Politecnico di Torino

Mobile Application Development

Slides recap from the course of Prof. Giovanni Malnati

A.A. 2021/22

Mobile Application Development

Introduction to Android

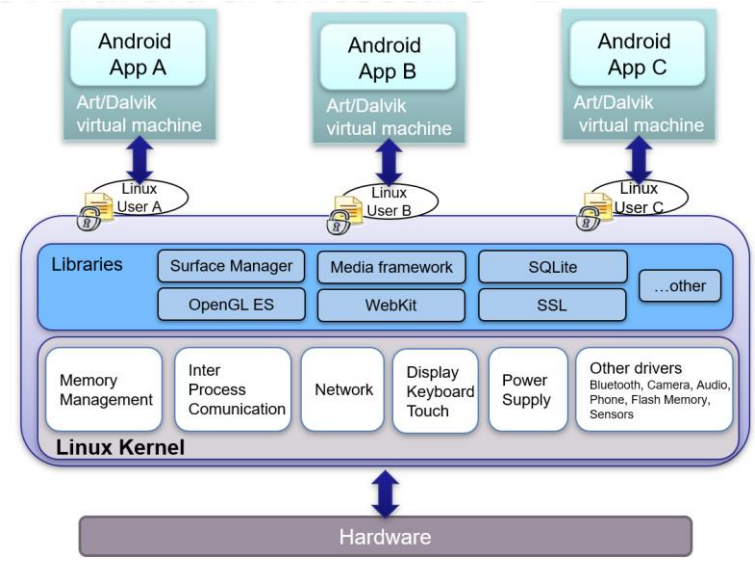
Android is an **open source** and **comprehensive** platform for mobile handsets. It includes the best features of previous mobile platforms. Development Tools are free and Android applications are (mostly) written in the **Kotlin** or **Java** programming languages. Alternatively, a C++ API is available.

There is **no distinction** between native and third-party applications: all the applications use the same Software Development Kit (SDK) and all the applications can access the underlying hardware.

The Android architecture

Android is composed of an **operating system** and a **software platform** for creating apps and games. Android includes a set of minimal applications (browser, email client). These basic features can be easily included in other applications. Android has been designed to be **robust**: it is based on the *Linux Operating System Kernel.* Every Android application runs in its own process, with its own instance of the virtual machine.

Java VM is not open source (while Java code is) so Android used Java code but compile in dex(Dalvik) not Java VM. Now it uses ART: Android Runtime. Android applications are node of components that obey to a strict lifecycle. Each installed application creates a user since Linux supports multi-users. Applications can R\W only in its own parts.



Processes are liquid here: Android continuously monitors the existing processes and resources consumption. The OS can kill at any time any process and gives a signal of it, so that processes can save their state. When the user will need it, it will be loaded again.

The **Linux Kernel** acts as a **hardware abstraction layer** (*HAL*) between the device physical layer and the Android software stack. The Linux kernel manages permissions and security, low level memory, processes and threads, network layer, display, keyboard, camera, flash memory, audio files.

Every Android application runs in its own process, with its own instance of the ART (Dalvik) virtual machine. This is conceptually like Java virtual machines but optimized to run on mobile devices as we previously said.

Security

Every application **runs with its own user**. Once the application is installed, the operating system creates a new user profile associated with it. File system permissions ensure that **one user cannot alter or read another user's files.** Every application must declare which shared resources will use, for example, making phone calls, using the camera or other sensors. Android will block applications which try to use not declared resources. Every application also requires the permission to access the user's private data such as *preferences*, *user location*, *user contacts*, … If the permission is not granted, the installation fails.

Android Application Development

The Android SDK exposes a set of APIs, which allows the access to the underlying hardware. There is no distinction between "native applications" and "third-party applications". Every application, if equipped with the appropriate permissions, can use them.

Android includes a set of minimal applications such as a browser, and an email client. Third-party provided applications can integrate, extend or even replace them.

Application Structure

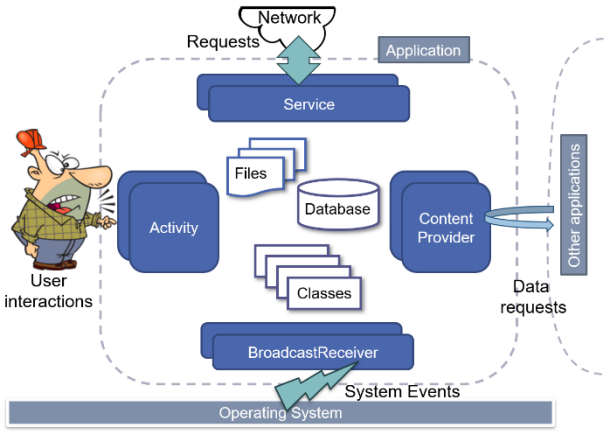
Conceptually, an application consists of a set of data and code designed to perform a given set of tasks. Such information is **stored in a private space** inside the file system (each app has a private folder) and executed in the user context that was created during the installation of the application.

Android applications do not have a single entry point, as it happens in other operating systems. Each application consists of one or more components, activated by the operating system, at its own will. The components of the same application are distributed in a single software "*package*" (*apk* - Android Package) and are described in a manifest file.

Android Components

Each application consists of **one** or **more** of the following components: Activity, Service, ContentProvider, BroadcastReceiver. Each kind of component takes care of a **specific interaction** with the operating system and/or the user. Component creation, operation, and destruction follow a well-defined, OS enforced, life-cycle, according to its kind.

Activity supports the user interaction. A service provide/execute long lasting operations. The content provider gives the data of the app visible to other apps. The broadcast receiver reacts to system events coming from the OS.



Activity

An **activity** is a software component that has a *Graphical User Interface*, *can perform a task inside the application*, *extends the class android.app.Activity*.

An application is composed of one or more activities. For example, an email client application could be composed of the following Activities:

- One listing unread inbox messages
- One for composing a new message
- One for reading a message

Service

A service is a component that can run in the background. It does not provide a user interface. Usually, services are used to **perform long tasks**. A service could play music while the user is using another application or service could gather network data without blocking the user interaction with another activity. A service is a *subclass* of *android.app.Service*.

Content Provider

A content provider manages application data. Data can be stored in the file system, in a database, on the web, … It implements a set of standard methods that allow other applications to fetch and to store data handled by the current application. Other applications do not call its method directly, but they interact via a **content resolver**. A Content Provider is a subclass of *android.content.ContentProvider*.

Broadcast Receiver

A Broadcast Receiver is a component which "**waits**" for messages. Some messages are created by the Operating System, for example, whenever the display is turned off, when the battery is low. Applications can produce messages too, for example, when a data transfer is completed.

A broadcast receiver does not have a Graphical User Interface, but it can generate notifications in the status bar to notify the user that a particular message is detected. A Broadcast Receiver is a subclass of *android.content.BroadcastReceiver*.

Application Lifecycle

The functionality provided by an application are defined by its **manifest file.** It is an XML document that "signs a sort of contract" between the application and the execution environment. It lists all the single components that compose the application, the requested permissions, and their configurations information. When an external event occurs, based on its type and on the components declared in the manifest file, Android creates a new process. Its owner is the one that was created when the application was installed. For each application in execution, Android instantiates in its process a **single object** of class *android.app.Application*. It is possible to specify a subclass of it in the manifest file. This object can be used to store global information shared by all the app components.

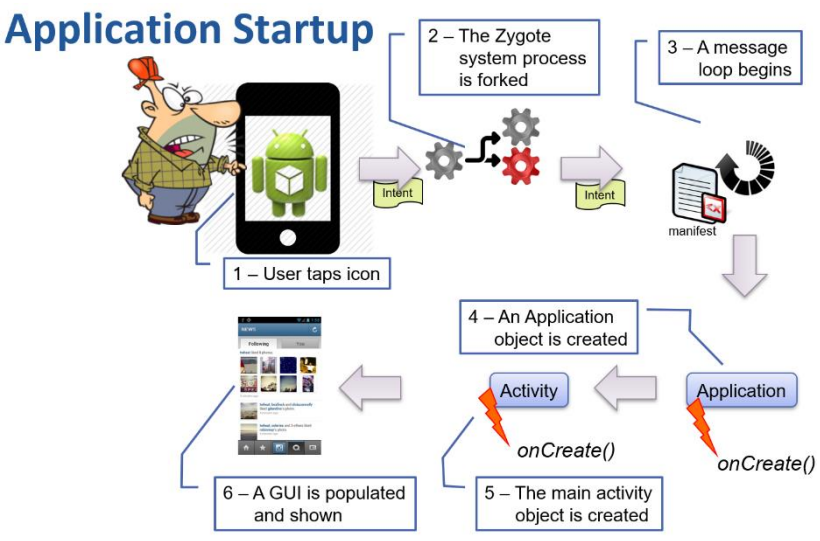
Android notifies the application object with the evolving status of the ongoing elaboration:

- *void onCreate()*
- *void onConfigurationChanged(…) (e.g., landscape/portrait)*
- *void onLowMemory()*
- *void onTerminate()*

The application object is the first one to be created and the last one to be destroyed. Once the application object has been created and initially notified of the beginning of the process, Android instantiates whatever individual component is needed based on the reason that caused the process to be started. The application object **stays in memory as long as there are active components**. The application object is removed from the memory when all the components end their lifecycle. It is possible to set up an user-defined Application subclass which plays the Model role in a Model-View-Controller pattern and can be used to store and manage application data.

Application process startup

Whenever an icon of the home screen is tapped, a new process is launched. Information regarding the chosen activity is packaged inside an **Intent**. The intent is delivered to a system process, named Zygote that forks a brand-new process. Zygote contains a pre-warmed VM: this reduces the overall start-up time. Only application specific classes need to be loaded, since general purpose ones have already been loaded in the parent process. The forked process starts a message loop and uses the information contained in the Intent to locate and load the corresponding APK and manifest file.



The main thread of the forked process instantiates an Application object and invokes the *onCreate()* method. Subsequently, the activity referred to by the received intent is instantiated and its *onCreate(…)* method is invoked.

Applications interaction

The modular mechanism in which applications are built enable the creation of synergies without building strong interdependency among classes. A particular kind of messages, named **Intents**, may be used to activate other applications and to exchange data with them. An intent defines an action to be performed and a set of data on which to operate. The operating system finds and instantiates the corresponding components that can handle the required action.

Intents

An intent is an **abstract description of an operation** to be performed which is late bound to components in order to activate them. It consists of several parts, the most important of which are:

- The **action**, a unique string describing what is requested or what has happened
- The **data** to operate upon, typically expressed as a URI
- The **category**, one or more strings containing additional information about the kind of component that should handle the intent

Intent filters and the manifest file

All the components exported by an application are listed in its manifest file together with zero or more intent-filters. Each filter describes a capability of the component, a set of intents that the component is willing to receive listing fields corresponding to the action, data, and category fields of an Intent object. When an intent is delivered, Android tries to match it against all filters, to detect which component should be activated. Filters are also used to learn something about the component itself: the launcher is populated with all activities that have filters reporting action MAIN and category LAUNCHER.

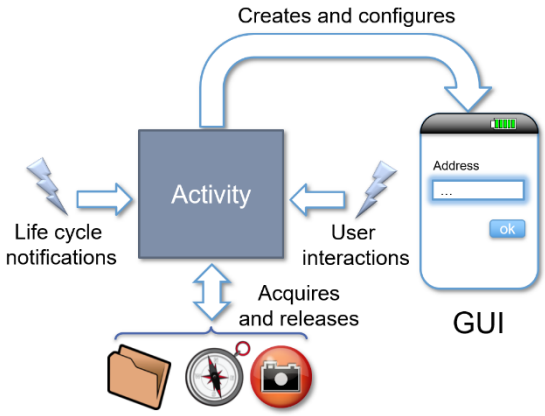
Activities

android.app.Activity is the base class that provides a GUI with which users can interact in order to do something. Applications extend this class to implement specific functionalities. The name of the derived class is specified in the manifest-file. The operating system creates the activity and manages its life cycle invoking some specific methods: **onCreate(…), onStart(), onResume(), onPause(), onStop(), onDestroy().**

Role and responsibilities

An activity manages user interaction:

- Acquires the required **resources** (requires the use of sensors, loads contents, …) to provide functionalities
- Builds and configures a **graphical user interface**
- Reacts to the **events** triggered by user interactions with widgets to implement the desired behavior
- Manages the notifications regarding its own **life cycle**, storing data collected by the interface and releasing resources when the operating system asks so.



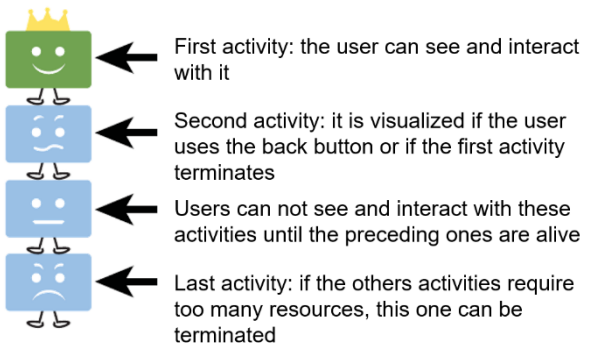
An activity typically shows a single user interface that allows only some types of user interaction. For this reason, an application contains several activities. Each one typically supports a single task (e.g., create a new

message, read a received message, manage the received messages, …). One of them is marked in the manifest file to be the first to be shown to the user, when the application is started by the launcher.

An **activity may start another activity**, by creating an **Intent** object. The called activity may be either part of the same application or may belong to a different one.

Activity stack

For each task started in the "home" screen of the device, the operating system builds an **activity stack** initialized with the default activity of the corresponding task. During its life cycle, an activity may request the system to start a new activity. The latter will be **created** and **inserted** at the **top of the stack** and will become visible and interactive. The previous activity is shifted one position and will remain in background until the new one is alive. If the new activity explicitly terminates or if the user presses the Back button, the previous activity reaches the top of the stack and becomes visible and interactive again.



Activity life cycle

The Android operating system allows the execution of several tasks at the same time provided that there is enough memory space and energy charge. Each task consists of one or more components (activities, services, …) and of all the related classes and resources. An activity can be interrupted and paused when given events are triggered (e.g., if a phone call is received).

Android sends different notifications to track the status of an application and its evolution according to user interaction, system-level events, and resource availability. The programmer has to react to these notifications, performing the actions that are needed to guarantee the correct management of all the application resources by releasing and re-acquiring them whenever it is necessary.

In addition to obvious events, the management of the life cycle also includes less obvious situations, often source of confusion. When you rotate a device, the **current activity is destroyed and recreated with new parameters.** The same happens if you set a different language on the control panel.

