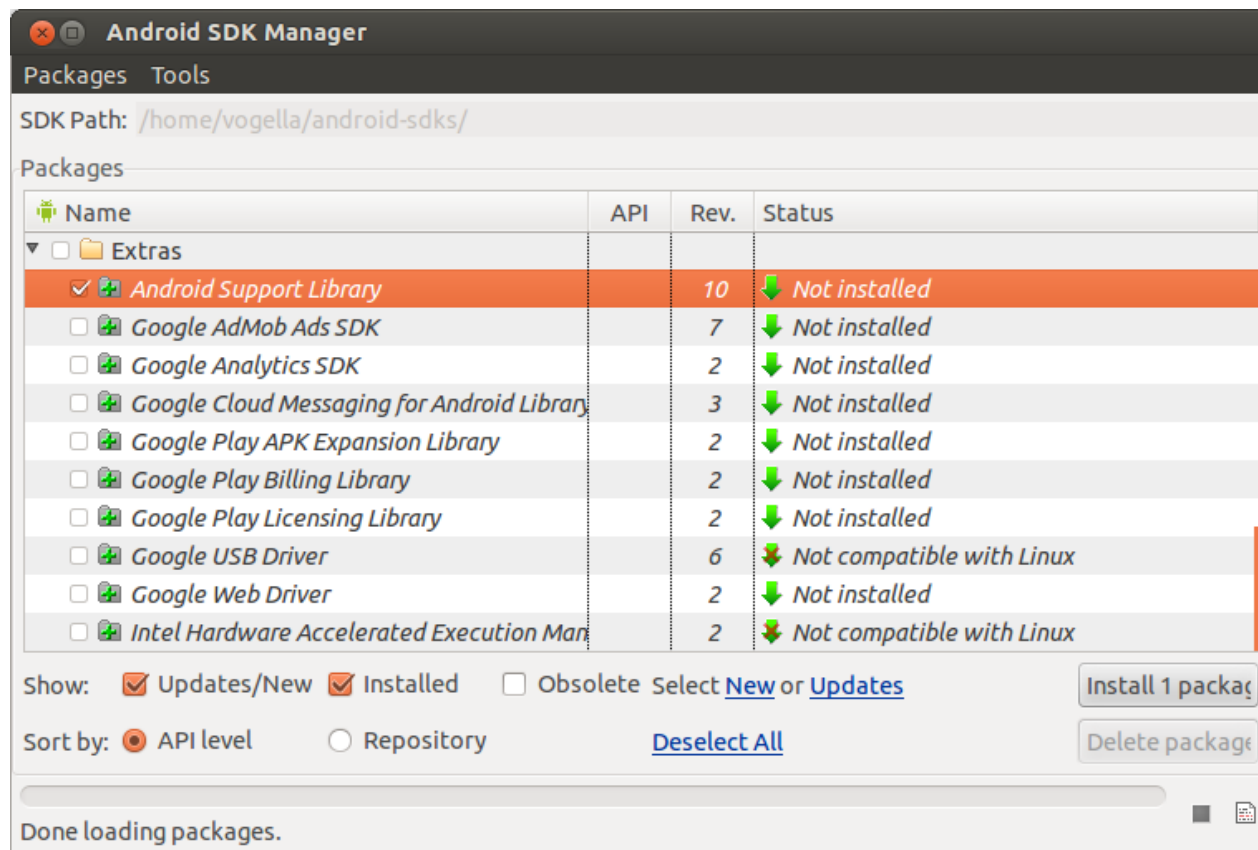
Select *Available packages* and open the *Third Party Add-ons* . Select the latest Google API of the SDK and press the *Install* button.



Press the *Install* button and confirm the license for all packages. After the installation completes, restart Eclipse.

8.6. Install support library

In the *Android SDK Manager* select *Extras* and install the *Android support library*.



8.7. Android Source Code

During Android development it is very useful to have the Android source code available.

As of Android 4.0 the Android development tools provides also the source code. You can download it via the Android SDK Manager by selecting the *Sources for Android SDK*.

The sources are downloaded to the source directory located in `path_to_android_sdk/sources/android-xx`. `xx` is the API level of Android, e.g. 15 for the Android 4.0.4 version.

To connect the sources with the `android.jar` file in your Android project, right click on your `android.jar` in the Eclipse Package Explorer and select *Properties* → *Java Source Attachment*. Type in the source directory name and press the *OK* button.

Afterwards you can browse through the source code.

9. Android virtual device - Emulator

9.1. What is the Android Emulator?

The Android Development Tools (ADT) include an emulator to run an Android system. The emulator behaves like a real Android device (in most cases) and allows you to test your application without having a real device.

You can configure the version of the Android system you would like to run, the size of the SD card, the screen resolution and other relevant settings. You can define several of them with different configurations.

These devices are called *Android Virtual Device* and you can start several in parallel.

9.2. Google vs. Android AVD

During the creation of an AVD you decide if you want an Android device or a Google device.

An AVD created for Android will contain the programs from the Android Open Source Project. An AVD created for the Google API's will also contain several Google applications, most notable the Google Maps application.

If you want to use functionality which is only provided via the Google API's, e.g. Google Maps you must run this application on an AVD with Google API's.

9.3. Emulator Shortcuts

The following shortcuts are useful for working with the emulator.

Alt+Enter Maximizes the emulator. Nice for demos.

Ctrl+F11 changes the orientation of the emulator.

F8 Turns network on / off.

9.4. Parameter

The graphics of the emulator can use the native GPU of the computer. This makes the rendering in the emulator very fast. To enable this, add the `GPU Emulation` property to the device configuration and set it to `true`.

Edit Android Virtual Device (AVD)

Name:

Target:

CPU/ABI:

SD Card:

☒ Size:

☐ File:

Snapshot: ☐ Enabled

Skin:

☐ Built-in:

☒ Resolution: x

Hardware:

Property	Value
Max VM application heap size	48
Device ram size	512
GPU emulation	yes

☐ Override the existing AVD with the same name

You can also set the `Enabled` flag for Snapshots. This will save the state of the emulator and will let it start much faster. Unfortunately currently native GPU rendering and Snapshots do not work together.

Android devices do not have to have hardware button. If you want to create such an AVD, add the *Hardware Back/Home keys* property to the device configuration and set it to *false*.

Edit Android Virtual Device (AVD)

Name:

Target:

CPU/ABI:

SD Card: ☒ Size: ☐ File:

Snapshot: ☐ Enabled

Skin: ☒ Built-in: ☐ Resolution: x

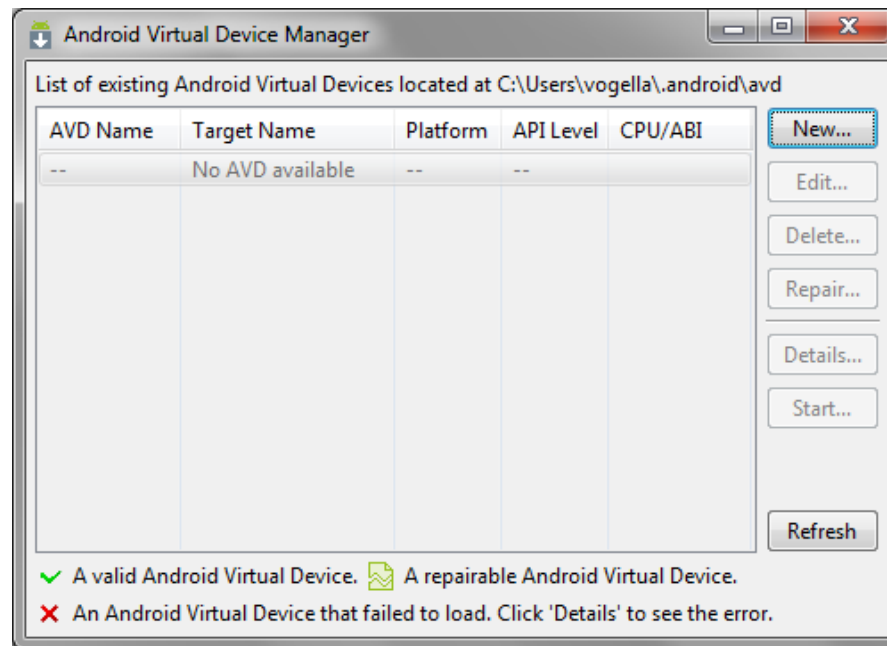
Hardware:

Property	Value
Hardware Back/Home keys	no
Abstracted LCD density	160
Max VM application heap size	48

☐ Override the existing AVD with the same name

10. Tutorial: Create and run Android Virtual Device

To define an Android Virtual Device (ADV) open the *AVD Manager* dialog via *Windows* → *AVD Manager* and press *New* button.



Enter the values similar to the following screenshot.

×

Create new Android Virtual Device (AVD)

Name:

Google

Target:

Google APIs (Google Inc.) - API Level 15

CPU/ABI:

ARM (armeabi-v7a)

SD Card:

☒ Size: 300 MiB

☐ File: Browse...

Snapshot:

☒ Enabled

Skin:

☒ Built-in: HVGA

☐ Resolution: x

Hardware:

Property	Value
Abstracted LCD dens	160
Max VM application h	48
Device ram size	512

New...

Delete

☐ Override the existing AVD with the same name

Cancel

Create AVD

Select the *Enabled for Snapshots* box. This will make the second start of the virtual device much faster. Afterwards press the *Create AVD* button. This will create the AVD configuration and display it under the *Virtual devices*.

To test if your setup is correct, select your your new entry and press the *Start* button

Create new Android Virtual Device (AVD)

Name:

Google

Target:

Google APIs (Google Inc.) - API Level 15

CPU/ABI:

ARM (armeabi-v7a)

SD Card:

☒ Size: 300 MiB

☐ File: Browse...

Snapshot:

☒ Enabled

Skin:

☒ Built-in: HVGA

☐ Resolution: x

Hardware:

Property	Value
Abstracted LCD dens	160
Max VM application h	48
Device ram size	512

New...

Delete

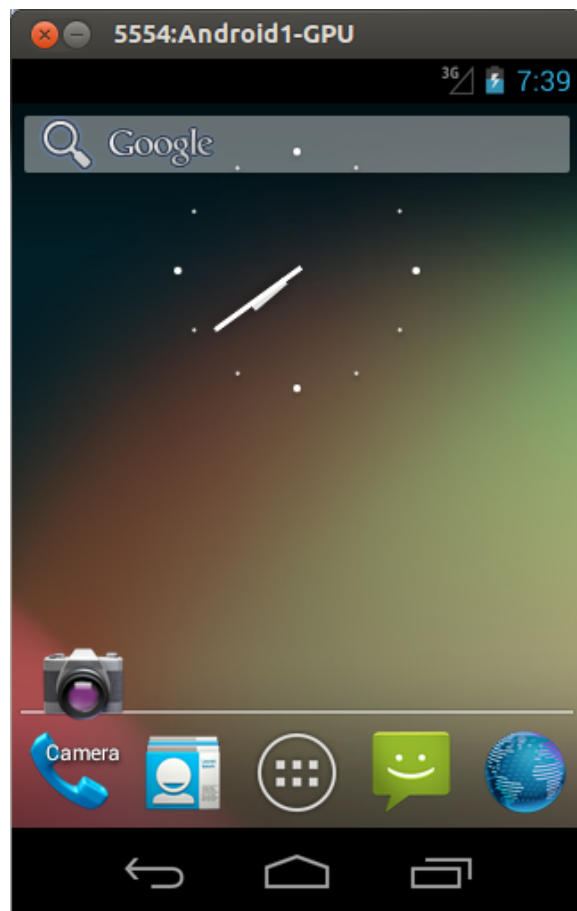
☐ Override the existing AVD with the same name

Cancel

Create AVD

After some time your AVD starts. Do not interrupt this startup process, as this might corrupt the AVD.

After the AVD started, you can use the AVD via the mouse and via the virtual keyboard of the emulator.



11. Solving Android development problems

Things are not always working as they should. You find a list of typical Android development problems and their solution under the following link: [Solutions for common Android development problems](#).

12. Conventions for the tutorials

12.1. API version, package and application name

The tutorials of this document have been developed and tested with Android 4.1, API Level 16. Please use this version for all tutorials in this tutorial. Higher versions of the API level should also work. A lower version

of the Android API might also work, but if you face issues, try the recommended version.

The base package for the projects is always the same as the project name, e.g. if you are asked to create a project called *de.vogella.android.example.test*, then the corresponding package name is *de.vogella.android.example.test*.

The application name, which must be entered on the Android project generation wizard, will not be predefined. Choose a name you like.

12.2. Warning messages for Strings

The Android development tools show warnings, if you use hard-coded strings, for example in layout files. For real applications you should use String resource files. To simplify the creation of the examples, we use Strings directly. Please ignore the corresponding warnings.

13. Tutorial: Your first Android project

13.1. Install the demo application

This application is also available on the Android Marketplace under **Android Temperature converter**.

