Introduction to Android

Implementing a Video Player

Systèmes Multimédia
2^e année IMA, Parcours Multimédia - Informatique

Sessions 1-3

Contents

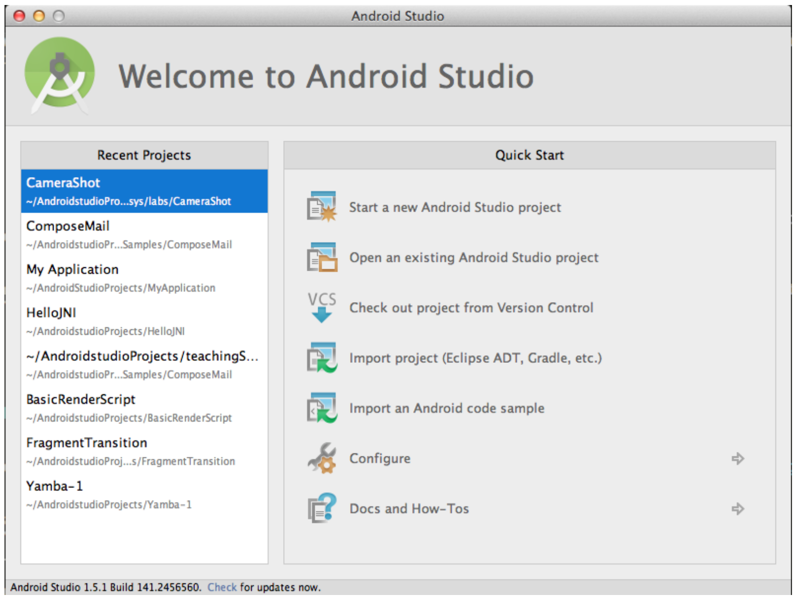
Objective

The objective of these first three TP sessions is to get you familiar with the Android development environment and to implement a multimedia application. In particular we will develop step by step an application to play videos from the internal SD card of the device as well as streaming remote videos from the Internet.

References

Here are some useful resources that may help you to complete the TP:

- A reminder of the main components of an Android application
  `https://developer.android.com/guide/components/fundamentals.html`
- A quick introduction to Android Studio
  `http://developer.android.com/tools/studio/index.html`
- Some explained examples of applications with specific components
  `https://developer.android.com/samples/index.html`
- The official documentation of the Android API.
  `https://developer.android.com/reference/packages.html`


Android Studio

Welcome to Android Studio

Recent Projects

CameraShot
~/AndroidstudioPro...sys/labs/CameraShot

ComposeMail
~/AndroidstudioPr...Samples/ComposeMail

My Application
~/AndroidStudioProjects/MyApplication

HelloJNI
~/AndroidstudioProjects/HelloJNI

~/AndroidstudioProjects/teachingS...
~/AndroidstudioPr...Samples/ComposeMail

BasicRenderScript
~/AndroidstudioProjects/BasicRenderScript

FragmentTransition
~/AndroidstudioProj...s/FragmentTransition

Yamba-1
~/AndroidstudioProjects/Yamba-1

Quick Start

Start a new Android Studio project

Open an existing Android Studio project

Check out project from Version Control

Import project (Eclipse ADT, Gradle, etc.)

Import an Android code sample

Configure

Docs and How-Tos

Android Studio 1.5.1 Build 141.2456560. Check for updates now.

Figure 1: The wizard that appears when you launch Android Studio for the first time.

1 First Application: web browsing using Intents

In this first warm-up exercise you are going to develop a simple application to visit web pages: the user types the URL of a web page and a button launches the Android default Web Browser to load the page.

1.1 Getting started with Android Studio

Before starting please clean up your account and free as much space as possible (empty the trash can, remove all teaching unrelated files *etc.*). In order to avoid any disk quota problem you should have ∼ 300MB of available space.

Android Studio is the official IDE for Android application development, based on IntelliJ IDEA, a Java IDE. Android Studio offers the same functionalities as IntelliJ for editing Java applications along with many other tools that have been added to support the development, the debugging and the testing of Android applications. In particular it integrates Gradle as building system, a rich layout editor to create the application GUI and tools for logging the applications. We will discover all these tools as we create our first application.

You can launch Android Studio with:

```
androidstudio &
```

Normally a wizard similar to the one in Figure 1 should pop up. The wizard allows to create a new project, load an existing one and other things that for the moment are out of the scope of this lab session.