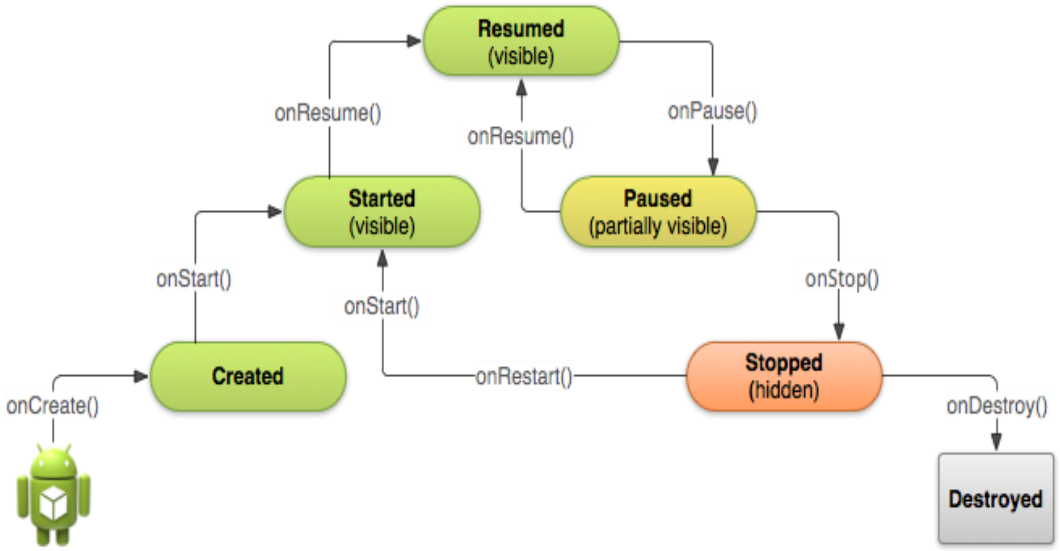
Callbacks and when they are called

- **onCreate(…)** - static initialization
- **onStart()** - when Activity (screen) is becoming visible
- **onRestart()** - called if Activity was stopped
- **onResume()** - start to interact with user
- **onPause()** - about to resume previous Activity or to start next one
- **onStop()** - no longer visible, but still exists and all state info preserved
- **onDestroy()** - final call before Android system destroys Activity

Activity life cycle and visibility

The operating system notifications indicate the different phases of the application:

- **onCreate(…) / onDestroy()** – The activity exists but is **not visible** to the user
- **onStart() / onStop()** – The activity is **visible**, but the **user cannot interact** directly with it (another partially obscuring window is on top of it)
- **onResume() / onPause()** – The activity is in the **foreground** and the user can interact with it



Managing the activity life cycle

If overridden, the methods related to the life cycle notifications should call the implementation of the **superclass**. Failing that, the system will throw an exception and exit the application.

```
class MainActivity : AppCompatActivity() {

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // other actions
  }

}
```

onCreate()

The method **onCreate(b: Bundle?)** is called in two different scenarios:

- When the activity is **run for the first time** (the Bundle parameter is null)
- When the activity is **launched again after being terminated** (due to lack of resources or for other reasons). In this case the Bundle parameter contains status information

In this method, it is appropriate to perform most resource allocation and data initializations such as those related to the management of the active screen and the data link.

onStart()

Method called when the application becomes visible to the user. If the activity has been previously visible, this method is preceded by the call to the **onRestart()** one. In most common cases, it is not necessary to redefine this method.

onResume()

This method is called when the activity **reaches the top of the activity stack** and becomes interactive. In this method, it is possible to start animations, videos, sounds, ... and to acquire temporary resources (like sensors and cameras).

onPause()

This method is called when the **activity has to be moved to the second position** of the activity stack. Here, sounds, videos and animations are usually stopped. In this method it is appropriate to release all resources that are not needed while the application is in the background:

- If the application handles persistent information (databases, files, preferences, ...), **here** data is usually **committed**.
- If the activity has registered any listener of system events, sensors, services, ..., **here** they are **unregistered**.

Warning: the next activity will not start until the onPause() method is running. Thus, it is necessary to **limit the duration** of the operations carried out here.

onStop()

Called when the application is no longer visible to the user. The next possible notification can be **onRestart()**, **onDestroy()**, or nothing at all depending on user interactions.

onDestroy()

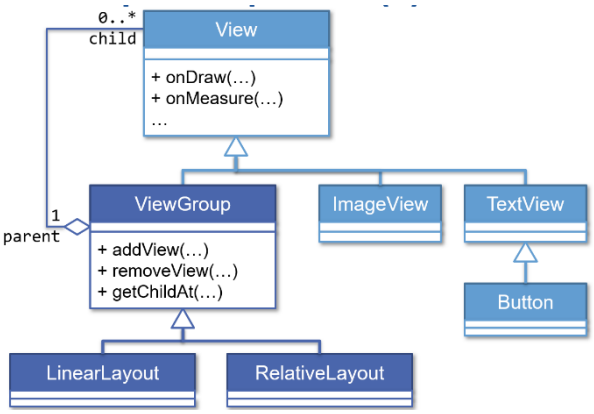
The activity has been terminated and will be removed from memory. The invocation may be the result of an explicit request of the application by having invoked the finish(...) method or of the need to free resources. The *isFinishing()* method allows us to distinguish between the two situations.

Preparing the GUI

In the *onCreate(...)* method you have to prepare a view (class that extends *android.view.View*) and make it visible via the *setContentView(...)* method. It is in charge of presenting content to the user and allow interaction. A view is usually made of a **set of elementary widgets** connected together to form a **visual tree.** The hierarchical structure of the tree reflects the usage of space inside the display. If an item is a “child of” another, it is completely contained in the space of the other.

The “composite” pattern

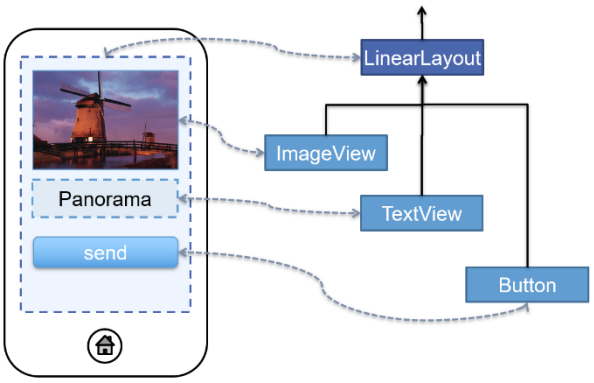
Like most other GUI frameworks, Android relies on the **composite pattern to model a hierarchy of visual components** (display-tree). A view is an instance of the View class (or of any subclass of it). Among the View subclasses, there are both "**elementary**" views (as text fields and buttons), and "**view containers**", which can internally host other views. A display tree is created by instantiating **containers as intermediate nodes** and **elementary views as leaves.**



Create the display hierarchy

Three approaches are possible:

- Create the view programmatically by manually instantiating the single views and configuring their parameters
- Describe the view in a layout XML file, and have it transformed in a view tree by an **inflater**
- Use the library **Jetpack Compose** and describe declaratively the structure and behavior of the view



Programmatical view creation

In the first approach, the various visual components are directly instantiated, and they are connected to each other so as to form the desired hierarchy. The content is configured, and event handlers are registered.

- **Pro**
  - Flexibility: You can insert/configure each component based on the data processed by the application
  - Control: everything that is created has been done explicitly by the programmer
- **Cons**
  - Maintenance is difficult
  - Poor support for internationalization

Create a view via XML

```
override fun onCreate(b: Bundle?) {
   super.onCreate(b)
   setContentView(R.layout.activity_main)
}
```

The XML files in the folder "*res/layout*" describe the visual hierarchy. The build environment automatically associates, to each such file, an integer constant in the class *R.layout.* The name of the constant is the name of the file, the value is automatically selected in order to make it unique. An overloaded version of the *setContentView(...)* method accepts an **integer** as a parameter. It is used to identify and load the XML file. For each element described in the file, the corresponding Java object is created and initialized with all its attributes. A powerful layout editor is available to make it easy to create the XML content.

While the hierarchy is described via XML, customizations can be made via code. Each element of the display tree has to be identified, so each element has an attribute named "**id**", which can be assigned a unique value using the syntax *android: id = "@+id/element_name"*. The compiler automatically creates a constant named *R.id.element_name*, and associates a unique value to it. After loading the XML file, you can get the reference to each widget using the generic method *findViewById<T: View>(id: Int): T?*

- **Pros**
  - Easy to maintain
  - You can use the visual editor provided in Android Studio
  - It supports string internationalization
- **Cons**
  - It is necessary to uniquely identify elements in the view in order to access them in code, in order to customize their behavior
  - Ids must be kept in sync between the XML representation and the Kotlin code

Using the Jetpack Compose library

It is a modern approach to describe GUIs inspired from the javascript React library. Views are described as functions labelled with the *@Composable* annotation. Their body consist of invocations to other composable functions, leveraging on a concise and idiomatic use of the Kotlin programming language. The outcome is fully compatible with views declared using the other approaches.

- **Pros**
  - One codebase consisting of concise and idiomatic Kotlin
  - Declarative and Compatible
  - Designed for and with Material Design
  - Fewer tools required
- **Cons**
  - Requires programming skills
  - Designers find the layout editor more intuitive

Saving the state of an activity

When the operating system needs to terminate the process that hosts an activity, it notifies it using the callback *onSaveInstanceState (outState: Bundle?).* The activity can react to this notification saving its own state. In the Bundle object it is possible to save simple objects (integers, boolean, String) or objects that implement the Serializable or Parcelable interfaces. When the operating system will recreate the process, it will **restore the bundle** and pass it to the methods *onCreate(…)* and *onRestoreInstanceState(…).* That can fetch the information contained therein and if the application terminates by calling the method *finish()*, this method is not called.

Intent

Asynchronous messaging mechanism used by the operating system to associate process requests with the appropriate activities and/or services. It can be directed to a specific component, or it can be broadcasted to other system applications. It can also be used to exchange data between activities. You can create **implicit** or **explicit** intents.

Implicit Intents

Indicate the action to be performed by letting the operating system finding the component that is able to handle it. It is defined in terms of:

```
val number = Uri.parse("tel:5555551212");
val dial =   Intent(Intent.ACTION_DIAL, number);
startActivity(dial);
```

- **Action** - a unique string taken from a pre-defined list or specified by the programmer
- **URI** - a resource identifier whose schema can be used to identify possible recipients
- **Category** - a string that provides additional details on the requested action
    - If an intent refers to one or more categories, only the components that declare to manage all the listed categories in the manifest file are considered.

To launch an activity of another application, we need to create an intent. The *action* parameter defines the action to be performed, the most common ones are defined by the Intent class (e.g., ACTION_MAIN, ACTION_EDIT, …) and the *uri* parameter is the resource referred to by the action.

Explicit Intents

Ask the OS to activate a specific component within the application process. The component is defined by the corresponding class.

```
val intent = Intent(this, MyActivity::class.java);

startActivity(intent);
```

Enriching an Intent

To pass more information to another activity, you can use the *putExtras()* method of the Intent class in the form of **key/value pairs** or collectively, as a **bundle.** A bundle is an object that contains the additional properties of the Intent. When an activity is killed it can be saved on disk for memory management.

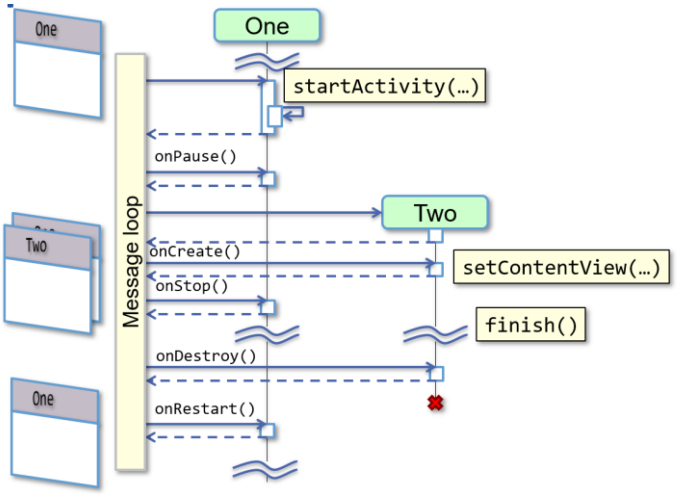
```
val intent = Intent(this, MyActivity::class.java);
val b = Bundle();
b.putString("DEFAULTTEXT", "Some value...");
intent.putExtras(b);

startActivity(intent);
```

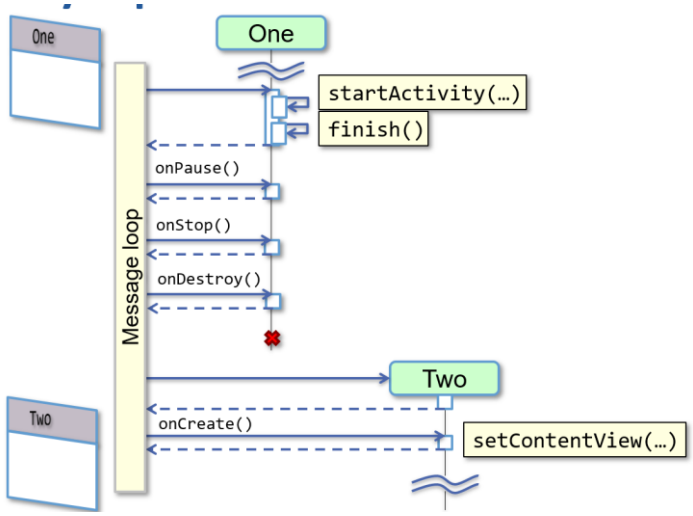
Transitions between activities

When an activity starts another one, the latter is created and pushed on top of the activity stack. Its user interface will usually hide the previous one and the corresponding callback notifications will be issued. When it finishes, it will be popped from the stack, revealing the previous one that will simply be notified of being again resumed. Some variations on this basic pattern exist: the new activity might replace the former one in the activity stack or the new activity might provide some kind of result to the former one.

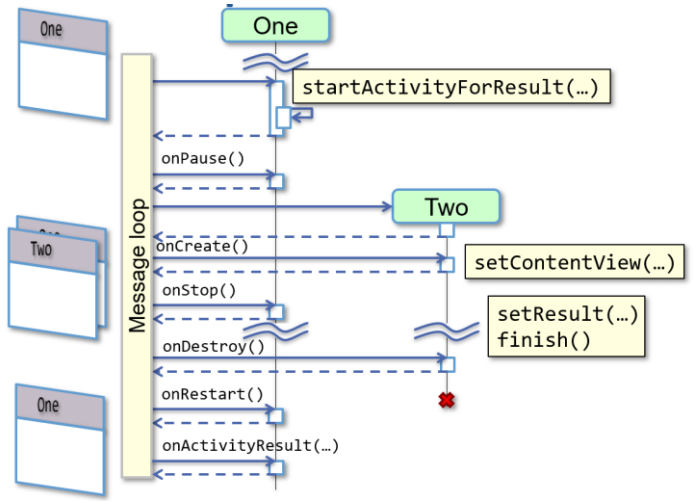
Simple invocation



Activity replacement



Invocation Request



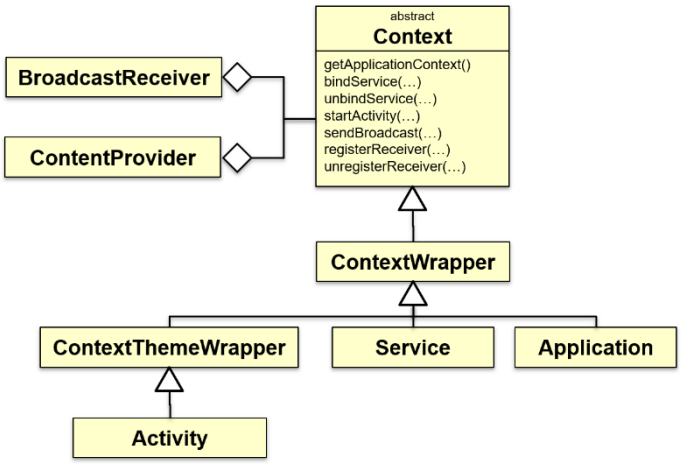
Broadcasting Intents

An activity can **broadcast an Intent object**, via the method *sendBroadcast (…)*. It will be received by all applications that registered a listener via the *registerReceiver(...)* method of their context. Typical usage is for **generating system wide notifications**.