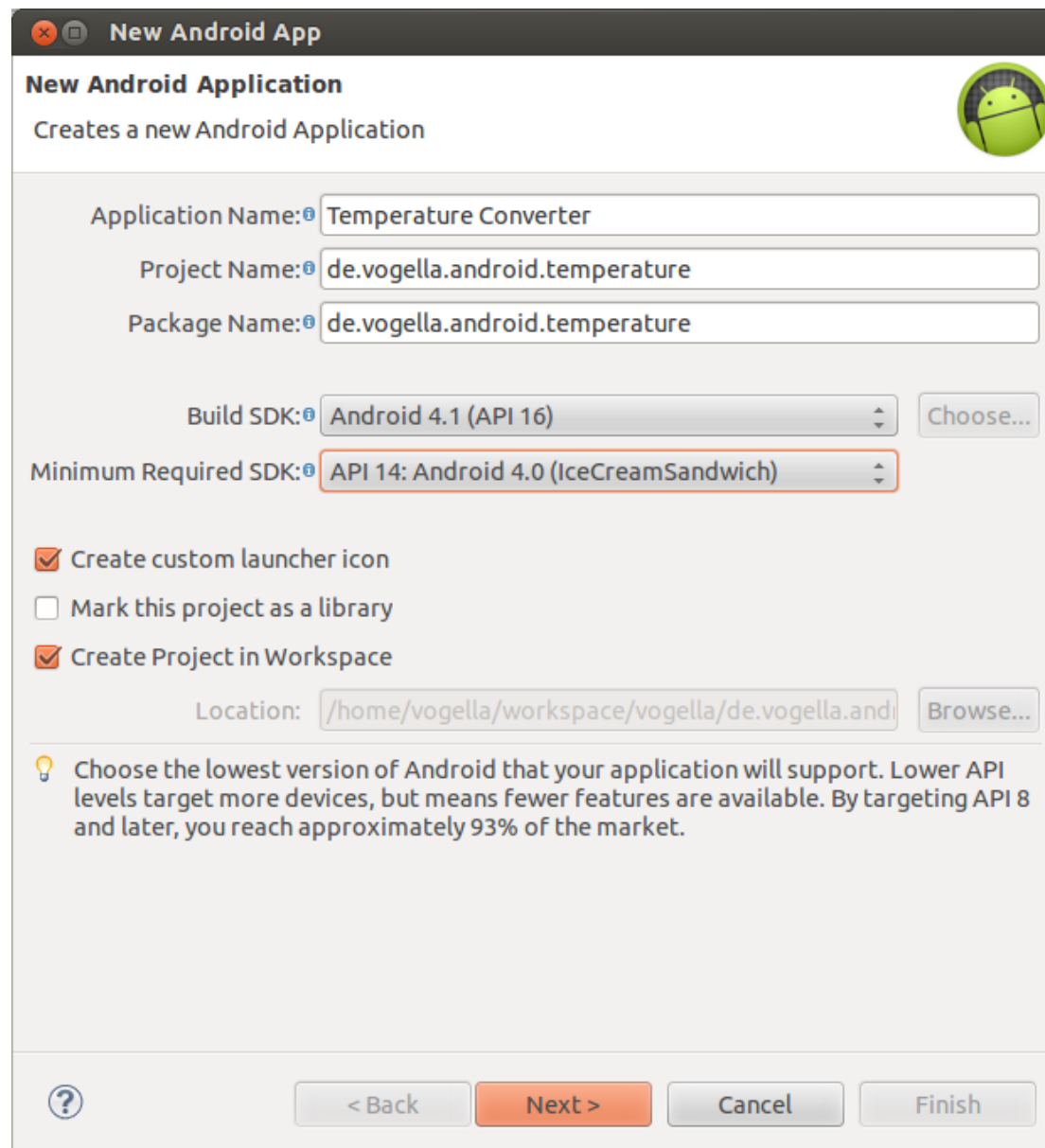
Alternatively you can also scan the following barcode with your Android smartphone to install it via the Google Play application.



13.2. Create Project

Select *File* → *New* → *Other* → *Android* → *Android Application Project* to create a new Android project.

Use *Temperature Converter* as *Application name* and *de.vogella.android.temperature* as project and package name. Select the latest Android SDK as *Build SDK* and *API 14* as *Minimum Required SDK*. After entering this data, press the *Next* button.



New Android App

New Android Application
Creates a new Android Application

Application Name:

Project Name:

Package Name:

Build SDK:


Minimum Required SDK:

☒ Create custom launcher icon

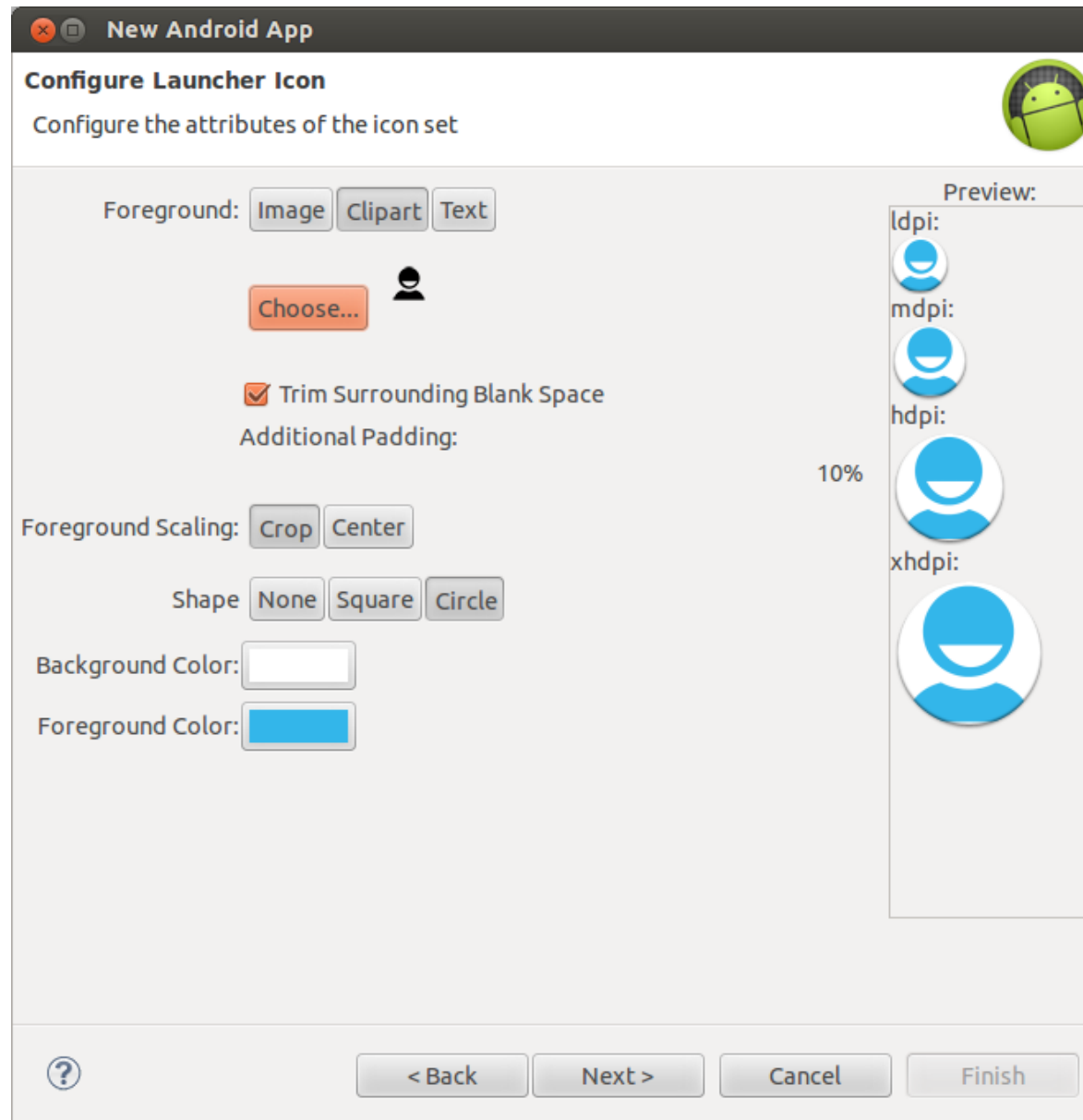
☐ Mark this project as a library

☒ Create Project in Workspace

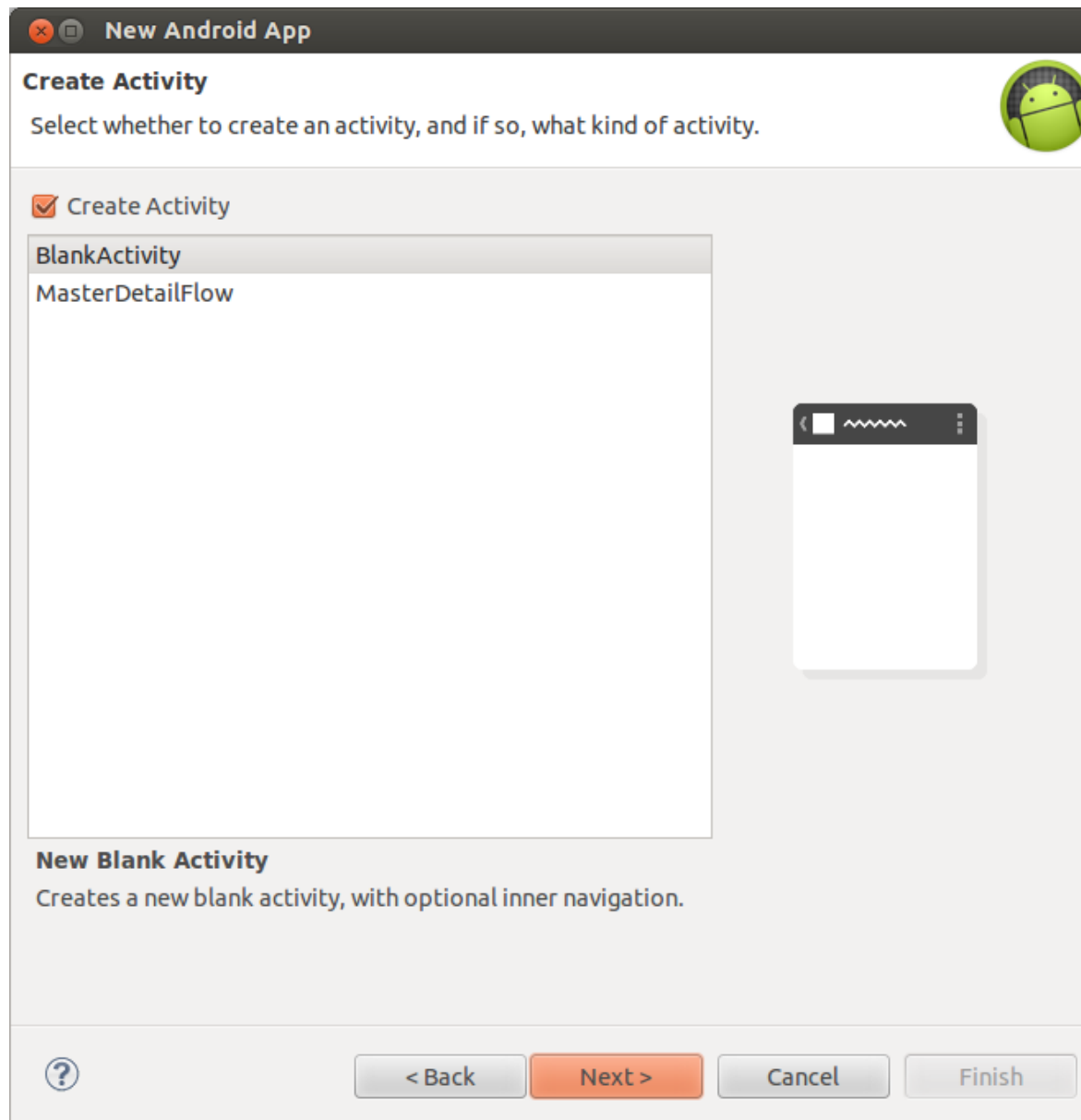
Location:

 Choose the lowest version of Android that your application will support. Lower API levels target more devices, but means fewer features are available. By targeting API 8 and later, you reach approximately 93% of the market.

The next screen allow you to create a *Launcher Icon* for your application. Accept the defaults and press the *Next* button.



Select the *BlankActivity* template and press the *Next* button.



On the next dialog change *Title* attribute to *Temperature Converter*. Leave the rest as is.

New Android App

New Blank Activity

Creates a new blank activity, with optional inner navigation.

Activity Name[?]MainActivity

Layout Name[?]activity_main

Navigation Type[?]None

Hierarchical Parent[?]

Title[?]Temperature Converter

💡 The name of the activity. For launcher activities, the application title.

?

