Lab di applicazioni mobili

per qualsiasi errore, potete segnalarmelo su telegram @LuigiBrosNin o qui con un commento, grazie per la collaborazione!

Teoria

▼ Android

https://developer.android.com/guide/components/fundamentals?hl=en

👆leggete qui per qualcosa di completo ed estensivo, official source lol

▼ Android studio (SDK)

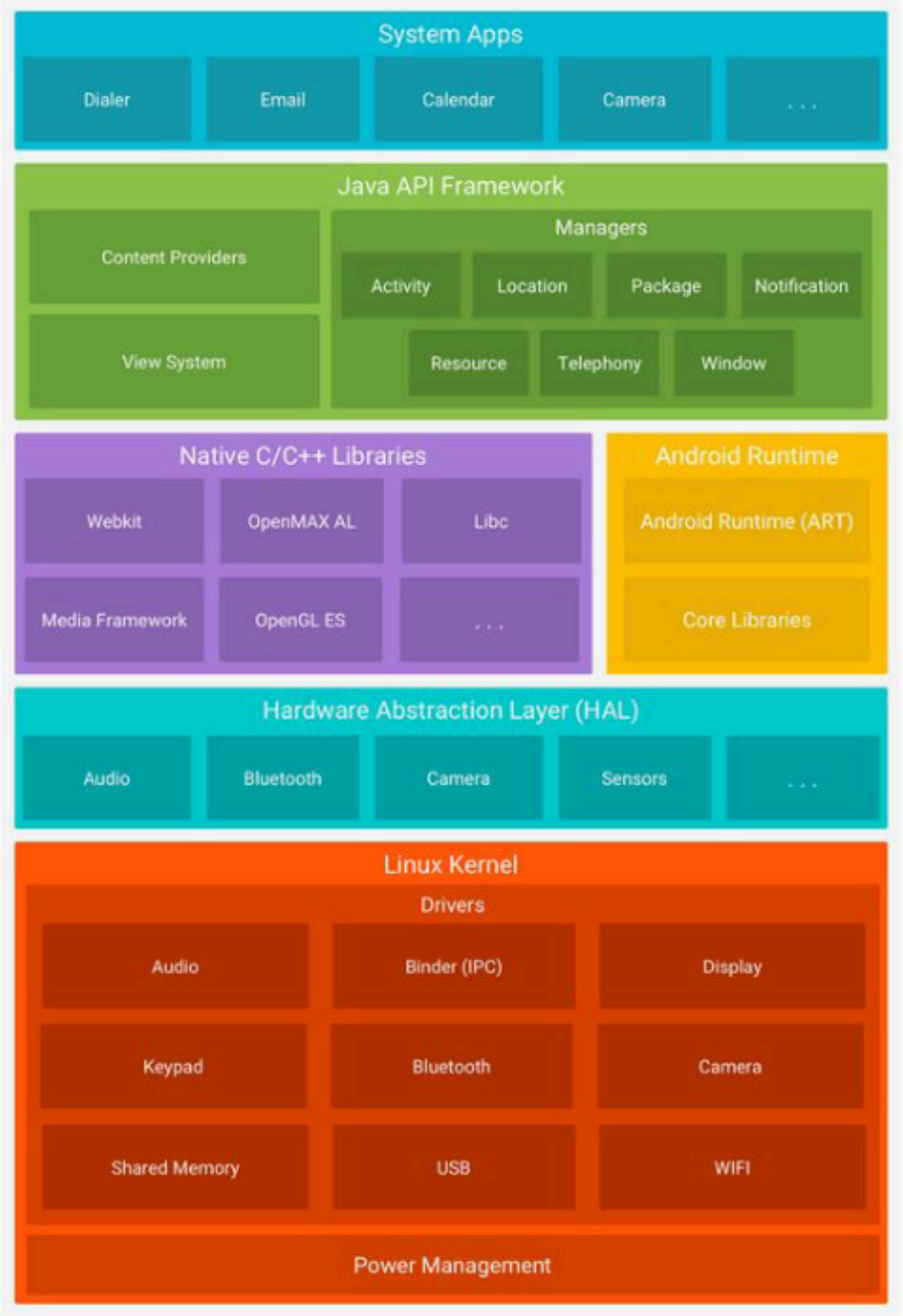
▼ System architecture

Android is a Linux-based platform for mobile touchscreen devices (except not really, we have android fridges)

Android has the Linux Kernel, but there are some distinct differences, for example

- every app is a user (from Linux's pov)

  this is done to grant security to apps, since user A can't access files from other users



HAL → mamages different devices of the same type and shadows the real device

C/C++ Libraries → Graphics, multimedia, SQLite, etc...

ART → java VM implementation (there's more to it)

i can compile in ahead of time compilation while the phone is idling

2 computing levels at runtime:

- complete compiling
- flash compiling → compiles bold methods (aka hot code, that gets executed many times)

API → global java classes, managers that concern the entire OS

Apps → Play store, entertainment, productivity, personalization, education etc

> 🎓 An Activity corresponds to a single screen of the application

an application can be composed of multiple screens, and the home activity is shown when the user launches an application

different activities can exchange information between them

each activity has graphics components, and the Views can interact with the user by handling events (aka stuff like buttons)

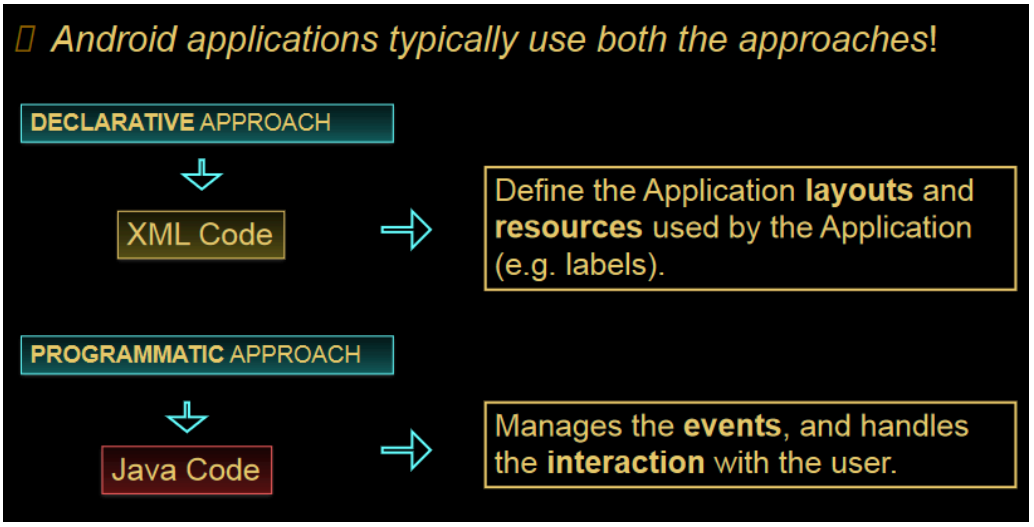
we can declare or program a build of the graphic interface

```
Button button = new Button (this);
TextView text = new TextView();
text.setText("Hello world");
```

```
< TextView android.text=@string/hello" android:textcolor=@color/blue
android:layout_width="fill_parent" android:layout_height="wrap_content" />
< Button android.id="@+id/Button01" android:textcolor="@color/blue"
android:layout_width="fill_parent" android:layout_height="wrap_content" />
```

we can build multiple layouts from different devices, and Android detects the configuration of the device at runtime and loads the appropriate resources

> XML does not need to be compiled 😀



Views can generate events that are managed through CALLBACKS

```
public void onClick(View arg0) {
   if (arg0 == Button) {
      // Manage Button events
   }
}
```

Activities have different states, such as starting, running, stopped, etc...

only one activity can be in the running state at a time

Activities are organized on a stack

most devices have constrained resource capabilities

we need to implement lifecycle methods to account for state changes of each Activity

```
public class MyApp extends Activity {

   public void onCreate() { ... }
   public void onPause() { ... }
   public void onStop() { ... }
   public void onDestroy(){ ... }
   ....
}
```

👆This is a reactive programming style

🎓 Android Intents are asynchronous messages to activate core Android components (like Activities)

an Explicit Intent has a component (Activity 1) that specifies the destination of the intent (Activity 2) like a login, for example

like when you click on a yt link on Telegram and Android asks you if you want to open youtube

🎓 Services are like activities, but run in background and have no UI

🎓 Content providers are a standard interface to access and share data among application, each application has a private set of data

🎓 Broadcast receivers are stuff like Notification bars, that receive external events

🎓 System API already touched on, it manages all the components described above

the Google API is integrated and serves as interface for the google services, like maps

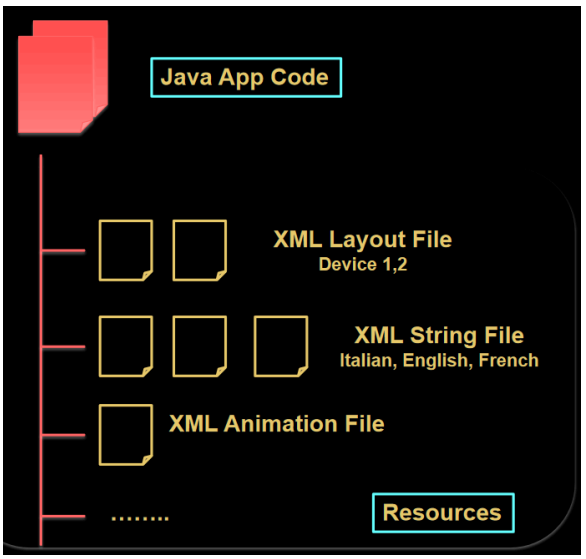
▼ Resources

an application is composed by code and resources

👩‍💻 Resources are everything that is not code, including XML layout files, language packs, media etc.

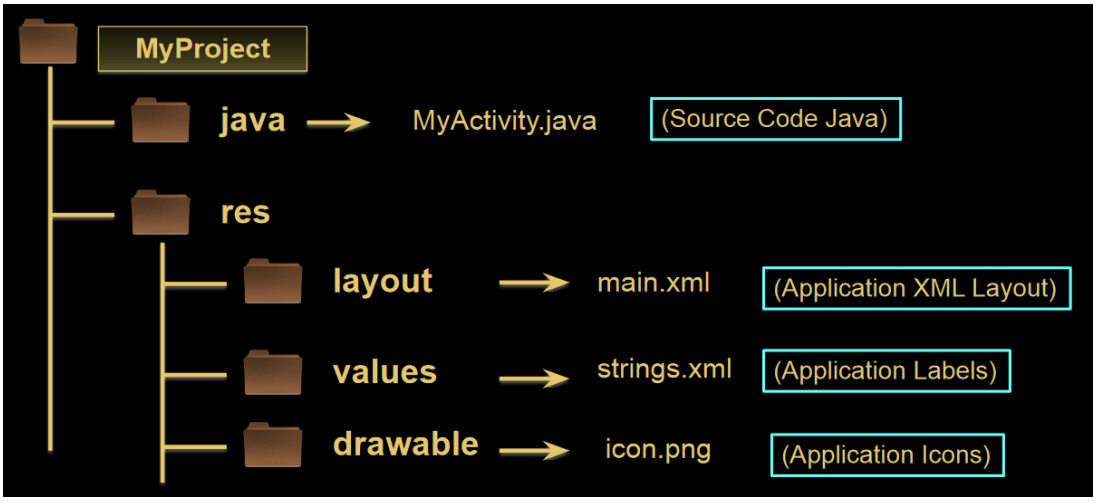
An Android application might run on heterogenous devices with different characteristics (e.g. screen size, language support, keyboard type, input devices, etc)

to fix this, we use separate code from application resources, using a declarative XML-based approach to define resources



Android automatically detects at runtime the current device configuration and loads the appropriate resources for the application

Resources are defined in the `res` folder



Resource Type	Resource contained
- res/animator	*XML files that define property animations.*
- res/anim	*XML files that define tween animations.*
- res/color	*XML files that define a state list of colors.*
+ res/drawable	*Bitmap files (.png, .9.png, .jpg, .gif) or XML files that are compiled into other resources.*
+ res/layout	*XML files that define a user interface layout.*
+ res/menu	*XML files that define application menus.*
- res/raw	*Arbitrary files to save in their raw form.*
+ res/values	*XML files that contain simple values, such as strings, integers, array.*
- res/xml	*Arbitrary XML files.*

Resources can be accessed in the Java code through the R class, that works as a glue between the world of java and the world of resources.

```
public final class R { // the class contains the resource IDs for all the resources in the res directory
   public static final class string {
   public static final int hello=0x7f040001;
   public static final int label1=0x7f040005;
   }
}
// to access the resource, we just need to call its ID
...
final String hello=getResources().getString(R.string.hello);
final String label=getResources().getString(R.string.labelButton);
Log.i(STRING_TAG," String1 " + hello);
Log.i(STRING_TAG," String2 " + label);
...
...
```

in XML, to access a resource, we use the syntax `@[<package_name>:]<resource_type>/<resource_name>` where:

- `<package_name>` is the name of the package in which the resource is located (not required when referencing resources from the same package)
- `<resource_type>` is the the name of the resource type
- `<resource_name>` is either the resource filename without the extension or the android:name attribute value in the XML element.

▼ example

```
<!-- STRING.XML -->
<?xml version="1.0" encoding="utf-8"?>
<resources>
<color name="opaque_red">#f00</color>
<string name="labelButton"> Submit </string>
<string name="labelText"> Hello world! </string>
</resources>

<!-- MAIN.XML -->
<?xml version="1.0" encoding="utf-8"?>
<resources>
<Textview android:id="@+id/label1" android:text="@string/labelText"
```

```
android:textcolor="@color/opaque_red">
</Textview>
<Button android:id="@+id/button1" android:text="@string/labelButton">
</Button>
</resources>
```

in Java, we have the syntax [`<package_name>.]` `R` `.<resource_type>.<resource_name>`

pretty much the same 😃

▼ example

```
// Get a string resource from the string.xml file
final String hello=getResources().getString(R.string.hello);

// Get a color resource from the string.xml file
final int color=getResources().getColor(R.color.opaque_red);

// Load a custom layout for the current screen
setContentView(R.layout.main_screen);

// Set the text on a TextView object using a resource ID
TextView msgTextView = findViewById(R.id.label1);
msgTextView.setText(R.string.labelText);
```

each View and ViewGroups in the example has a number of attributes `android:id="@+id/label_1`

@ means “parse and expand the rest of the string as an id resource”

+ means “ this is going to be added as a new id in `R.java`

▼ Types of resources

Resource Type	File	Java constant	XML tag	Description
string	Any file in the res/values/	R.string.<key>	<string>	**String** value associated to a key.
integer	Any file in the res/values/	R.integer.<key>	<integer>	**Integer** value associated to a key.
array	Any file in the res/values/	R.array.<key>	<string-array> <item> <item> </string-array>	**Array of strings**. Each element is a described by an <item>
array	Any file in the res/values/	R.array.<key>	<integer-array> <item> <item> </integer-array>	**Array of integers**. Each element is a described by an <item>

types of resources

▼ code example

```
<!-- MYVALUES.XML -->
<?xml version="1.0" encoding="utf-8"?>
<resources>

   <string name="app_title"> Example Application </string>
   <string name="label" > Hello world! </string>
   <integer name="val" > 53 </integer>
```

```
    <string-array name="nameArray">
        <item> John Bonham </item>
        <item> Frank Zappa </item>
    </string-array>
    <integer-array name="valArray">
        <item> 1 </item>
        <item> 2 </item>
    </integer-array>

</resources>
```

```
// MYFILE.JAVA
// Access the string value
final String hello=getResources().getString(R.string.app_title);
// Access the string-array values
final String[] names=getResources().getStringArray
(R.array.nameArray);
// Access the integer-array values
final int[] val=getResources().getIntArray(R.array.valArray);
```

Resource Type	File	Java constant	XML tag	Description
layout	Any file in the res/layout/	R.layout.<key>	<layout>	Defines a layout of the screen
animation	Any file in the res/animator/	R.animator.<key>	<animator>	Defines a property animation (not the only method!)
menu	Any file in the res/menu/	R.menu.<key>	<menu>	User-defined menus with multiple options

Resource Type	File	Java constant	XML tag	Description
color	Any file in the res/values/	R.color.<key>	<color>	Definition of **colors** used in the GUI
dimension	Any file in the res/values/	R.dimen.<key>	<dimen>	Dimension **units** of the GUI components
style/theme	Any file in the res/values/	R.style.<key>	<style>	**Themes** and **styles** used by applications or by components