Prior to version 8.0, a broadcast receiver had to be declared in the manifest file. It should extend the *android.content.BroadcastReceiver* class and implement the *onReceive(…)* method. On Android 8.0+ the receiver must be registered in code using the *registerReceiver(…)* method. When it is no more needed, it can be removed via *unregisterReceiver(…)*.

Inheritance hierarchy

Activities, services, and applications are modelled by the relative classes, that compose an inheritance hierarchy. They have a common root: the **Context class**. Context provides the functionality needed to access resources and application-specific classes and to interact with the operating system: start other activities, send and receive intent, access the file system, ...



Each active component forms its context. Its life cycle coincides with that of the component. There is also a global context, relative to the entire application accessible via the method *getApplicationContext()*.

Context and resources

One of the main features offered by the context is the access to **resources:** set of strings, images, xml files, ..., which are aggregated to the executable file to be uniformly and independently accessible from the installation folder. Each resource is identified by a unique number, and it is automatically generated at compile time and stored in a constant in the R class.

Access to resources and preferences

You can access a resource using the getResources() method by specifying the unique identifier of the resource that was inserted into the file R.java. You can get the application preferences using the *getSharedPreferences()* method. The *SharedPreferences* class can be used to store simple data application as configuration settings in the form of key/value pairs.

The Context class also allows you to:

- Start activity instances
- Getting assets included in the application
- Request system services
- Manage files, directories, and databases accessible only to the current application
- Check and activate application authorizations

Graphical User Interfaces

The use of graphical tools facilitates, at the operational level, the design of user interfaces by allowing to create (even sophisticated) screens in a few clicks. However, this subject remains highly problematic for two kinds of reasons:

- Cognitive issues
- Operational issues

Cognitive challenges

Building an interface that is **pleasing, functional, consistent, easy to learn** and **use** in real operating environments is a major challenge. If the task requires the user to go beyond the most basic interactions ("write a text and press Ok"), you must design the informational flow and the organization of graphic contents very carefully. Failing that, the user will be confused and will *stop using* the application, without giving a second chance.

Operational challenges

Regardless of how contents are organized, mobile devices vary widely in terms of:

- Physical **dimensions** (readability, interactivity)
- Screen **resolution** (amount of overall displayable information)
- Dot **density** (ratio between the two previous parameters)
- Screen **orientation** (layout and logical organization of content)

Some properties can vary of one or more orders of magnitude: you cannot "draw" a single interface and fit it to all cases, you should design a set of possible alternative configurations and manage diversity. In addition to device diversity, it is necessary to consider the human variability:

- Different languages and writing systems
- Different cultures and metaphors
- Different levels of confidence with the device and/or to the proposed functionality (novice/expert)

Designing mobile applications

There is no “operating manual” which ensures an effective design. It is possible to try to identify some general guidelines:

- The first step is related to the **analysis of the context and the processes** that you intend to support.
- The second step tries to subdivide complex tasks into smaller ones, identifying typical “**interaction patterns**”.
- The third step relies on the general principles of **interaction design**, to ensure that basic principles are not violated.

Users	Behavior	Feedback and flexibility
Know your users and their profile	Identify elementary tasks	Show what the app is doing
Expertise levels	Ask as little as possible	Make the user aware of the interaction flow
Expectations	Introduce a priority order	Insert feedback mechanisms

Layout

Having figured the design of the user interface, you can start building it. In general, a GUI consists of a set of elementary components (widgets) suitably arranged on the screen. The disposition can be described via code or in a layout xml-file. The XML-based solution facilitates the decoupling between presentation and logic.

Although it is possible to place the individual widgets at a specific (and fixed) location on the screen, it is usually a bad idea. The great **variability of the screen sizes** would make the application unusable in most cases. Here is a list of problems:

- Problems related to the **resolution** (number of pixels):
  - Screens with lower resolution would not be able to accommodate all components
  - Wider screens would present large empty areas
  - The variability is of about one order of magnitude (from 320x240 to 1920x1080 and beyond)
- Problems related to the **physical size**:
  - The widget size is proportional to the diagonal length: very small screens make both reading and interacting very difficult
  - Variability of two orders of magnitude (from 2.5 "to 10" tablets to 50"+ smart TVs)
- Problems related to the **density** (pixels per inch)
  - Screens with a density much higher than the design one would make widgets much smaller (difficult touch interaction, reading difficulties) than expected
  - Low density screens would lead to disproportionately large components
  - Seemingly limited variability (from 120 to 300+ dpi) but area variations are proportional to the square of the density variations
- Problems related to the **aspect ratio** (width/height)
  - The logical organization of content becomes complex
  - Variability by a factor of ~ 3 (16:9 to 9:16)

Hierarchical layout

In general, you can create a composite view through an instance of the *ViewGroup* class to which are added, as components, the individual widgets or other ViewGroup objects. Android provides a rich family of classes derived from ViewGroup specialized in the way they manage the space assigned to them. **By selecting suitable management policies, it is possible to mitigate the effects of the devices variability.**

Specifying the dimension

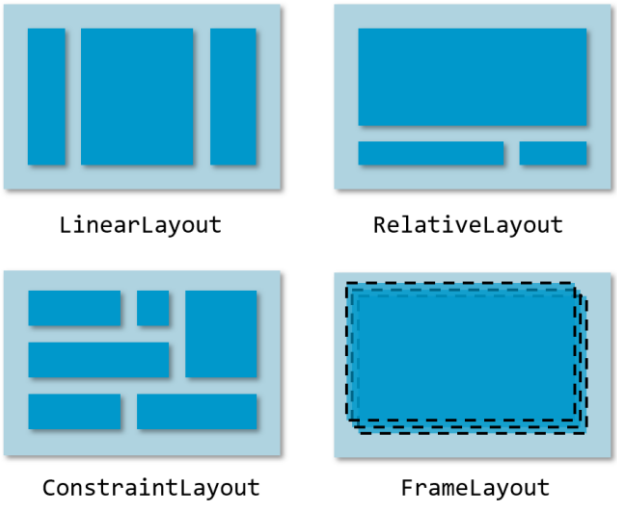
If you create a hierarchy of views via code, all methods that refer to positions/length/size use pixels as measure unit. To relate these values to absolute magnitudes, refer to the *android.util.DisplayMetrics* class.

Specifying the dimension

In the xml layout, you should indicate the measure units adopted for each value by using an appropriate suffix: Pixels (px), Inches (in), Millimeters (mm), Points (pt, 72pt = 1in), Density independent pixel (dp, 160dp = 1in), Scale-independent pixels (sp, 1sp = 1DP * font_zoom_factor).

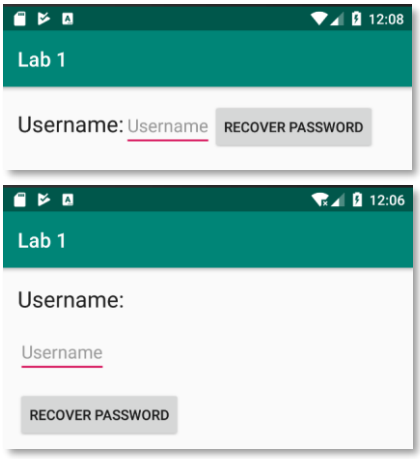
Instead of directly enter constants in the description of the layout, you can refer to resources of type "*@dimen*". The actual value is specified in an xml file in the "*./res/values*" folder. This increase flexibility in case of dependencies on other factors (themes/language/culture).

Types of layouts



Linear Layout

Child elements are arranged one after the other on a column or a row on the basis of the orientation of the container. Some properties are defined on the container element (e.g., the orientation) while others are specified in each child element. The name of these properties begins with "*layout_*":



- Dimensions (layout_width and layout_height)
- Weight (layout_weight)
- Contents arrangement (layout_gravity)
- External spacing (layout_margin)

The **android:orientation** property indicates whether the elements of the linear layout are **vertically** or **horizontally** arranged. Its value can be horizontal or vertical. You can change this property also via code using the *setOrientation(...)* method.

The properties **android:layout_width** e **android:layout_height** can assume the following values:

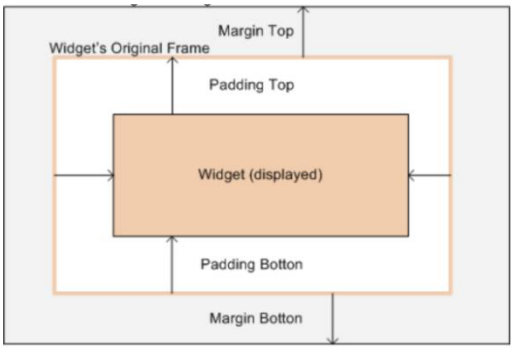
- A number followed by its units of measure. For example, 100 dp
- The constant **wrap_content** that occupies the space they need the total number of children with the corresponding margin
- The constant **match_parent.** In this case the view tries to fill all available space within the container of the parent view

The property **android: layout_weight** indicates how to divide any extra space among the children's elements. The default value is 0. Residual space is assigned in proportion to the weight of each element. Divided by the sum of the weights of all the elements of the layout.

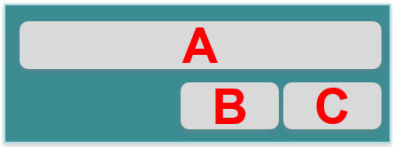
The *android: layout_gravity* property allows you to set the alignment of a widget relative to its base container. By default, the widget has left and top alignment. Values can be: left, center, right, top, bottom, ...

Padding and margin

Any widget can indicate the presence of empty space around itself, using the padding property. It is the attribute *android:padding* in XML or via the *setPadding(..)* method. When a widget is added to a container, additional outer space can be added using *layout_margin* or via its sub-properties *layout_marginLeft*, *layout_marginRight*, ...



Relative Layout



It is a layout where the positions of the children can be described in relation to each other or to the parent. In the example: A is placed at the top of the parent container, C is positioned below A, with its right edge aligned with the right edge of A and B is placed below A, to the left of C. The alignment constraints with the parent container are expressed using boolean properties (e.g., layout_alignParentTop).

A widget can be constrained to some sibling element using the following properties: layout_above, layout_below, layout_toLeftOf, layout_toRightOf, etc… Siblings are referenced using their id property.

FrameLayout

Show the children superimposed on the same area. The most recently added element is the last to be designed (and then it appears above the others). By using the layout_gravity property, it is possible to place the various sub-elements in different parts of the FrameLayout rectangle. This is the base class used for deriving more advanced containers.

ScrollView

The ScrollView control is used when a view has to display more data than the ones that can be displayed on a single screen. It implements only the **vertical scroll and** can often be replaced by NestedScrollView. The *HorizontalScrollView* allows to **scroll horizontally**.

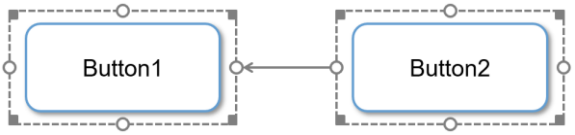
ConstraintLayout

Extension of the mechanisms available in LinearLayout (weight) and in RelativeLayout (component alignment) allowing to flatten down the visual hierarchy and improve expressivity and performance. It allows to create **flat, fast, and effective visual hierarchies** supporting almost all needs of graphical designers (e.g., Alignment, (non) uniform spacing, aspect-ratios). It is highly integrated with the visual editor of Android Studio. Blueprint view provides an effective way to setup and manipulate most information about constraints.

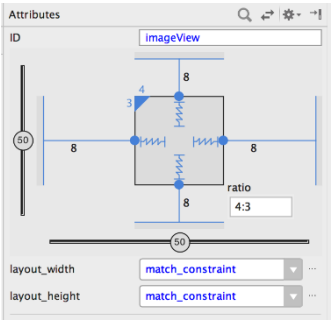
Constraints

A constraint is a one-way relationship between two widgets. It controls how they will be positioned within the layout. It can be created by dragging handles to other element in design or in blueprint view:

- Square handles resize component
- Circular handles set constraints



The "Attributes" pane allows to add and remove constraints to the container borders:



- Visually define margins
- And change how width and height will be computed (fixed, wrap_content, match_constraint)

Constraints can be written also via XML notation. It is introduced by the "*app:*" namespace, being the *ConstraintLayout* a library inside the project rather than an element of the Android framework.

Chains

When a constraint is duplicated in both directions between two widgets, a chain is built. It allows to share space between views and control how much of it is given to each view. There are three modes for chaining elements:

- **Spread** mode (default): available space is divided in (n+1) blocks, and the chained widgets are evenly separated
- **Spread_inside** mode: available space is divided in (n-1) blocks; first and last elements touch the container border
- **Packed** mode: available space is divided in half, and concentrated to the beginning and end of the chain

If a horizontally chained element has its width set to *match_constraint* (0dp), available space is divided according to its *horizontal_weight* similarly to LinearLayout. Weights do not work in packed chains.

Guidelines

Guidelines are special views with no appearance (0 pixels-high and -wide) that may be used to align other elements. A guideline can be either horizontal or vertical. Its position can be set relative to the start margin, to the end margin, or to a percentage of the size of the container.

Aspect Ratio

The attribute "constraintDimensionRatio" allows to propagate vertical constraints to horizontal ones and vice versa. It can be set to any floating-point value or can be an expression made of a letter ('h' or 'v') indicating which direction drives, and a ratio expressing how to derive the other dimension *app_layout_constraintDimensionRatio="h,16:9".*

Barriers

They allow to express constraints concerning multiple elements, so that the largest one will drive the actual calculations. Each barrier has a "barrierDirection" attribute determining the direction of the element. Barriers reference the ids of the constrained views.

Other layout policies

Some devices have so different characteristics from each other that, despite the flexibility introduced by the different layout types, it becomes difficult to find a harmonious and functional arrangement for each of them. We often need to create a number of alternative layouts that adapt to different configurations. The **resources management system** provides an automatic support to use different layout on the basis of the actual characteristics of a device. By adding the **appropriate suffix** to the folder res/layout, you can specify the layout for each configuration. It is necessary to create, in each sub-folder, a file with the same name (and the same fundamental elements). The system will automatically choose which to load.