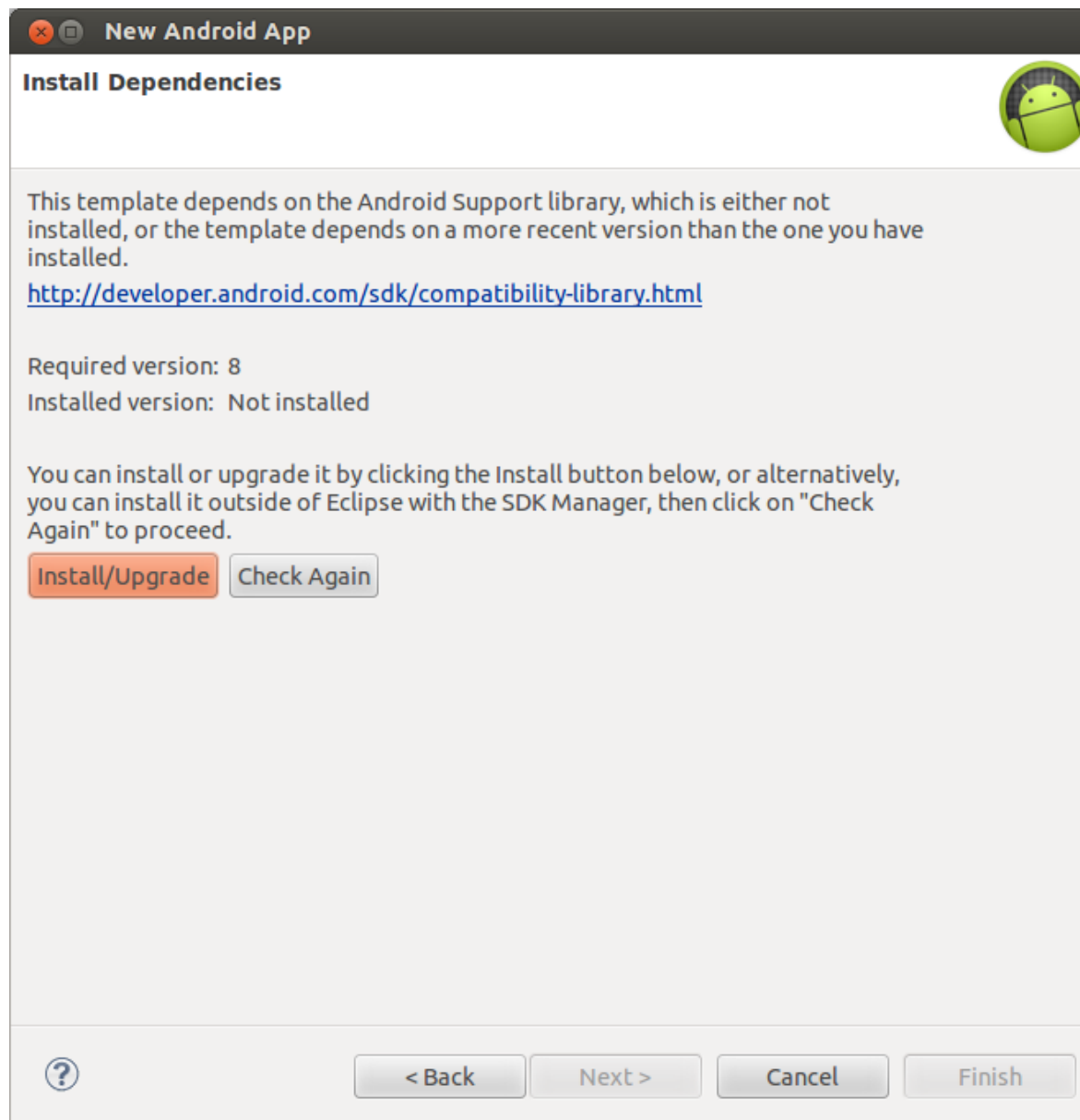
< Back

Next >

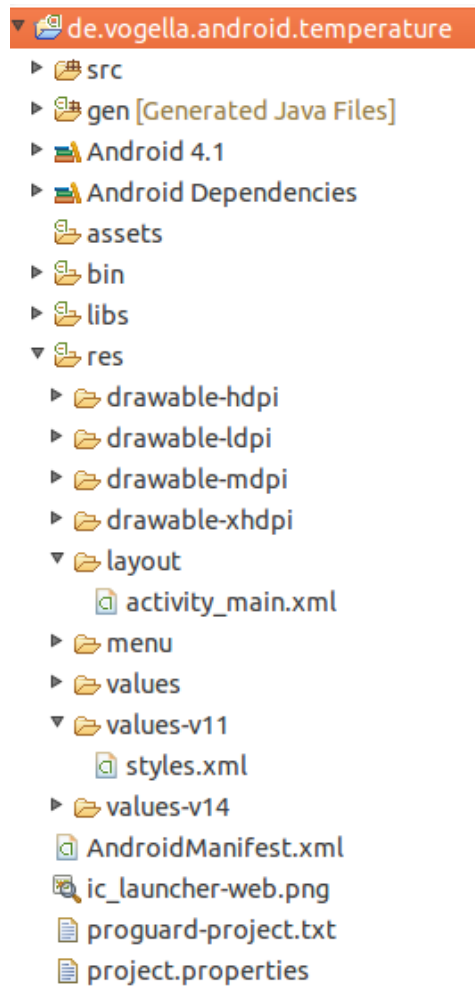
Cancel

Finish

Press the *Finish* button. The wizard may prompt you to install the Support library. If you are prompted, select to install it.



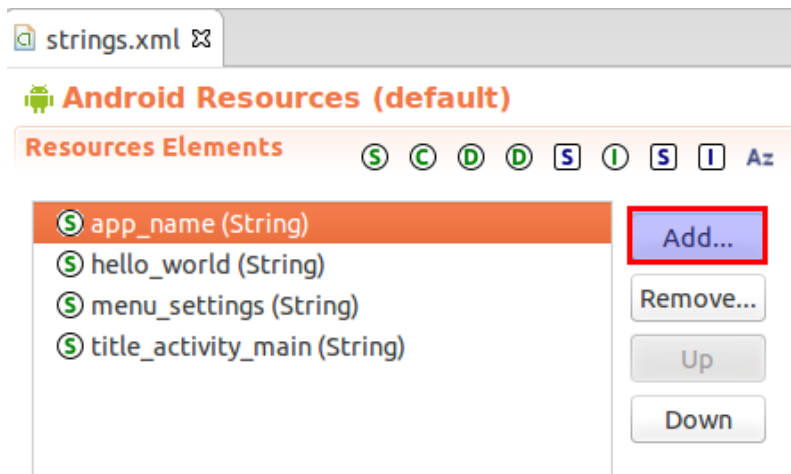
After the wizard finishes the following project should be created.



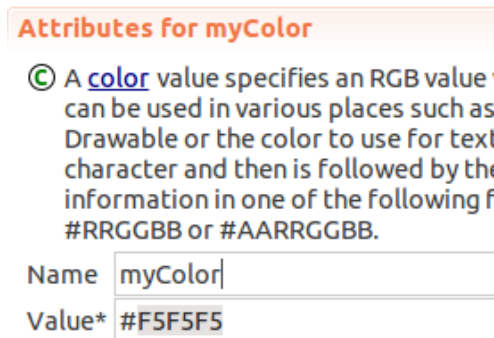
13.3. Create attributes

Android allows you to create static attributes, e.g. Strings or colors. These attributes can for example be used in your XML layout files or referred to via Java source code.

Select the `res/values/string.xml` file and press the *Add* button.



Select the *Color* entry in the following dialog and press the *OK* button. Enter *myColor* as the name and *#F5F5F5* as the value.



Add more attributes, this time of the *String* type. String attributes allow the developer to translate the application at a later point.

Table 2. String Attributes

Name	Value
celsius	to Celsius
fahrenheit	to Fahrenheit

Switch to the XML representation and validate that the values are correct.

```
<resources>

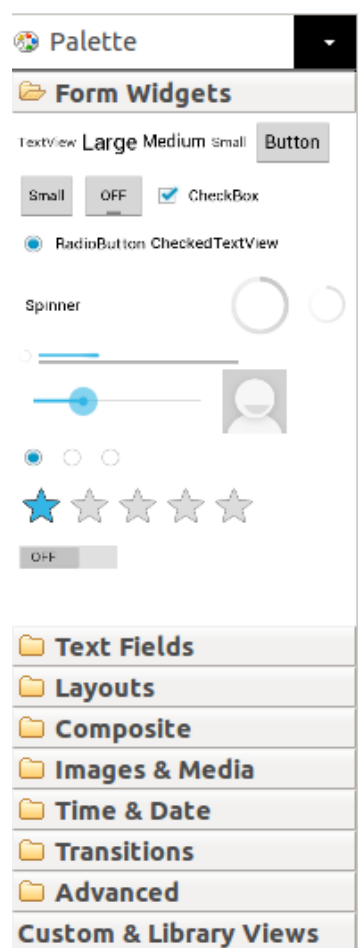
    <string name="app_name">Temparature Convertor</string>
    <string name="hello_world">Hello world!</string>
    <string name="menu_settings">Settings</string>
    <string name="title_activity_main">Temparature Convertor</string>
    <color name="myColor">#3399CC</color>
    <string name="celsius" >to Celsius</string>
    <string name="fahrenheit">to Fahrenheit</string>
    <string name="calc">Calculate</string>

</resources>
```

13.4. Add Views

Select the `res/layout/activity_main.xml` file and open the Android editor via a double-click. This editor allows you to create the layout via drag and drop or via the XML source code. You can switch between both representations via the tabs at the bottom of the editor. For changing the position and grouping elements you can use the Eclipse *Outline View*.

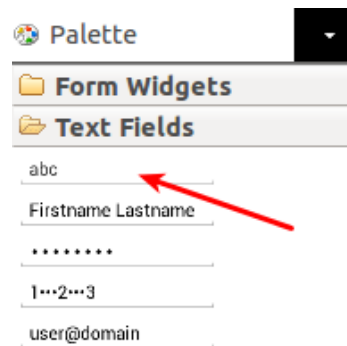
The following shows a screenshot of the *Palette* side of this editor. from which you can drag and drop new user interface components into your layout. Please note that the *Palette* view changes frequently so your view might be a bit different.



You will now create the layout for your Android application.

Right-click on the existing *Hello World!* text object in the layout. Select *Delete* from the popup menu to remove the text object.

Afterwards select the *Text Fields* section in the *Palette* and locate the *Plain Text* (via the tooltip).



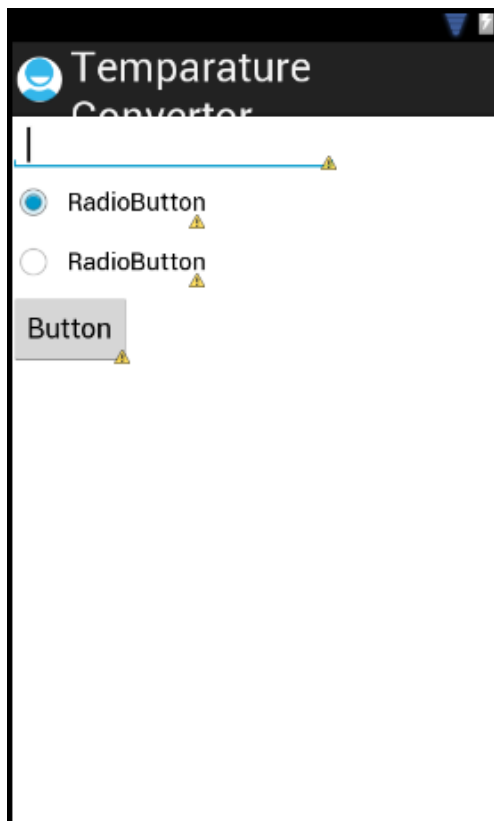
All entries in the *Text Fields* section define text fields. The different entries define additional attribute for them, e.g. if a text field should only contain numbers.

Drag this onto the layout to create a text input field.

Afterwards select the *Form Widgets* section in the *Palette* and drag a *RadioGroup* entry into the layout. The number of radio buttons added to the radio button group depends on your version of Eclipse. Make sure there are two radio buttons by deleting or adding radio buttons to the group.

Drag a Button from the *Form Widgets* section into the layout.

The result should look like the following screenshot.



Switch to the XML tab of your layout file and verify that the file looks similar to the following listing. ADT changes the templates very fast, so your XML might look slightly different.