▼ example

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="red"> #FF0000 </color>
```

```
    <color name="red_trasparent" > #66DDCCDD</color>
</resources>
```

dp = pixel, the ones we're gonna use

sp = same thing, but used for text

Code	Description
px	Pixel units
in	Inch units
mm	Millimeter units
pt	Points of 1/72 inch
dp	Abstract unit, independent from pixel density of a display
sp	Abstract unit, independent from pixel density of a display (font)

These units are relative to a 160 dpi (dots per inch) screen, on which 1dp is roughly equal to 1px. When running on a higher density screen, the number of pixels used to draw 1dp is scaled up by a factor appropriate for the screen's dpi. Likewise, when on a lower density screen, the number of pixels used for 1dp is scaled down

▼ example (usage)

```
<!-- MYVALUES.XML -->
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <dimen name="textview_height">25dp</dimen>
    <dimen name="textview_width">150dp</dimen>
    <dimen name="font_size">16sp</dimen>
</resources>

<!-- MAIN.XML -->
<TextView
    android:layout_height="@dimen/textview_height"
    android:layout_width="@dimen/textview_width"
    android:textSize="@dimen/font_size"/>
```

 A Style is a set of attributes that can be applied to a specific component of the GUI (View) or to the whole screen or application (in this case, it is also referred as "theme").

Styles can be organized in a hierarchical structure. A style can inherit properties from another style, through the parent attribute.

Use <style></style> tags to define a style in the res/ folder.

Use <item> to define the attributes of the style.

▼ example

```
<!-- MYVALUES.XML -->
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="CustomText" parent="@style/Text">
        <item name="android:textSize">20sp</item>
        <item name="android:textColor">#008</item>
    </style>
```

```
</resources>

<!-- MAIN.XML -->
<EditText style="@style/CustomText"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Hello, World!" />
```

the Drawable resource is a general concept for a graphic that can be drawn on the screen, such as images and XML resources, here's the complete list

Drawable type	Description
BitMap File	A bitMap Graphic file (.png, .gif. .jpeg)
Nine-Patch File	A PNG file with stretchable regions to allow resizing
Layer List	A Drawable managing an array of other drawables
State List	A Drawable that references different graphics based on the states
Level List	An XML managing alternate Drawables. Each assigned to a value
Transition	A Drawable that can cross-fade between two Drawable
Inset	A Drawable that insets another Drawable by a specific distance
Clip	A Drawable that clips another Drawable based on its current level
Scale	A Drawable that changes the size of another Drawable
Shape	An XML file that defines a geometric shape, colors and gradients

XML and RAW types

Resource Type	File	Java constant	XML tag	Description
xml	Any file in the res/xml/	R.xml.<key>	<xml>	User-specific XML file with name equal to key
raw	Any file in the res/raw/	R.raw.<key>	<raw>	Raw resources, accessible through the R class but not optimized

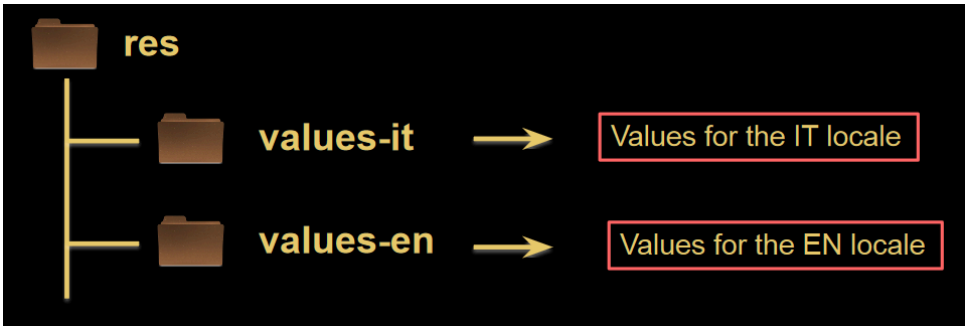
Used to define resources for which no run-time optimization must be performed (e.g. audio/video files). They can be accessed an a stream of bytes, by using Java **InputStream** objects:

InputStream is= getResources().**openRawResource**(R.raw.videoFile)

Alternatives are those configurations that apply differently on different android devices, remember?

Name of the folder: <resources_name>-<config_qualifier>

so we just need to ad a - and a qualifier to make an alternative!



Configuration	Values Example	Description
MCC and MNC	mcc310, mcc208, etc	mobile country code (MCC)
Language and region	en, fr, en-rUS, etc	ISO 639-1 language code
smallestWidth	sw320dp, etc	shortest dimension of screen
Available width	w720dp, w320dp, etc	minimum available module width
Available height	h720dp, etc	minimum available module height
Screen size	small, normal, large, ...	screen size
Screen aspect	long, notlong	aspect ratio of the screen
Screen orientation	port, land	screen orientation (can change!)
Screen pixel density (dpi)	ldpi, mdpi, hdpi	screen pixel density
Keyboard availability	keysexposed, etc	type of keyword
Primary text input method	nokeys, qwerty	availability of qwerty keyboard
Navigation key availability	navexposed, etc	navigation keys of the application
Platform Version (API level)	v3, v4, v7, etc	API supported by the device

attribute importance in case of dupes

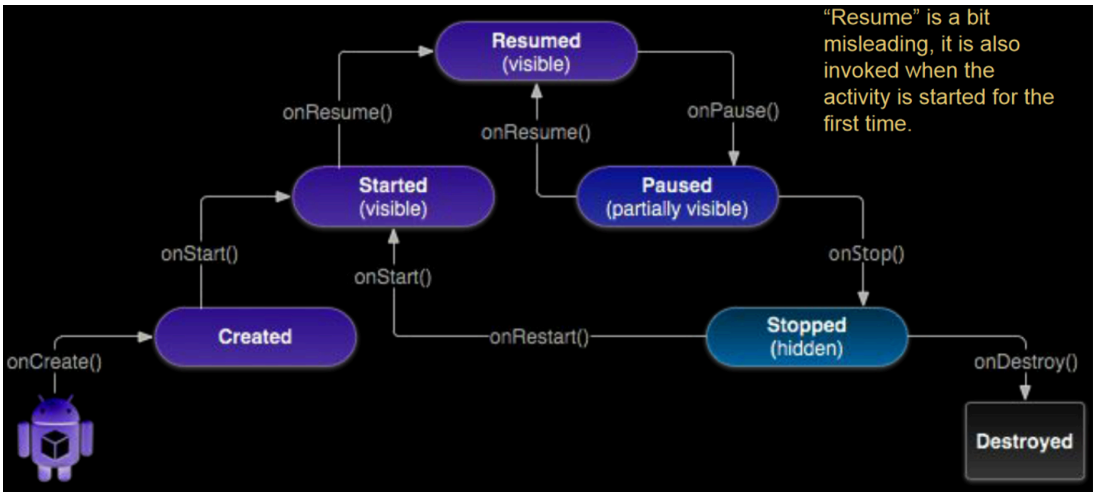
at runtime, android deletes all unmatched configuration folders

the best practice would be giving default resources for your apps, provide alternatives based on the target market of your app, avoid unused resources alternatives and use alias to reduce the dupes

▼ Activities

Activities are what the device starts, an application can be composed of multiple activities

we call activity a screen state



▼ activity lifecycle

activity lifecycle is important because it prevents the app from crashing or bugging out when the user is running something else, does not consume unnecessary resources

- `onCreate()` → initialization operations, calls `onStart()`
- `onStart()` → called before it is visible to the user, on focus calls `onResume()`, otherwise calls `onStop()`
- `onResume()` → activity is ready
- `onPause()` → called when another activity comes to the foreground, stops cpu-consuming processes

  paused activities keep the activity's UI in memory, an activity gets paused on stuff like pop ups, calls and such
- `onRestart()` → similar to `onCreate()`
- `onStop()` → when activity is no longer visible
- `onDestroy()` → destroy 💀

activities should be declared in the Manifest

when an activity is destroyed and recreated, android saves a data bundle called instance state.

if you want to save data, we can override `onSaveInstanceState()` and `onRestoreInstanceState()`

▼ example

```
static final String STATE_SCORE = "playerScore";
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    super.onSaveInstanceState(savedInstanceState);
    savedInstanceState.putInt(STATE_SCORE, mCurrentScore);
}
```

when tagging free strings, it's good practice to create a constant label for that string to operate

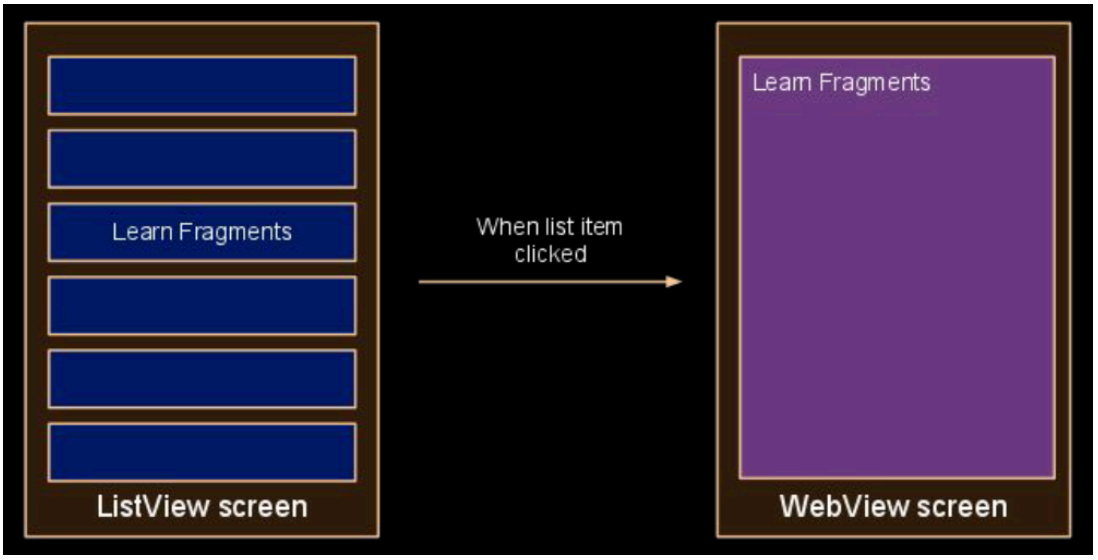
▼ pratically

```
final String COUNTER_LABEL = "COUNTER"
// chiamo COUNTER_LABEL invece che la stringa libera
// this is done because the IDE can't recognize typos in free strings, since it has no way to tell if the free stri
```

▼ Fragments

Fragments are a portion of the UI in an activity, a modular selection of that activity.

by this we can see an activity as a collection of fragments



▼ defining a fragment

to define a new Fragment → create a subclass of Fragment.

`public class MyFragment extends Fragment { ... }`

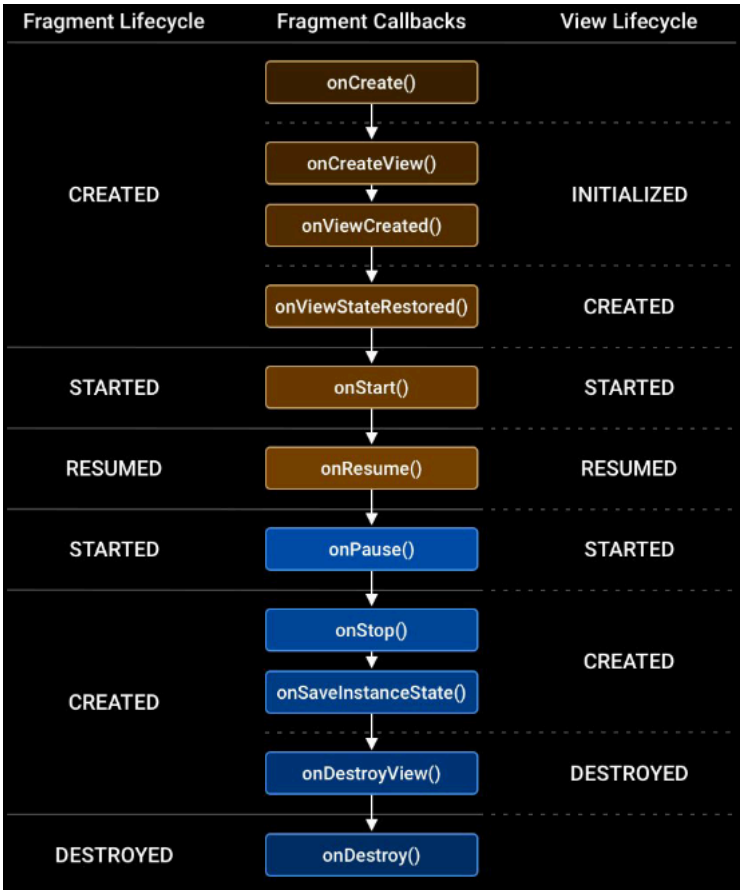
Proprieties

1. Has its own lifecycle (partially connected with the Activity lifecycle)
2. Has its own layout (or may have)
3. Can receive its own input events
4. Can be added or removed while the Activity is running.
5. Cannot run by itself (always hosted by an Activity)

we can also extend

- `DialogFragment`: better than the normal dialog helper, 'cause you can reuse it.
- `ListFragment`: a good alternative for managing lists quite similar to `ListActivity`
- `PreferenceFragmentCompat`: typically used for displaying preference screens.

▼ Lifecycle and methods



- `onCreate()` → called when creating the Fragment (elements retained when stopped).
- `onCreateView()` → called when it is time for the Fragment to draw the user interface the first time (or coming back from the backstack).
  Good to set the properties in `onViewCreated()`. Must return the View associated to the UI of the Fragment (if any).
- `onPause()` → called when the user is leaving the Fragment (commit changes in need of persistence).

The lifecycle of the Activity in which the Fragment lives directly affects the lifecycle of the Fragment.

- `onPause (Activity)` → `onPause (Fragment)`
- `onStart (Activity)` → `onStart (Fragment)`
- `onDestroy (Activity)` → `onDestroy (Fragment)`

Fragments have also extra lifecycle callbacks to enable runtime creation/destruction.

```
// a layoutinflater will help you return the View associated to the UI you need.
// container is the parent ViewGroup of the Fragment
// inflate takes the resource to be inflated and where it should be inflated.

public class ExampleFragment extends Fragment {

   @Override
   public View onCreateView(LayoutInflater inflater,
   ViewGroup container, Bundle savedInstanceState) {
      return inflater.inflate(R.layout.example_fragment,
   container, false);
   }
}
// A recent alternative is to pass the layout over to the super constructor
```

```
public class ExampleFragment extends Fragment {
    public ExampleFragment () {
        super(R.layout.example_fragment);
    }
} // this is not as customizable tho
```

▼ adding it to the UI

the layout proprieties get edited as if it was a view

```
<?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:name="it.cs.android30.FragmentOne" <!-- fragment class --
    android:id="@+id/f1"
    android:layout_width="wrap_content"
    android:layout_height="fill_parent"
    />
<fragment android:name="it.cs.android30.FragmentTwo"
    android:id="@+id/f2"
    android:layout_width="wrap_content"
    android:layout_height="fill_parent"
    />
<!-- these proprieties are permanent -->
</LinearLayout>
```

once specified, the system assigns the layout to the activity, creates all fragments and calls `onCreate()`.

it calls the `onCreateView()` → loads the content and puts it on screen trough the inflater.

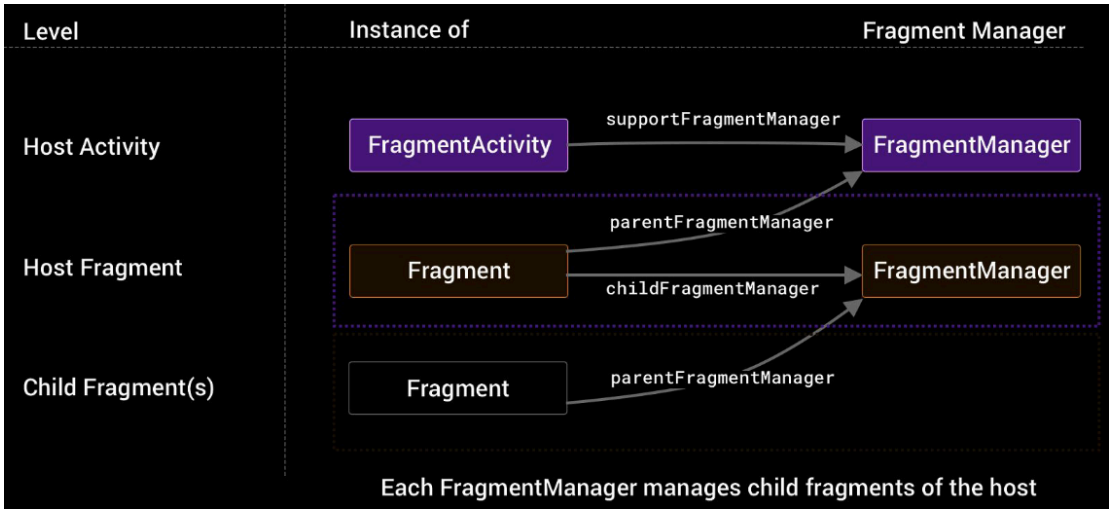
▼ Managing Fragments

FragmentManager, a support API element that handles the Fragments' lifecycle and scheduling:

`getSupportFragmentManager()` → from within an activity

`getParentFragmentManager()` → from within a Fragment

The **FragmentManager** manages the Fragment associated to the current Activity.



A Fragment can get a reference to the Activity ...

`getActivity()`

An Activity can get a reference to the Fragment ...

`ExampleFragment fragment=(ExampleFragment)`

`getSupportFragmentManager().findFragmentById(R.id.example_fragment)`

Before a Fragment enters the lifecycle, it calls its `onAttach()` method right when it gets passed to the **FragmentManager**.

The dual is `onDetach()`.

if you need a reaction from an Activity to a Fragment event, we need an interface from the Activity and Fragment checks for it in the `onAttach()`