Resources

In an Android project, the resource files are stored separately from those of Java classes. Most types of resources are represented in XML. It is possible to save as a resource also raw file and images. The Eclipse ADT plugin generates and compiles the *R.java* file, which contains the identifiers of each resource.

All the names of the resource files must be lower case and simple: only letters, numbers, and the underscore character. You can save different types of resources:

- **Simple**, such as strings and integers
  - They are stored in XML format in the directory /res /values
  - Usually the resources are stored in separate files that reflect their type, such as "strings.xml"
- **Complex**, such as images, animations, menu and file
  - They are not saved under the directory /res /values
  - But in directories whose name reflects the type,

For example, animations reside in the directory */res/anim*.

The mechanism of resources specialization applies not only for layouts, but for any resource type

- To internationalize the application
- /res/values/strings.xml (strings in the default language)
- /res/values-it/strings.xml (strings in Italian)
- /res/values-en/strings.xml (strings in English)

To manage images for screens with different densities

- /res/drawable-ldpi/myLogo.png (low density screen)
- /res/drawable-mdpi/myLogo.png (medium density screen)
- /res/drawable-hdpi/myLogo.png (high density screen)

Selection rules

During execution, the system always looks for the most specific version of the resource, matching the current device configuration. If not available, a more generic version will be selected. If no generic version is present, a failure will happen. Alternative resources must be **named exactly as the default** one and use the **same identifiers**. The creation of alternative resources affects the size of the application.

Managing density

It is better to keep code independent from the density of the screen. If the size of the TextView and the drawable bitmap are specified in pixels a UI element appears larger in a low-density screen and smaller in a high-density one.

If the size of the widgets is specified in dp (device independent pixels), with:

- A medium density you get the same results previously seen
- A low and high density, the system scale the values of dp of the components, creating components of approximately the same physical size

As an alternative to dp, you can specify the value "*wrap_content*" for the attributes "*android: layout_width*" or "*android: layout_height*". In this case the component will occupy the space needed to present its content (plus any padding).

Android Architecture Components

The Jetpack project

It is a set of components, libraries, and tools provided to developers with the following aims:

- **reduce development time**
- **Reduce boilerplate code**
- **Build robust and high quality apps**

It is divided in four main areas:

- **Foundation** – Backward compatibility, testing, kotlin support
- **Architecture** – Properly structure projects to make them robust, testable and mantainable
- **Behaviour** – Integration between apps and standard Android services (notifications, sharing, permissions, assistant)
- **UI** – widgets and helpers to make app delightful to use

All classes are part of *androidx.** packages. Many of them are a re-write of existing ones while others are brand new.

Jetpack Foundation

Areas common to all apps:

- Backward compatibility (AppCompat)
- Unit and runtime tests (Test)
- Performance assessment (Benchmark)
- Security and access control (Security)

Special usage contexts:

- Vehicle integration (Auto)
- On demand TV (TV)
- Wearable devices (Wear)

Development support:

- Idiomatic usage of Kotlin (KTX)
- Support for large apps (Multidex)

Jetpack Architecture

Classes and interfaces that help make a robust, testable, and maintainable app:

- Declaratively bind UI elements to variables (DataBinding)
- Manage activity and fragment lifecycles (Lifecycles)
- Notify views of any data changes (LiveData)
- Handle everything needed for in-app navigation (Navigation)
- Gradually load information on demand from your data source (Paging)
- Fluent SQLite database access (Room)
- UI-related data management in a lifecycle-conscious way (ViewModel)
- Background jobs management (WorkManager)

Jetpack Behavior

The behavior components help in the integration with standard Android services:

- Schedule and manage large downloads in background with auto retry support (DownloadManager)
- Add professional camera capabilities (CameraX)
- Backwards compatible APIs for media playback and routing (Media & Playback)
- Backwards-compatible notification API with Wear and Auto support (Notification)
- Check and request permissions (Permissions)
- Manage preferences (Preferences)
- Improve content sharing across app (Sharing)
- Display rich, dynamic, and interactive content in the Google Search app and the Google Assistant (Slices)

Jetpack UI

Graphical components and helper classes that allows to improve the overall UX:

- Managing complex choreographies (Animation & Transition)
- Updated emoji font (Emoji)
- Improve fragment management (Fragment)
- More layout components (Layout)
- Create views with a functional pattern (Compose)
- Dynamic management of color paletts (Palette)

Application architecture

Android applications are built by putting together a series of components that are managed directly by the OS such as Activities, Services, Content providers, Broadcast receivers. Components can be launched individually and out-of-order, and can be destroyed at anytime by the user or the system. This may lead to **unexpected sequences of events** that makes designing and testing an app a difficult task.

Separation of concerns

No component should store, inside its own properties, **application data or information about its state**. Moreover, no component should depend on the (active) existence of another, unless a specific API provides this kind of guarantee (e.g., a bound service). **Asynchronous operations** (like interacting with a remote server) are **particularly problematic**. When the response arrives, the component that originated the request, **may exist no more**.

Classes extending Activity and Fragment should **only deal with UI or operating system interactions**. These classes are only **glue classes** and can be **destroyed at any moment** by the user or by the OS. The content of the UI should reflect a **model**, preferably a persistent one. **Users will not lose data if the app gets destroyed**. Your app will work even on problematic network connections.

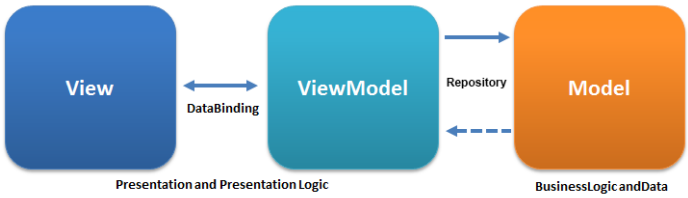
Architecting a general solution

The problem of how to properly organize code in order to cope with these problems has received several responses which, in turn, have led to different architectural patterns:

- Model-View-Controller (MVC)
- Model-View-Presenter (MVP)
- Model-View-ViewModel (MVVM)

Google highly recommends the last one and supports it with several enhancements in its libraries.

The MVVM Architecture



- The **Model c**ontains all the data classes, database classes, API and repository
- The **View i**s the UI part that represents the current state of information that is visible to the user
- The **ViewModel** contains the data required by the View and transforms the data stored in Model in a way suitable to be presented inside the View. ViewModel and View are connected through DataBinding and the observable LiveData.

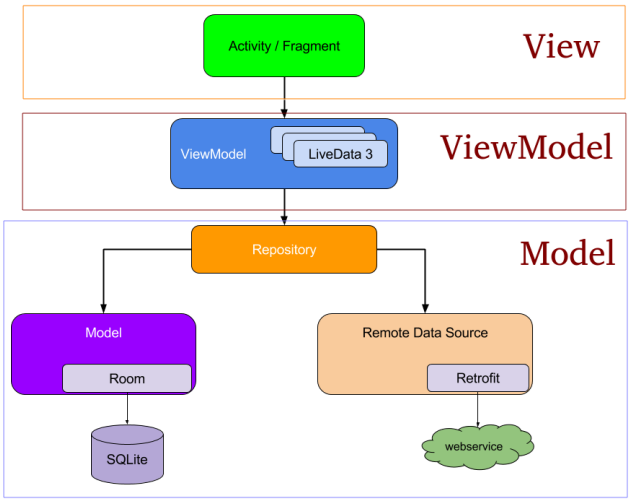
MVVM features

The features are:

- **Makes the project loosely coupled s**eparating the business logic from the presentation
- **Easier to maintain** since the view is not aware of the computation happening behind the scenes and improves testable code
- **It gives great structure to your project and makes it easier to navigate** and understand our code. Each block has specific responsibilities and it is simple to add a new feature or remove existing ones.

Google's implementation

There can be several ViewModel. Pay attention to the "Model" word. It represents the whole block composed of Repository, Remote Data Source and Model (which should be called database). The repository provides a façade access, then data can be fetched remotely (via Retrofit that is an HTTPS client) and through Room that is an ORM-compliant library (Object-To-Relational mapping). If the network is off the repository will be directed to the local database (SQLite).



LiveData

It is an **observable** data holder class that notifies its observer when the underlying data change. It is **Lifecycle-aware** which means that it respects the lifecycle of other app components. Only observers in components that are in an active lifecycle state are updated (STARTED or RESUMED). The advantages to use LiveData are:

- No memory leaks
- No crashes due to stopped activities
- No manual lifecycle handling (nothing to do onStart(), onStop(), ...)
- Always up to date data
- Proper configuration change handling
- Sharing resources: you can wrap system services with LiveData. The LiveData object connects once to the system service and observers just watch the object for changes

SKIPPED CODE SLIDES 21-22-24 CAP A06

Models

Model is responsible providing app data either fetching it from the local storage or from a remote web service. It is further divided into sub-components:

- The **Repository** provides a façade to access the enclosed data
- The **Remote Data Source** contains the code necessary to access a remote server, that represents the primary information source
- The **Model** (!) contains the code necessary to persist, in the local storage (often a DBMS) the received information

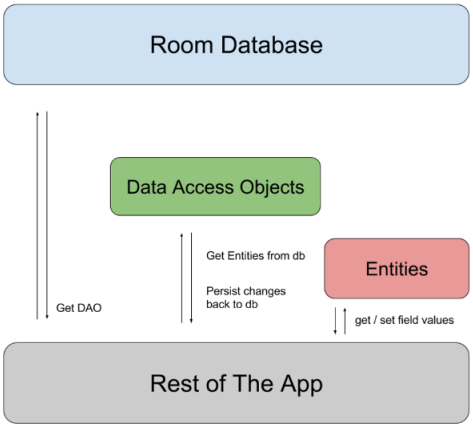
Models are independent from Views and any other Android component. They are **isolated** from the complex components' lifecycle. Managing separately the UI code from the application logic makes it easier to maintain the app and enables testing. Models provide **consistency** to your app.

Asynchronous updates

Model state evolves because of many different reasons: some may be local to the app, some remote. In most cases, these updates are **asynchronous.** They may happen at **any time**, possibly conflicting with the evolution of the components' lifecycle. Models provide their **contents as LiveData objects** or arrange a way to propagate content update via call-backs.

The Room Persistence Library

Room provides an abstraction layer over SQLite It is an ORM (Object-to-Relational Mapping) library provided by Google. The **room-compiler** is an **annotation processor** that operates in the Gradle build pipeline and generates Java code starting from your annotated classes. It is useful when dealing with domain specific languages, DSL's.



The components of Room are:

- **Database**: contains the database holder and represents the main entry point to the persisted relational data
- **Entity class:** represents a table within the database. Its instances represent single rows in the table
- **DAOs – Data Access Objects**: declares the methods that will be used to access the DB (an interface that will be implemented by the Room Annotation Processor)

Database

Abstract class that extends RoomDatabase annotated with **@Database**. The annotation should list all the entities referred to by the DAO object. It must contain an abstract method with no arguments that returns the DAO class. It may contain a static method for DB creation, but Database instances must be singletons.

Look code slides 33-34-35 cap A06.

```
@Entity(tableName = "users")
data class User (
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,

    @ColumnInfo(name = "first_name")
    val firstName: String,

    @ColumnInfo(name = "last_name")
    val lastName: String

)
```

Defining data using Entities

Database schema is driven by the class structure: by default Room creates a column for each field. With **@Ignore** you can flag data that must not be persisted. You can change the column name via the annotation **@ColumnInfo**(name = "first_name"). Each entity must define at least one primary key with the **@PrimaryKey** annotation. To add indices to an entity, include the indices within the @Entity annotation

@Entity(indices = {@Index("name"), @Index(value = {"last_name", "address"})})

Uniqueness can be specified with the property **unique** of a @Index annotation set to **true.** Relationships are expressed through the **@ForeignKey** annotation passed as a property to the @Entity one

```
@Entity(foreignKeys = @ForeignKey(entity = User.class,
                                  parentColumns = "id",
                                  childColumns = "user_id"))
```

SQLite handles **@Insert(onConflict = REPLACE)** as a set of *REMOVE* and *REPLACE* operations instead of single update one. This method could affect the foreign keys constraints. By setting *onDelete = CASCADE* in the ForeignKey annotation you can tell SQLite to delete all related items if the source record is removed.

Accessing data

Data can be retrieved either as instances of the entities classes or it can be wrapped by LiveData objects or reactive types (RxJava2 Flowable's). If necessary, data can be accessed also via Cursor objects that allow direct access to the DB. See slide 39 cap A06.

Maintaining the DB

As you add and change features in your app your DB evolves, and your entity classes change. Room provides the **migrations mechanism** for this particular need. See slide 40 cap A06.

SKIPPED SLIDES FROM 41 TO 44 CAP A06

ViewModel

Designed to store and manage UI-related data in a lifecycle conscious way and to propagate change requests to the underlying model. It is indirectly instantiated inside an activity or a fragment using a **delegate property**. This allows to rebind the same ViewModel when the activity/fragment is destroyed and recreated as a consequence of a configuration change (device rotation, change in the UI language, …). When the owning component is finished, the ViewModel is notified via the onCleared() method. ViewModels must extend the *androidx.lifecycle.ViewModel* helper class or a subclass of it, like AndroidViewModel.

What VMs should not contain

A ViewModel must **never reference** a view, Lifecycle, or any class that may hold a reference to the activity context: Views, fragments, adapters, drawables, lifecycles, lifecycle-aware observables, …

Resources must only be referred to by their **ids**, and should be loaded by the view. The only exception to this rule is the **Application context**, since that object lifecycle spans the entire duration of the process, it is safe to store it in a ViewModel. Class AndroidViewModel, which can be used super-class of custom ViewModels, **requires the application object to be constructed**. Android code that instantiates a class deriving from it will automatically inject the application object into the ViewModel.

Buffering view state data

A ViewModel is the natural place where view state can be buffered:

- Immutable public LiveData values are typically provided to observe data
- MutableLiveData private variables allow the VM to update its own state
- If a VM needs to propagate an existing LiveData coming from the Repository, Transforms method can be used

Methods that propagate change request to the model may require switching to a different thread. Android forbids, in fact, performing long running operations from the main thread. ViewModels can also be used as a **communication layer between the different fragments** hosted inside the same activity. This **decouples** the **data exchange** allowing fragments to interact without necessarily being alive at the same time.

Lifecycle of a ViewModel

It is created the first time an activity calls onCreate. When an activity is recreated, it receives the **same VM instance**. When the owning activity/fragment is finished/detached, VM's onCleared is called.