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <EditText
        android:id="@+id/editText1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true"
        android:ems="10" >

        <requestFocus />
    </EditText>

    <RadioGroup
```

```

        android:id="@+id/radioGroup1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_below="@+id/editText1" >

        <RadioButton
            android:id="@+id/radio0"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:checked="true"
            android:text="RadioButton" />

        <RadioButton
            android:id="@+id/radio1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="RadioButton" />
    </RadioGroup>

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_below="@+id/radioGroup1"
        android:text="Button" />

</RelativeLayout>

```

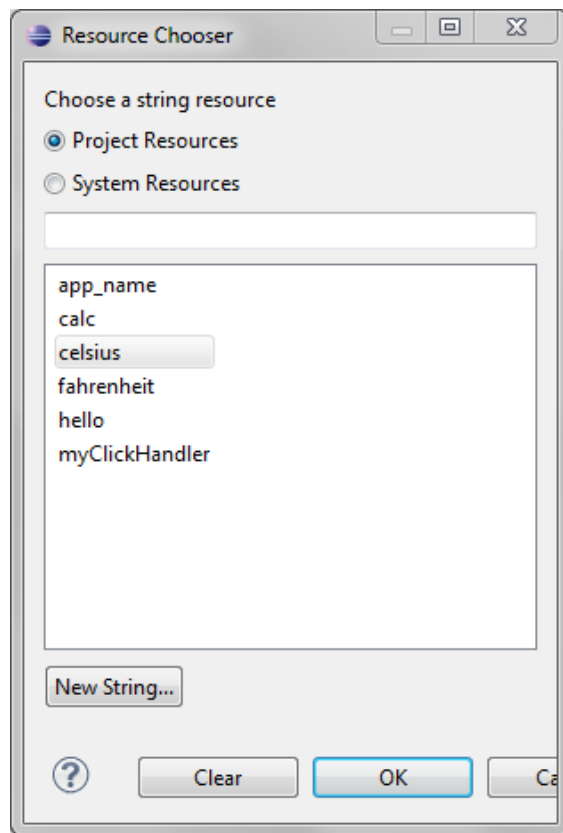
13.5. Edit View properties

If you select a user interface component (an instance of `View`), you can change its properties via the Eclipse *Properties View*. Most of the properties can be changed via the right mouse menu. You can also edit properties of fields directly in XML. Changing properties in the XML file is much faster, if you know what you want to change. But the right mouse functionality is nice, if you are searching for a certain property.

Open your layout file.

Use the right mouse click on the first radio button to assign the `celsius` String attribute to its `text` property. Assign the `fahrenheit` string attribute to the `text` property of the second radio button.

Edit Text...	
Edit ID...	
Edit Style...	
Layout Width	▸
Layout Height	▸
Other Properties	▸
Extract Include...	
Extract Style...	
Wrap in Container...	
Remove Container...	
Change Widget Type...	
radioGroup1	▸
RelativeLayout	▸
Select	▸
Cu <u>t</u>	Ctrl+X
<u>C</u> opy	Ctrl+C
Paste	Ctrl+V
<u>D</u> ele <u>t</u> e	Delete
Play Animation	▸
Export Screenshot...	
Show Included In	▸
Sho <u>w</u> In	▸
Input Methods	▸



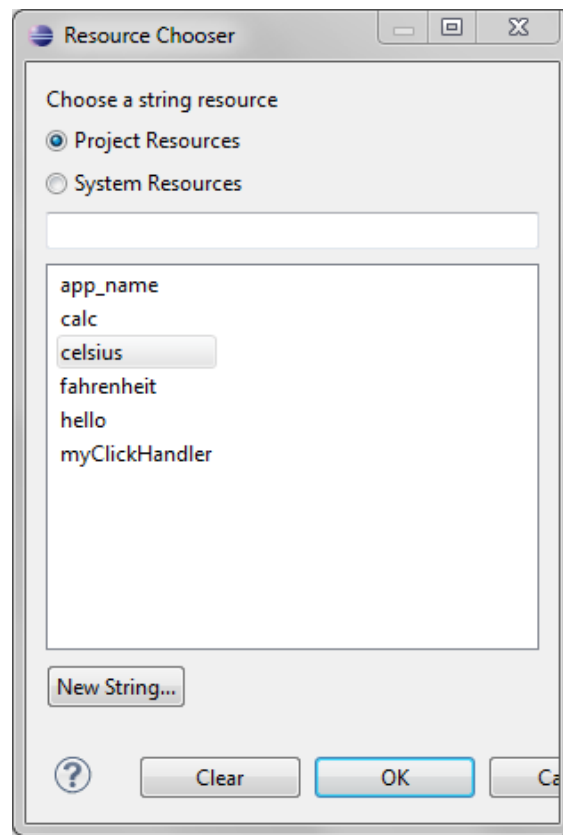
From now on, I assume you are able to use the properties menu on user interface components. You can always either edit the XML file or modify the properties via right mouse click.

Set the *Checked* property to true for the first *RadioButton*.

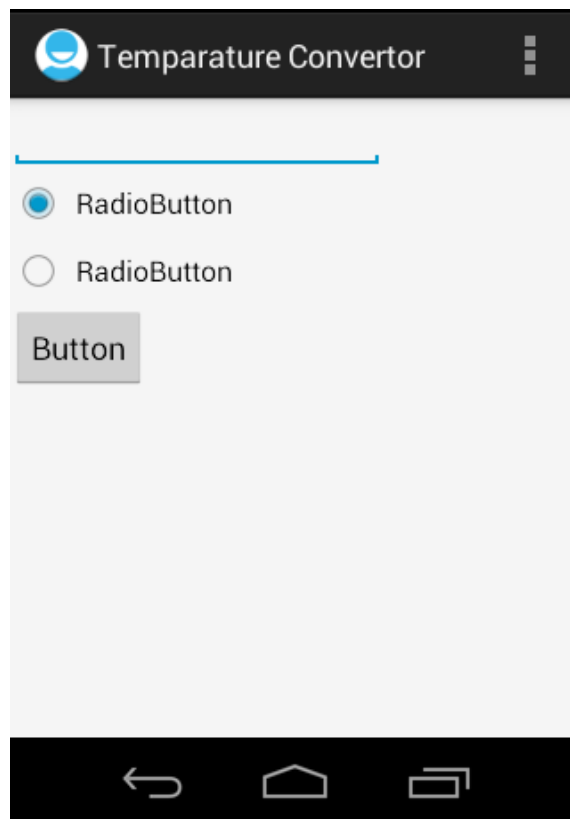
Assign *calc* to the text property of your button and assign the value *onClick* to the *onClick* property.

Set the *Input type* property to *numberSigned* and *numberDecimal* on the *EditText*.

All your user interface components are contained in a layout. Assign a background color to this *Layout*. Right-click on an empty space in *Graphical Layout* mode, then select *Other Properties* → *All by Name* → *Background*. Select *Color* and then select *myColor* in the dialog.



Afterwards the background should change to the *whitesmoke* color. It might difficult to see the difference.



Switch to the `activity_main.xml` tab and verify that the XML is correct.

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/myColor" >

    <EditText
        android:id="@+id/editText1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true"
        android:ems="10"
        android:inputType="numberSigned|numberDecimal"
        >

        <requestFocus />
    </EditText>
```

```

<RadioGroup
    android:id="@+id/radioGroup1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentLeft="true"
    android:layout_below="@+id/editText1" >

    <RadioButton
        android:id="@+id/radio0"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:checked="true"
        android:text="@string/celsius" />

    <RadioButton
        android:id="@+id/radio1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/fahrenheit" />
</RadioGroup>

<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentLeft="true"
    android:layout_below="@+id/radioGroup1"
    android:onClick="onClick"
    android:text="@string/calc" />

</RelativeLayout>

```

13.6. Change the Activity source code

During the generation of your new Android project you specified that an Activity called MainActivity should be created. The project wizard created the corresponding Java class.

Change your MainActivity class to the following listing. Note that the onClick will be called based on the onClick property of your button. I use the same name as this is easier to remember.

```

package de.vogella.android.temperature;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.RadioButton;
import android.widget.Toast;

```

```

public class MainActivity extends Activity {
    private EditText text;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        text = (EditText) findViewById(R.id.editText1);
    }

    // This method is called at button click because we assigned the name to the
    // "OnClick property" of the button
    public void onClick(View view) {
        switch (view.getId()) {
            case R.id.button1:
                RadioButton celsiusButton = (RadioButton) findViewById(R.id.radio0);
                RadioButton fahrenheitButton = (RadioButton) findViewById(R.id.radio1);
                if (text.getText().length() == 0) {
                    Toast.makeText(this, "Please enter a valid number",
                        Toast.LENGTH_LONG).show();
                    return;
                }

                float inputValue = Float.parseFloat(text.getText().toString());
                if (celsiusButton.isChecked()) {
                    text.setText(String
                        .valueOf(convertFahrenheitToCelsius(inputValue)));
                    celsiusButton.setChecked(false);
                    fahrenheitButton.setChecked(true);
                } else {
                    text.setText(String
                        .valueOf(convertCelsiusToFahrenheit(inputValue)));
                    fahrenheitButton.setChecked(false);
                    celsiusButton.setChecked(true);
                }
                break;
            }
        }

        // Converts to celsius
        private float convertFahrenheitToCelsius(float fahrenheit) {
            return ((fahrenheit - 32) * 5 / 9);
        }

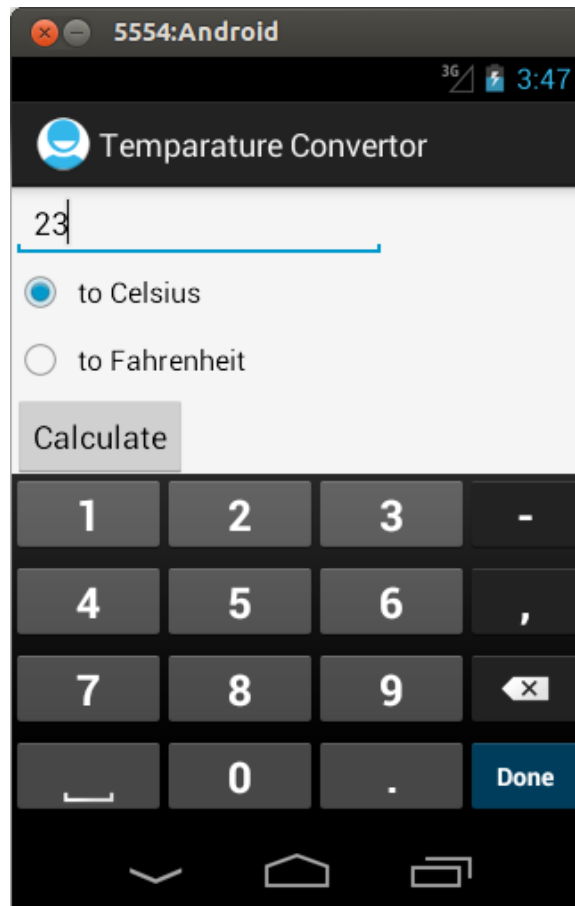
        // Converts to fahrenheit
        private float convertCelsiusToFahrenheit(float celsius) {
            return ((celsius * 9) / 5) + 32;
        }
    }
}

```

13.7. Start Project

To start the Android Application, select your project, right click on it, and select *Run-As* → *Android Application*. If an emulator is not yet running, it will be started. Be patient, the emulator starts up very slowly.

Type in a number, select your conversion and press the button. The result should be displayed and the other option should get selected.

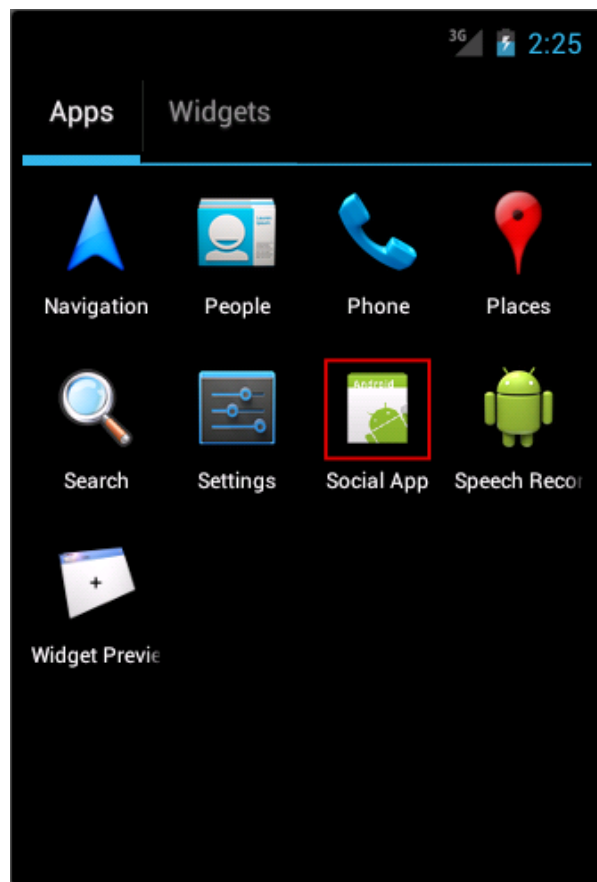


14. Starting an installed application

After you run your application on the virtual device, you can start it again on the device. If you press the

Home button you can select your application.





15. Layout Manager and ViewGroups

15.1. Available Layout Manager

A layout manager is a subclass of `ViewGroup` and is responsible for the layout of itself and its child Views. Android supports different default layout managers.