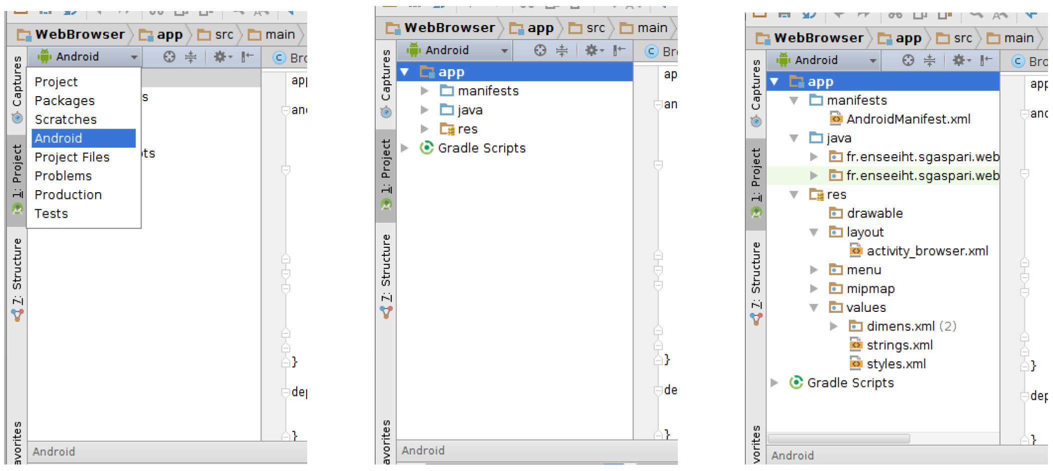


Figure 2: The "Android" project view displaying how the application is organized in different folders.

1.2 Create a new Android Studio project

Thanks to Android Studio, we can create a new Android application stub in quite few steps. From the "Quick Start" panel of the wizard click "Start a new Android Studio project" to start the wizard that creates all the files needed for the application. Following the various steps that are proposed:

- in the first step create a new project with the following settings:
  - Application name: `Web Browser`.
  - Company domain: `YOUR_LOGIN.enseeiht.fr`
- In the second step select only "Phone and Tablet" with "Minimum SDK: API 15: Android 4.0 (IceCreamSandwich)".
- In the third step of the wizard select "Blank Activity", click Next, set the activity name as Activity `BrowserActivity`, and then click Finish.

Once you complete the wizard, Android Studio will generate all the classes and all the resources needed for the application, and it will show them in the editor. Let's have a quick look at the Android Studio editor.

1.3 Android Studio editor

Take a moment to look at the Android Studio editor. As it has been explained in class, the interface of the editor is composed of 3 mains parts, the project views, the editor itself and the Android toolbar for logging and debugging. By default, Android Studio displays your project files in the *Android* project view (see Figure 2). This view shows a flattened version of your project's structure that provides quick access to the key source files of Android projects and helps you work with the Gradle-based build system. The Android project view:

- Shows the most important source directories at the top level of the application hierarchy.

- Groups resource files for different locales, orientations, and screen types in a single group per resource type (see Figure 2). In particular you can see that the application files are organized in 3 main folders:
  - `java/` contains the Java source files for the application.
  - `manifests/` contains the manifest files for the application.
  - `res/` contains the resource files for the application.
- Groups the build files for all modules in a common folder.

Another interesting view is the *Problem* view, which displays links to the source files containing any recognized coding and syntax errors, such as missing an XML element closing tag in a layout file.

1.4 The Virtual Device

We will use a virtual device to test our application. As we have seen in classes, Android Studio allows to create an emulator of an Android device quite easily:

- Open the AVD manager from the icon on the menu bar or from the menu Tools > Android > AVD Manager
- Create a new device with a Nexus 4 device (768×1280 screen), with Lollipop API 22 for x86_64 processor.
- In the "Advanced Settings" you can choose 1907MB of RAM, and a SD Card of (at least) 20MB (you can set more space according to your account quota...). Also in "Skin" you can choose the Nexus 4 skin (this is just to shape the emulator like the real device).
- Once you created the emulator, you can start it by clicking on the green arrow (like the common "play" icon) in the AVD manager window.
- For your reference, here you can find the keyboard shortcuts to control the virtual device.

1.5 Start the application

Now you can finally run the application clicking on the Run icon (the green arrow) on the toolbox bar or from the Run > Run 'app' menu. Obviously the application is pretty dumb and it does nothing. But now we have a stub from which we can develop our application.

1.6 The Log class and the LogCat view

Since Android applications run on external devices without shell, the debugging could be really troublesome. Fortunately, Android provides a logging system for collecting and viewing system debug output. Logs from various applications and portions of the system are collected in a series of buffers, which then can be viewed and filtered out using the LogCat view (you may need to click on the tab `6 Android` at the bottom of the window). Now let's try to see how the logging works:

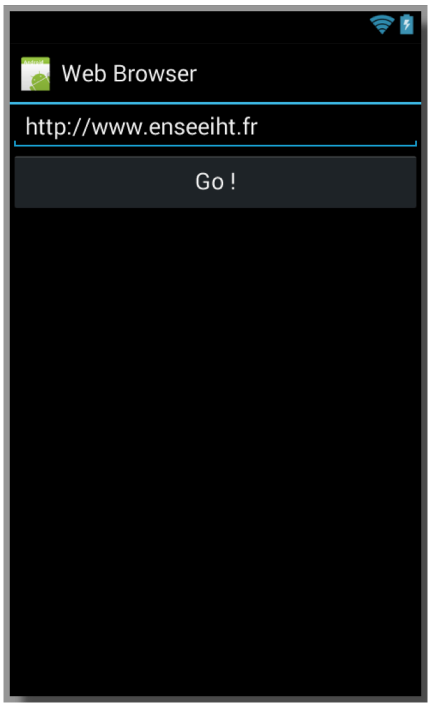


Figure 3: The graphical layout of the Web Browser.