```
public static class FragmentA extends ListFragment {
   OnArticleSelectedListener listener;
   ...
   @Override
   public void onAttach(Context context) {
      super.onAttach(context);
      try {
         listener = (OnArticleSelectedListener) context;
      } catch (ClassCastException e) {
         throw new ClassCastException(context.toString() + " must implement OnArticleSelectedListener");
      }
   }
}
```

▼ Transactions

Fragments can be **added, removed** and **replaced** during Activity runtime, these changes are called **Transactions**.

We can save Transactions so that the user can navigate them (e.g. the “Back” button)

```
//1. ACQUIRE an instance of the FRAGMENT MANAGER
FragmentManager fm = getSupportFragmentManager();
FragmentTransaction transaction = fm.beginTransaction();
transaction.setReorderingAllowed(true);
//2. CREATE new Fragment and Transaction (changes you want at the same time)
FragmentExample newFragment = new FragmentExample();
transaction.replace(R.id.fragment_container, newFragment);
//3. SAVE to backStack and COMMIT
transaction.addToBackStack(“FragmentExample”);
transaction.commit();
```

FragmentActivity then automatically retrieves fragments from the back stack via `onBackPressed()`

A Transaction is not performed till the commit ...

- If `addToBackStack()` is not invoked the Fragment is destroyed and it is not possible to navigate back.
- If `addToBackStack()` is invoked the Fragment is stopped and it is possible to resume it when the user navigates back.
- `popBackStack()` simulates a Back from the user.

You can’t replace a fragment declared in the static XML.

Androidx with `FragmentContainerView` makes all fragments dynamic when declared on the XML Layout

```
<fragment android:name="it.cs.android30.FragmentOne"
   android:id="@+id/f1"
   android:layout_width="wrap_content"
   android:layout_height="match_parent"
/>

<androidx.fragment.app.FragmentContainerView android:name="it.cs.android30.FragmentOne"
```

```
    android:id="@+id/f1"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
/>
```

With FragmentContainerView new Fragments can be replaced easily

- Layout of Fragments have always to be within a FrameLayout (not necessarily true for <fragment>).
- If FragmentContainerView has an `android:name` or a class then it triggers a Fragment Transaction when the Activity starts up.
    - i.e. it is not static, but it behaves like so...

▼ Intents

An Android application can be composed of multiple Activities.

Each activity must be declared in the file `AndroidManifest.xml` Unless we're using an external activity.

```
<application>
    <activity android:name=".MyActivity" />
    <activity android:name=".SecondActivity" />
</application>
```

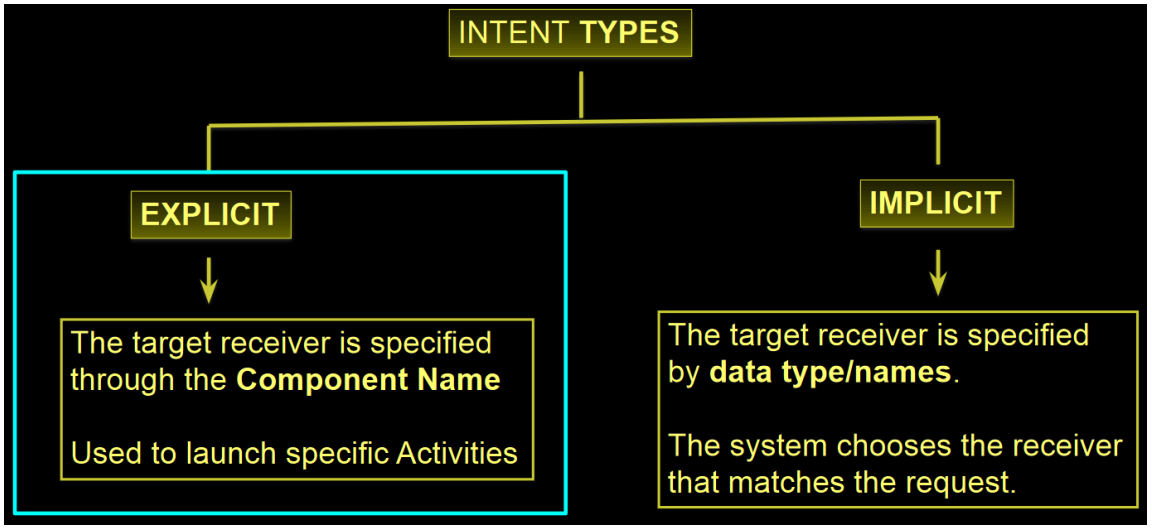
> 🎂 Intent: facility for late run-time binding between components in the same or different applications. an intent is a message object

late run-time: viene risolto nel momento che viene lanciata la funzione che usa l'intent specificato

it's very important to test them for this reason, as compiling and starting the app won't call them at all, so if they don't work well... 😅

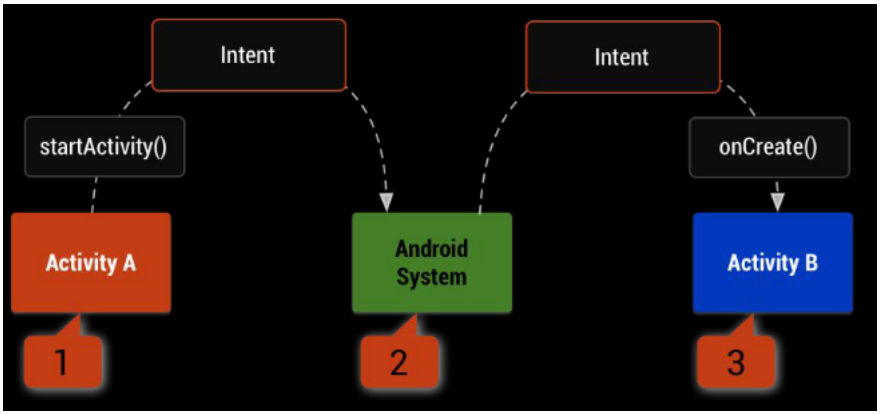
- Call a component from another component
- Possible to pass data between components
- Components: Activities, Services, Broadcast receivers
- Something like:
    - "Android, please do that with this data"
- Reuse already installed applications and components

structure of an activity



since Activities can return results, we can call Intents that give us back data!

```
Intent intent = getIntent();
setResult(RESULT_OK, intent);
intent.putExtra("result", resultValue);
finish();
```



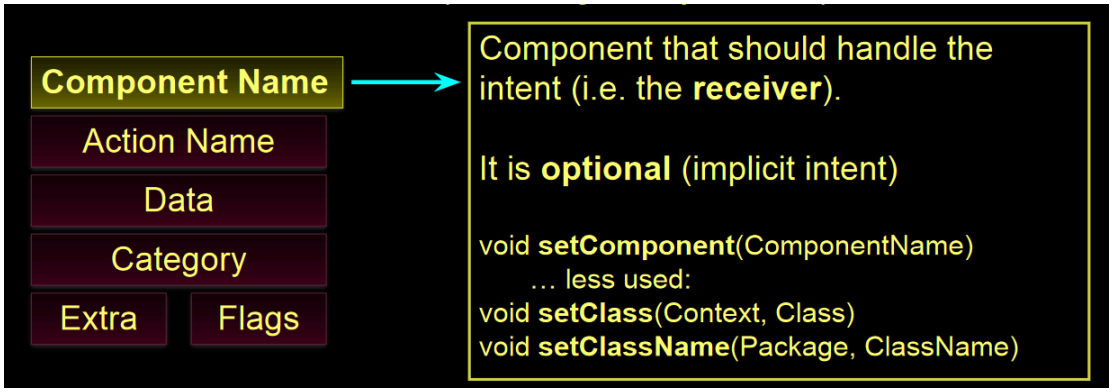
Activity A fires an Intent

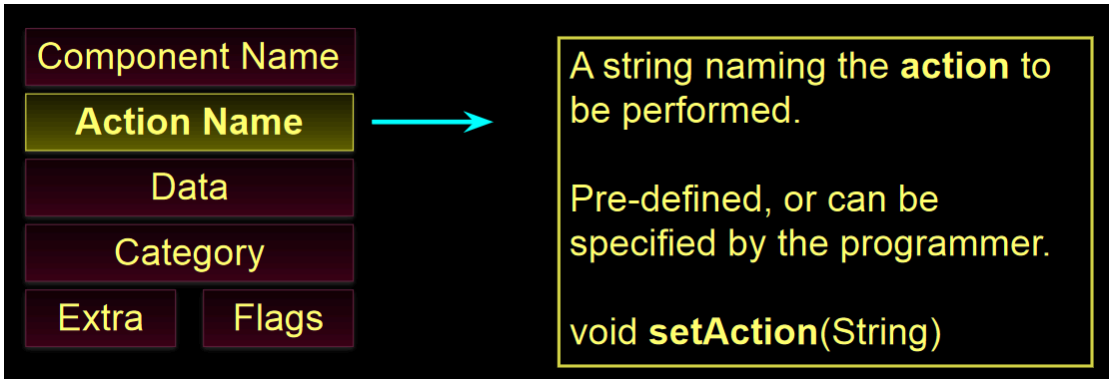
Android System looks for suitable activities (looks at the manifest of other apps)

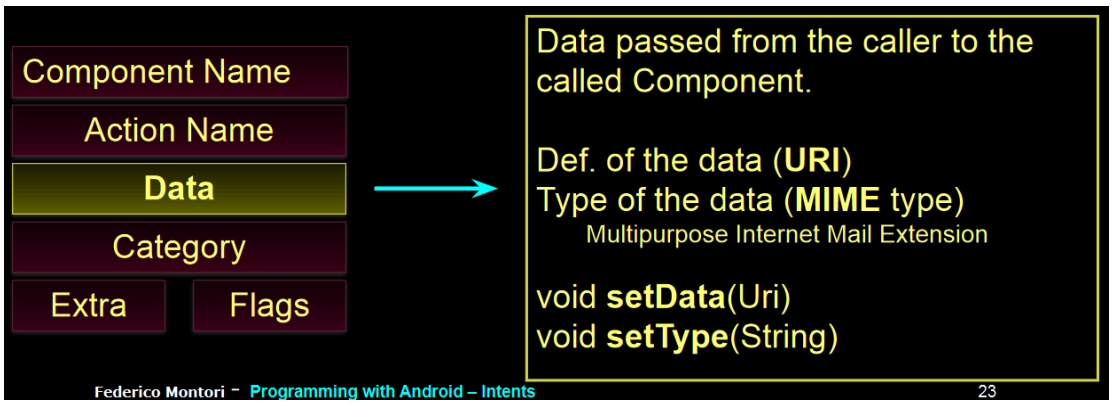
when at least one is found, it is called

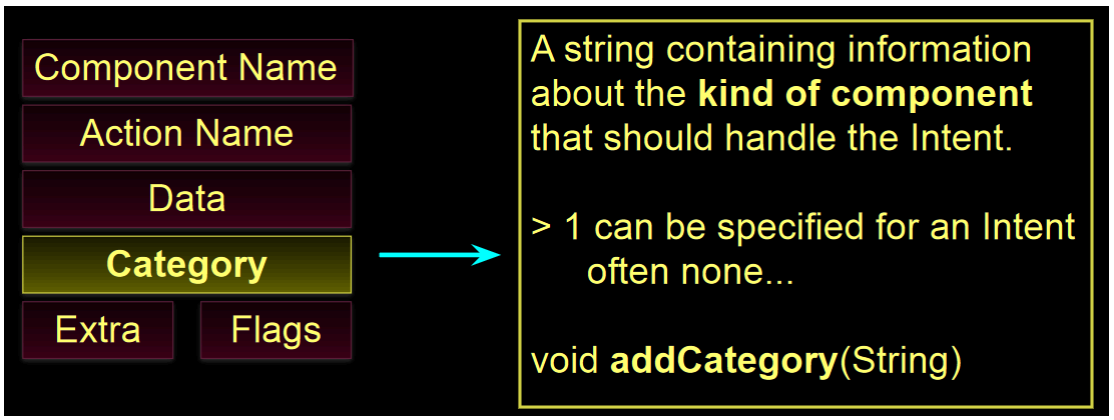
if multiple are found, we can choose.

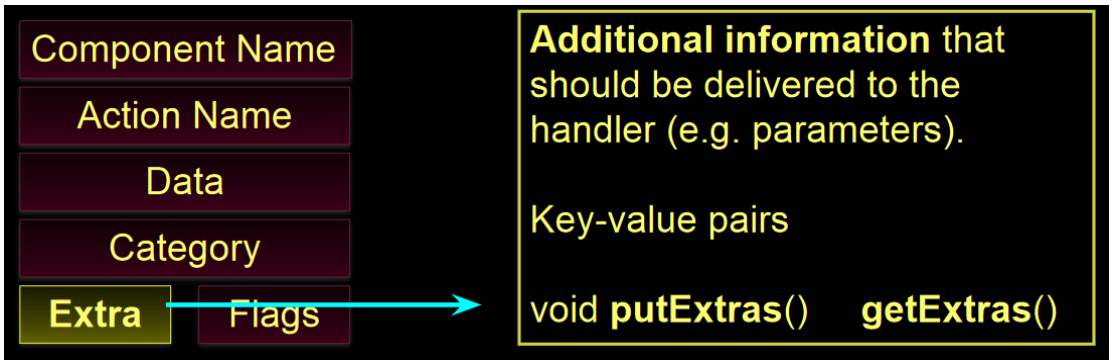
▼ Intent components

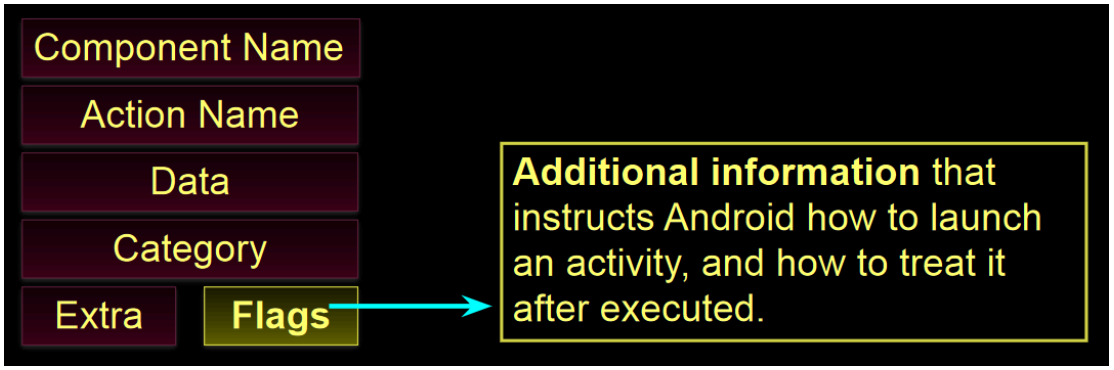



Component Name
Action Name
Data
Category
Extra
Flags
A string naming the **action** to be performed.
Pre-defined, or can be specified by the programmer.
void **setAction**(String)


Component Name
Action Name
Data
Category
Extra
Flags
Data passed from the caller to the called Component.
Def. of the data (**URI**)
Type of the data (**MIME** type)
Multipurpose Internet Mail Extension
void **setData**(Uri)
void **setType**(String)
Federico Montori - Programming with Android – Intents
23


Component Name
Action Name
Data
Category
Extra
Flags
A string containing information about the **kind of component** that should handle the Intent.
> 1 can be specified for an Intent
often none...
void **addCategory**(String)


Component Name
Action Name
Data
Category
Extra
Flags
Additional information that should be delivered to the handler (e.g. parameters).
Key-value pairs
void **putExtras**() **getExtras**()



▼ types of actions

You can use Actions to determine what the called Activity is supposed to do, for example:

- ACTION_VIEW is called when the receiving Activity shows something to the user (e.g. a photo in the gallery, an address on the map).
- ACTION_SEND is called when the receiving Activity is able to send the data received through the Intent using some dedicated channel (e.g. an e-mail or a message in a social app)

Action Name	Description
ACTION_EDIT	Display data to edit
ACTION_MAIN	Start as a main entry point, does not expect to receive data.
ACTION_PICK	Pick an item from the data, returning what was selected.
ACTION_VIEW	Display the data to the user
ACTION_SEARCH	Perform a search
ACTION_SEND	Send some data through another component

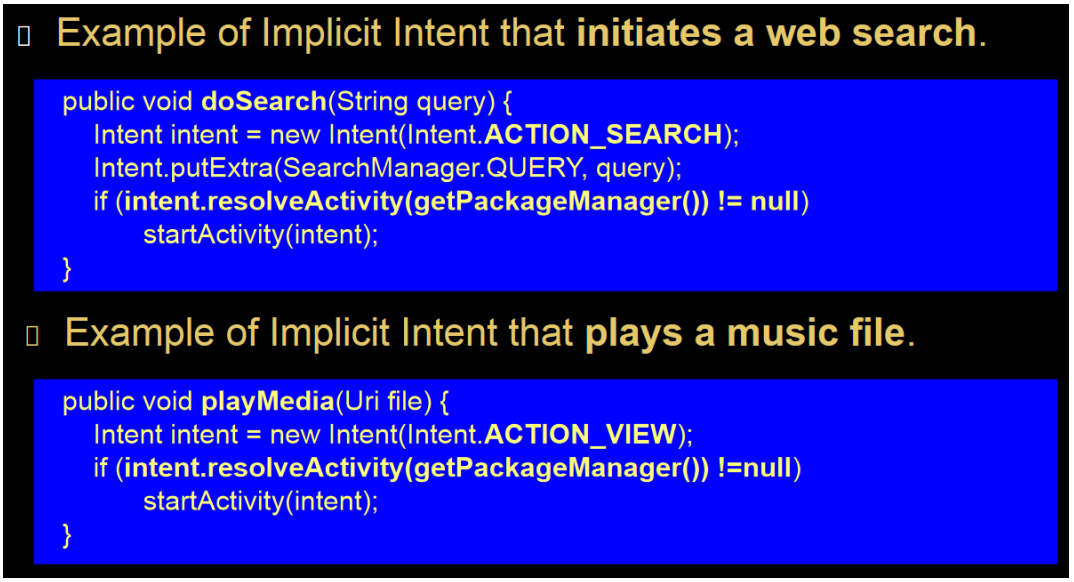
we can also define actions

`it.example.projectpackage.FILL_DATA (package prefix + name action)`

Special actions:

Action Name	Description
ACTION_IMAGE_CAPTION	Open the camera and receive a photo
ACTION_VIDEO_CAPTION	Open the camera and receive a video
ACTION_DIAL	Open the phone app and dial a phone number
ACTION_SENDTO	Send an email (email data contained in the extra)
ACTION_SETTINGS	Open the system setting
ACTION_WIRELESS_SETTINGS	Open the system setting of the wireless interfaces
ACTION_DISPLAY_SETTINGS	Open the system setting of the display

▼ examples i can't bother to write manually nor copy + paste



among types, we have MIME (Multipurpose Internet Mail Extensions)-type

MIME type is important to help the system handle better the intent... although the type of data supplied is often dictated by the Action (e.g. if Action is ACTION_EDIT, then the data is the Uri of the document to edit).

> ⚠ Do not call `setData()` and `setType()` if you need to set both because they nullify each other: call `setDataAndType()`

Category Name	Description
CATEGORY_HOME	The activity displays the HOME screen.
CATEGORY_LAUNCHER	The activity is listed in the top-level application launcher, and can be displayed.
CATEGORY_PREFERENCE	The activity is a preference panel.
CATEGORY_BROWSABLE	The activity can be invoked by the browser to display data referenced by a link.

Category: string describing the kind of component that should handle the intent

▼ Implicit intents