As of Android 4.0 the most relevant layout managers are `LinearLayout`, `FrameLayout`,

`RelativeLayout` and `GridLayout`.

All layouts allow the developer to define attributes. Children can also define attributes which may be evaluated by their parent layout.

`AbsoluteLayout` is deprecated and `TableLayout` can be implemented more effectively via `GridLayout`

Children can specify their desired width and height via the following attributes.

Table 3. Width and height definition

<code>android:layout_width</code>	Defines the width of the widget.
<code>android:layout_height</code>	Defines the height of the widget.

Widgets can use fixed sizes, e.g. with the *dp* definition, for example `100dp`. While *dp* is a fixed size it will scale with different device configurations.

The *match_parent* value tells the widget to maximize the widget in its parent. The *wrap_content* value tells the layout to allocate the minimum amount so that the widget is rendered correctly.

15.2. `LinearLayout`

`LinearLayout` puts all its child elements into a single column or row depending on the `android:orientation` attribute. Possible values for this attribute are `horizontal` and `vertical`, `horizontal` is the default value.

`LinearLayout` can be nested to achieve more complex layouts.

`LinearLayout` supports assigning a weight to individual children via the `android:layout_weight` layout parameter. This value specifies how much of the extra space in the layout is allocated to the `View`. If for example you have two widgets and the first one defines a *layout_weight* of 1 and the second of 2, the first will get 1/3 of the available space and the other one 2/3. You can also set the `layout_width` to zero to have always a certain ratio.

15.3. RelativeLayout

RelativeLayout allow to position the widget relative to each other. This allows for complex layouts.

A simple usage for RelativeLayout is if you want to center a single component. Just add one component to the RelativeLayout and set the `android:layout_centerInParent` attribute to `true`.

```
<?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"
    android:orientation="vertical" >

    <ProgressBar
        android:id="@+id/progressBar1"
        style="?android:attr/progressBarStyleLarge"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        />

</RelativeLayout>
```

15.4. GridLayout

GridLayout was introduced with Android 4.0. This layout allows you to organize a view into a Grid. GridLayout separates its drawing area into: rows, columns, and cells.

You can specify how many columns you want for define for each View in which row and column it should be placed and how many columns and rows it should use. If not specified GridLayout uses defaults, e.g. one column, one row and the position of a View depends on the order of the declaration of the Views.

15.5. ScrollView

The ScrollView class can be used to contain one View that might be too big to fit on one screen. ScrollView will in this case display a scroll bar to scroll the context.

Of course this View can be a layout which can then contain other elements.

16. Tutorial: ScrollView

Create an android project "de.vogella.android.scrollview" with the activity "ScrollView". Create the following layout and class.

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

    <LinearLayout
        android:id="@+id/LinearLayout01"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical" >

        <TextView
            android:id="@+id/TextView01"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:paddingLeft="8dip"
            android:paddingRight="8dip"
            android:paddingTop="8dip"
            android:text="This is a header"
            android:textAppearance="?android:attr/textAppearanceLarge" >
        </TextView>

        <TextView
            android:id="@+id/TextView02"
            android:layout_width="wrap_content"
            android:layout_height="match_parent"
            android:layout_weight="1.0"
            android:text="@+id/TextView02" >
        </TextView>

        <LinearLayout
            android:id="@+id/LinearLayout02"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" >

            <Button
                android:id="@+id/Button01"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_weight="1.0"
                android:text="Submit" >
```

```

        </Button>

        <Button
            android:id="@+id/Button02"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1.0"
            android:text="Cancel" >

        </Button>
    </LinearLayout>
</LinearLayout>

</ScrollView>

```

```

package de.vogella.android.scrollview;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;

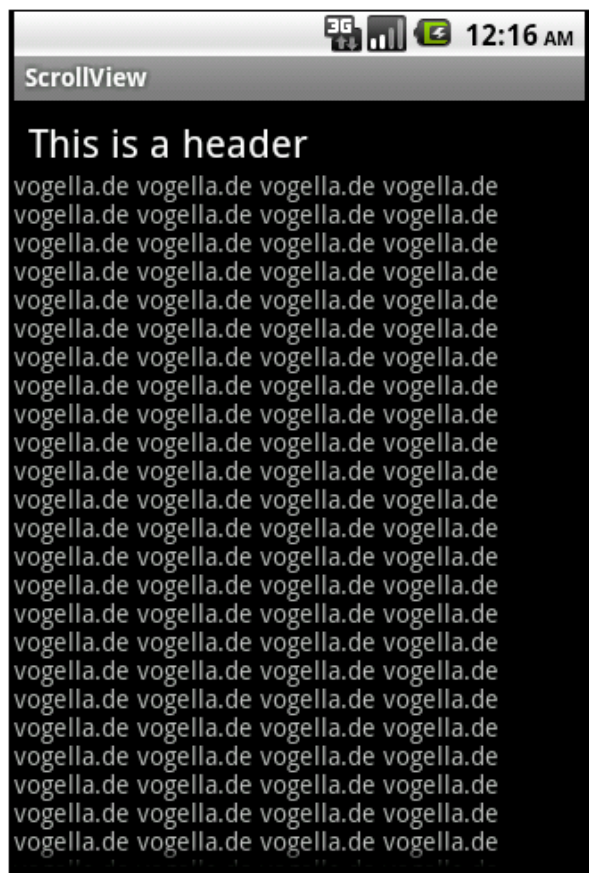
public class ScrollView extends Activity {

    /** Called when the activity is first created. */

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        TextView view = (TextView) findViewById(R.id.TextView02);
        String s="";
        for (int i=0; i < 100; i++) {
            s += "vogella.com ";
        }
        view.setText(s);
    }
}

```

The attribute "android:fillViewport="true"" ensures that the scrollview is set to the full screen even if the elements are smaller than one screen and the "layout_weight" tell the android system that these elements should be extended.



17. Fragments

17.1. Fragments Overview

Fragments are components which run in the context of an *Activity*. A *Fragment* components encapsulate application code so that it is easier to reuse it and to support different sized devices.

Fragments have their own lifecycle and their own user interface. They can be defined via layout files or via coding.

If an *Activity* stops its *Fragments* will also be stopped; if an *Activity* is destroyed its *Fragments* will also get destroyed.

If a *Fragment* component is defined in an XML layout file, the `android:name` attribute points to the corresponding class.

The base class for *Fragments* is `android.app.Fragment`. For special purposes you can also use more special classes, like `ListFragment` or `DialogFragment`.

The `onCreateView()` method is called by Android once the *Fragment* should create its user interface. Here you can inflate an layout via the `inflate()` method call of the `Inflator` object passed as a parameter to this method. .

The `onStart()` method is called once the *Fragment* gets visible.

Fragments can be dynamically added and removed from an *Activity* via *Fragment* transactions. This will add the action to the history stack of the *Activity*, i.e. this will allow to revert the *Fragment* changes in the *Activity* via the back button.

17.2. When to use Fragments

Fragments make it easy to re-use components in different layouts, e.g. you can build single-pane layouts for handsets (phones) and multi-pane layouts for tablets.

This is not limited to tablets; for example you can use *Fragments* also to support different layout for landscape and portrait orientation. But as tablets offer significantly more space you typically include more views into the layout and *Fragments* makes that easier.

The typical example is a list of items in an activity. On a tablet you see the details immediately on the same screen on the right hand side if you click on item. On a handset you jump to a new detail screen. The following discussion will assume that you have two *Fragments* (main and detail) but you can also have more. We will also have one main activity and one detailed activity. On a tablet the main activity contains both *Fragments* in its layout, on a handheld it only contains the main fragment.

To check for an fragment you can use the `FragmentManager`.

```
DetailFragment fragment = (DetailFragment) getFragmentManager().
    findFragmentById(R.id.detail_frag);
if (fragment==null || ! fragment.isInLayout()) {
    // start new Activity
}
else {
```

```
fragment.update(...);  
}
```

To create different layouts with *Fragments* you can:

- Use one activity, which displays two *Fragments* for tablets and only one on handset devices. In this case you would switch the *Fragments* in the activity whenever necessary. This requires that the fragment is not declared in the layout file as such *Fragments* cannot be removed during runtime. It also requires an update of the action bar if the action bar status depends on the fragment.
- Use separate activities to host each fragment on a handset. For example, when the tablet UI uses two *Fragments* in an activity, use the same activity for handsets, but supply an alternative layout that includes just one fragment. When you need to switch *Fragments*, start another activity that hosts the other fragment.

The second approach is the most flexible and in general preferable way of using *Fragments*. In this case the main activity checks if the detail fragment is available in the layout. If the detailed fragment is there, the main activity tells the fragment that it should update itself. If the detail fragment is not available the main activity starts the detailed activity.

It is good practice that *Fragments* do not manipulate each other. For this purpose a *Fragment* typically implements an interface to get new data from its host *Activity*.

17.3. Configuration changes in Fragments

Fragments instance can be configured to retained across configuration changes. Just use the `setRetainInstance(boolean)` method.

This only works if

18. Fragments Tutorial

18.1. Overview

The following tutorial demonstrates how to use *Fragments*. The application will use layouts with different fragments depending on portrait and landscape mode.

In portrait mode the *RssfeedActivity* will show one *Fragment*. From this *Fragments* the user can navigate to another *Activity* which contains another *Fragment*.

In landscape mode *RssfeedActivity* will show both *Fragments* side by side.

18.2. Create Project

Create a new Android project with the following data.

Table 4. Android project

Property	Value
Application Name	RSS Reader
Project Name	com.example.android.rssfeed
Package name	com.example.android.rssfeed
Template	BlankActivity
Activity	RssfeedActivity
Layout	activity_rssfeed

18.3. Create standard layouts

Create or change the following layout files in the *res/layout/* folder.

Create a new layout file called *fragment_rssitem_detail.xml*. This layout file will be used by the *DetailFragment*.

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

```

<TextView
    android:id="@+id/detailsText"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    android:layout_gravity="center_horizontal|center_vertical"
    android:layout_marginTop="20dip"
    android:text="Default Text"
    android:textAppearance="?android:attr/textAppearanceLarge"
    android:textSize="30dip" />

</LinearLayout>

```

Create a new layout file called *fragment_rsslist_overview.xml*. This layout file will be used by the *MyListFragment*.

```

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

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Press to update"
        />

</LinearLayout>

```

Change the existing *activity_rssfeed.xml* file. This layout will be the default layout for *RssfeedActivity* and shows two *Fragments*.

```

<?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="horizontal" >

    <fragment
        android:id="@+id/listFragment"
        android:layout_width="0dp"
        android:layout_weight="1"
        android:layout_height="match_parent"
        android:layout_marginTop="?android:attr/actionBarSize"
        class="com.example.android.rssfeed.MyListFragment" ></fragment>

    <fragment

```

```

        android:id="@+id/detailFragment"
        android:layout_width="0dp"
        android:layout_weight="2"
        android:layout_height="match_parent"
        class="com.example.android.rssfeed.DetailFragment" >
        <!-- Preview: layout=@layout/details -->
    </fragment>

</LinearLayout>

```

18.4. Activity

RssfeedActivity will remain unmodified.

```

package com.example.android.rssfeed;

import android.app.Activity;
import android.os.Bundle;

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

```

18.5. Create Fragment classes

Create now the Fragment classes.

Create the DetailFragment class.

```

package com.example.android.rssfeed;

import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;

public class DetailFragment extends Fragment {

```



```

@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
    Bundle savedInstanceState) {
    View view = inflater.inflate(R.layout.fragment_rssitem_detail,
        container, false);
    return view;
}

public void setText(String item) {
    TextView view = (TextView) getView().findViewById(R.id.detailsText);
    view.setText(item);
}
}

```

Create the `MyListFragment` class. Despite its name it will not display a list of items, it will just have a button which allow to send the current time to the details fragment.

```

package com.example.android.rssfeed;

import android.app.Fragment;
import android.content.Intent;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.Button;

public class MyListFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_rsslist_overview,
            container, false);
        Button button = (Button) view.findViewById(R.id.button1);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                updateDetail();
            }
        });
        return view;
    }

    // May also be triggered from the Activity
    public void updateDetail() {
        String newTime = String.valueOf(System.currentTimeMillis());
        DetailFragment fragment = (DetailFragment) getFragmentManager()
            .findFragmentById(R.id.detailFragment);
        if (fragment != null && fragment.isInLayout()) {

```

```

        fragment.setText(newTime);
    } else {
        Intent intent = new Intent(getActivity().getApplicationContext(),
            DetailActivity.class);
        intent.putExtra("value", newTime);
        startActivity(intent);
    }
}
}

```

18.6. Run

Run your example. Both fragments should be displayed.

19. Fragments Tutorial - layout for portrait mode

19.1. Create layouts for portrait mode

The *RssfeedActivity* should use a special layout files in portrait mode. For this reason create the *res/layout-port* folder.

In portrait mode Android will check the *layout-port* folder for fitting layout files. If Android does not find a fitting layout file it uses the *layout* folder.

Therefore create the following *activity_rssfeed.xml* layout file in the *res/layout-port* folder.