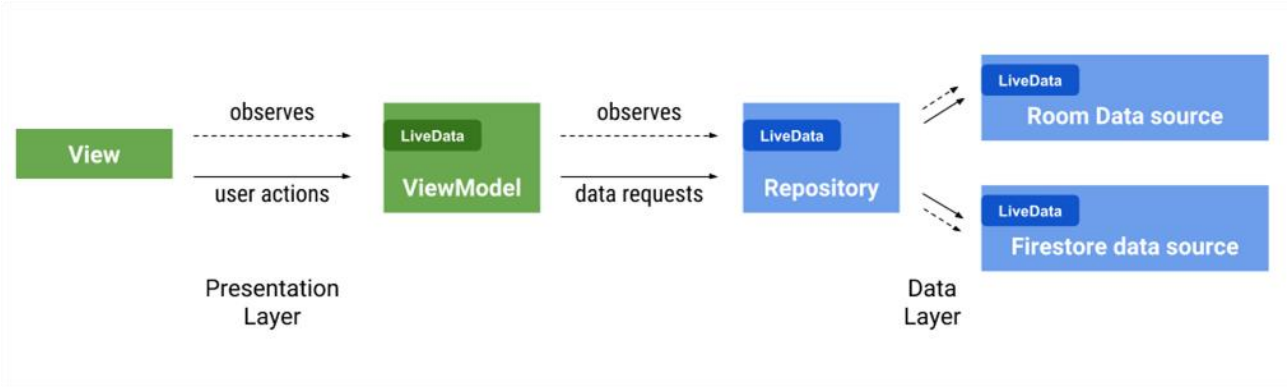
Android defines the androidx.lifecycle.ViewModel class and make some provisions to keep it alive across configuration changes. This requires a special way to instantiate ViewModels. They need to be created in association with a **scope** (a Fragment or an Activity) and they are retained while the scope is alive.

The viewModels() function

Extension function applied to activity/fragment returning a ViewModel tied to the invoker object. It is part of the *activity-ktx* library. It **lazily** returns a *viewModel* of the target class (i.e., the class of the variable to be assigned) and stores it in the activity's *ViewModelStore* using the class name as a key. If the same class will be requested in future by an activity having the same ViewModelStore, the **stored instance will be returned**. This is what makes the ViewModel "**special**" with respect to the activity life-cycle.

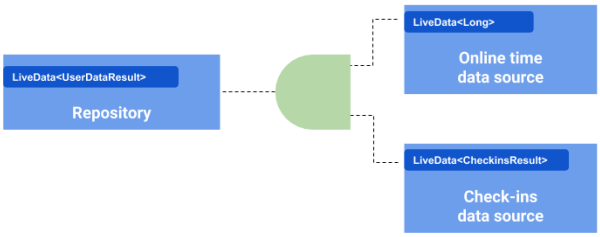
ViewModel in fragments

A fragment can be tied to its own ViewModels (*by viewModels()*) or reuse a ViewModel tied to the enclosing activity (*by activityViewModels()*) or even one tied to the current NavigationGraph (*by navGraphViewModels(R.id.graph)*). It is useful for managing the business logic of related parts of the app. A *ViewModelFactory* can be provided to these functions in order to pass parameters to it.



Merging multiple LiveData sources

Dealing with multiple asynchronous data is difficult since there are multiple callbacks to handle. *MediatorLiveData* can be used to merge this data together by creating a **single source** of the merged data.



SKIPPED SLIDES FROM 60 TO END CAP A06

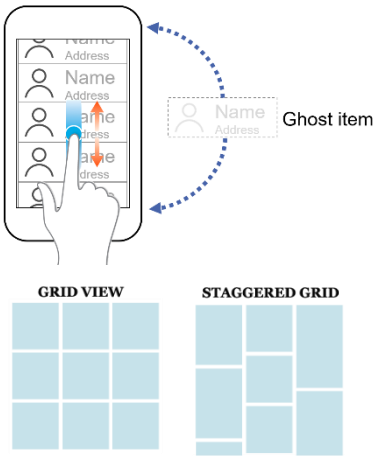
Displaying collections

We often need to show lists or other kind of information collections whose **size is not known a priori** and may be large. Dynamically populating a view is not, usually, a good solution. If the number of entries to display grows significantly, the visual hierarchy **grows and consumes a lot of resources**.

Originally, the Android framework offered a bunch of classes to deal with this problem: ListView, GridView, Gallery, … which are subclasses of a more general one, AdapterView. Today, these classes are considered legacy, and their use is discouraged in favour of a more modern, flexible, and performant approach to displaying lists, based on RecyclerView.

Recycling views

Usually, only a limited set of items can be displayed on the screen. Instead of creating as many views as items in the collection, only a **small number** of them can be created: those that are visible plus one extra **ghost item**. The ghost item will become part of the visual hierarchy when the list is scrolled. At the same time, since the item at the opposite end will become invisible, it will take the role of the ghost item.

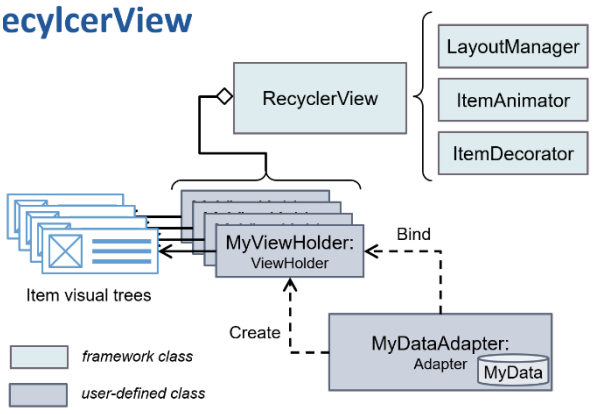


The same principle may be employed if the collection is shown as a grid or with another kind of layout. Simply, more ghost items are needed. This approach can **drastically improve performance** by skipping initial layout inflation or construction, which require a lot of CPU cycles.

androidx.recyclerview.widget.RecyclerView

It is a class of the Jetpack library that manages the presentation of potentially large data sets:

- Presented items are managed by an **Adapter**, which is responsible for providing and filling views with relevant content from the data set
- Each item in the data set is presented in a recyclable visual hierarchy which is managed by a **ViewHolder**
- The visual presentation of items is delegated to a **LayoutManager**, possibly helped by an **ItemAnimator** and an **ItemDecorator**

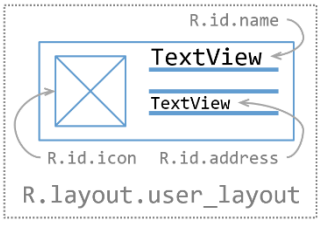


Implementing the Adapter

The RecyclerView.Adapter instance has three responsibilities:

- It stores the data model and offers a method to retrieve the total number of items in the model (**getItemCount()**)
- It supports the creation of a new ViewHolder and of the corresponding visual tree (**onCreateViewHolder(…)**)
- It populates the created visual tree, via the ViewHolder with the data of a specific item (**onBindViewHolder(…)**)

Given an item data model, a corresponding layout for presenting its content is usually created.

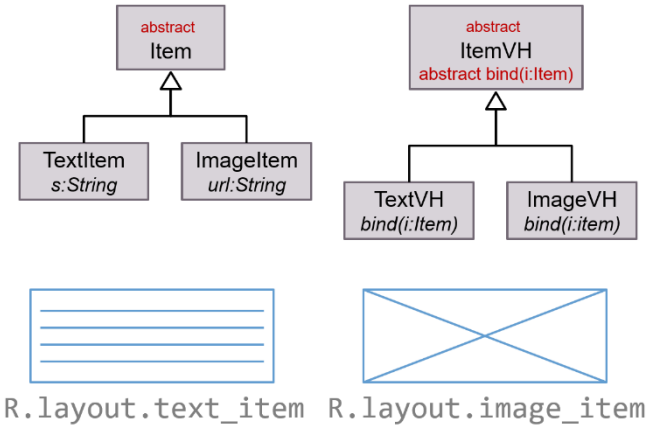


SKIPPED SLIDES 13-16 CAP A07

Displaying heterogeneous views

Sometimes, items inside the collection need different representations depending on their content. RecyclerView can manage this, provided that the adapter cooperates. Adapters can override method *getItemViewType(pos: Int): Int* that can return an arbitrary integer.

When a ViewHolder is created, the requested type is passed as a parameter. Items which have the same type can recycle the same ViewHolder.



SKIPPED SLIDES 18-23 CAP A07

Interacting with the RecyclerView

If any kind of interaction is needed within the single item presentation a suitable handler should be installed when the ViewHolder is bound to a specific data item. Callbacks can be removed when the ViewHolder is recycled (*fun bind()* and *fun unbind()*).

Manipulating the data model

If any change occurs to the data list, a proper notification should be raised. RecyclerView.Adapter offers several useful methods that trigger an update of the RecyclerView:

- *notifyItemInserted(position: Int)*
- *notifyItemRemoved(position:Int)*
- *notifyItemChanged(position:Int)*
- *notifyItemMoved(fromPos: Int, toPos: Int)*
- *notifyDataSetChanged()*

When a notification is received, the RecyclerView updates its presentation, performing an **animation**. This will not happen if *notifyDataSetChanged()* is invoked, because it has no clues about what changes did happen. In order to cope with this, the **DiffUtil** class can be used.

DiffUtil calculates the **difference** between two lists and output a list of update operations, based on the Myers' algorithm. A class implementing the *DiffUtil.Callback* interface must be provided. See slides 28-29 cap A07.

Legacy data containers

Initial Android versions did offer several classes specialized in the presentation and interaction with data collections: ListView displays a vertical list of views, while GridView displays a two-dimensional grid of views. These classes are now considered deprecated in favor of RecyclerView, which is simpler and more efficient.

Fragments

Managing the user interface may be complex. It is difficult to choose a presentation that nicely fits to different classes of devices. Screens are heterogeneous in several ways:

- **Resolution** (from 320x240 to 1280x1024 and more) affects the amount of information that can be displayed and its readability
- **Physical size** (from 3 "to 10" and more) affects the interactivity
- **Orientation** (vertical or horizontal) and aspect ratio (3/4 to 9/16) affect the arrangement of contents

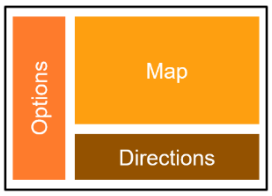
To help the designer in the creation of usable layouts, the concept of "**fragment**" (*android.app.Fragment*) was introduced:

- It makes the graphical user interfaces more **adaptive**
- It facilitates the **creation of tabbed** views (tab)
- It supports the **creation of custom dialog** boxes
- It allows to **design custom navigation** graphs

Fragments are intended to encapsulate major components in a **reusable way**, hiding the complexity of handling the user interaction within the component and leaving a high level of customization and integration with other components that may be visible at the same time or not. Typical examples are *maps, time & date picker, shopping carts*, …

Modularity

Each fragment is designed to support a well-defined kind of interaction with the user. By dividing complex activity code into several fragments, better organization and maintenance is achieved.

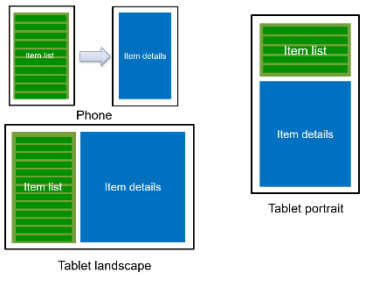


Reusability

The behaviour supported by a single fragment may be re-used by multiple activities or it can appear several times in the same one.

Adaptability

To simplify the task of adapting an interface to different types of screens, you can divide it into smaller blocks and combine them in different ways, according to the characteristics of the device. The different blocks are kept together by the activity that hosts them. The activity acts as a hierarchical controller of the features offered by each block.



Fragments

Introduced as core components of the operating system from version 3.1 of the operating system (API 12). They are available as an add-on module for older devices. It is part of the Jetpack Library and usable on over 99.7% of the devices in use today. For the sake of compatibility, two kinds of fragment classes exists:

- android.app.Fragment, aka “platform fragments” (we will not use these)
- androidx.fragment.app.Fragment, aka “compatible fragments”

They are (almost) functionally equivalent. The second ones can only be hosted inside classes that derive from *androidx.fragment.app.FragmentActivity* (e.g. *androidx.appcompat.app.AppCompatActivity*). A Fragment is an object that, conceptually, **stands between Activity and View.**

- As an Activity, it has a **complex life cycle**, managed by numerous callbacks and participates to the management of the interaction flow between the user and the app.
- It owns a **hierarchy of views** that may become part of the host activity visual tree.

The class Fragment need to be sub-classed in order to implement a specific behaviour. Differently from activities, which are completely managed by the Android framework, **fragments are manipulated** (created, removed, shown, hidden, …) **directly by the programmer**. Their constructor should not perform any real task: real initialization should happen **inside the onAttach(ctx: Context) method**. The fragment UI can be described by an **XML file** that defines its layout or it can be dynamically populated via code.

The **onCreateView(...)** method of the Fragment class has the task of returning the created view. It receives, as parameters, a **LayoutInflater**, the **ViewGroup** that will host the fragment and a **Bundle** that contains its previous state (if available). The main layout of the activity usually contains one or more *FragmentContainerView* the purpose of which is to host the fragments that will be dynamically appended to it.

When the activity loads its own layout the *FragmentContainerView* will act as a placeholder where fragments may be appended. The activity is responsible of **instantiating** the required fragments and of **displaying** them inside the placeholder. The latter operation occurs thanks to the help of a special component, the **FragmentManager** which is accessible via the *supportFragmentManager* property of *FragmentActivity* and its subclasses.

Dynamic fragment management

The FragmentManager allows you to dynamically change the fragments shown inside an activity. Changes should be enclosed within a transaction.

```
class MyActivity : AppCompatActivity() {
  //…
  fun showFragment(f: Fragment) {
    supportFragmentManager
      .beginTransaction()
      .replace(R.id.fragment_container, f)
        //other operations: insertion,
        //removal, visibility change, …
      .commit()
  }
}
```

Fragment insertion

The *add(…)* method of *FragmentTransaction* adds the fragment to the activity state. If the fragment has a view and a container id is specified, the fragment view is inserted in the container. A tag (unique string) can be optionally associated to the fragment, in order to make it easier to retrieve it via *FragmentManager.findFragmentByTag(…)*.

Fragment removal

The *remove(f: Fragment)* method removes a fragment from the view hierarchy and detaches it from the activity. The *replace(containerId: Int, f: Fragment)* method removes all the fragments currently present in the view identified by containerId and inserts the fragment f in the same view.

Avoiding double instantiation

When an activity, designed to host dynamically created fragments, is created for the first time, no fragments are present. Typical code in *onCreate(…)* will start a FragmentTransaction and add those that are needed. If the activity, for any reason, is destroyed when it is not yet finished, when it will later be re-created, the fragments that were part of it will be automatically re-created as well.

If the initialization code does not take this into account, **two copies** of the fragment(s) will exist and be part of the activity state. You may **check whether this is the first instantiation or not** using the savedInstanceState parameter of onCreate(…), if it is **null**, this is the first instantiation (Is bundle null?).