- Android provides the class `android.util.Log` to send messages to the system. Check out the brief explanation here
  http://developer.android.com/tools/debugging/debugging-log.html
- define a private constant string named `TAG` containing an identifier for all the messages sent by our activity:

  ```
  private static final String TAG = "Browser Activity"
  ```

- now you can add a log message to the method `onCreate()`:

  ```
  Log.d( TAG, "OnCreate() called!" )
  ```

- Re-run the application and check the LogCat to see the message sent by the activity.
- Note that in this document the clickable hyperlink to the class API documentation is given for some classes and some class methods (*e.g.* `findViewById()`).

LogCat is a very useful tool to debug your application. Adding text messages may help you to follow the application flow and find the bug(s).

1.7 Setting up the application

Now let's start adding the elements to develop our browser application:

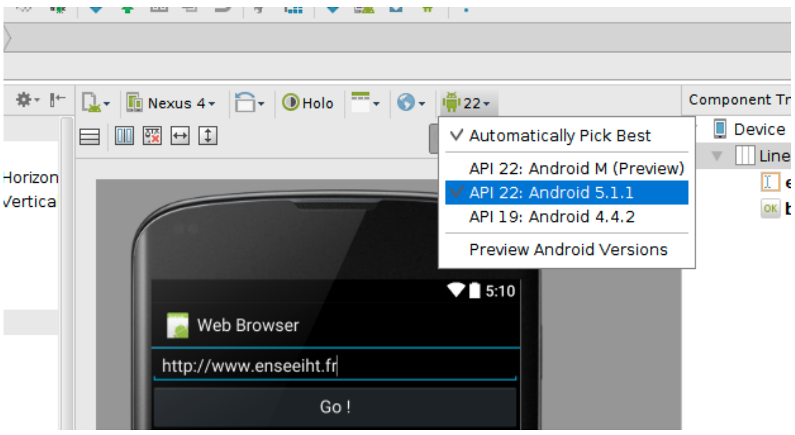


Figure 4: Check that the correct version of the API is setting for the visualization in Design mode.

- open the layout from `res/layout` and switch to the Design Mode. If you get an error "Rendering Problems", be sure that the settings for the Android version are correct by selecting "API 22: Android 5.1.1" from the toolbar menu (see Figure 4)
- modify the GUI layout in order to add an editable field where the user can write the web address to load (`EditText`), and a button that asks the Web Browser to load the webpage (`Button`).
- the button must display the text "GO!", while the `EditText` must display the URL `http://www.enseeiht.fr` as default text, so that it is displayed automatically when the application is launched for the first time; as we saw in classes, all the strings must be collected in `value/string.xml`[1].
- Re-run the application and verify that the GUI is displayed correctly (see Figure 3).

1.8 Action and. . . GO!

The last thing to do is to wire the button so that the web browser is called when the user presses the button: to this end we need to implement the button callback, and create the `Intent` that invokes the default Android browser.

- in `onCreate()` get the button from the resources using the function `findViewById()`;
- get also the reference to the text field, as we need it to get the web address typed by the user;
- once we have the reference to the button, set its callback listener using its method `setOnClickListener()`: among other choices, you can pass an anonymous class with `new View.OnClickListener()` and if you use the Android Studio auto-completion, it will automatically generate the stub of the method to implement (`OnClick()`);

[1] In order to create a new string you can either define it in the `strings.xml` and reference it in the `Text` attribute of the widget, or use the graphical GUI editor and follow the guided wizard by clicking on the small button in the `Text` attribute of the widget.

- the callback listener has to create a new `Intent` to open the web address contained inside the `EditText`.