```

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

    <fragment
        android:id="@+id/listFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_marginTop="?android:attr/ActionBarSize"
        class="com.example.android.rssfeed.MyListFragment" />
</LinearLayout>

```

Also create the *activity_detail.xml* layout file. This layout will be used in the *DetailActivity*.

Please note that we could have create this file also in the `res/layout` folder, but it is only used in portrait mode hence we place it into this folder.

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

    <fragment
        android:id="@+id/detailFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        class="com.example.android.rssfeed.DetailFragment" />

</LinearLayout>
```

19.2. DetailActivity

Create a new Activity called DetailActivity with the following class.

```
package com.example.android.rssfeed;

import android.app.Activity;
import android.content.res.Configuration;
import android.os.Bundle;
import android.widget.TextView;

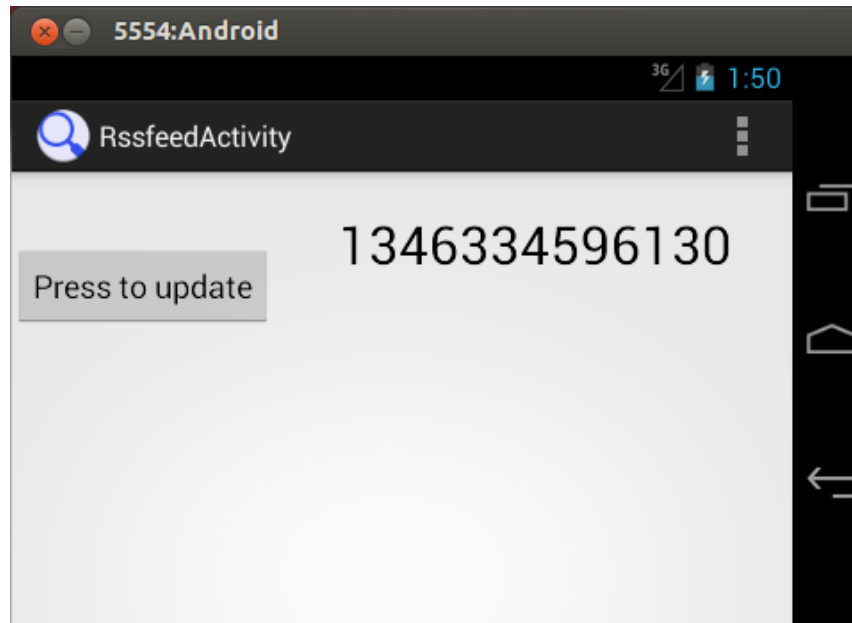
public class DetailActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Need to check if Activity has been switched to landscape mode
        // If yes, finished and go back to the start Activity
        if (getResources().getConfiguration().orientation == Configuration.ORIENTATION_LANDS
CAPE) {
            finish();
            return;
        }
        setContentView(R.layout.activity_detail);
        Bundle extras = getIntent().getExtras();
        if (extras != null) {
            String s = extras.getString("value");
            TextView view = (TextView) findViewById(R.id.detailsText);
            view.setText(s);
        }
    }
}
```

```
}
```

19.3. Run

Run your example. If you run the application in portrait mode you should see only one *Fragment*. Use Ctrl+F11 to switch the orientation. In horizontal mode you see two *Fragments*. If you press the button in portrait mode the a new *DetailActivity* is started and shows the current time. In horizontal mode you see both *Fragments*.

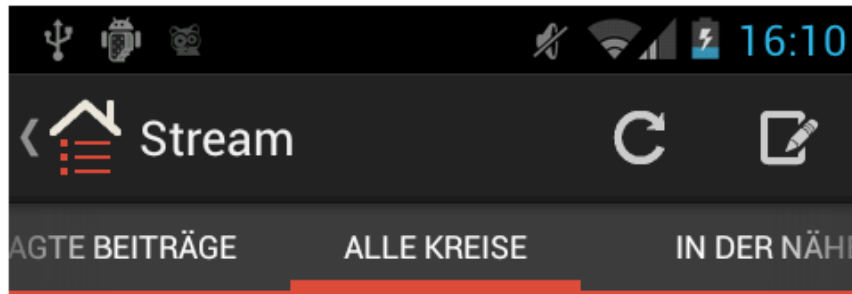


20. OptionMenu and ActionBar

20.1. ActionBar

The *ActionBar* is located at the top of the *Activity* that may display the *Activity* title, navigation modes, and other interactive items.

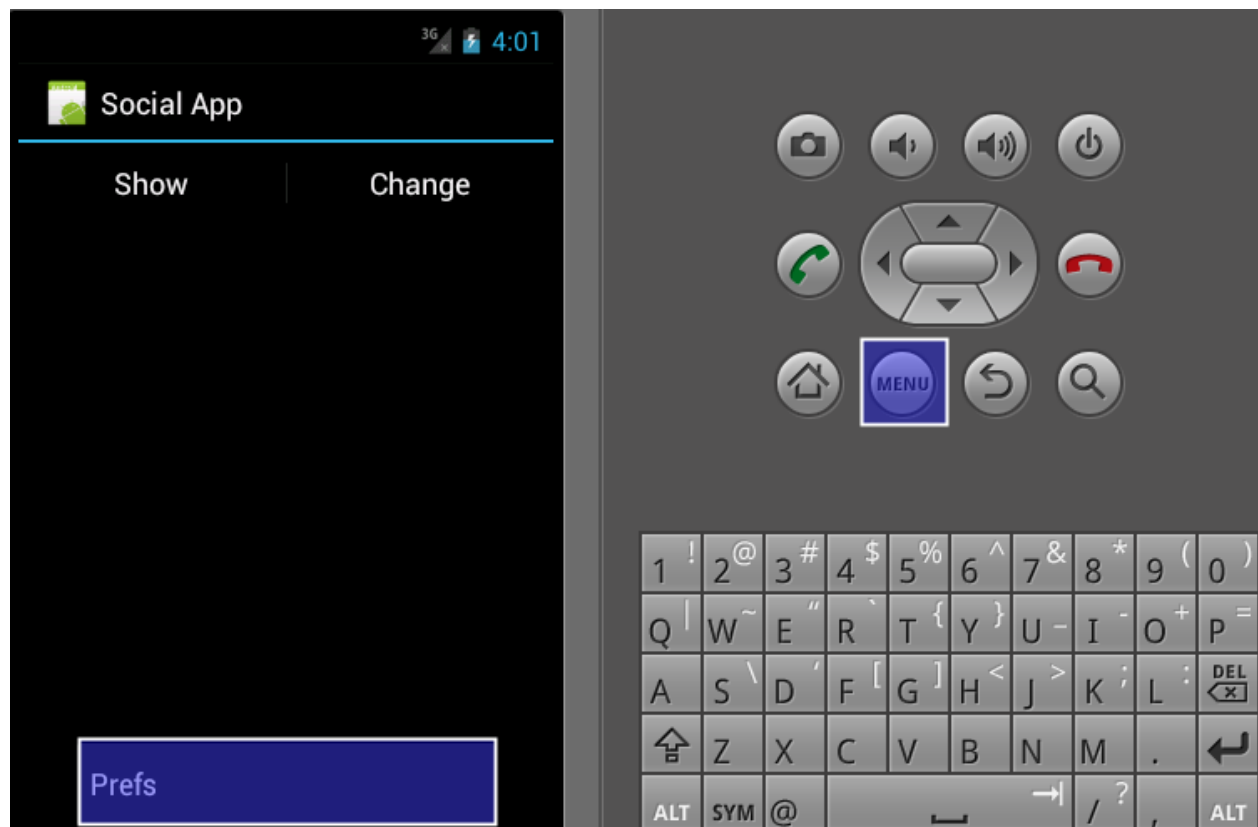
The following picture show the *ActionBar* of a typical Google Application with interactive items and a navigation bar.



20.2. OptionsMenu

The application can also open a menu which shows actions via a popup menu. This `OptionsMenu` is only available if the phone has a hardware *Options* button. Even if the hardware button is available, it is recommended to use the *ActionBar*, which is available for phones as of Android 4.0.

The following picture highlights the hardware button and the resulting menu as popup.

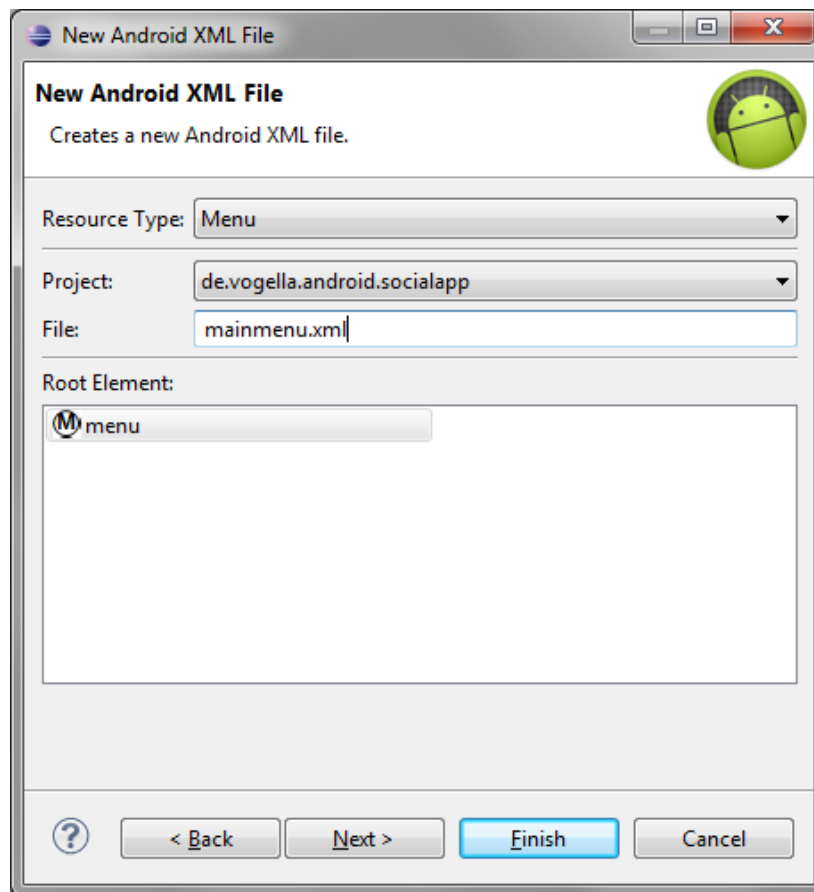


One of the reasons why the *ActionBar* is superior to the *OptionsMenu*, is that it is clearly visible, while the *OptionsMenu* is only shown on request and the user may not recognize that options are available.

20.3. Creating the menu

The menu is typically created as an XML resource. To create one select your project, right click on it and select *New* → *Other* → *Android* → *Android XML File*.

Select the *Menu* option, enter the filename of your choice and press the *Finish* button.



This will create a new menu file in the `res/menu` folder of your project. Open this file and select the *Layout* tab of the Android editor. Via the *Add* button you can add new entries.

In your *Activity* you can assign this menu to the *ActionBar* via the `onCreateOptionsMenu()` method. The `MenuInflater` class allows to inflate menu entries defined in XML to the menu. `MenuInflater` can get accessed via the `getMenuInflater()` method in your *Activity*

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

20.4. Changing the menu

The `onCreateOptionsMenu()` method is only called once. If you want to change the menu later have to call `invalidateOptionsMenu()` method. Afterwards `onCreateOptionsMenu()` is called again.

20.5. Reacting to menu entry selection

If a menu entry is selected then the `onOptionsItemSelected()` method is called. As parameter you receive the menu entry which was selected so that you can react differently to different menu entries.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.menuitem1:
            Toast.makeText(this, "Menu Item 1 selected", Toast.LENGTH_SHORT)
                .show();
            break;
        case R.id.menuitem2:
            Toast.makeText(this, "Menu item 2 selected", Toast.LENGTH_SHORT)
                .show();
            break;
        default:
            break;
    }

    return true;
}
```

21. Advanced ActionBar

21.1. Using the home icon

The *ActionBar* also shows an icon of your application. You can also add an action to this icon. If you select this icon the `onOptionsItemSelected()` method will be called with the value `android.R.id.home`. The recommendation is to return to the main Activity in your program.

```
// If home icon is clicked return to main Activity
case android.R.id.home:
```



```
Intent intent = new Intent(this, OverviewActivity.class);
intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
startActivity(intent);
break;
```

As of Android 4.1 this code is not required anymore, you can simply set the *parentActivityName* in the *AndroidManifest.mf* file, pointing to the parent *Activity*.

21.2. Custom Views in the ActionBar

You can also add a custom View to the *ActionBar*. The following code snippet demonstrates that.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    ActionBar actionBar = getActionBar();
    // add the custom view to the action bar
    actionBar.setCustomView(R.layout.actionbar_view);
    actionBar.setDisplayOptions(ActionBar.DISPLAY_SHOW_CUSTOM
        | ActionBar.DISPLAY_SHOW_HOME);
}
```

21.3. Contextual action mode

A contextual action mode activates a temporary *ActionBar* that overlays the application *ActionBar* for the duration of a particular sub-task.