Fragment state

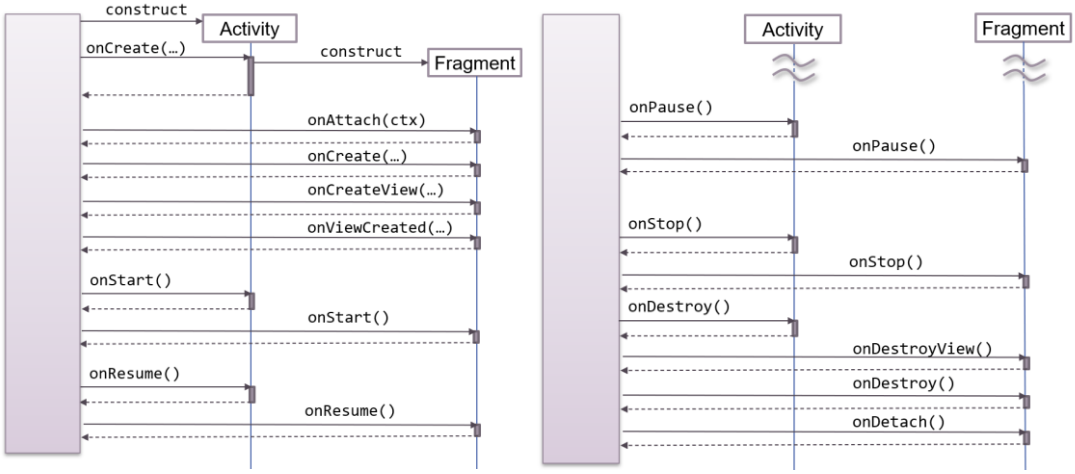
A fragment, during its life cycle, can be in one of three macro-states:

- **Existing as an object, unbounded from any activity**
- **Being bounded to an activity, but not visible**
- **Being bounded to an activity and visible**

On each state change, a corresponding call to method onXXX(...) of the fragment instance occurs. When a fragment is bound to an activity, the activity state changes are reflected on the fragment state.

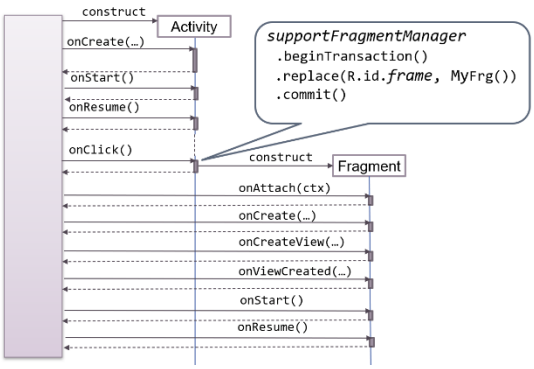
Fragment lifecycle

A fragment may be either attached to an activity (and, thus, have a context) or not. If a fragment is detached, it is not part of application state. Corresponding notifications are: *onAttach(ctx: Context)*, *onDetach()*. When attached, a fragment will follow the evolution of the state of the corresponding activity. Notifications will be received either immediately before or after the corresponding activity ones: **onCreate(b: Bundle), onStart(), onResume(), onPause(), onStop(), onDestroy()**. Note that fragments are not notified of activity's restarts. Notifications of a fragment itself: **onCreateView(...), onDestroyView()** require the fragment to create/release its view and **onActivityCreated(b: Bundle)** indicates that the activity has finished its creation.



If a fragment is dynamically added or removed from an activity, the above diagrams change. The *onAttach(...)* method is called when the fragment is added, even if the activity is already in the "Resumed" state. Similarly, for all the other methods.

The two life cycles are only weakly interconnected: the cycle of construction and destruction of a fragment can be completely **contained** in a single phase of the life cycle of the activity.

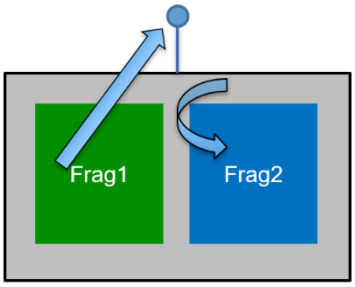


Interaction with the stack of activities

If you dynamically change a fragment, the user perceives the presence of a new screen. It is reasonable to expect that the back button brings back the previous screen, but normally this does not happen. You can tie a transaction on the fragments to an event on the back stack through the **addToBackStack(String)** method. It must be called before closing the transaction.

Activity coordination

A fragment can be displayed depending on what happens in another fragment. The activity that includes them both can act as a **shared controller**. We can define an interface that exposes the possible events. The activity implements this interface and is responsible for invoking the appropriate methods on the different fragments. A fragment can find the **reference to the containing activity** via the **getActivity()** method.



Depending on layouts, the two fragments may be not visible at the same time. It is necessary to verify their presence and act accordingly. All fragments are bound to a **string tag.** *supportFragmentManager* allows you to access each fragment via its Tag (*getFragmentByTag (...)*). If the fragment is not null and is visible, then it is possible to invoke a method on it.

Alternatively, both fragments may refer to the common activity viewModel, and use that object as a way of coordinating. It can be access using the following block of code: **val vm: someViewModel by activityViewModels()**. Care has to be taken not to store anything that has a LifeCycle inside a fragment:

- If one fragment executes an action that requires another fragment to react, the viewModel is updates, which causes a LiveData to change
- The other fragment can observe the changed LiveData and react accordingly
- If it may happen that the second fragment is not currently displayed, the activity shall act as an observer and trigger its visualization

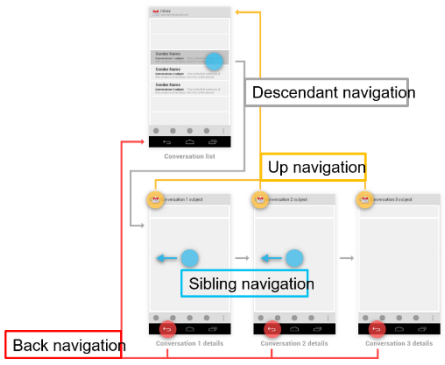
Fragment inheritance hierarchy

Although it is often enough to extend the Fragment class, there are several cases where it is convenient to extend one of its sub-classes to leverage a set of default and well-tested behaviors, avoiding the need to write repetitive code. The sub-classes provide support for *managing lists*, *preferences*, *dialog boxes* and *web views.*

Content navigation

Android supports different forms of navigation: **Temporal (back) navigation**, **Up/descendants navigation**, **Sibling navigation, Drawer based navigation**. Fragments can help in most situations.

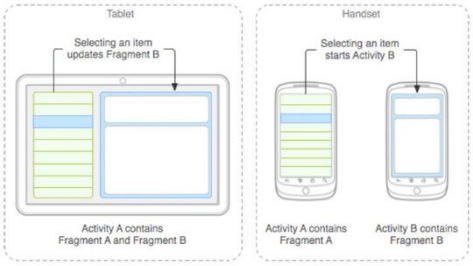
The **back navigation** is pbtained by default, thanks to Activity stacks. It can be changed, by overriding method *onBackPressed()*. Override it **only if it is necessary** to satisfy user expectations. With fragments, there is the possibility to invoke *addToBackStack(…)* in FragmentTransactions.



The **hierarchical navigation** is composed of:

- **Parent screen**: screen that enables navigation down to child screens. Typically, the main activity screen.
- **Collection siblings:** screen enabling navigation to a collection of child screens (e. g., a list)
- **Section sibling:** screen with some content that can be navigated via lateral swipes

The main screen is implemented by a single activity. Child screens can be implemented both as fragments and as activities. Collection siblings screens can be shown as lateral panes on landscape tablets, and with multiple screens on phones.



The **descendant navigation** is introduced by many controls:

- Navigation drawer
- Buttons, image buttons on main screen
- Other clickable views with text and icons arranged in horizontal or vertical rows, or as a grid
- List items on collection screens
- Cards

The Jetpack Navigation library

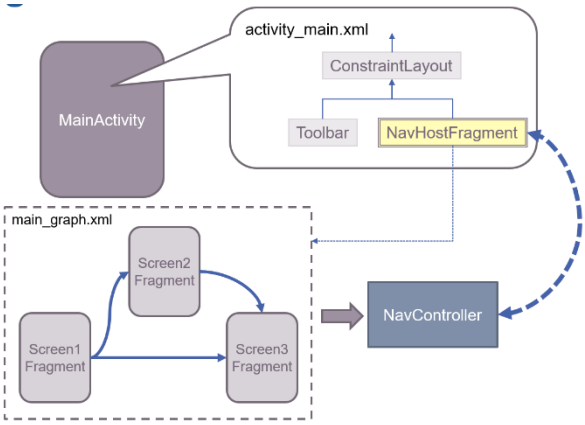
This library offers a compact, flexible, and easy to use solution to handle screen navigation. Two dependencies must be added to the build.gradle file, in order to use it.

Navigation UX principles

Some principles for navigation UX:

- The app should have a fixed starting screen
- A stack is used to represent the “navigation state” of an app
- The Up button never exits your app
- Up and Back are equivalent within your app’s task
- Deep linking to a screen or navigating to the same destination should yield the same stack

Navigation architecture



Navigation is defined in a **declarative way as an XML** resource describing a graph of screens. They are stored in the *res/navigation* sub-folder. It may be edited as a text file or using an ad hoc editor inside Android Studio. Each screen can be a single fragment or a whole activity. Fragment are used for internal navigation while activities for navigation amongst different apps. Activities hosting internal navigation via fragments, rely on a *NavHostFragment* contained in their visual hierarchy. It provides an area within the screen where navigation will occur. It offers a *NavController* instance which is in charge of reading the graph and dynamically instantiate and show screen fragments as navigation occurs.

NavController

Library provided class, in charge of:

- Handling transactions of single screen fragments
- Handling "up" and "back" navigation
- Creating standard input and output animations for fragments
- Implementing and handing deep-links to single screens
- Supporting drawer- and bottom-navigation with a minimal effort by the programmer
- Provide graph-wide ViewModels to simplify information sharing

Navigation editor

It allows the designer to visually describe the navigation graph:

- Screens (both fragments and activities) can be added, removed, changed at will
- Each navigation arc connecting a pair of screens can be labelled with an input animation and an output one
- Deep links may be inserted to allow other apps to access a specific screen

Managing navigation events

Inside each screen fragment a navigation request may be activated. Each **arc of the graph** is **labelled** with a unique ID representing the navigation **action**. Navigation is triggered by method *navigate(…)* offered by NavController. There are several overloaded versions, allowing to specify the navigation action id or the destination id to navigate to.

Accessing the NavController

Fragments offer method *findNavController()* which return an instance of it. NavHostFragment registers its navigation controller at the root of its view subtree. **Any descendant** can obtain the controller instance through Navigation helper class methods *Navigation.findNavController(view: View)*. This frees event listeners that need to trigger a navigation event from being tightly coupled to their navigation host.

Enriching navigation

Some of the overloaded versions of the *navigate(…)* method allow an extra parameter of class Bundle. This can be used to **pass information** to the target fragment. This becomes the value of the arguments property.

The safeArg gradle plug-in

It automatically generates classes that incapsulate arguments to navigate to a given target. Both the main "build.gradle" file (related to the project) and the app module one need to be modified. Generated classes are named out of the starting screen followed by 'Directions' (e.g., LoginFragmentDirections). These classes contain as many methods as outgoing arcs. Their parameters are the mandatory arguments for the given destination.

Introduction to Cloud Firestore

The course project will rely, for its backend on the **Cloud Firestore database**, part of Google Firebase platform. It is a **document based**, **NoSQL backend as a service** (BaaS) owned and supported by Google. It can be used both for small and for large applications. It is a versatile and proven platform with many bindings: Android, iOS, Javascript, REST, …

NoSQL

Broad term that refers to **any DBMS that does not rely on the SQL language** for querying data. NoSQL DBMSs typically provide support for many other ways to store data, not necessarily tables made of simple tuples. It is designed for spreading over several server machines in order to scale both in terms of number of records and in terms of number of concurrent users. Scalability comes at a price: **consistency is eventual**. Data is not immediately updated, and this is consequence of the C,A,P theorem.

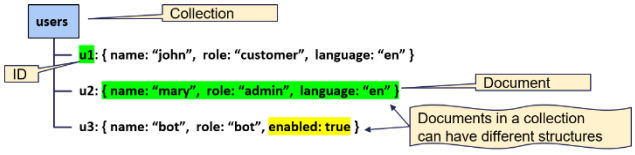
Types of NoSQL DBMS

There are several types of NoSQL DBMS:

- **Key/Value stores**
  - Big distributed hash-table
  - Each data item has a unique key
  - Actual data is not understood by the server
  - Examples: Redis, Riak
- **Document stores**
  - Similarly to Key/Value, each data item has a unique key
  - Data is a structured document, often represented in JSON format
  - Examples: MongoDB, CouchDB
- **Column data stores**
  - Single records represent data columns, not data rows as in relational DBs
  - This approach simplifies data aggregation
  - Examples: Cassandra, Hbase
- **Graph stores**
  - Data is organized in graphs, using nodes and links
  - Examples: Neo4J, HyperGraphDB, JanusGraph

Cloud Firestore architecture

Firestore is a **schema-less document store**, which means that data is organized in **collections** and **documents**. Every collection has a **name** and may contain zero or more documents. A document consists of a **JSON data structure** having textual keys and boolean, numerical, textual, temporal, geographical values. Values can also consist of list of values, sub-objects (maps), or references to other objects (caution!). Documents can contain sub-collection, too. A document is limited to 1MB in size, whatever its internal structure. Every document has an **ID** assigned by the system or by the programmer. The ID is unique within the collection where the document is stored and is part of its path.



Expressive querying

Firestore allows to retrieve individual documents inside a collection provided their id is known or to retrieve all documents of the collection that match a set of constraints. Queries can contain **multiple filters** and **sorting criteria**. Thanks to **indexing**, query performance depends on the result set size and not on the data set size.

Realtime updates

Firestore sets up a **data synchronization channel** between the database and each connected device. Whenever the data set changes, a notification is sent to each device in order to synchronize. Queries can be expressed as **one-time fetch only**, in order to improve overall efficiency. Queried data is **cached automatically**, thus allowing offline operations. Both read and write access is possible when the device is offline: as soon as it gets connected again, data is synchronized again with the main store. A document consists of named pieces of data.

```
{
  "title" : "Mobile Application Development",
  "groups" : {
    "g1" : [ "name1", "name2", "name3", "name4" ],
    "g2" : [ "name5", "name6", "name7", "name8" ]
  },
  "tableOfContents" : ["Kotlin", "Android", "Firestore"
        "User Centered Design", "Hybrid Apps" ],
  "year" : 2022
}
```

Collections

A complete database can be as simple as a single document or it may contain thousands of **collections** each storing a huge amount of documents. Document may have any structure and a single collection can contain heterogenous documents but, usually, a more conservative approach is chosen, putting similar documents inside single collections. Every document in Cloud Firestore is uniquely identified by its **location within the database.** A reference is a lightweight object that points to a location in the database, **whether it exists or not:**

- *val mad =db.collection("courses").document("mad")*
- *val alwaysMad = db.document("courses/mad")*

Sub-collections

A given document might contain many **sub-items**, although it is possible to store them inside the document itself, usually this is not a good idea. Single documents have size limitations: they must be smaller than 1Mbyte, have no more than 20,000 properties and have no more than 20 level deep nesting. A sub-collection is a collection associated with a specific document. References to items in sub-collections can be created easily:

val student1 = db.collection("courses").document("mad").collection("students").document("student1")

Sub-collections can be nested up to 100 levels deep.