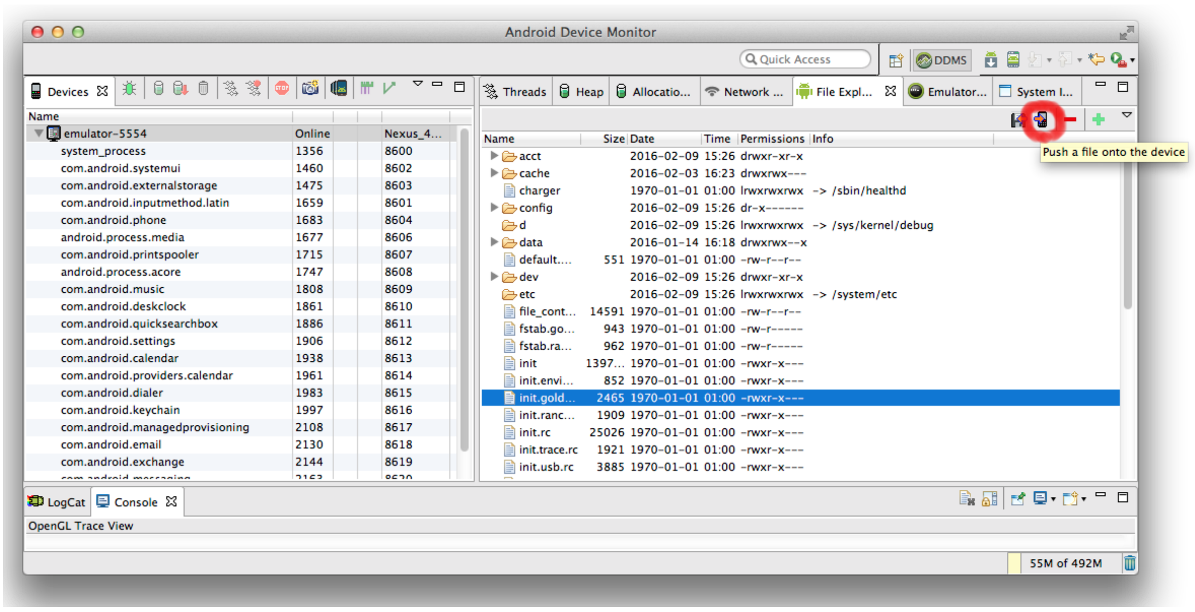
  An Intent is a messaging object you can use to request an action to be executed by another application component and it contains an abstract description of the operation to be performed. In this case you will need to use an implicit intent, as you are asking a general action to perform, which allows a component from any other available application to handle it.

  There are two main things that you need to specify when you create an intent, the action and the data. The **Action** that you are requiring to perform: when you create a new object of the class `Intent` you need to pass one of the possible predefined action available in Android. For example, `ACTION_VIEW` is used when you have some information that an activity can show to the user, such as a photo to view in a gallery application, or an address to view in a map application; `ACTION_SEND` is used when you have some data that the user can share through another application, such as an email or social sharing application (see more here). The **Data** is the URI (a `Uri` object) that references the data to be acted on and/or the MIME type of that data. The type of data supplied generally depends on the intent's action. For example, if the action is `ACTION_SEND`, the data should contain the URI of the document to edit. In order to set the data of the intent you can use either the intent method setData() or setDataAndType().

  Finally, once the intent is created and its data set, you can start the request with the method `startActivity()` of the class `Activity`

  Set up the intent action (`ACTION_VIEW`) and the data field as an URI obtained from `EditText` (remember to use the `Uri.parse()`). A quick review about `Intents` is available here
  `http://developer.android.com/guide/components/intents-filters.html`

- Re-run the application (. . . and debug it if necessary!)



Name			
emulator-5554	Online		Nexus_4...
system_process	1356		8600
com.android.systemui	1460		8602
com.android.externalstorage	1475		8603
com.android.inputmethod.latin	1659		8601
com.android.phone	1683		8604
android.process.media	1677		8606
com.android.printspooler	1715		8607
android.process.acore	1747		8608
com.android.music	1808		8609
com.android.deskclock	1861		8610
com.android.quicksearchbox	1886		8611
com.android.settings	1906		8612
com.android.calendar	1938		8613
com.android.providers.calendar	1961		8614
com.android.dialer	1983		8615
com.android.keychain	1997		8616
com.android.managedprovisioning	2108		8617
com.android.email	2130		8618
com.android.exchange	2144		8619

Name	Size	Date	Time	Permissions	Info
acct		2016-02-09	15:26	drwxr-xr-x	
cache		2016-02-03	16:23	drwxrwx---	
charger		1970-01-01	01:00	lrwxrwxrwx	-> /sbin/healthd
config		2016-02-09	15:26	dr-x------	
d		2016-02-09	15:26	lrwxrwxrwx	-> /sys/kernel/debug
data		2016-01-14	16:18	drwxrwx--x	
default....	551	1970-01-01	01:00	-rw-r--r--	
dev		2016-02-09	15:26	drwxr-xr-x	
etc		2016-02-09	15:26	lrwxrwxrwx	-> /system/etc
file_cont...	14591	1970-01-01	01:00	-rw-r--r--	
fstab.go...	943	1970-01-01	01:00	-rw-r-----	
fstab.ra...	962	1970-01-01	01:00	-rw-r-----	
init	1397...	1970-01-01	01:00	-rwxr-x---	
init.envi...	852	1970-01-01	01:00	-rwxr-x---	
init.gold...	2465	1970-01-01	01:00	-rwxr-x---	
init.ranc...	1909	1970-01-01	01:00	-rwxr-x---	
init.rc	25026	1970-01-01	01:00	-rwxr-x---	
init.trace.rc	1921	1970-01-01	01:00	-rwxr-x---	
init.usb.rc	3885	1970-01-01	01:00	-rwxr-x---	

Figure 5: Check that the correct version of the API is setting for the visualization in Design mode.