The contextual action mode is typically activated by selecting an item or by long clicking on it.

To implement this, call the `startActionMode()` method on a View or on your Activity. This method gets an `ActionMode.Callback` object which is responsible for the lifecycle of the contextual *ActionBar*.

21.4. Context menus

You can also assign a context menu to a View. A context menu is also activated if the user "long presses" the view.

If possible the contextual action mode should be preferred over a context menu.

A context menu for a view is registered via the `registerForContextMenu(view)` method. The `onCreateContextMenu()` method is called every time a context menu is activated as the context menu is discarded after its usage. The Android platform may also add options to your View, e.g. `EditText` provides context options to select text, etc.

22. Tutorial: ActionBar

22.1. Project

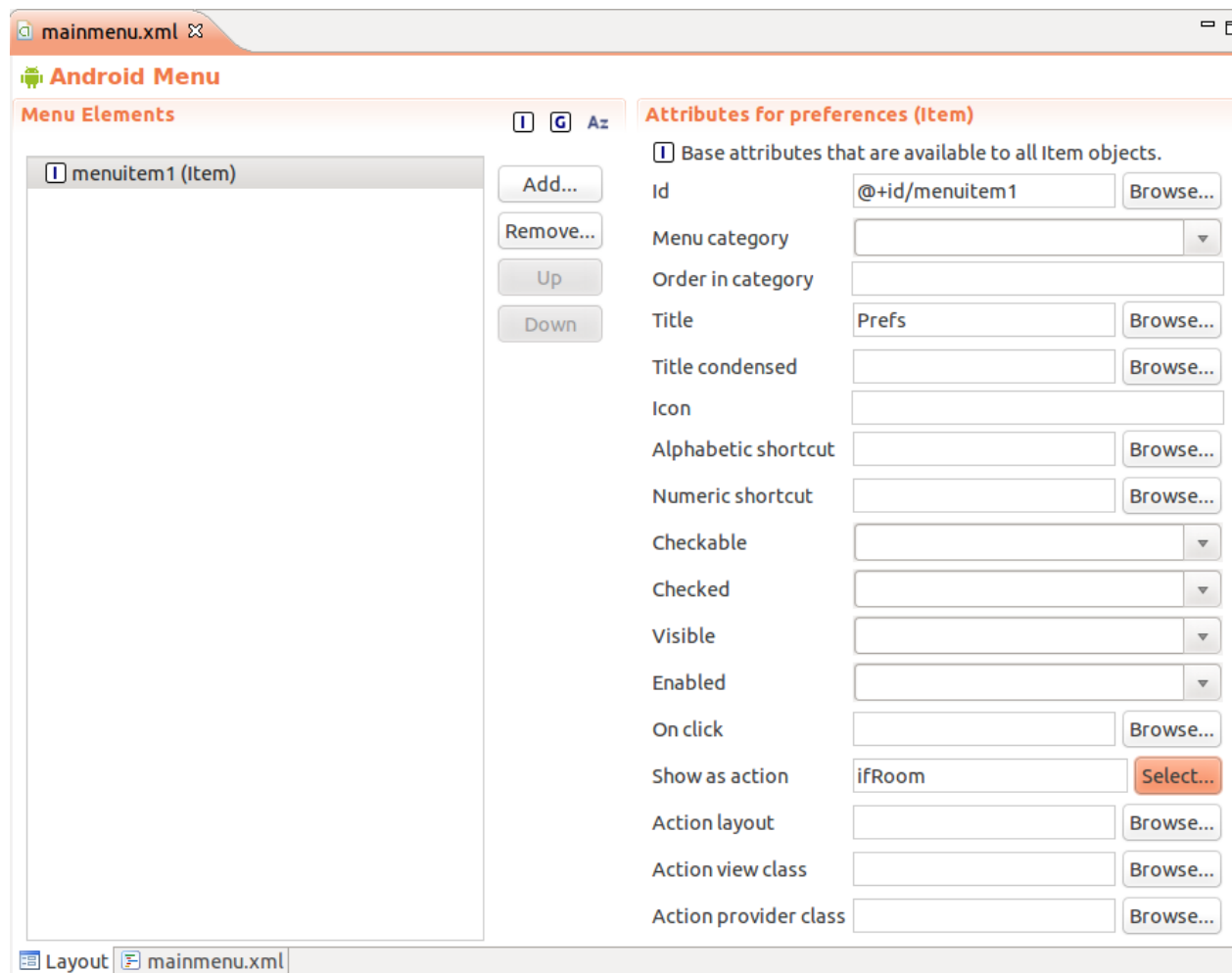
This chapter will demonstrate how to create items in the *ActionBar* and react to the selection of the user.

Create a project called *de.vogella.android.socialapp* with the Activity called *OverviewActivity*.

22.2. Add a menu XML resource

Create a new *Menu* XML resource called *mainmenu.xml*. Open the *mainmenu.xml* file and select the *Layout* tab of the Android editor.

Press the *Add* button and select the *Item* entry. Maintain a entry similar to the following screenshot. Via the *ifRoom* attribute you define that the menu entry is displayed in the *ActionBar* if there is sufficient space available.



Add a similar entry to the menu with the *ID* attribute set to "`@+id/menuitem2`", and the *Title* attribute set to "Test". Also set the *ifRoom* flag.

The resulting XML will look like the following.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android" >

    <item
        android:id="@+id/menuitem1"
        android:showAsAction="ifRoom"
        android:title="Prefs">
    </item>
```

```
<item
    android:id="@+id/menuitem2"
    android:showAsAction="ifRoom"
    android:title="Test">
</item>

</menu>
```

Change your OverviewActivity class to the following.

```
package de.vogella.android.socialapp;

import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.widget.Toast;

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

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.mainmenu, menu);
        return true;
    }

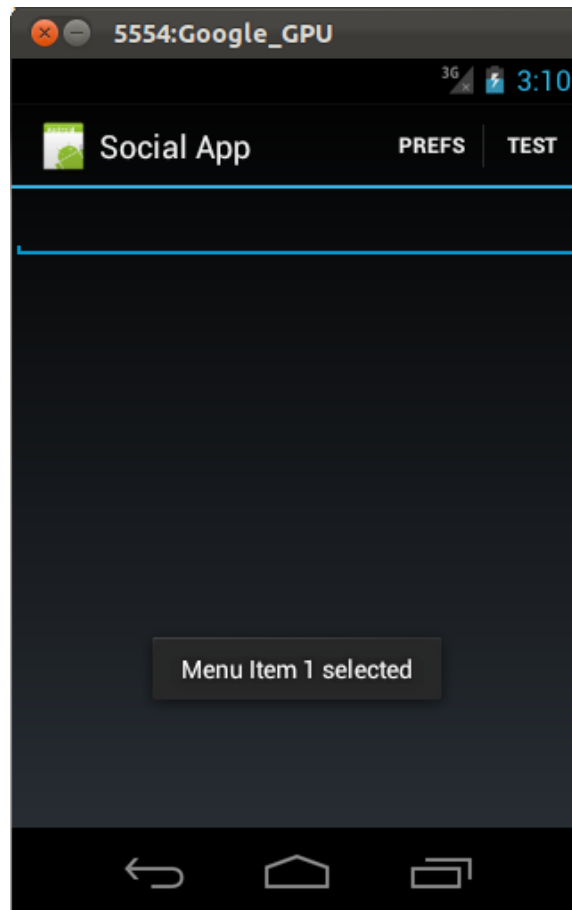
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case R.id.menuitem1:
                Toast.makeText(this, "Menu Item 1 selected", Toast.LENGTH_SHORT)
                    .show();
                break;
            case R.id.menuitem2:
                Toast.makeText(this, "Menu item 2 selected", Toast.LENGTH_SHORT)
                    .show();
                break;

            default:
                break;
        }

        return true;
    }
}
```

```
}
```

Run your application. As there is enough space in the *ActionBar* otherwise you may see the Overflow menu or you have to use the *Option* menu button on your phone. If you select one item, you should see a small info message.



23. Tutorial: Using the contextual action mode

Add a `EditText` element your `main.xml` layout file.

```
<?xml version="1.0" encoding="utf-8"?>
```

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <EditText
        android:id="@+id/myView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:ems="10" >

        <requestFocus />
    </EditText>

</LinearLayout>

```

Create a new menu XML resource with the *contextual.xml* file name.

```

<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
    <item
        android:id="@+id/toast"
        android:title="Toast">
    </item>

</menu>

```

Change your Activity to the following.

```

package de.vogella.android.socialapp;

import android.app.Activity;
import android.os.Bundle;
import android.view.ActionMode;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.widget.Toast;

public class OverviewActivity extends Activity {
    protected Object mActionMode;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        // Define the contextual action mode
        View view = findViewById(R.id.myView);
        view.setOnLongClickListener(new View.OnLongClickListener() {
            // Called when the user long-clicks on someView

```

```

    public boolean onLongClick(View view) {
        if (mActionMode != null) {
            return false;
        }

        // Start the CAB using the ActionMode.Callback defined above
        mActionMode = OverviewActivity.this
            .startActionMode(mActionModeCallback);
        view.setSelected(true);
        return true;
    }
});
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.mainmenu, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    Toast.makeText(this, "Just a test", Toast.LENGTH_SHORT).show();
    return true;
}

private ActionMode.Callback mActionModeCallback = new ActionMode.Callback() {

    // Called when the action mode is created; startActionMode() was called
    public boolean onCreateActionMode(ActionMode mode, Menu menu) {
        // Inflate a menu resource providing context menu items
        MenuInflater inflater = mode.getMenuInflater();
        // Assumes that you have "contextual.xml" menu resources
        inflater.inflate(R.menu.contextual, menu);
        return true;
    }

    // Called each time the action mode is shown. Always called after
    // onCreateActionMode, but
    // may be called multiple times if the mode is invalidated.
    public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
        return false; // Return false if nothing is done
    }

    // Called when the user selects a contextual menu item
    public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
        switch (item.getItemId()) {
            case R.id.toast:
                Toast.makeText(OverviewActivity.this, "Selected menu",
                    Toast.LENGTH_LONG).show();
                mode.finish(); // Action picked, so close the CAB
                return true;
        }
    }
};

```

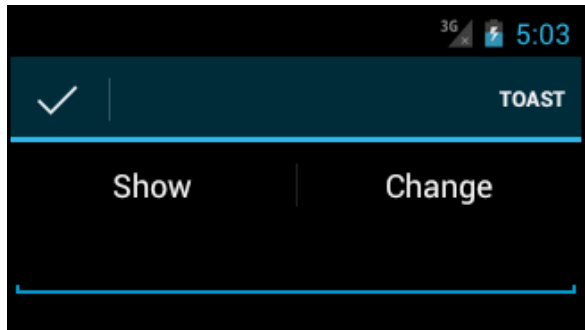
```

        default:
            return false;
        }
    }

    // Called when the user exits the action mode
    public void onDestroyActionMode(ActionMode mode) {
        mActionMode = null;
    }
};
}

```

If you run this example and long press the EditText widget, your contextual *ActionBar* is displayed.



24. ActionBar navigation with Fragments

Fragments can also be used in combination with the *ActionBar* for navigation. For this your main Activity needs to implement a *TabListener* which is responsible for moving between the tabs.