```
Intent i = new
   Intent(android.content.Intent.ACTION_VIEW, //action to perform
   Uri.parse("http://informatica.unibo.it")); //data to perform the action to
startActivity(i);
```

Implicit intents are very useful to re-use code and to launch external applications

More than a component can match the Intent request

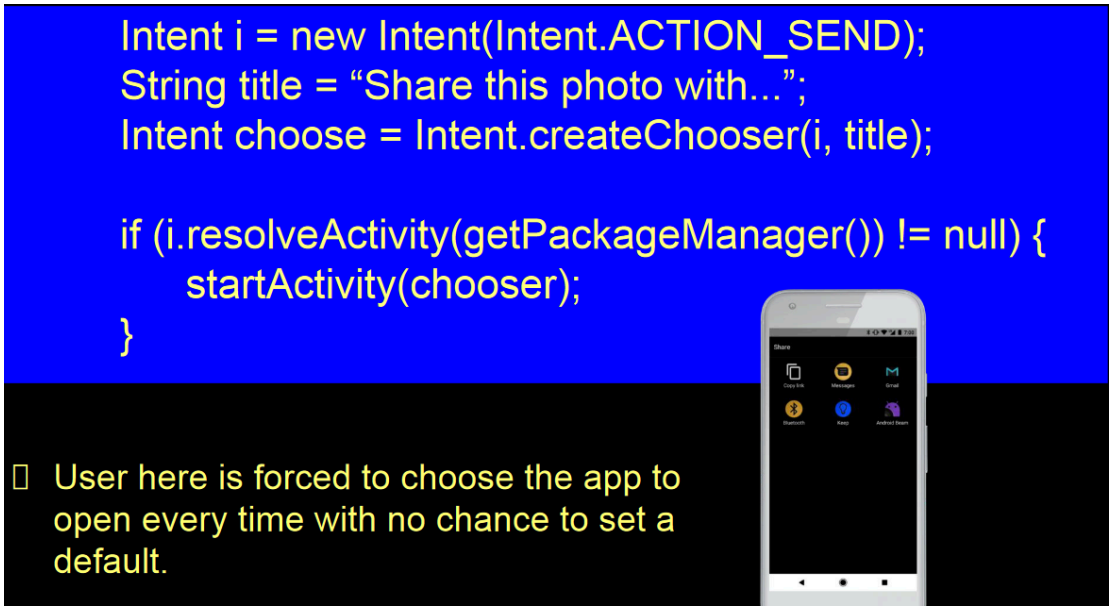
How to define the target component?

```
String msg = "This has been sent through an Intent!";
Intent i = new Intent();
i.setAction(Intent.ACTION_SEND);
i.putExtra(Intent.EXTRA_TEXT, msg);
i.setType("text/plain");

if (i.resolveActivity(getPackageManager()) != null) {
```

```
    startActivity(i);
}
```

receiver is not determined by the data Uri, but by the `ACTION_SEND` and the data Type.



`<intent-filter> tag in AndroidManifest.xml` to declare which intents i’m able to handle

```
<intent-filter>
    <action android:name=“my.project.ACTION_ECHO” />
</intent-filter>
```

If you specify more than one instance of the same tag (e.g. more than one action) → Your activity should handle each combination of these

▼ Intent resolution

The intent resolution process resolves the Intent-Filter that can handle a given Intent.

Three tests to be passed:

- Action field test → The action specified in the Intent must match one of the actions listed in the filter
- Category field test → Every category in the Intent must match a category of the filter
- Data field test → The URI of the intent is compared with the parts of the URI mentioned in the filter (this part might be incomplete)

if it passes, it’s selected to handle the intent.

▼ Activity results

From Androix 1.2.0 you can use the Activity Result API which leverages a whole new set of functions (all async).

They did this to overcome cases when the calling activity is destroyed and recreated while the called one is running

```
ActivityResultLauncher<String> mGetContent = registerForActivityResult( //register returns a launcher that
    new GetContent(), //returns stuff, in this case an URI
    new ActivityResultCallback<Uri>() {
        @Override
        public void onActivityResult(Uri uri) { /* Handle the returned Uri */ }
});
```

`mGetContent.launch("image/*");` ← you can fire this by passing in the data type you want

contract-less method:

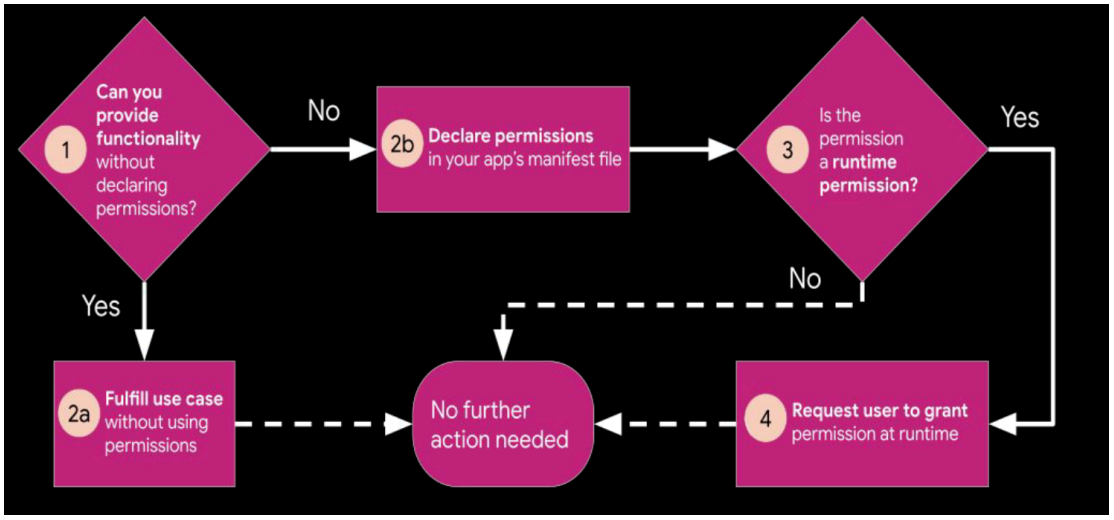
```
ActivityResultLauncher<Intent> mLauncher = registerForActivityResult(
    new StartActivityForResult(),
    new ActivityResultCallback<ActivityResult>() {
    @Override
    public void onActivityResult(ActivityResult result) {
        if (result.getResultCode() == Activity.RESULT_OK) { Intent intent = result.getData(); }
    }
}); --> mLauncher.launch(new Intent(this, resultActivity.class))
```

Custom contract:

```
public class PickRingtone extends ActivityResultContract<Integer, Uri> {
    @Override
    public Intent createIntent(@NonNull Context context, @NonNull Integer ringtoneType) {
        Intent intent = new Intent(Intent.ACTION_GET_CONTENT);
        intent.putExtra(RingtoneManager.EXTRA_RINGTONE_TYPE, ringtoneType.intValue());
        return intent;
    }

    @Override
    public Uri parseResult(int resultCode, @Nullable Intent result) {
        if (resultCode != Activity.RESULT_OK || result == null) {
            return null;
        }
        return result.getParcelableExtra(RingtoneManager.EXTRA_RINGTONE_PICKED_URI);
    }
}
```

▼ Permissions



android's permission system

Up to 6.0 (excluded)

Just declare them in the manifest

`<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>`

Starting from 6.0

User can only grant a subset of the permission set User can revoke permission after installing the app Declare them in the manifest And check if the permission is granted

```
if (ContextCompat.checkSelfPermission(thisActivity, Manifest.permission.ACCESS_FINE_LOCATION)
!= PackageManager.PERMISSION_GRANTED) {
   // Permission is not granted
}
//if it is not requested, ask for it
ActivityCompat.requestPermissions(thisActivity,
   new String[]{Manifest.permission.ACCESS_FINE_LOCATION},
   MY_PERMISSIONS_LOCATION);

// wait for the user response asynchronously
@Override
public void onRequestPermissionsResult(int requestCode, String permissions[], int[] grantResults) {
   switch (requestCode) {
      case MY_PERMISSIONS_LOCATION: {
         if (grantResults.length > 0
               && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
               // GRANTED
               } else { /* DENIED*/ }
               return;
         }
   }
}
```

▼ Views, Layouts, Events

Layouts are the placement of elements on the screen, Views are the elements to be placed

▼ Views

View objects are basic building blocks for user interface components

some examples are GoogleMap, WebView, Widgets, ...

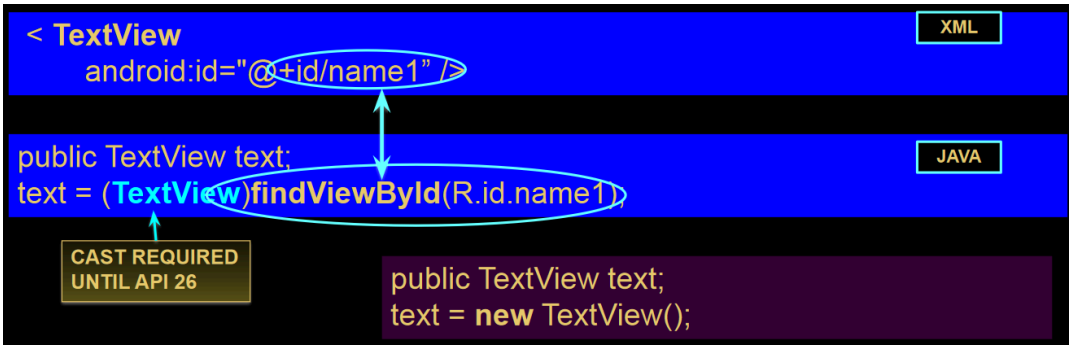
▼ View defined in

XML

```
< TextView
   android:id="@+id/textLabel”
   android:width=“100dp”
   android:height=“100dp”
   android:layout_width="match_parent"
   android:layout_height="wrap_content”
   android:visibility="visible”
   android:enabled="true”
   android:scrollbars="vertical”
   ....
/>
```

Java

(we can create views on Java or access them from XML)



Each View can generate events, that can be captured by Listeners (or other methods) that define the appropriate actions to be performed in response to each event.

Each View can have a focus and a visibility, based on the user's interaction.

The user can force a focus to a specific component through the `requestFocus()` method.

The user can modify the visibility of a specific component through the `setVisibility(int)` method.

Views are interactive components, they can generate events trough user interactions

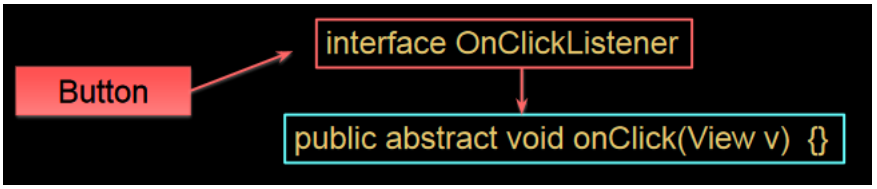
we can handle events trough XML, Event handlers and Event listeners (the recommended one)

▼ code for XML

```
<Button
   android:text="@string/textButton"
   android:id="@+id/idButton"
   android:onClick="doSomething"
/>

//Java
public void doSomething(View w) {
   // Code to manage the click event
}
```

for event Listeners, we have nested interfaces that handle single type of events and contains a `callback` method.



To handle OnClick events through the ActionListener:

1. Implement the nested interface in the current Activity
2. Implement the `callback` method (`onClick`)
3. Associate the ActionListener to the Button through the `View.setOnClickListener()` method