2 Second Application: Building your custom Video Player

In this exercise we want to create an application that plays a video using a custom video player. We will build the application incrementally, starting with a basic application that uses the default Android Video Player, and then adding all the components to play the video inside our application.

The first version of the application is very similar to the Web Browser application you just created: using that application as reference, create a new Android project named `Video Player`, with a different package name (`fr.enseeiht.YOUR_LOGIN.player`), an activity named `PlayerActivity`, and, again, a GUI composed of an editable text field and a $\boxed{\text{GO!}}$ button. The text field contains the name of the video to load and the button launches the Video Player.

2.1 Adding a video to the AVD SD card

Before starting, let's add a video on the virtual device SD card that we can use to test our application:

- open the Android Device Monitor from the Android icon on the menu bar in order to access to the file manager of the device (Tools > Android > Android Device Monitor);
- add the video using `Push File on device` button (see Figure 5) and load on the SD card the video from `/mnt/n7fs/ens/tp_android/TP_01/documentariesandyou.mp4`[2]
- if you want, you can add other video files (mind your disk quota...); here are the media

[2]You may need to restart the emulator to see the file...

formats supported by Android
http://developer.android.com/guide/appendix/media-formats.html

2.2 A new Intent

The user can type the name of a video file in the text field in order to load it inside the default Android video player:

- set the default value displayed by the text field to the name of the video file you added to the SD card (`documentariesandyou.mp4`);
- before creating the intent, you need to get the name of the file along with its full path, which will be pass as intent data. In order to get the base path of the video you can use:
  `Environment.getExternalStorageDirectory().getPath()`
  Take a moment to read the documentation of the method `getExternalStorageDirectory()`
- when you create the intent, remember to set the type of data (`setDataAndType()...`);
- run and test the application.

2.3 A new layout

Now we can add the video playback feature directly in the main activity of our application. We first need to modify the layout in order to add the controls typical of a video player (see Figure 6 as reference):

- a new button Play from start ;
- a new button Play / Resume ;
- a `SurfaceView` , used to display the video;
- a `SeekBar` , which displays the current position of the video and allows to jump to any position of the video
- Note that the main Activity has now to implement the class `SurfaceHolder.Callback` , *i.e.*

```
public class PlayerActivity extends Activity
                                implements SurfaceHolder.Callback {
...
```

2.4 The MediaPlayer class

Android provides the class `MediaPlayer` as the primary API for playing sound and video files. An object of this class can fetch, decode, and play both audio and video with minimal setup. Let's start setting up the graphical part to display the video.

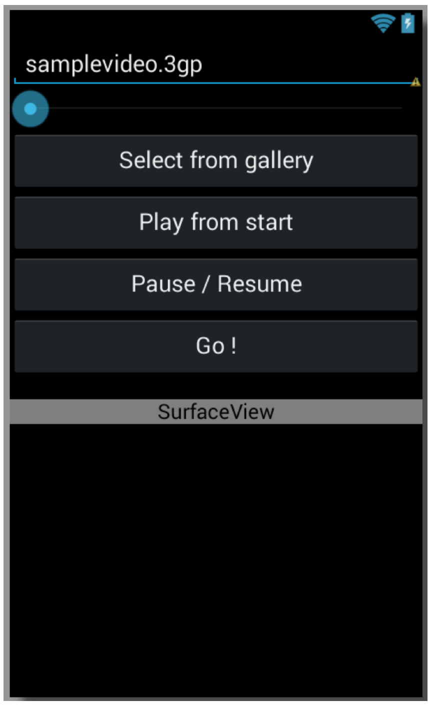


Figure 6: The graphical layout of the Video Player.

2.4.1 Setting up the SurfaceView and the MediaPlayer

The `MediaPlayer` class continuously draws the video frames inside the `SurfaceView`. The `SurfaceView` provides a dedicated drawing surface embedded inside of the view hierarchy. In order to manage and access to the underlying surface, the `SurfaceHolder` interface is provided, which can be retrieved by calling `getHolder()`. Take a moment to read the "Class Overview" on the online documentation to better understand the use of these classes.

The surface holder must be initialized inside the `onCreate()` method of `PlayerActivity`:

```
// recover the SurfaceView from resources
surfaceView = (SurfaceView) findViewById( R.id.surfaceview );
// Get the surfaceHolder from it
surfaceHolder = surfaceView.getHolder();
// and assign to it the call back this class implements
surfaceHolder.addCallback( this );
// this is a compatibility check, setType has been deprecated since HoneyComb,
// it guarantees back compatibility
if( Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB )
     surfaceHolder.setType( SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS );
```

then the `SurfaceHolder` can be associated to the `MediaPlayer`

```
// the surfaceHolder.addCallback implemented by PlayerActivity
public void surfaceCreated( SurfaceHolder holder )
{
    mediaPlayer.setDisplay(surfaceHolder);
}
```

2.4.2 Playing the video

The `MediaPlayer` is actually a state machine that controls the audio/video playback. Again, take your time to read the online documentation of the class, in particular the state diagram (it may seems complicated at first, but it's really straightforward and intuitive to follow).