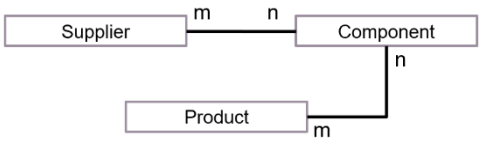
Data structure comparison

- **Nested data in documents**
  - Good when simple, fixed list of data need to be kept together, in order to speed up retrieval
  - If data becomes large, load time increases, and upper limits may be reached
- **Sub-collections**
  - Allow easy querying of sub-items and let the parent document size unaffected by the number of items
  - Difficult to delete: if the parent document is deleted, the sub-collection stays there
- **Root-level collections**
  - Extremely flexible and scalable
  - Difficult to manage relationships between items belonging to sibling collections

Modelling relationships

If a given data item is related to other items, two modelling approaches are possible:

- Referencing data in other collections
- De-normalising data



Referencing data

A document reference is a kind of URL. As such, it can be stored in the source document in order to allow the retrieval of the target document. A **new collection must be created to implement many-to-many relationships**. Its documents contain references to two related parties. Not having columns, **Firestore does not support column joins**. Two requests need to be issued: one for getting the set of relationship documents, the second to get the related items. The first request implies a search (an index on each referencing field need to be built). Firestore **does not support cascading**. If one of the documents is deleted, the other keeps on referencing it, **no way to propagate automatically the deletion**.

De-normalising data

As an alternative, a data duplication may be introduced. Products, components, and suppliers can be stored in separate collections. Each supplier can have an array field listing the set of components it produces, and each product has an array field with the set of components it is made of. Each component has two array fields: one listing the set of possible suppliers, the other listing the set of products that use it.

De-normalization introduces redundancy and possible conflicts. What happens if only one side of the relationship is updated? If two pieces of information that should be equal are different, which one must be trusted?

The Firebase console

Allows the programmer to create, configure, and modify the structure, the access rules, and the behaviour of a Firebase-powered application. A Google account is required. It is possible to manage authentication rules, data-storage, back-end functions, and machine learning.

Cloud Firestore and Android

A complete library is available for accessing a Cloud Firestore database from an Android application. It allows programmer to query and update data in a seamless way. At the end of the process, a *google-services.json* file will be generated and put inside the android project.

Accessing Firestore

All operations involving the database start with an object of class FirebaseFirestore

val db = FirebaseFirestore.getInstance()

Typically stored as an activity property or a property of a singleton object. The database configuration is automatically derived from the *google-services.json* file. References to collections and objects are created via

- val coll1 = db.collection("courses")
- val doc1 = db.collection("courses").document("mad")
- val doc2 = db.document("courses/mad")

All reading and writing operations are **asynchronous.** Callbacks are used to get results or failure indications.

SKIPPED SLIDES FROM 22 TO 25 CAP A09

Kotlin type mapping

Each elementary value stored in the Firestore JSON object is mapped to the corresponding Kotlin class:

- JSON numbers becomes either Long or Double instances, according to their decimal part
- JSON strings are mapped to Kotlin Strings
- Dates are mapped to Timestamp
- Boolean literals are mapped to Boolean instances

- JSON objects are mapped to Map<String, Any>
- JSON lists are mapped to List<Any>

The **toObject(c: Class)** method can be used to have the returned data mapped to a data class.

Listening to real time updates

It is possible to subscribe to a single document to intercept any change it may undergo. A SnapshotListener is invoked as soon as the document is subscribed to and whenever it changes. Each time it is invoked, the listener will receive a copy of the most recent document content. If a query is specified, it is possible to detect any change to the result set (e.g., a document is added, removed, or modified)

Deleting a listener

If the app is no more interested in subscribed data (e.g., because the activity/fragment is quitting) the listener should be cancelled. The object returned by *addSnapshotListener(…)* offers a **remove()** method to this purpose. If an error arises, the listener will be invoked with a non-null error argument. After an error, the listener will not receive any more events, and there is no need to detach the listener.

Query limitations

A document or collection reference object offers a set of methods to perform data queries such as *whereEqualsTo(field, value)* or *whereGreaterThan(field, value)* and so on… They all return a **Query object** which can be *read via get()*, *subscribed to*, or further constrained via where methods.

Queries with **range filters** on different fields **are not supported.** Queries with a **!=** clause are **not supported**. The query can be split into a greater-than query and a less-than query. Cloud Firestore provides **limited support for logical OR queries**: the in and array-contains-any operators support a logical OR of up to 10 equality (==) or array-contains conditions on a single field. For other cases, create a separate query for each OR condition and merge the query results in your app

Ordering and limiting data

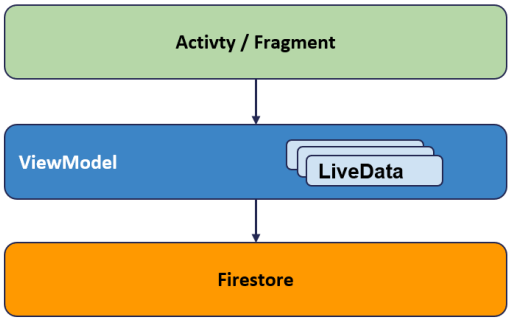
Method *.orderBy(fieldname)* allows to specify the sort order of the retrieved data as well as act as a filter, removing all document that lacks the specified field. By default, ascending order is applied. Method *.limit(size)* sets an upper bound on the number of returned documents. Methods *.startAt(position)* and *.endAt(position)* allows to set a range of results to be retrieved. Similarly to *.startAfter(position)* and *.endBefore(position)*.

Using the MVVM pattern

If an app is backed by Firestore, the MVVP pattern can be largely simplified.

There is **no need** to create a Repository with the corresponding Model and RemoteDataSource, since Firestore provides its own local cache and support offline data persistence.

LiveData objects inside the viewModel can be populated by subscribing to a query snapshot at creation time and removing the subscription when the viewModel is cleared.



Accessing data offline

Cloud Firestore **supports offline data persistence**. Data that is actively used is cached on device and is accessible when offline: read, write, listen and query operations are supported. When the device is again online, local changes are synchronized with the remote backend. Offline persistence is **enabled by default**. The FirebaseFirestoreSetting.Builder() class can be used to disable it or to set the cache size.

Indexing data

Query performance can be increased creating **indices**. This is mandatory for making compound queries with ranges. Indices are created and destroyed from the Firebase console or from the Firebase command line interface (CLI). Indexes can take a **few minutes to build**, depending on the size of the query and are subject to some limitations (200 max composite indices per db and some other constraints).

Concurrent modifications

Multi-user applications must deal with **concurrent data updates** (e.g., up-votes on a blog post). Firebase supports some **ready-made operations** (*increment*, *arrayRemove*, *arrayUnion*). A **transaction** is a set of read and write operations on one or more documents executed atomically. A **batched write** is a set of write operations on one or more documents executed atomically.

Transactions

Transaction operations are **described as a pure lambda function**, with no side-effects. An **optimistic locking** approach can cause the function to be executed several times, until all its pre-conditions are met. Read operations must come before write ones. Transaction will fail if the device is offline. A value can be returned by the lambda to provide the transaction outcome. Any exception thrown inside the lambda, causes the transaction to roll-back.

Security

By default, Read/Write operations need the user **to be authenticated**. Rules are specified in a Domain Specific Language (DLS) in the Firebase Console in the "Rules" tab. It is possible to have different rules for different Collections/Objects.

Cloud Firestore Security Rules always begin with the following declaration.

Basic rules consist of a **match** statement specifying a document path and an **allow** expression. A **read** clause can be split into **get** (a single document) and **list** (a collection or a query). **Write** clauses can be split into **create** (a non-existing document), **update** (an existing document) and **delete.**

```
service cloud.firestore {
  match /databases/{database}/documents {

    match /courses/{course} {
      allow read: if <condition>;
      allow write: if <condition>;
    }
  }
}
```

SKIPPED CAP A10A (BASIC WIDGETS)

Material Design

Material Design is:

- **Design guidelines**
- **Visual language**
- Combine classic principles of good design with innovation and possibilities of technology and science

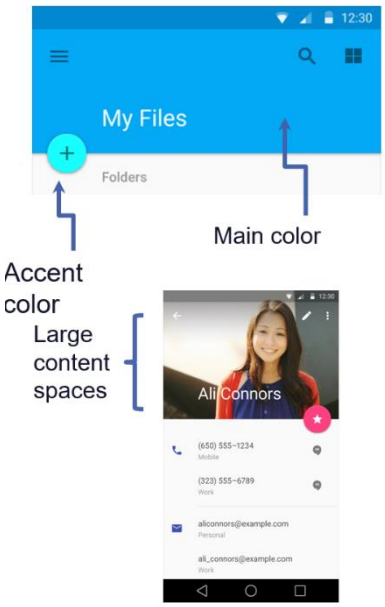
In June 2014 Android proposed a new style for mobile interfaces which deepens flat design principles: *minimalist, consistent, rationale*.

Material metaphor:

- Three-dimensional environment containing light, material and shadows
- Surfaces and edges provide visual cues grounded in reality
- Fundamentals of light, surface and movement convey how objects move, interact and exist in space and in relation to each other

Elements in your Android app should behave similarly to real world materials:

- Occupy space
- Cast shadows
- Interact with each other



Some guidelines

- Choose colors deliberately
- Fill screen edge to edge
- Use large-scale typography
- Use white space intentionally
- Emphasize user action
- Make functionality obvious

Imagery

Images help you communicate and differentiate your app. They should be:

- Relevant
- Informative (rational meaning)
- Delightful

Best practices are:

- Use together with text
- Choose original images
- Provide point of focus
- Build a narrative

Typography

Roboto is the standard typeface on Android. It is present in different flavours: thin, light, regular, medium, bold, black. Regarding font styles and scale:

- Too many sizes is confusing and looks bad
- Adopt a limited set of sizes that work well together

Setting text appearance

Fonts as resources:

- Bundle fonts as resources in app package (APK)
- Create font folder within res, add font XML file to font
- Reference fonts as @font/myfont in layouts, R.font.myfont in code
- Use the Jetpack library androidx.appcompat:appcompat:1.4.1

Downloadable fonts:

- Reduces APK size
- Increases the app installation success rate
- Improves the overall system health, saves cellular data, phone memory, and disk space

Colors

Material Design recommends using **a primary color**, **along with some shades and an accent color**. Android Studio creates a color palette for you with the **AppTheme definition** in *styles.xml*.

- **colorPrimary** - AppBar, branding
- **colorPrimaryDark** - status bar, contrast
- **colorAccent** - draw user attention, switches, FAB

Colors are defined in *colors.xml*.

Text color and contrast

- Contrast for visual separation
- Contrast for readability
- Contrast for accessibility
- Not all people see colors the same
- Theme handles text by default
    - *Theme.AppCompat.Light* - text will be near black
    - *Theme.AppCompat.Light.DarkActionBar* - text near white

Motion

Motion in Material Design describes **spatial relationships**, **functionality** and **intention**. Motion should be: **responsive, natural**, **aware**, **intentional**.

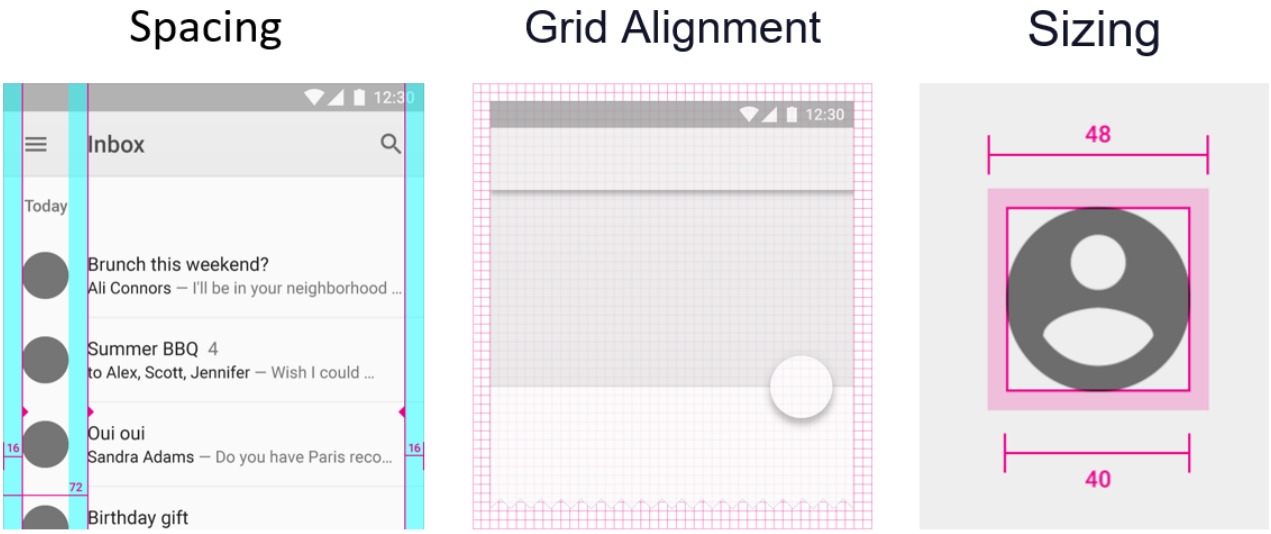
Motion in app allow to:

- Maintain continuity
- Highlight elements or actions
- Transition naturally between actions or states
- Draw focus
- Organize transitions
- Responsive feedback

Layout for Material Design

- Density independent pixels for views - dp
- Scalable pixels for text - sp
- Elements align to a grid with consistent spacing
- Plan your layout
- Use templates for common layout patterns

Layout planning



Material design

New classes and abstractions have been introduced in the library to support this style. Most features are available as compatible classes with previous Android versions. In order to properly support material design in all versions, some work has been done. Material Design is a drop-in replacement for Android's Design Support Library (almost all visual components).

Themes

"A **style** is a collection of properties that specify the look and format for a View or window. A **theme** is a style applied to an entire Activity or application, rather than an individual View"

Customize the colors

- In *values/styles.xml* you can find the theme of your app
- In *values/colors.xml* you can find the colors of your app (colorPrimary, colorPrimaryDark, ColorAccent)

Toolbar

A Toolbar is a **generalization** of Action Bar (more flexible version). Action Bar can be replaced with a Toolbar. It inherits from ViewGroup and can have Children Views as well. Can be placed anywhere on screen (top/bottom).

A toolbar can have:

- Navigation button: Up Arrow, Menu Toggle, …
- App Icon
- Title/subtitle
- One ore more custom views
- An action menu
    - Consisting of frequent/important/typical actions along with an optional overflow menu for additional items

SKIPPED SLIDE 24 CAP A10B

Cards

Cards contain content and actions about a single subject. The basic implementation is in androidx.cardview.widget.CardView. Specialization com.google.android.material.card.MaterialCardView.