```
// LAMBDA notation with java 8
Button btn = findViewById(R.id.buttonNext);
btn.setOnClickListener(v -> {
      // Event management
   }
});
```

▼ some `ActionListener`

- interface OnClickListener

  abstract method: `onClick()`
- interface OnLongClickListener

  abstract method: `onLongClick()`
- interface OnFocusChangeListener

  abstract method: `onFocusChange()`
- interface OnKeyListener

  abstract method: `onKey()`
- interface OnCheckedChangeListener

  abstract method: `onCheckedChanged()`
- interface OnItemSelectedListener

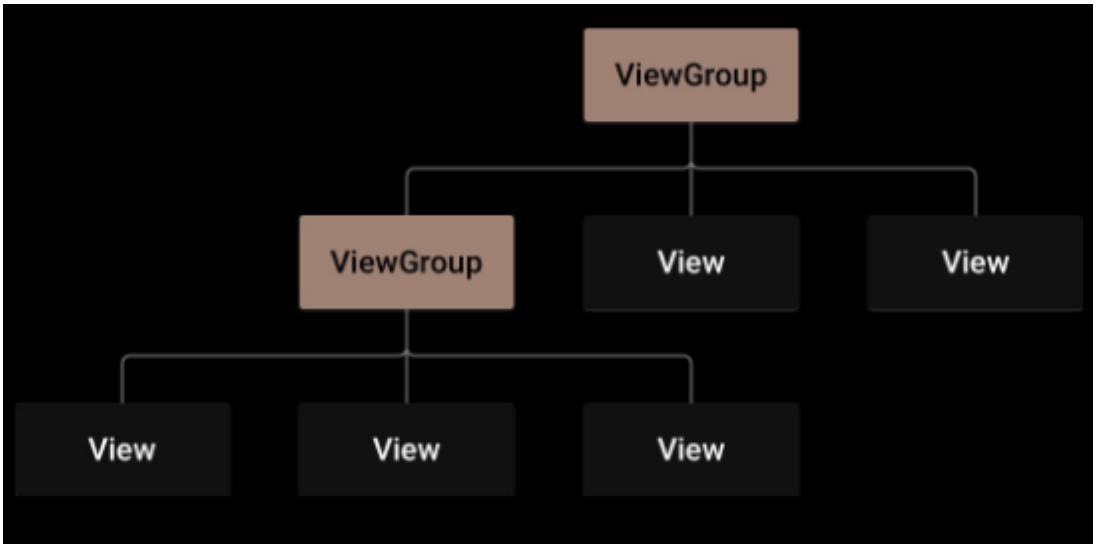
  abstract method: `onItemSelected()`
- interface OnTouchListener

  abstract method: `onTouch()`
- interface OnCreateContextMenuListener

  abstract method: `onCreateContextMenu()`

▼ Layouts

 ViewGroup objects are invisible containers that define a layout structure for the Views declared in it

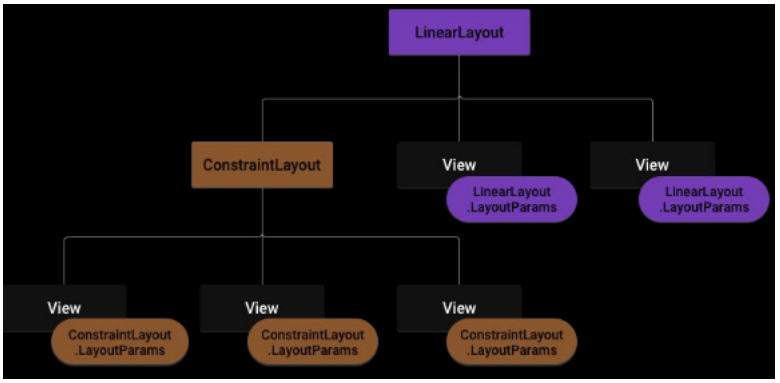


every view has an ID findable with `findViewById(int id)`

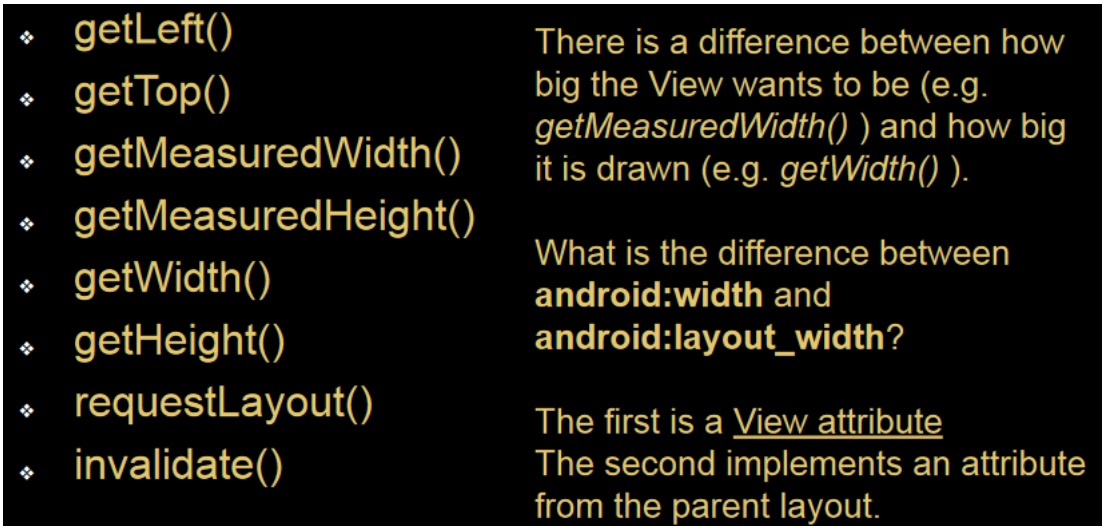
we can customize views to our likings and single views are referred to widgets (NOT app widgets)

a Layout is an extended ViewGroup

XML layour attrubutes are named `layout_something`



Useful methods regarding Layouts:



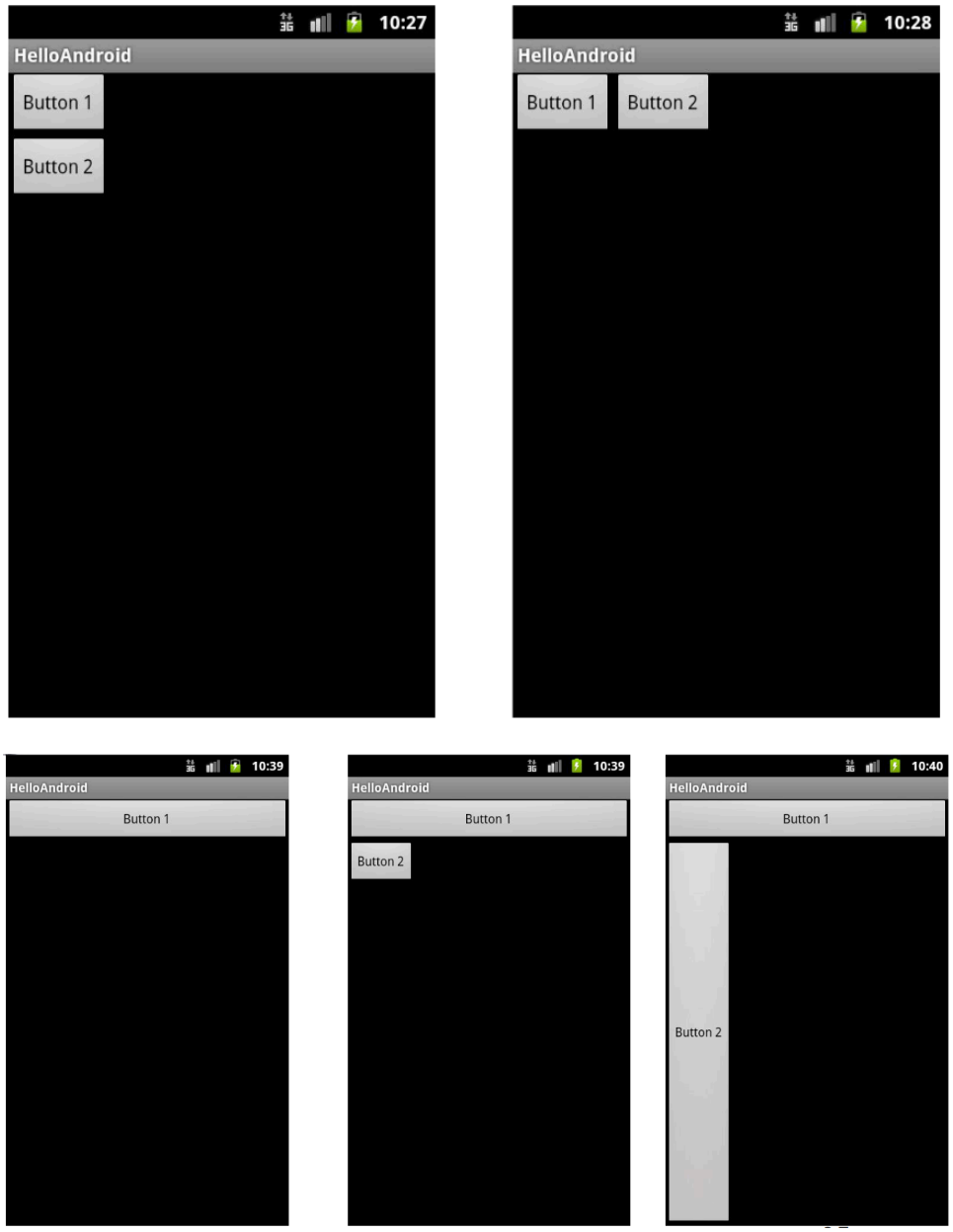
Android supports both padding and margin

padding is a view propriety `android:padding`

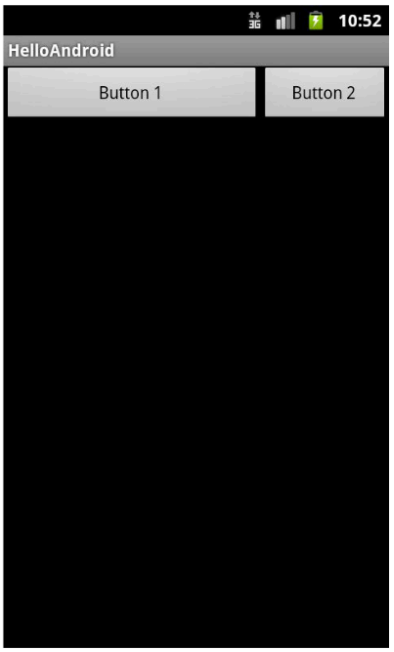
margin is a layout propriety `android:layout_margin`

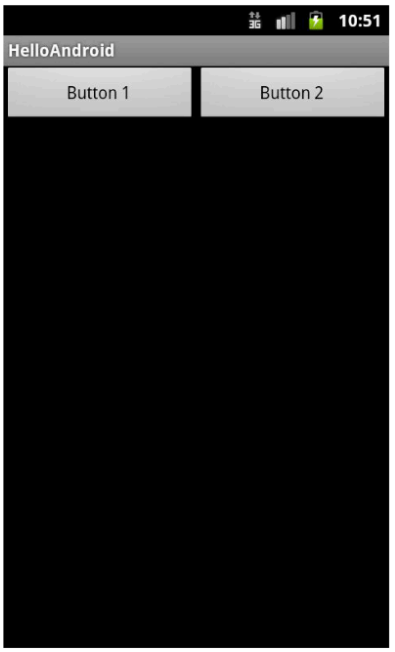
the most commons layouts are

▼ `LinearLayout`

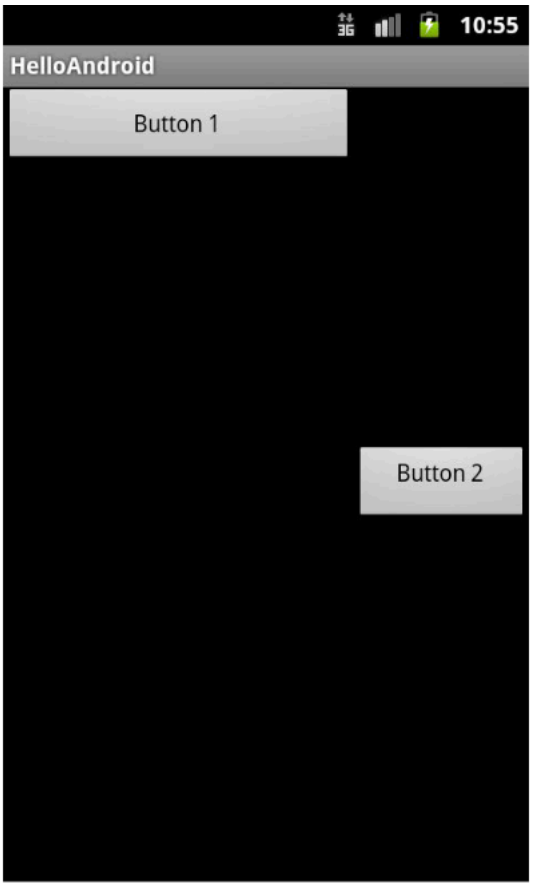


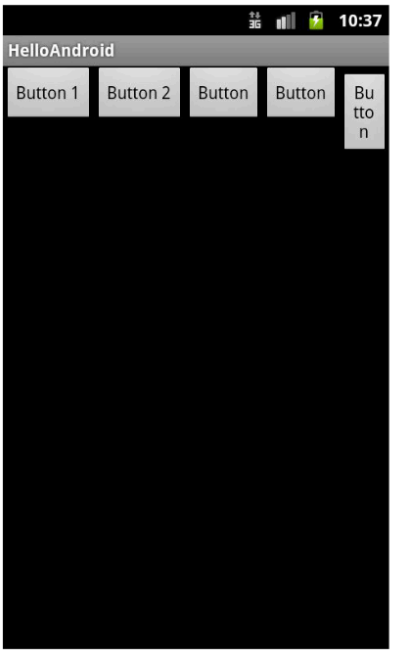
there's a `weight` assigned to every View, as well as `layout_width` that, when set at 0db, means "fill the available space)





`gravity` is another propriety





This happens with weights…

Without weights views can even disappear...

40

▼ RelativeLayout

Disposes views according to the container or according to other view

The gravity attribute indicates what views are more important to define the layout

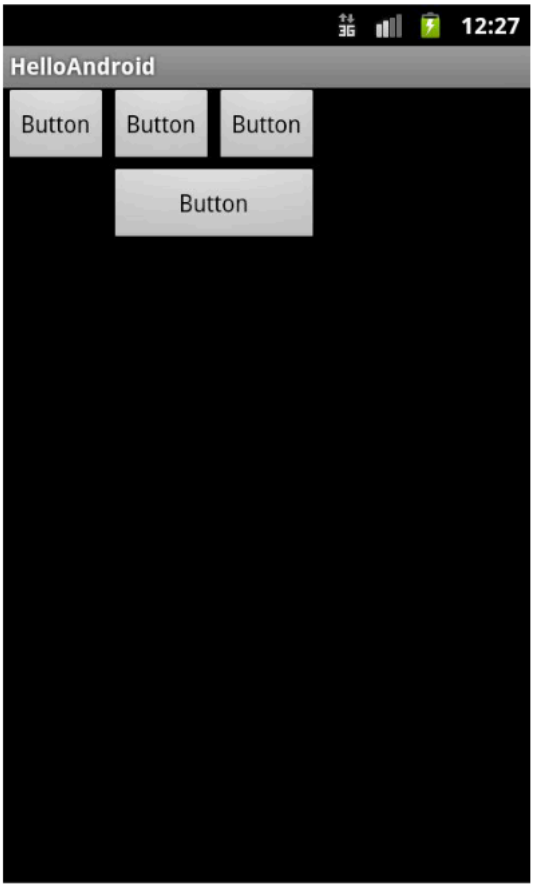
Useful to align views in “blocks”

▼ TableLayout

As the name say, similar to a Table, and has some attributes to customize the layout:

- `android:layout_column`
- `android:layout_span`
- `android:stretchColumns`
- `android:shrinkColumns`
- `android:collapseColumns`

Each row is inside a `<TableRow>` element



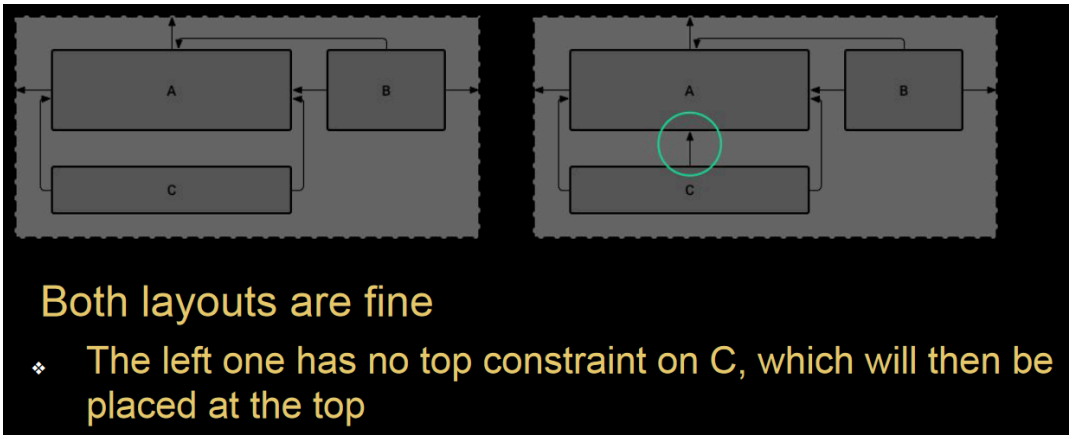
▼ `FrameLayout`

- Adds an attribute, `android:visibility`
- Blocks out portion of the screen to suit (typically) only one object.
- Size equal to the size of its largest (non GONE) child.

▼ `ConstraintLayout`

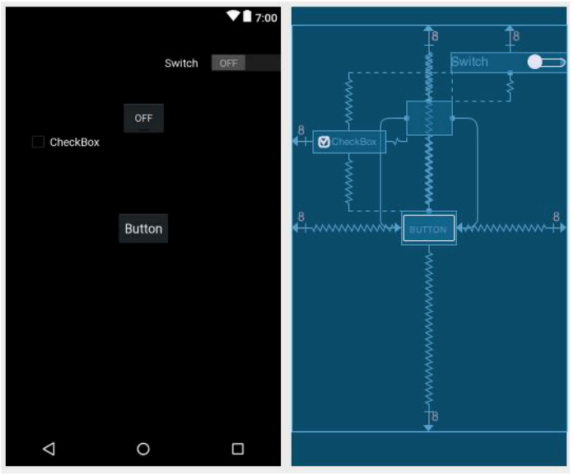
- Flat view hierarchy
- Similar to `RelativeLayout`
- > Android 2.3
- Overarching idea: define constraints

  (top/bottom/left/right) for each view
- Each constraint has to be defined to another (previously declared) view, another layout or an invisible guideline.



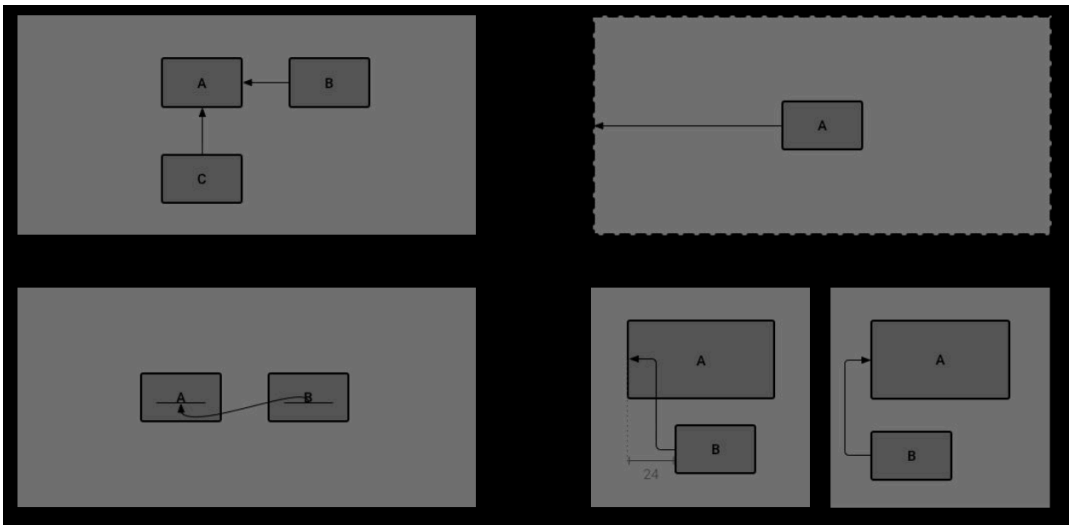
Both layouts are fine

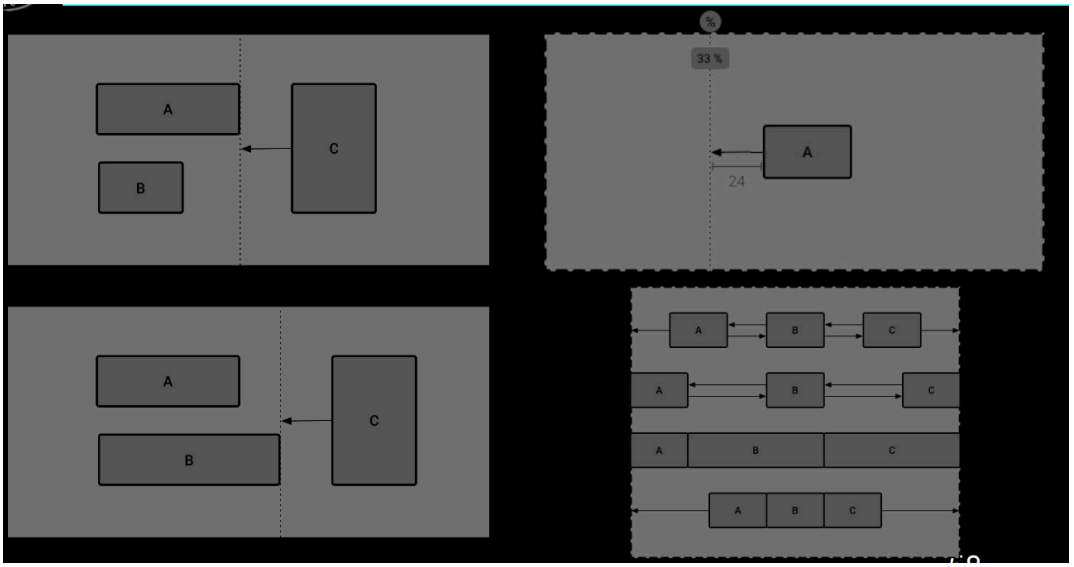
- The left one has no top constraint on C, which will then be placed at the top



- In the layout editor you'll see on the right the constraints, and on the left a preview
- *Layouts are drawn according to the available space*

- Each view needs at least one constraint per plane
  (plane = vertical | horizontal)
- Constraints can be defined only between anchor points sharing the same plane
- Each handle can define one constraint
- Multiple handles can define a constraint to a single anchor point
- Adding 2 opposite constraints places the view in the middle





we can declare Layouts within layouts

Fondamento android: se impari qualcosa, domani sarà deprecato.

[UNFINISHED]

iOS

▼ Introduzione ad iOS

▼ iOS architecture

▼ Core OS → si occupa delle funzioni base per il funzionamento del SO e della sicurezza

- OSX Kernel
- Power Management
- Mach 3.0
- Keychain Access
- BSD
- Certificates
- Sockets
- File System
- Security → 👮

- Core services → Collezioni, file access, SQLite, preferenze, etc etc, i servizi importanti per la base di tutte le applicazioni

▼ Media → Auto esplicativo

- Core Audio
- OpenAL
- Audio Mixing
- Audio Rec
- Video Playback
- JPEG,PNG,TIFF
- PDF
- Quartz (2D)
- Core Animation

- OpenGL ES

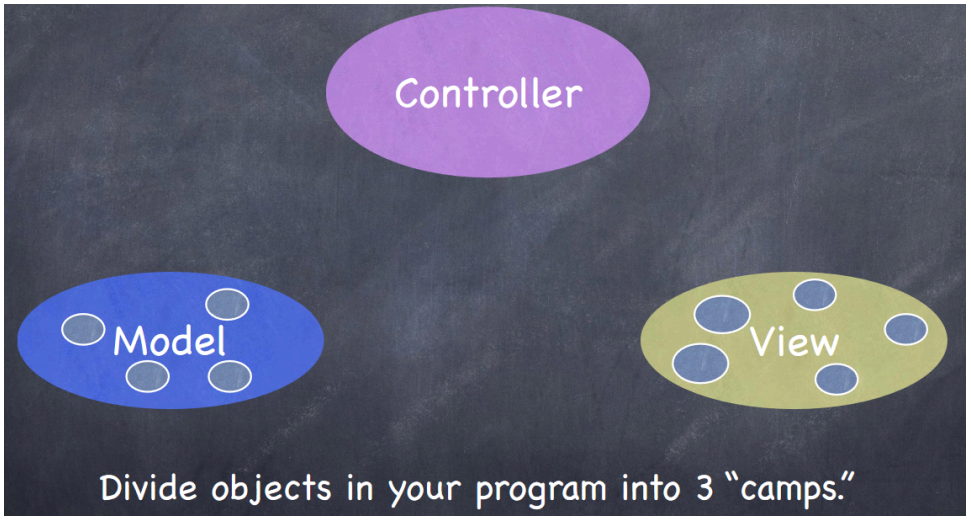
- Cocoa Touch → si occupa di tutto ciò che riguarda il touch e utility, gestione movimenti, multi-touch, web view etc.

Platform Components

- Tools → Xcode, instruments
- Languages → swift + something from Objective-C
- Frameworks → Foundation, corde data, UIKit + Cocoa Touch features
- Design Strategy → MVC

▼ MVC

MVC is an Object-oriented design pattern, and it works as follows

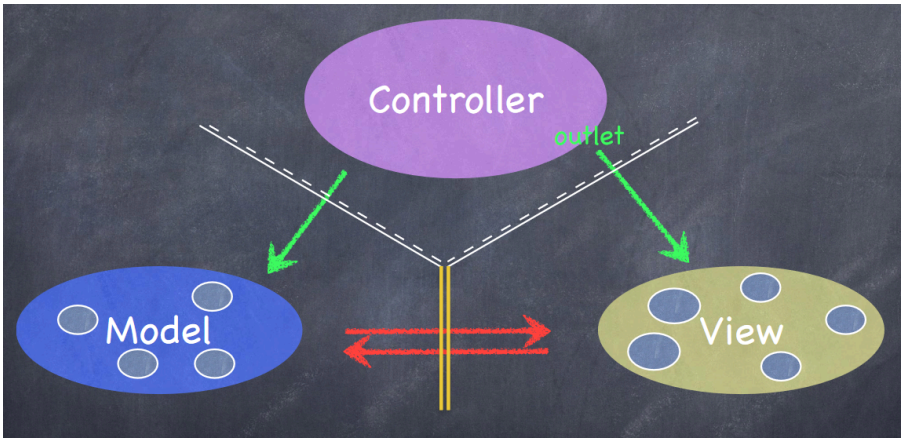


Model = What your application is (but not how it is displayed)

Controller = How your Model is presented to the user (UI logic)

View = Your Controller's minions (yes, the despicable me ones)

it's all about managing the communication between camps

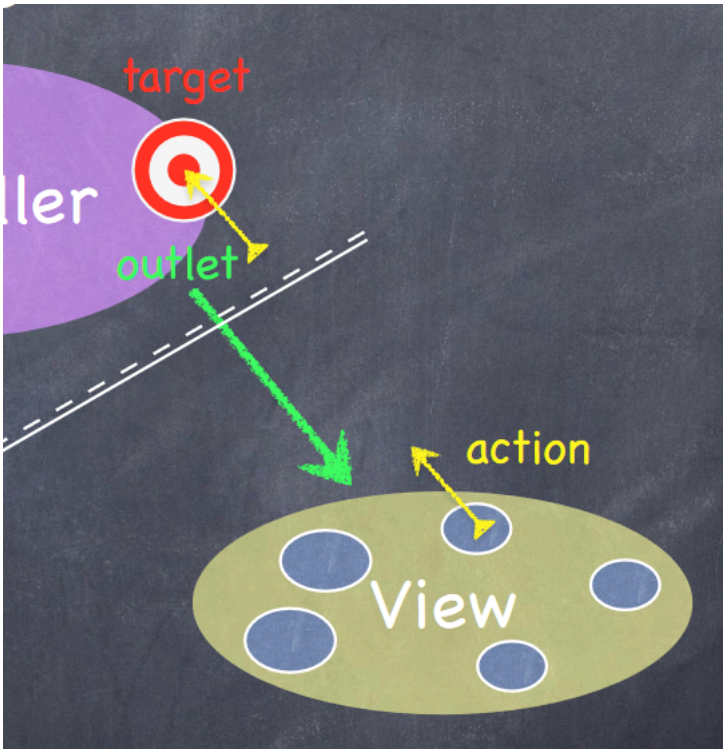


the controller should communicate with both the model and the view, while view and model should never communicate together

(the lines represent the actual lines on the road, so the controller can communicate with model and view, but the model or the view can't communicate with the Controller)

although, the View, being the UI, can interact with the controller, because that's how the user can input things.

the View sends the action when things happen in the UI

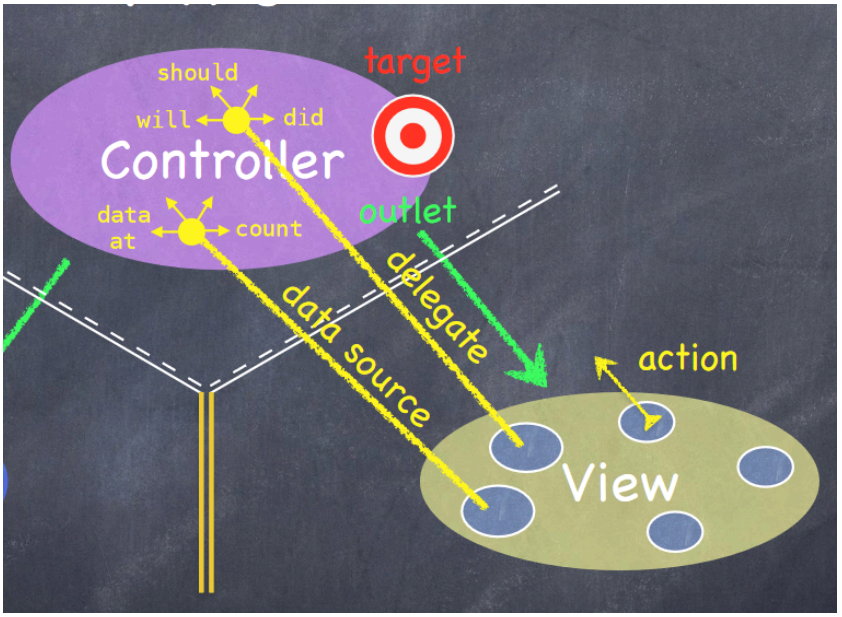


- Delegates (delegati):

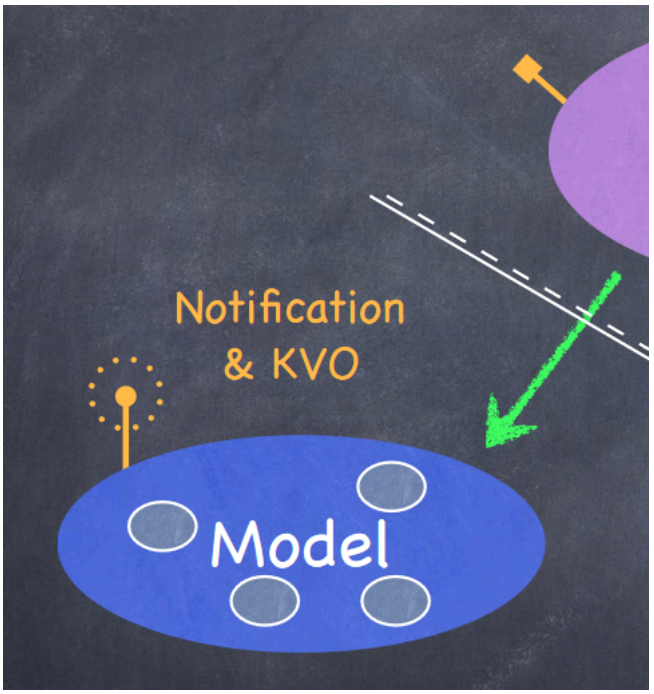
  sono metodi invocati dal sistema ogni volta che si verifica una determinata condizione. Questi metodi sono implementati dal controller tramite i protocols (protocolli) che sono una definizione di una collezione di metodi senza la classe associata (tipo le interfacce). i delegati iniziamo con parole del tipo "will, should, did".

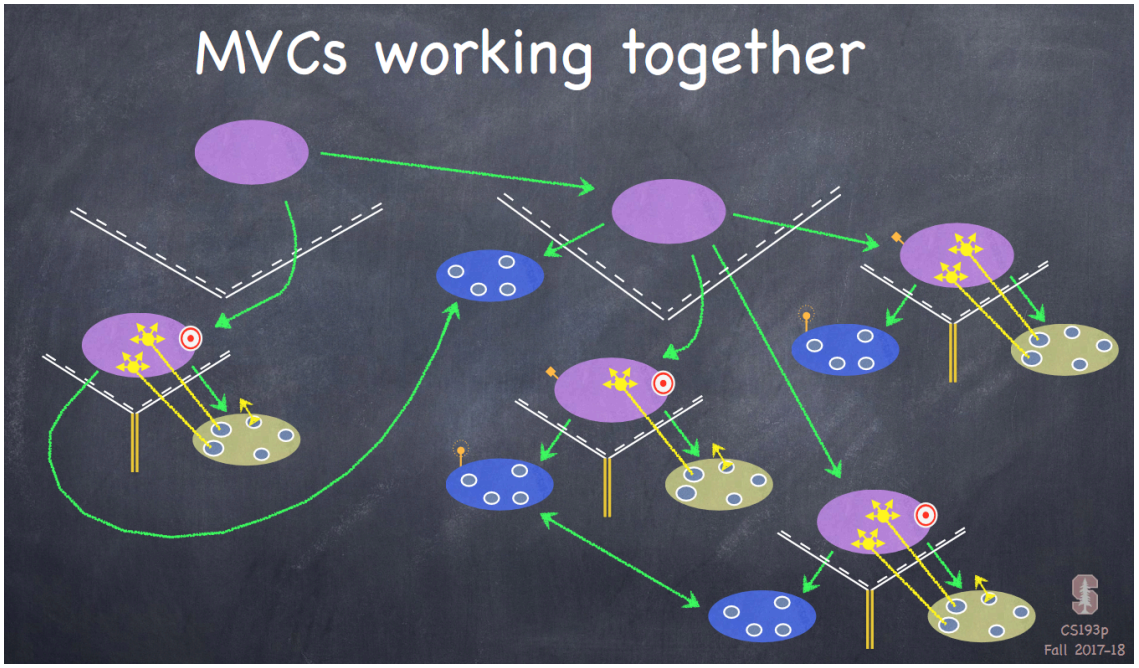
- Data source

  sono metodi che permettono la comunicazione tra view e modello passando dal controller.



se il modello ha informazioni da aggiornare o altro, usa un sistema broadcast, e il controller ascolta quello che gli interessa





buona pratica (e necessaria)

[UNFINISHED]

Hybrid

[NOT EVEN STARTED LOL]

Orale

Montori chiede sempre le solite 6 cosette, ed eventuali domande bonus sono correlate in base alla vostra risposta

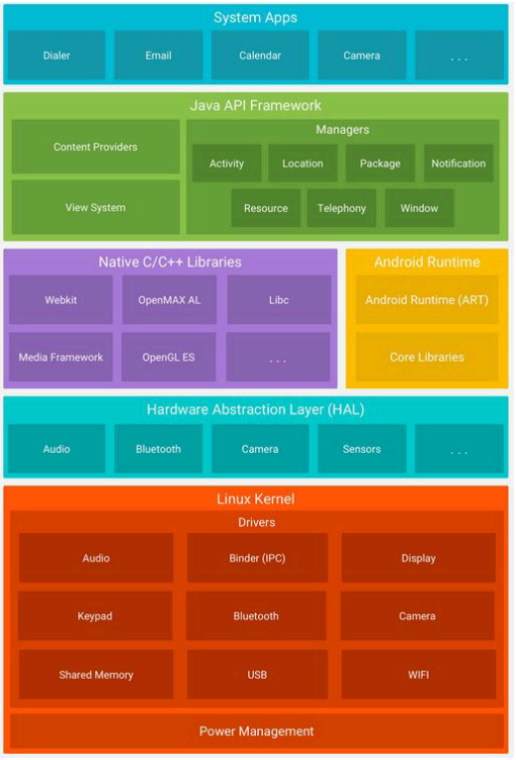
- Differenza tra Thread e Service/Processo
- MVVM vs MVC
- HAL (Hardware abstraction layer)
- Activity ed i suoi stati / lifecycle
- Intents, come sono strutturati
- Message passing nei Thread

Sunti Android (ITA)

▼ **Architettura di sistema**

Android è una piattaforma Linux-based per dispositivi touchscreen.

- utilizza un modello basato su permessi
- ogni processo è isolato (un utente Linux a sé)



HAL (Hardware Abstraction Layer) → interfaccia che gestisce diversi hardware

Librerie C/C++ → per gestione di oggetti, grafiche, etc

ART → VM Java con librerie core

API → tutto quello che richiamiamo nei file ed usiamo

Apps → auto esplicativo

Elementi di design

- GUI
- gestione Eventi
- gestione Dati
- operazioni background
- notifiche

Componenti d'applicazione

- Activities
  - singolo schermo di un'applicazione, esse possono comunicare tra di loro.