- Add a callback listener for the Play button and implement the function that sets up the `MediaPlayer` and starts playing the video:
  - get the name of the video from the `TextView` widget as you already did for the GO! button;
  - set it as the source to be play by `MediaPlayer` (`setDataSource()`);
  - prepare the `MediaPlayer` for the playback (`prepare()`)
  - start the video (`start()`)
- Add another callback listener for the Pause button to pause and resume the video according to its state; you can use a simple boolean variable `isPlaying` to keep track of the player state.

2.4.3 The activity life-cycle

In order to avoid any resource conflict, we need to stop and release the `MediaPlayer` whenever the user leaves the activity and saves the current position of the video playback, so that, when she comes back, the video resumes from the current position[3]. Therefore, we need to override the life-cycle callbacks `onPause()` and `onResume()` of `PlayerActivity` to manage the release and restart of `MediaPlayer`[4]:

- whenever the user leaves the activity we need to save the current position (`getCurrentPosition()`) and release the `MediaPlayer` with `reset()` and `release()` . You can use a simple `int` variable to save the position.
- whenever the user comes back to the activity, we need to reinitialize the `MediaPlayer` (as we already did for the button listener) and start the video from the last saved position (use `seekTo()`)

[3]Contrary to audio playback, it makes less sense to keep the video playing in the background when the user leaves the application. As we saw in classes, in order to get the video to continue playing in background we would rather need to implement a `Service`.

[4]Astuce: select the `PlayerActivity` class on the editor, press Ctrl + O : Android Studio will propose a wizard to automatically generate the code skeleton of the class method to override.

Try to run the application and load a video. If you implemented correctly all the steps... you may get an error and some exceptions! Well, we are accessing the memory card, so we need...? Fix the manifest file!

Now the video should be displayed and you can control the playback with the relevant buttons.

2.4.4 The MediaPlayer life-cycle

The `MediaPlayer` class provides a series of callbacks `MediaPlayer.setOn*()` to control each state transition of its state machine. We can implement these callbacks to display some messages on the graphical interface (as well as in the `LogCat`).

- first add a new text field `TextView` below the control buttons, this will contain the messages to display;
- set the default string to blank (empty string), and reduce the font size so that the messages are readable but they don't take too much space;
- implement the main callback listeners for the state transition of `MediaPlayer`:
  - `OnBufferingUpdateListener` is called whenever there is an update during the buffering, it allows to show the buffering progress (see Figure 8);
  - `OnCompletionListener` is called when the media player finishes to read the video: a message "Playback completed" can be displayed when it occurs;
  - `OnErrorListener` allows to manage the errors: check the documentation to see what kind of errors can occur and display a message for the most significant ones;
  - `OnSeekCompleteListener` is called whenever the seek operation is terminated: a relevant message can be displayed;
- as usual you can either implement the listener as a private object member of the class `PlayerActivity` or make the `PlayerActivity` directly implement the interface, *e.g.* for `OnErrorListener` you can either
  - implement the listener

```java
// inside PlayerActivity
OnErrorListener OnErrorListener = new OnErrorListener()
{
        public boolean onError( MediaPlayer mp, int what, int extra )
        {
                // TODO Auto-generated method stub
                return false;
        }
};
```

    and then set the listener as
    `MediaPlayer.setOnErrorListener( OnErrorListener );`
  - or implement the interface

```
public class PlayerActivity extends Activity
                        implements OnErrorListener, ... {
...
        public boolean onError( MediaPlayer mp, int whatError, int extra )
        {
                // TODO Auto-generated method stub
                return false;
        }
```

and then set the listener as
`MediaPlayer.setOnErrorListener( this );`

2.4.5 The seek bar

We can use the `SeekBar` to navigate and jump to (or, "seek to") a specific position of the video. The `SeekBar` can be wired to the playback so that it displays the playback progress. We first implement the "seek to" function of the bar, and then we will create a thread dedicated to the position update.