The *ActionBar* allows to add tabs to it via the `newTab()` method. The following code shows such an Activity. It uses two *Fragments*, called *DetailFragment* and *ImageFragment*. At this point you should be able to create these two *Fragments* yourself.

```

package de.vogella.android.fragment;

import android.app.ActionBar;
import android.app.ActionBar.Tab;
import android.app.ActionBar.TabListener;
import android.app.Activity;

```



```

import android.app.Fragment;
import android.app.FragmentTransaction;
import android.os.Bundle;

public class MainActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        // setup action bar for tabs
        ActionBar actionBar = getActionBar();
        actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);
        actionBar.setDisplayShowTitleEnabled(false);

        Tab tab = actionBar
            .newTab()
            .setText("First tab")
            .setTabListener(new MyTabListener<DetailFragment>(this, "artist",
                DetailFragment.class));
        actionBar.addTab(tab);

        tab = actionBar
            .newTab()
            .setText("Second Tab")
            .setTabListener(new MyTabListener<ImageFragment>(this, "album",
                ImageFragment.class));
        actionBar.addTab(tab);
    }

    public static class MyTabListener<T extends Fragment> implements
        TabListener {
        private Fragment mFragment;
        private final Activity mActivity;
        private final String mTag;
        private final Class<T> mClass;

        /**
         * Constructor used each time a new tab is created.
         *
         * @param activity
         *             The host Activity, used to instantiate the fragment
         * @param tag
         *             The identifier tag for the fragment
         * @param clz
         *             The fragment's Class, used to instantiate the fragment
         */

        public MyTabListener(Activity activity, String tag, Class<T> clz) {
            mActivity = activity;
            mTag = tag;
            mClass = clz;
        }
    }
}

```

```

}

/* The following are each of the ActionBar.TabListener callbacks */

public void onTabSelected(Tab tab, FragmentTransaction ft) {
    // Check if the fragment is already initialized
    if (mFragment == null) {
        // If not, instantiate and add it to the activity
        mFragment = Fragment.instantiate(mActivity, mClass.getName());
        ft.add(android.R.id.content, mFragment, mTag);
    } else {
        // If it exists, simply attach it in order to show it
        ft.setCustomAnimations(android.R.animator.fade_in,
            R.animator.animationtest);
        ft.attach(mFragment);
    }
}

public void onTabUnselected(Tab tab, FragmentTransaction ft) {
    if (mFragment != null) {
        ft.setCustomAnimations(android.R.animator.fade_in,
            R.animator.test);
        ft.detach(mFragment);
    }
}

public void onTabReselected(Tab tab, FragmentTransaction ft) {
}
}
}

```

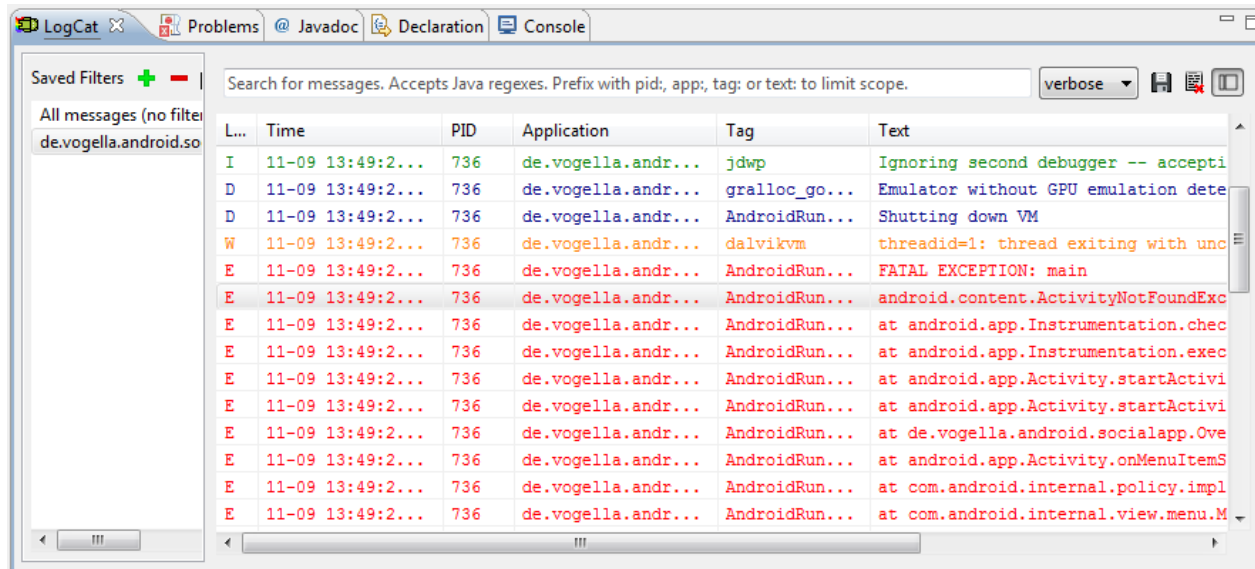
25. DDMS perspective and important views

25.1. DDMS - Dalvik Debug Monitor Server

Eclipse provides a perspective for interacting with your Android (virtual) device and your Android application program. Select *Window* → *Open Perspective* → *Other* → *DDMS* to open this perspective. It includes several Views which can also be used independently and allows for example the application to place calls and send SMS to the device. It also allows the application to set the current geo position and allows you to perform a performance trace of your application.

25.2. LogCat View

You can see the log (including `System.out.print()` statements) via the LogCat view.



25.3. File explorer

The file explorer allows to see the files on the Android simulator.

Name	Size	Date	Time	Permissions	Info
data		2009-10-22	01:34	drwxrwx--x	
+ anr		2009-11-03	15:26	drwxrwx--x	
- app		2009-10-22	01:34	drwxrwx--x	
ApiDemos.apk	2184352	2009-10-22	01:34	-rw-r--r--	com.examp...
SoftKeyboard.apk	35613	2009-10-22	01:34	-rw-r--r--	com.examp...
de.vogella.android.first.apk	14677	2009-11-03	18:24	-rw-r--r--	de.vogella...
de.vogella.android.network.html.apk	15107	2009-11-05	09:54	-rw-r--r--	de.vogella...
de.vogella.android.test.apk	13300	2009-11-03	15:36	-rw-r--r--	de.vogella...
de.vogella.android.test2.apk	13301	2009-11-03	15:34	-rw-r--r--	de.vogella...
+ app-private		2009-11-03	15:26	drwxrwx--x	
+ backup		2009-11-03	15:27	drwx-----	
+ dalvik-cache		2009-11-03	15:26	drwxrwx--x	
+ data		2009-11-03	15:26	drwxrwx--x	
+ dontpanic		2009-11-03	15:26	drwxr-x---	
+ local		2009-11-03	15:26	drwxrwx--x	
+ lost+found		2009-11-03	15:26	drwxrwx---	
+ misc		2009-11-03	15:26	drwxrwx--t	
+ property		2009-11-03	15:26	drwx-----	
+ system		2009-11-03	15:26	drwxrwxr-x	
+ sdcard		1969-12-31	16:00	d---rwxr-x	
+ system		2009-10-22	01:33	drwxr-xr-x	

26. Shell

26.1. Android Debugging Bridge - Shell

You can access your Android emulator also via the console. Open a shell, switch to your "android-sdk" installation directory into the folder "tools". Start the shell via the following command "adb shell".

```
adb shell
```

You can also copy a file from and to your device via the following commands.

```
// Assume the gesture file exists on your Android device
adb pull /sdcard/gestures ~/test
// Now copy it back
adb push ~/test/gesture /sdcard/gestures2
```

This will connect you to your device and give you Linux command line access to the underlying file system, e.g. ls, rm, mkdir, etc. The application data is stored in the directory `"/data/data/package_of_your_app"`.

If you have several devices running you can issue commands to one individual device.

```
# Lists all devices
adb devices
#Result
List of devices attached
emulator-5554 attached
emulator-5555 attached
# Issue a command to a specific device
adb -s emulator-5554 shell
```

26.2. Uninstall an application via adb

You can uninstall an android application via the shell. Switch the data/app directory (`cd /data/app`) and simply delete your android application.

You can also uninstall an app via adb with the package name.

```
adb uninstall <packagename>
```

26.3. Emulator Console via telnet

Alternatively to adb you can also use telnet to connect to the device. This allows you to simulate certain things, e.g. incoming call, change the network "stability", set your current geocodes, etc. Use `"telnet localhost 5554"` to connect to your simulated device. To exit the console session, use the command `"quit"` or `"exit"`.

For example to change the power settings of your phone, to receive an sms and to get an incoming call make the following.

```
# connects to device
telnet localhost 5554
# set the power level
power status full
power status charging
# make a call to the device
gsm call 012041293123
```

```
# send a sms to the device
sms send 12345 Will be home soon
# set the geo location
geo fix 48 51
```

For more information on the emulator console please see [Emulator Console manual](#)

27. Deployment

27.1. Overview

In general there are no restrictions how to deploy an Android application to your device. You can use USB, email yourself the application or use one of the many Android markets to install the application. The following describes the most common ones.

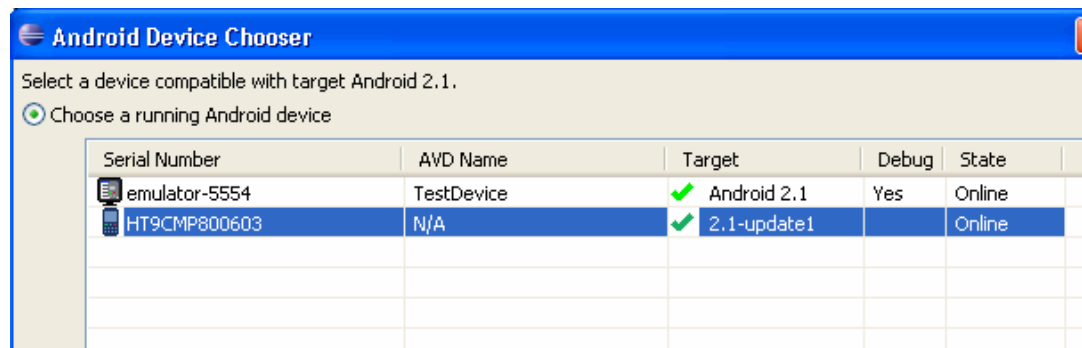
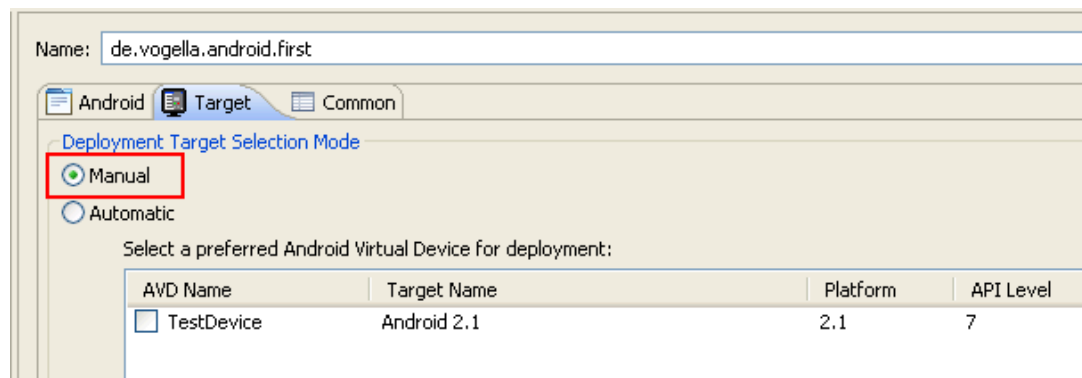
27.2. Deployment via Eclipse

Turn on *USB Debugging* on your device in the settings. Select in the settings of your device *Applications* → *Development*, then enable USB debugging.

You may also need to install the a driver for your mobile phone. Linux and Mac OS usually work out of the box while an Windows OS typically requires the installation of a driver.

For details please see [Developing on a Device](#). Please note that the Android version you are developing for must be the installed version on your phone.

If you have only one device connected and no emulator running, the Android development tools will automatically deploy to this device. If you have several connected you can select which one should be used.



27.3. Export your application

Android application must be signed before they can get installed on an Android device. During development Eclipse signs your application automatically with a debug key.

If you want to install your application without the Eclipse IDE you can right click on it and select *Android Tools* → *Export Signed Application Package*.

This wizard allows to use an existing key or to create a new one.

Please note that you need to use the same signature key in Google Play (Google Market) to update your application. If you loose the key you will NOT be able to update your application ever again.

Make sure to backup your key.

27.4. Via external sources

Android allow to install applications also directly. Just click on a link which points to an .apk file, e.g. in an email attachment or on a webpage. Android will prompt you if you want to install this application.

This requires a setting on the Android device which allows the installation of non-market application. Typically this setting can be found under the "Security" settings.

27.5. Google Play (Market)

Google Play requires a one time fee, currently 25 Dollar. After that the developer can directly upload his application and the required icons, under **Google Play Publishing**.

Google performs some automatic scanning of applications, but no approval process is in place. All application, which do not contain malware, will be published. Usually a few minutes after upload, the application is available.

28. Thank you

Please help me to support this article:



29. Questions and Discussion

Before posting questions, please see the **vogella FAQ**. If you have questions or find an error in this article please use the **www.vogella.com Google Group**. I have created a short list **how to create good questions** which might also help you.

30. Links and Literature

30.1. Source Code

[Source Code of Examples](#)

30.2. Android Resources

[Android ListView and ListActivity](#)

[Android SQLite Database](#)

[Android Widgets](#)

[Android Live Wallpaper](#)

[Android Services](#)

[Android Location API and Google Maps](#)

[Android Intents](#)

[Android and Networking](#)

[Android Homepage](#)

[Android Developer Homepage](#)

[Android Issues / Bugs](#)

[Android Google Groups](#)

[Android Live Folder](#)

30.3. vogella Resources

[vogella Training](#) Android and Eclipse Training from the vogella team

[Android Tutorial](#) Introduction to Android Programming

[GWT Tutorial](#) Program in Java and compile to JavaScript and HTML

[Eclipse RCP Tutorial](#) Create native applications in Java

JUnit Tutorial Test your application

Git Tutorial Put everything you have under distributed version control system