  - sono composti da dei Components chiamati Views (scritti in XML) che permettono interazioni con degli elementi (es. bottoni)
  - 5 stati
    - Starting
    - Running
    - Stopped
    - Destroyed
    - Paused
- Fragments
  - parte di un'applicazione, compone un'Activity e può fare circa tutto quello che un'activity può fare (solo come parte di essa obv)

- Intents
  - messaggi asincroni per attivare componenti android (es. Activities)
  - Intent esplicito → il component specifica la destinazione
  - Intent implicito → il component specifica il tipo di intent, che può essere soddisfatto da più scelte (es. apri un file video)
- Services
  - come le Activities, ma runnano in bg e non hanno UI
  - 3 stati
    - Starting
    - Running (on bg)
    - Destroyed
- Content Providers
  - interface standard per accedere e condividere dati tra applicazioni
- Broadcast Receivers
  - un'app può essere segnalata da eventi esterni, come cambiamenti nei sensori

▼ App Resources

tipi di risorse

- dichiarazioni di risorse
- integer, string, array
- color, dimension, style
- drawable, raw, xml
- risorse configuration-specific

Le risorse sono tutto ciò che non è codice.

sono dichiarate in XML e ogni risorsa ha un identificatore, accessibili tramite la classe java R

`[<package_name>.]R.<resource_type>.<resource_name>`

oppure tramite chiamata XML con

`@[<package_name>:]<resource_type>/<resource_name>`

un Drawable è un concept generale per una grafica disegnabile a schermo (immagini, vettori, icone etc)

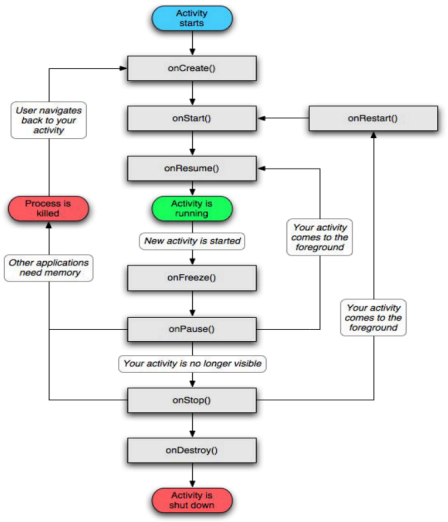
Android può rilevare e caricare solo alcune risorse se indicato (es. telefono inglese → carica solo stringhe indicate in lingua inglese)

▼ Activities & Fragments

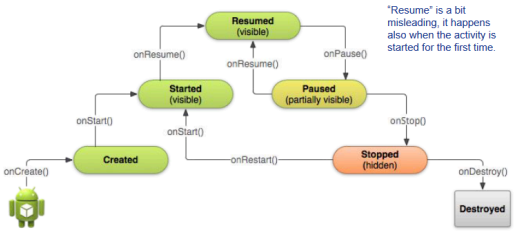
un'Activity è ciò che viene fatto partire dal dispositivo, contiene le informazioni dell'applicazione, reagisce a particolari eventi, e sono anche chiamate "screen state"

Vengono mantenute in uno stack

Lifecycle:



- onCreate → inizializzazione
- onStart → dopo OnCreate, chiama onResume o onStop, a seconda
- onResume → chiamata per mandare l'attività in running
- onPause → chiamata quando un'altra attività và in foreground o l'app và in bg
- onRestart → quando l'attività è stata precedentemente fermata (onStop)
- onStop → stessi criteri di onPause ma l'attività è completamente nascosta
- onDestroy → metodo finish() chiamato



Stati:

- Resumed → in foreground
- Paused → overlaid da un'altra activity
- Stopped → in background

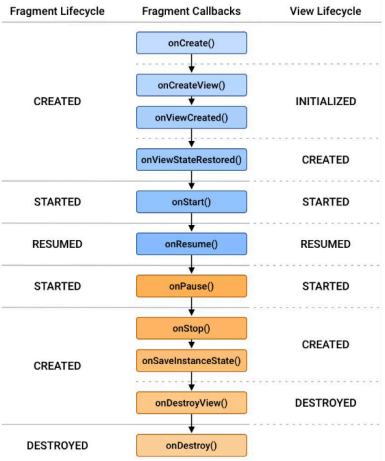
salviamo un Bundle chiamato "Instance State" per ricreare l'activity così com'era quando necessario

possiamo decidere di salvare dati aggiuntivi nel bundle con onSaveInstanceState() (chiamato prima di onStop())

Un fragment è una porzione dell'UI in un'attività, una selezione modulare dell'attività.

Esso ha il suo layout, lifecycle, e eventi input, può essere aggiunto / rimosso mentre l'attività è in RUNNING, deve sempre dipendere da un'activity.

Lifecycle (strettamente legato all'activity):



- onCreate → quando viene creato il fragment
- onCreateView → quando il fragment deve inizializzare la view, deve ritornarla, solitamente usando l'oggetto LayoutInflater
- onPause → quando il fragment viene lasciato

un fragment può prendere la reference all'attività padre con getActivity() e l'activity può prendere la reference di un fragment figlio

è possibile legare gli eventi del fragment all'activity con onAttach(context)

Transaction → cambiamenti all'activity, usati per navigare tra fragment (con il pulsante indietro ad es)

si aggiungono ad uno stack tramite addToBackStack()

le transizioni dell'attività sono tenute dal sistema, le transizioni dei fragment tenuti dall'attività.

▼ Intents & Permissions

un Intent è una chiamata da un componente ad un altro, con la possibilità di passare dati → un oggetto messaggio che contiene informazioni

"Android, fai *azione* con dati x"

2 tipi di intent:

- Espliciti → richiedono un componente nello specifico
- Impliciti → richiedono un componente capace di eseguire un'azione generica, a volte scelto dal sistema, altre dall'utente

Intent su attività possono tornare risultati (dati)

chiamano Services, Activities, BroadcastReceivers

Struttura:

- Component name → componente richiesto (opzionale)
- Action name → stringa che identifica l'azione da fare, negli intent impliciti Android cerca nei manifest delle altre app, puoi specificare cosa l'action debba fare (es, mandare qualcosa, visualizzare, ...)
- Data → URI + MIME type
- Category → stringa che specifica il tipo di component da chiamare
- Extra → parametri, chiavi, whatev

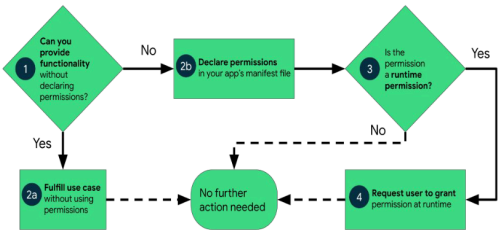
- Flags → extra info che specifica ad Android come far partire un'activity etc.

il processo di Intent Resolution risolve l'intent lanciato passando 3 test

- Action field → action deve corrispondere alle azioni listate dal filtro
- Category field → ogni categorie deve corrispondere ad una categoria del filtro
- Data field → URI comparato con parti dell'uri menzionate nel filtro

---

Permissions



Android 6.0+ ha introdotto permessi a runtime

Le slide sono completamente deprecate su questo, non vi serve sapere nulla se non che una volta chiesti i permessi si aspetta una risposta in modo asincrona e si reagisce con onActivityResult

:^)

▼ Views, Layouts & Events

Views → gli elementi da piazzare (pulsanti, widgets, etc), sono creati in XML o java

- può generare eventi, catturati da Listeners o Handlers
- hanno un focus e una visibilità, controllabili programmaticamente

Eventi → generati da azioni dell'utente su Views (click, long click, focus, drag etc)

Event handlers → sono i callbacks (`onTouchEvent()` , dichiarabili anche nell'XML)

Listeners → interfacce java che gestisce un singolo tipo di eventi (es. `CoordinateListener`)

- per usarle le dichiariamo come nested interface nell'Activity (`public class ExampleActivity extends Activity implements OnClickListener`) e la associamo nel codice
- i callbacks si possono invocare da codice (es. `performClick()`)

Layouts → piazzano gli elementi nello schermo

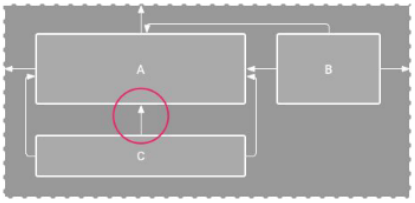
ViewGroup → oggetti container invisibili che definiscono la struttura layout di ciò che contengono

attributi negli oggetti View: `android:id="@+id/foo"`

- @ → richiama una risorsa tramite id
- + → nuovo id

tipi predefiniti:

- LinearLayout → singola riga / colonna
- RelativeLayout → uguale ma toglie delle view a seconda di alcuni attributi
- TableLayout → Tabella 🧑‍🎨
- FrameLayout → aggiunge visibilità
- ConstraintLayout → tipo relative ma con constraints



Esistono layout dinamici (ListView, GridView, CardView)

- AdapterView → stabilisce il suo contenuto tramite un adapter, che usa dati dinamici per generare contenuto (pensate alla lista della rubrica)
- RecyclerView → separa data e layout, gestisce meglio gli eventi, ha un adapter

View organizzate in gerarchie

alcune View:

- TextView → visualizza testo statico, può creare link
- EditText → visualizza testo cambiabile
- AutocompleteTextView → c'è un array di opzioni che ti appaiono mentre digiti per autocompletare il testo
- Button → superclasse di TextView, genera eventi
- CompoundButton → un checkbox praticamente
- Spinners → dropdown menu
- ImageView → ha dei metodi per essere manipolata
- CheckedTextView → testo con toggle

- Toast → piccoli messaggini sull'activity

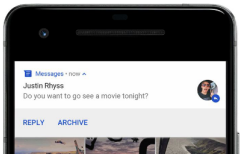