- Inside `onCreate()` get the reference to the `SeekBar` from the resources, and set a new callback with `setOnSeekBarChangeListener()`: this manages the events related to the bar due to the user interaction.
- create a new object implementing the interface class `OnSeekBarChangeListener` private to `PlayerActivity`; Android Studio creates the stubs for the 3 methods to implement. We are interested only on the method `onProgressChange`: the method is invoked whenever the progress level changes, and can distinguish between user-initiated changes (*i.e.* the user touched the bar) and those occurred programmatically (*i.e.* the playback position has been updated).
- if the change occurred because of the user interaction, set the new playback position in `MediaPlayer` (`seekTo(progress)`);

In order to continuously update the playback position on the bar, a thread can be created to query the current position from `MediaPlayer` with a given frequency, and update the bar accordingly:

- create a new inner class `BarUpdater` that implements `Runnable`. Android Studio automatically generates the code stub for the method `run()` to implement;
- create and start the thread inside the callback listener of the Play button:

```
progressUpdate = new Thread(BarUpdater);
progressUpdate.start()
```

- implement the `run()` method so that the thread gets the current position from `MediaPlayer`, and updates the position of the seek bar (`SeekBar.setProgress(progress)`). Insert a pause of 50 ms between each query `Thread.sleep(50)`.
- Finally, update the life-cycle of the thread whenever the video is stopped, paused or the application is going on pause or resuming.
- Run and test the application

2.4.6 A final touch...

You can improve the application by controlling the properties of some buttons according to the playback state: for example you can enable / disable the Pause button whether the video is playing (`setEnable(true|false)`) or not, and you can hide the seek bar when there is no video playing (`setVisibility()`).

2.5 Picking videos from the Gallery

Writing the name of the file is not really handy when you are using a mobile device. We want to improve the application so that the user can select the video from the Android Gallery application. We then need to create an implicit `Intent` to ask for an available activity able to pick a video from all the video available on the device, and load the result on our video player.

- Add a new button Select from Gallery on top of the control buttons;
- implement a new listener for the button, and create a new implicit `Intent` to pick a video from the gallery:
  - the action related to the intent has to be the "pick a video" kind: look at the documentation of the action `ACTION_GET_CONTENT` and compare it with the usual `ACTION_PICK`;
  - set the type of data to "any kind of video": the activity that receives the intent will only display videos among which the user can choose.
  - here is a general reference about starting Activities to get a result `http://developer.android.com/training/basics/intents/result.html`
- start the activity with `startActivityForResult`, use an arbitrary value for the `requestCode`, *e.g.* you can declare a constant for it

  ```
  private static final int SELECT_VIDEO = 100;
  ```

- override the method `onActivityResult()` of the activity `PlayerActivity` to grab the result when the other activity is done. Verify that the request code matches with the one sent, and that the user has actually selected an element (`resultCode`);
- the result intent gives back the URI of the selected video: get the result data from the intent and show it inside the `EditText`; the URI has the following form:
  `content://media/external/images/media/62`
- when the user presses the button Play the relevant callback reads the content from the `EditText`: now `EditText` already contains the video URI, thus you can eliminate the previous instructions to build the full path to the file. From now on we assume that the user only uses the "select from gallery" feature to load a video;
- change the default text of `EditText` to a blank string;
- run the application and test it.

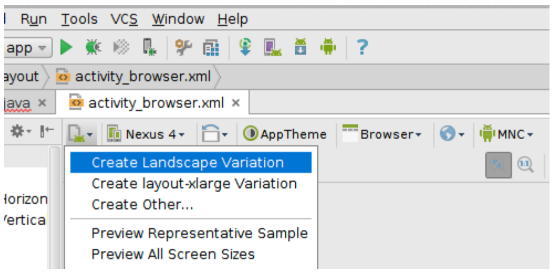

Run Tools VCS Window Help
app
ayout activity_browser.xml
java activity_browser.xml
Nexus 4
AppTheme
Browser
MNC
Create Landscape Variation
Create layout-xlarge Variation
Create Other...
Preview Representative Sample
Preview All Screen Sizes
Horizon
Vertica

Figure 7: How to generate a landscape variation for the portrait layout.

2.6 Video streaming from the Internet

Another feature we want to implement for our application is the video streaming from the Internet. It turns out that it is a very easy feature to implement: `MediaPlayer` can play both local videos and remote videos, it is enough to provide an URL to the video instead of the full local path.

- We can let the user type the URL of the video in the `EditText`; once the Play button is pressed its content is read and passed to `MediaPlayer`;
- if we are playing a remote video it means that we need the access to the Internet connection, so. . . Update the manifest file;
- try with the following video
  http://www.androidbegin.com/tutorial/AndroidCommercial.3gp

2.7 Managing different orientations

So far our application runs just fine. Try to run the application on the emulator and press Ctrl + F12 . The virtual device has switched to the landscape mode (Ctrl + F11 to get back). The result is quite disappointing, though: the control buttons take the whole screen and there is no space for displaying the video. We can then redefine a more optimized layout for the landscape mode. As we have seen in classes, Android keep the resources and the visual elements of the application separated from the source code. Therefore, to fix the orientation problem we just have to redefine a proper layout file for the landscape mode (see Figure 8):

- open the layout file in Design Mode, click on the first combo box of the toolbar menu and choose “Create Landscape Variation” (see Figure 7). A new layout with the same name is created in `res/layout-land` . This layout is loaded every time the device switch to the Landscape mode.
- create a new XML file with the same name as the original one; in `Available qualifiers` choose `Orientation` and the `Landscape` mode. A new file `res/layout-land/` is created, containing the layout for the landscape orientation;
- now you can redefine the position and the size of the buttons as well as the text they display in order to save space and maximize the space for the video. In Figure 8 you can see an example of a possible layout. It is important that you use the same ID for each

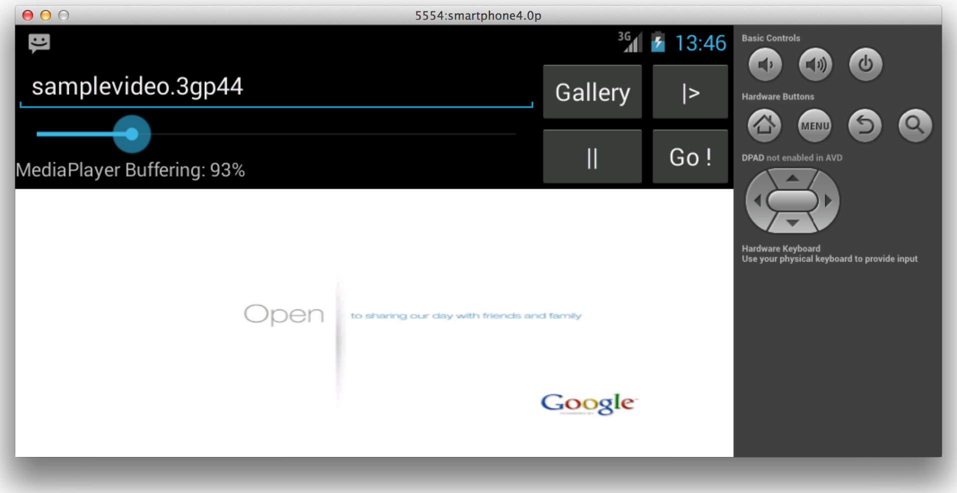


Figure 8: The landscape layout of the Video Player.

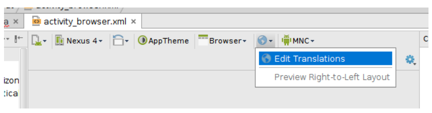
component, while you can change the text using other strings to be placed as usual in `strings.xml`;

- in order to gain more space you can remove the action bar on top of the screen with the name of the application and the icon: change the application style to `Theme.Holo.NoActionBar` using the Style button of the toolbar menu of the Design Mode;
- run again the application and see how it works in landscape mode.

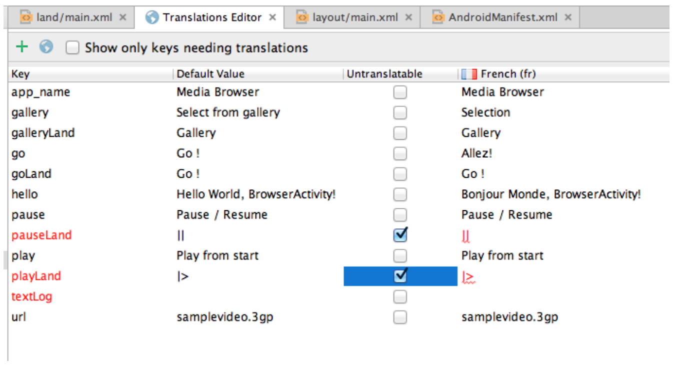
2.8 Managing internationalization and localization

Finally, in order to create a best-seller application it's important to add the support for the internationalization, so that you can reach many markets and much more potential customers:

- open the layout file in Design Mode and click on the button with a "world" icon and select "Edit Translations". This will open the translation editor (see Figure 9.a).
- In the translation editor click again on the button with a "world" icon and select the French country code `fr` (or `en` if you already used French for the text of the GUI).
- Give the translation for the new language for all the string. Note that you can check a box in case the string is not translatable or you do not want to translate it (in this case, *e.g.*, for the button play and pause which are not exactly text, see Figure 9).
- You should notice that a new file `string.xml` has been added to the `res/value` folder.
- Go to the emulator and change the global settings to the French language: now your application automatically loads the French version. This is a good way to keep the visual representation of the application separated from the source code.



(a)



Key	Default Value	Untranslatable	French (fr)
app_name	Media Browser	☐	Media Browser
gallery	Select from gallery	☐	Selection
galleryLand	Gallery	☐	Gallery
go	Go !	☐	Allez!
goLand	Go !	☐	Go !
hello	Hello World, BrowserActivity!	☐	Bonjour Monde, BrowserActivity!
pause	Pause / Resume	☐	Pause / Resume
pauseLand	||	☑	||
play	Play from start	☐	Play from start
playLand	|>	☑	|>
textLog		☐	
url	samplevideo.3gp	☐	samplevideo.3gp

(b)

Figure 9: The steps for managing the localization of the application.