[Kotlin saltato]

▼ Operazioni Background

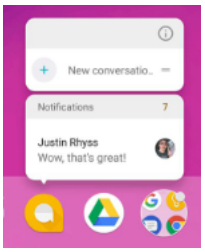
Notifiche → messaggi dall'applicazione, possono reagire al click con operazioni

Tipi:

- Heads up



- Icon Badge



Composta da:

- Icona
- Titolo
- Messaggio
- Pending Intent → azione da lanciare quando la notifica viene selezionata
- Canale (API 26+)

Opzioni:

- Ticket-text message
- Alert-sound
- Vibrate setting
- Flashing LED setting
- Customized layout

è possibile raggruppare notifiche e aggiornarle

⏰ Qualsiasi operazione che richiede più di pochi millisecondi dovrebbe essere eseguita in un thread di background.

Android supporta nativamente il multi-threading

Thread pool → collezione di threads in una queue

i Thread hanno meccanismi di message passing tra loro per comunicare

Looper → classe per gestire i messaggi nei thread

Service → componente che esegue operazioni a lungo termine in background, no UI, eseguita sul main thread, non viene distrutta quando l'app và in background

Service Lifetime:

- onCreate → quando il service è creato
- onStartCommand → `startService()`
- onDestroy → `stopService()` / `selfStop()`

Tipi di service:

- Intent Service → servizi semplici
- Foreground Service → servizio attivo nella status bar
- Bound Service → i service possono essere bonded a componenti con `bindService()`, bisogna creare un'interfaccia per interagirci (con IBinder o AIDL)

Broadcast Receiver → componente che viene attivato solo quando avviene un evento

l'evento è un intent

registriamo il Receiver all'evento (XML o Java) e gestiamo l'evento

- onReceive → unico stato del lifetime, al termine viene distrutto

si possono mandare intents, in modo ordinato o meno.

▼ Data Management

Diversi modi:

- Preferences → coppia Key - Value
- Text Files → leggi file direttamente
- XML Files
- SQLite Db → Tabelle

- Content provider → rendono i dati accessibili ad altre app

Sistema SharedPreferences → modo conveniente di definire parametri di configurazione, possono essere private o meno

editabili con SharedPreferences.Editor

editabili da utente con un PreferenceScreen (tipo le impostazioni android)

ci sono dati non condivisibili (`res/raw` , `res/xml` , App storage per supporto esterno)

`/data/data/<package>/files` → file dell'app

`res/raw` → file di testo, li puoi riempire ma non editare

XML → nulla di nuovo

File I/O → gestibile programmaticamente, accessibili con `getFilesDir()` o `getExternalFilesDir()` se esterne

è possibile condividere dati con un intent ACTION_SEND

Android usa SQLite per tenere tabelle di informazioni e fornisce librerie per gestirle, come un Cursore che mantiene le risposte delle query

---

Content Providers → simili a servizi web REST, danno l'accesso a path specificati per prendere dati (un db ad esempio)

▼ Operazioni Network

Ci servono i permessi NETWORK_STATE e INTERNET

le operazioni Network sono costose

dobbiamo gestirle, possiamo usare le librerie per i dettagli sulla connessione ed effettuare decisioni

WebView → un embed praticamente

DownloadManager → servizio che gestisce i download HTTP

esistono classi native per interfacciarsi con il protocollo HTTP

OKHttp → client per applicazioni java usato

Volley → libreria http molto customizzabile

Android può usare java.net.Socket per le comunicazioni TCP/UDP, con delle classi apposite come DatagramSocket e ServerSocket

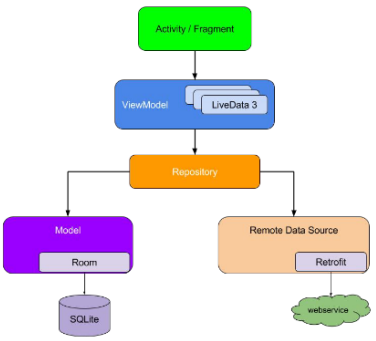
P2P → informazioni veloci tra dispositivi

Wifi direct → protocollo standard per il P2P

gestibile tramite il WifiP2pManager

▼ Linee guida per Components

Model View ViewModel (MVVM)



ViewModel → componente che mantiene dati UI in modo lifecycle-aware

Observables → classi di dati che notificano quando avvengono cambiamenti nei dati osservati

Implementiamo LifeCycle Awareness con un observer al LifeCycle

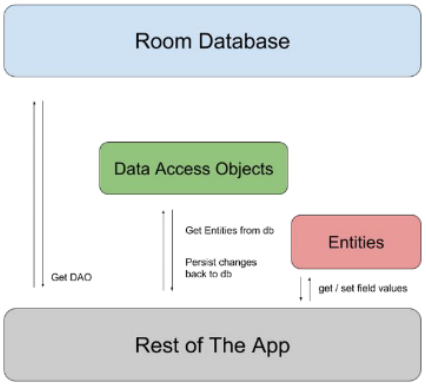
LiveData → Observables content-aware che notificano gli iscritti quando sono nello stato attivo (RESUMED o STARTED), i valori interni sono settabili

MVC → Model View Controller

uguale a MVVM, cambiano dove sono separati gli interessi, ma generalmente:

in MVC, il controller è la parte attiva invece che la View

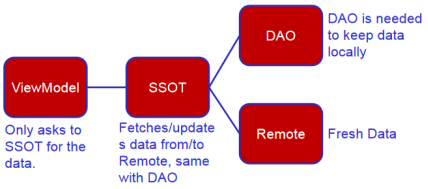
Room → layer di astrazione di SQLite



DAO → Data Access Objects, interfaccia con metodi di accesso al db, non accessibile senza

supporta la migrazione dei database e le observable query.

SSOT model → modello per richiedere dati ad una singola fonte



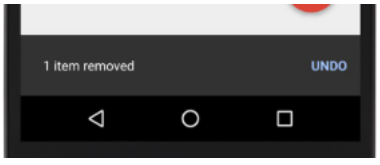
Retrofit → type-safe HTTP client per java, traduce XML e JSON in POJO (Plain-Old Java Objects), è molto simile a room come funzionamento

Firebase → piattaforma di sviluppo Google per sviluppare la parte backend della nostra app

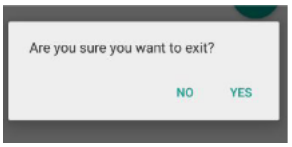
▼ Linee guida per navigazione UI

Menu → appare quando l'utente preme il menu button, creabili da Java e XML (res/menu)

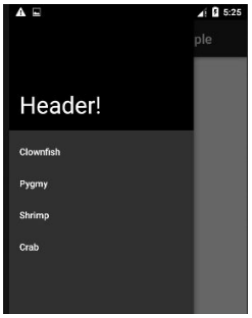
Snackbar → un toast attaccato ad una view, ha la possibilità di ascoltare eventi e contenere azioni (tipo le notifiche)



Dialoghi → interazioni dirette con l'utente, piccoli messaggi, possono avere layout custom



NavigationDrawer → Nav bar, swipe per farla apparire, può ascoltare per eventi



Toolbar → come NavBar ma si vede che esiste grazie all'hamburger (le 3 linee)

ShareIntent → intent semplice per condividere dati

Navigation → framework per gestire la navigazione, tramite questo è possibile definire interfacce con un'interfaccia grafica (in android Studio)

Principi di design materiale:

- Materiale è la metafora

  razionalizza lo spazio e il movimento come se fossero nella vita vera

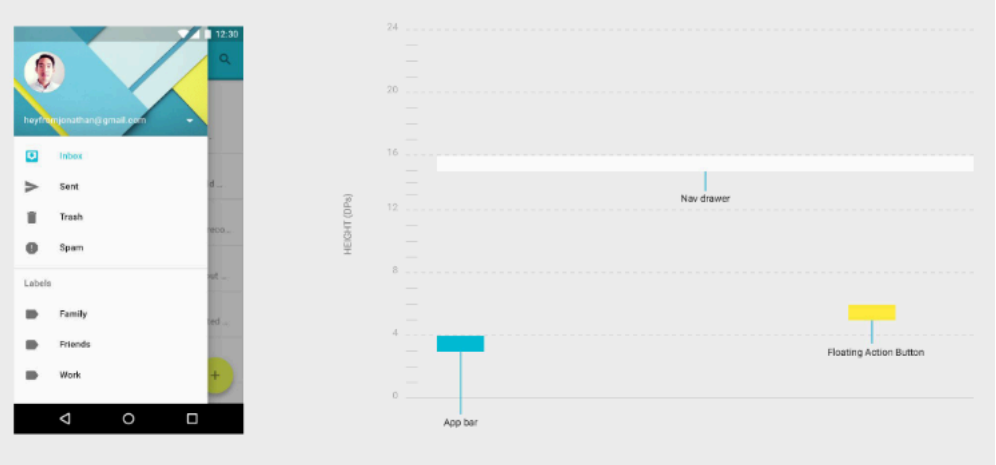
- Bold, grafico, intenzionale

  crea una gerarchia, sfrutta il colore per funzionalità

- Il movimento provvede significato

  i movimenti dell'utente devono corrispondere a del feedback

MaterialDesign → ambiente 3d di sviluppo, mette a disposizione layer e funzioni



▼ Geolocalizzazione e servizi Google maps

Geolocalizzazione → identificazione della locazione geografica del mondo vero di un utente finale.

reso possibile da hw radio transceivers + sw localization algorithms

Localizzazione attraverso GPS, WI-FI, Network cellulare

Context aware computing → possibilità del sistema di fare computazioni dipendenti da contesto

tipi di contesto:

- Primari → raw data, sensori, gps, tempo
- Secondari → stagione (calcolata), identificare un volto

If This Then That (IFTTT) → esempio di Context aware computing, definisce regole per eseguire azioni in base a stati precedenti

è possibile ricavare contesto da qualsiasi cosa

GPS → global positioning system, basato su satelliti

i satelliti mandano posizione e current time

i GPS receiver:

1. ricevono data passivamente
2. computano il delay
3. computano la distanza = delay * c
4. dalla distanza di 3 satelliti (minimo), determina posizione

ogni satellite trasmette su 2 frequenze su banda UHF

- L1
- civilian data

posizione tramite WiFi interface → tramite MAC e SSID dei router nei dintorni, query a Google che si tiene un db di posizioni conosciute

posizione tramite Network → riconoscimento della torre cellulare a cui siamo attaccati, un po' come il metodo sopra e contesto (timing, angolo di arrivo, etc)

Android usa tutti e 3 con il Location manager

Location Listener → oggetto con metodi callback per gestire i cambiamenti di provider/stato/location

FusedLocationProvider → servizio google per semplificare lo sviluppo

[la parte che parla di google maps l'ho saltata sinceramente, molto pratica ma di teoria ci sono cose ultra specifiche che non credo valga la pena di trattare, in più sono standard di molte mappe API, robe come marker, tiles, customizzazioni varie]

Google Map → altamente customizzabile, ci sono diversi tipi di mappa e proprietà

LatLng → classe per definire punti sulla mappa

GeoCoding → google map library technique, converte un indirizzo in coordinate

Georeferencing →tecnica che crea dei boundaries per capire quando un utente esce da una certa area

▼ System Services

lista infinita di servizi, tipo BatteryManager, LayoutInflater etc etc

quelli trattati dal prof sono

Battery Manager → ricevere callbacks e info sulla batteria

Alarm Service → per lanciare intent nel futuro

Work Manager → API che semplifica il lanciare task asincrone e deferenziabili

Periodic Work → per schedulare lavori periodici con un WorkManager

Montor & Chain → monitora cambiamenti tramite LiveData

Sensor Service → interazioni con i sensori (acceller. , giroscopio, bussola, etc)

Audio Service → controllare il suono, aggiustare il volume, e task legate all'audio

Telephony Service → interagisce con le chiamate, ritorna informazioni sullo stato

SMS Service → manda messaggi

Connectivity Service → controlla lo stato del network, wifi, gprs, lte

Wi-Fi Service → gestisce la connessione wifi

Sunti IOS (ITA)

▼ Paradigma MVC

IOS:

Core OS → ispirato a BSDX, componente per il power management e diverse componenti di sicurezza

Core Services → astrazione successiva tramite strutture dati, contiene preferences

Media → integrazione efficace dei vari media nei dispositivi mobili

- gestisce ambienti smart
- gestisce realtà virtuale ed aumentata
- gestisce grafica 2d e 3d

Cocoa Touch → librerie e processi che gestiscono l'interazione con l'utente, è composta da:

- gerarchia delle viste
- localizzazione linguistica
- web view
- controllo periferiche

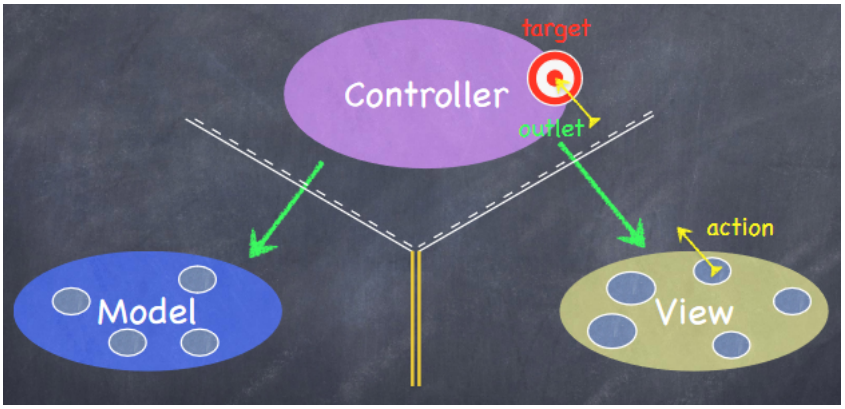
sviluppo seguendo il modello MVC o MVM.

▼ Model View Controller

Model → area che contiene i dati che rappresentano lo stato dell'esecuzione dell'applicazione (variabili, strutture dati, oggetti, ecc...), ma non informazioni su come queste cose sono mostrate a schermo.

V**iew** → elementi che mostrano all'utente qualcosa sia a livello grafico che in formato audio o altro, che cosa devono mostrare è invece gestito dal controller.

Controller → luogo dove si trova la logica del programma e le politiche di visualizzazione dei dati (UI logic).



Outlet → puntatore alla view, controlla la comunicazione tra Controller e View

Target → metodi definiti dal controller per permettere alla view di mandare azioni al controller.

Delegates → metodi invocati dal sistema, implementati dal controller tramite i protocols (collezione di metodi senza classe associata).

Data source → metodi che mettono in comunicazione view e modello tramite il controller

▼ Swift

Range → for i in stride(from: 0.5, through: 15.25, by: 0.3), equivalente di un for che usa floating point numbers

Tupla → gruppo di valori utilizzabile come tipo

Computed propriety → variabile con un get o set sovrascritti, quindi altamente personalizzabili

Access Control → protezione dell'accesso ai campi delle classi

Extensions → si possono estendere strutture aggiungendo metodi e campi, non si possono sovrascrivere metodi, e i campi non hanno un'area di memoria associata, quindi non sono utilizzabili (che merda tbh)

Enum → tipo che può assumere degli stati specificati al suo interno, si possono associare attributi e possono avere metodi e campi (sempre come computed properties)

Optionals → tipo, che può avere il tipo non definito (`let i`)

hanno una sintassi brutta e lunga che non specifico

▼ Swift Data structures

Classi → Swift è OO. ogni oggetto di una classe viene passato per puntatore

Struct → contiene più campi

Protocolli → collezione di diciture di metodi, tipo le interfacce java

Stringhe → array di caratteri unicode, l'indice è un oggetto di tipo index che gestisce eccezioni particolari

NSAttributedString → stringhe con attributi aggiuntivi (font, colore)

Tipi funzione → le funzioni sono implicitamente associate ad un tipo

Casting → esiste

▼ Views

una view è un'area rettangolare disegnabile.

le view sono organizzate in gerarchia partendo dalla UIView, che è l'area più grande possibile e che comprende anche la barra di stato.

sono basate su coordinate, hanno tipi speciali CGFloat, CGPoint e CGSize per essere gestiti.

Xcode ha UIView generiche che possono essere usate per costruire le View (tipo android) tramite lo storyboard

CALayer → meccanismo di disegno per Views

le views possono avere trasparenza, fonts, immagini, etc.

Sunti Hybrid (ITA)

▼ Ionic & Angular

App ibride → una mobile application ibrida è sviluppata usando librerie native e tecnologie web per arrivare ad avere il meglio di entrambi i mondi.

→ la loro interfaccia è in embedded HTML

→ la parte nativa comunica con il web-based server backend

→ ci sono dev tool 3rd party che usano containers che arrivano quasi a una performance nativa

PWAs → progressive web applications, usano gli stessi tool per lo sviluppo di siti mobile, HTML5, e librerie TP di JS come Dojo o jQuery.

Native, Ibride e PWA hanno ogniuna dei pro e cons

Native → miglior performance, peggior dev mantainance costs

Hybrid → alta performance, alti costi, più bilanciato

PWA → leggera, limitata e con i costi più bassi

Angular 2+ → app design framework per single-page apps

→ JS framework

→ codice Typescript compilato in javascript

→ estende HTML

→ architettura component based ispirata a MVC/MVVC

Ionic → toolkit open source per mobile UI, per creare sia web apps che native, usa HTML, CSS e JS

Capacitor → open source native runtime per costruire app web native, usa HTML, CSS e JS

▼ React Native

React Native → framework per costruire il frontend di app native

ReactJS → JS framework per costruire interfacce web

JSX → estensione XML /HTML-like a Javascript

Virtual DOM → concetto programmatico dove una rappresentazione ideale dell'ui è tenuta in memoria separata dal vero DOM, ma sincronizzata.

JavaScriptCore → JS engine che usa safari, usato dall'ambiente react native

Fast Refresh → feature di React Native che permette di avere feedback quasi istantaneo dall'app quando i componenti cambiano

Expo CLI → tool per costruire intorno React Native

Progetto/Esame

Protip assoluto:

se potete lavorare in più persone al progetto, fatelo.

i requisiti di integrazione per due persone sono solo una piccola feature che si sviluppa con poco e per il lavoro dimezzato del progetto ne vale assolutamente la pena di lavorare in due.

fare un progetto così complesso per chi è alle prime armi è un suicidio, almeno avrete da piangere in due.

---

relazione app mobili.pdf

GitHub - mettil13/transceiver-go: Android mobile app for collecting data about network connectivity, wifi signal and noise level

Android mobile app for collecting data about network connectivity, wifi signal and noise level - GitHub - mettil13/transceiver-go: Android mobile app for collecting data about network connectivity,...

https://github.com/mettil13/transceiver-go

Memotti

Contesto: 4~ mesi di lavoro in due persone, e normalmente il progetto è per una sola persona.