Class CardView extends FrameLayout. It's a ViewGroup: children may be added to it. MaterialCardView is an extended version of CardView. Besides all the features of CardView, it **adds attributes for customizing the stroke** and uses an **updated Material style** by default. It can be checked and dragged.

Material buttons

It is a customizable button component with updated visual styles. It is made of a text label, a container and an optional icon.

- **Text button**: background-less buttons used for less-pronounced actions
- **Outlined button: a** medium-emphasis button that contains an action that is important, but isn't the primary one
- **Contained button:** a high-emphasis button, distinguished by its use of elevation and fill, default style. It typically triggers the primary action in the app.
- **Toggle button: c**an be used to select from a group of choices. To emphasize groups of related toggle buttons, a group should share a common container. Toggle buttons may consist of text or be icon only.

Floating action button

A FloatingActionButton displays the **primary action** in an application. It is a round icon button that's elevated above other page content. They provide quick access to important or common actions within an app:

- Performing a common action
- Displaying additional related actions
- Update or transforming into other UI elements on the screen

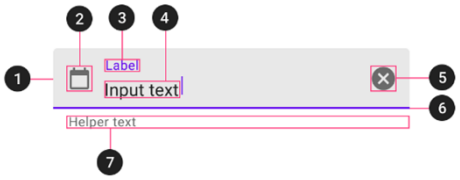
It is typically used as a child of a CoordinatorLayout. Use the *show()* and *hide()* methods to animate the visibility of a FAB. The *show()* animation grows the widget and fades it, while the *hide()* animation shrinks it and fades it out. The size of a FAB is, by default, **responsive to the size of the window.** Custom dimensions are supported via the app:fabCustomSize attribute.

Text fields

Text fields let users enter end edit text and come in two flavours: filled or outlined. Text fields support rich formatting. They are created as a combination of a *TextInputLayout* directly wrapping a *TextInputEditText* component.

The TextInputLayout (1) offers the following attributes:

- Leading icon (2)
- Label (3)
- Trailing icon (5)
- Activation indicator (6)
- Helper/error/counter text (7)
- Custom prefix/suffix/placeholder (not shown)



The TextInputEditText (4) extends the EditText. Classical attributes apply here.

Snackbars

Temporary widgets that provides quick feedback messages to the bottom of the screen. They disappear automatically after a timeout, can be swiped off the screen or a dismissed interacting elsewhere on the screen.

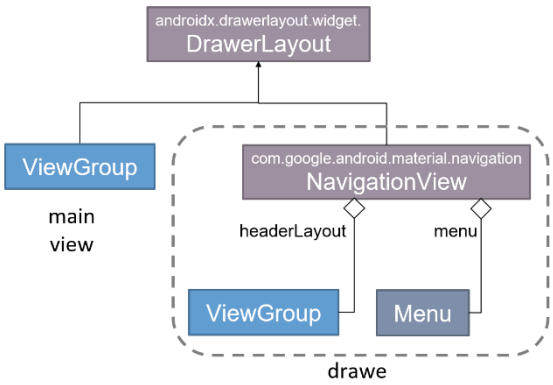
A snackbar may be associated with an action. It is typically used for supporting retring or undoing another action. They work best when displayed inside a CoordinatorLayout. This enables the swipe-to-dismiss behaviour and automatically move widgets like FABs.

Chips

A chip represents a complex entity in a small block: a contact, a keyword, a tag, … It consists of a label, an optional chip icon and an optional close icon. It can either be clicked or toggled, the latter only if it is checkable.

Navigation drawer

It is a combination of a *DrawerLayout* enclosing a layout manager and a *NavigationView*. The former wraps whatever content has to shown on the page, the latter creates a modal elevated dialog used to display in-app navigation links.



The NavigationView is made of:

- A headerLayout, typically conveying some graphics
- A menu, where each item represents a target view to be shown in the layout manager

SKIPPED SLIDE 42 OF CAP A10B.

Supporting navigation

The main content area can be managed by the Jetpack Navigation library. The main area may directly contain the *NavHostFragment* or can be based on a more elaborate scheme (CoordinatorLayout including a Toolbar, the NavHostFragment, and possibly a FAB). The NavigationView can directly operate with the NavController via its *.setupWithNavController(…)* method.

Custom views

The **android.view.View** class is the root class of all Android visual components. A View occupies a **rectangular area** on the screen and is responsible for **drawing itself** and for **handling events** that occur inside its area. It can be instantiated directly from the programmer or as the consequence of inflating a layout resource. In order to get access to the app resources, the View class keeps a **reference to the activity that owns it**, known as its **context**. The constructor must be able to process XML attributes, to allow the inflation process to take place.

The View constructors

- **View(context: Context!): s**imple constructor to use when creating a view from code
- **View(context: Context!, attrs: AttributeSet?):** called when inflating a view from XML
- **View(context: Context!, attrs: AttributeSet?, defStyleAttr: Int)**: perform inflation from XML and apply a class-specific base style from a theme attribute
- **View(context: Context!, attrs: AttributeSet?, defStyleAttr: Int, defStyleRes: Int)**: perform inflation from XML and apply a class-specific base style from a theme attribute or style resource.

Create your own view

In order to create a custom view, either the View class should be extended or any of its subclasses: TextView, Button, ViewGroup, LinearLayout, ... Any visual component has three main responsibilities:

- **Negotiate with its container the space** needed to be painted on the screen
- **Layout its own children** in the area received by its container
- **Draw the custom view** inside that area

Moreover, it may introduce its own custom XML attributes in order to be configured in the visual layout editor.

Creating custom XML attributes

Custom attributes may be defined as resources in the *res/values/attrs.xml* file. A <declare-styleable> element must be defined, setting its name to the custom view class name (without any package). Inside it, attributes are listed, with their names and formats.

```
<declare-styleable name="CustomView">
  <attr name="borderColor" format="color" />
  <attr name="borderWidth" format="dimension" />
</declare-styleable>
```

The declared attributes can be used in layouts by prefixing them with the "app:" namespace. For each attribute, a corresponding property need to be introduced in the custom view. When a custom view is constructed, these properties need to be initialized from the constructor parameter representing the XML description.

Saving the view state

Being part of the visual hierarchy also means that the view may be **destroyed** and **recreated** each time that the owning activity does so. The custom view is **responsible of properly saving and restoring its state** by overrding the functions *onSaveInstanceState()* and *onRestoreInstanceState(…)*.

The view lifecycle

When the view hierarchy comes into existence, three things have to happen before it appears on the screen:

- **Measuring it** → *onMeasure(…)*
- **Laying it out** → *onLayout(…)*
- **Drawing it** → *onDraw(…)*

Each of these operations entails a depth-first traversal of the whole view hierarchy from the parent node to children to children's children, all the way down. These operations can be invoked many (!) times along the lifecycle of a view. Their implementation must thus be very efficient.

Measuring a view

A given view may derive the amount of space it needs from several sources:

- The size explicitly assigned to it by the programmer *(layout_width = "100dp")*
- The content it encapsulates *(layout_width = "wrap_content")*
- The size of the containers *(layout_height = "match_parent")*
- A mix and match of the three or any other source of information *(layout_height = "match_constraints")*

Depending on the content that a given view wraps, there may be many possible arrangements of it. Thus the parent container start a kind of "**dialogue**" with each child view, proposing it a given amount of space. The child view will compute, given those constraints, how large it wants to be and then it will store that measure inside a specific attribute.

Negotiating the space

When a view needs some (more) space, it invokes its own *requestLayout()* method. The entire view tree to which the view belongs, should undergo the layout procedure. The Android UI framework puts an event in the **event queue**. When it is processed, the framework allows the container to ask each of his child how much space they need to draw themselves. The process is split into two phases:

- Measuring all child views
- Calculating their sizes and positions

The measure phase

The objective of this phase is to give each view the opportunity to indicate the desired space to draw itself. First, the framework starts the process by invoking the **measure()** method of the root in the view tree. Each container asks each child view the desired space. The call is **propagated** to all the descendants, depth-first, so that all views have the opportunity to propose their own size to their container. The container will then take all the child views measurements and decide how large it needs to be.

The measure() method is **final** and **cannot be overwritten**. Internally, it calls *onMeasure(int, int)*, which must be overridden by the views that **wish to implement their own space management** procedure. In order to converge to a feasible solution, the method *onMeasure(...)* can be called several times.

onMeasure(w: Int, h:Int) arguments indicate how much space (width and height) the parent view offers to its children. The offered size can range between 0 and 2^{30} pixels. The two highest bits of each parameter are used to encode the specification mode.

To extract from the parameters the actual sizes in pixel and the measurement specification modes, use static methods

- *MeasureSpec.getSize(val: Int)*
- *MeasureSpec.getMode(val: Int)*

The measurement specification mode describes how to interpret the provided values:

- *MeasureSpec.EXACTLY*: the parent has determined an exact size for the child
- *MeasureSpec.AT_MOST*: the child can be as large as it wants up to the specified size
- *MeasureSpec.UNSPECIFIED*: the parent has not imposed any constraint on the child

A child has to indicate to its parent **how much space it needs** via the method *setMeasuredDimensions(...)* that allows to set the desired width and height. The parent can retrieve these values via the *measuredWidth* and *measuredHeight* properties. If you override onMeasure(...) but you do not call setMeasuredDimensions(...), the Android system launches an IllegalStateException exception.

The base implementation of onMeasure is the following:

- If the parent specifies *MeasureSpec.UNSPECIFIED*, the *setMeasuredDimensions(...)* method uses the **default size** of the view, the value of the suggestedMinimumWidth and suggestedMinimumHeight properties.
- If the parent view specifies any other mode, *setMeasuredDimensions(...)* uses the size **proposed by the parent** view

A child view **may not obtain all the requested space**, as the parent view will eventually determine how much space to allocate to each child.

The container can call onMeasure(...) many times. Usually, the process is divided into three sub-phases:

- Method *onMeasure(...)* is called with *MeasureSpec.UNSPECIFIED*: each child view can indicate its preferred size
- Method *onMeasure(...)* is invoked with *MeasureSpec.AT_MOST* or *MeasureSpec.EXACTLY*: specifying an appropriate size with respect to the available space

The onMeasure(...) method implementation must be **idempotent.** Apart setting the measured size, no other modifications should take place in the view object.

The layout step

After negotiating the space, the **parent lays out its children inside the rectangle it owns.** This phase takes place by calling **layout()** on the root of the view tree. This invokes the *onLayout(...)* method and then notifies all the LayoutChangeListeners registred on the view.

The onLayout(…) method is responsible of:

- Calculating the boundaries of the rectangle that will be provided to each child view
- Calling the layout() method for each of them

Given the complexity of laying out sub-views, it might be better to leverage on proven algorithms as those provided by existing containers.

The draw phase

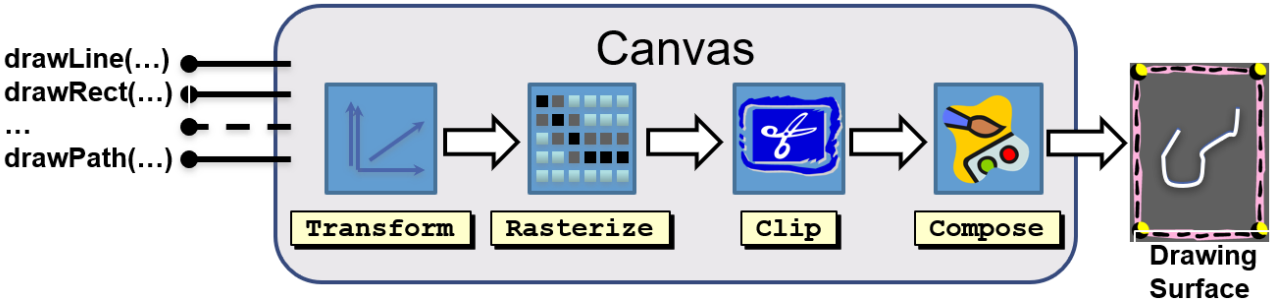
After allocating space for a component, the Android system has to draw it whenever the component acquires space on the screen but also when the *invalidate()* method of the view is called. This causes an event to be inserted in the message queue, which will be later processed.

When the event is processed, the system calls the *draw(...)* method on the root of the view tree:

- This draws the background, first, if not null
- Then it calls onDraw(...). By overriding this method, you can specify your drawing logic
- Then it invokes dispatchDraw(...). If this view contains any child views, these are drawn recursively, in the order in which they were added
- Finally, it draws the scrollbars if any, by invoking onDrawScrollbars(...)

Drawing a view

The **onDraw(…)** method receives as a parameter a **Canvas object** that provides access to the two-dimensional graphics pipeline. Internally, the Canvas stores different attributes that define its behaviour and it also embeds a reference to the surface on which the drawing process will take place.



Canvas operations

The methods of the Canvas class draw different graphical primitives on the drawing surface: segments, arcs, rectangles, circles, paths, images, texts, ... Some of the graphics pipeline operations are driven by the attributes of the Canvas instance (like matrix and clip). Once set, they keep their state between invocations. Others are driven by specific parameters supplied as arguments of each drawing primitive via instances of the **Paint class**.

Rendering of graphic primitives

Most of the drawXXX(...) methods offered by the Canvas class requires a Paint parameter. It controls the graphics primitives rendering, providing the necessary parameters for the rasterization and composition done within the pipeline.

A Paint object defines:

- The color or the pattern used for painting
- The style of rendering (only the edge, just inside, edge and interior)
- The fonts and attributes to use for texts
- The parameters for the joints of the paths
- Any special effect to apply

Drawing lines

A line is transformed during rasterization in a two-dimensional surface with a thickness controlled by the *strokeWidth* property of the Paint object. The edges of the line can be rounded or truncated, as defined by the *strokeCap* property.

The vertices of a broken line may be connected in different ways as defined by *strokeJoin* and *strokeMiter*. If a PathEffect instance has been set, this is applied on the path before rendering the individual lines. For example, any CornerPathEffect blunt corners in circular arcs.

Drawing an area

If the style of a Paint object is set to *FILL* or *FILL_AND_STROKE*, any internal area of its contour is filled by using the color associated with the Paint. The definition of what is inside is derived from the single primitive that is drawn (*fillType* property of Path). The color applied to each pixel is the one contained in the Paint or that obtained from the *Shader object* stored in the Paint.

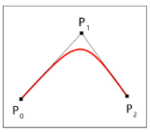
Drawing a text

The Paint class provides mechanisms to manage the presentation of text on a canvas, such as *setTypeface(...)* that specifies the font or setTextSize(...) that indicates the height, and so on.

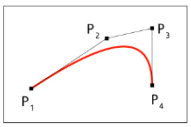
The Path class

It is the abstract vector representation of a geometric shape (open or closed) composed of

- Line segments, polylines, closed polygonal
- Bezier curves, quadratic and/or cubic
- Rectangles (eventually rounded), ellipses and arcs of ellipse
- Other paths



Quadratic Bezier



Cubic Bezier

A path is created by **repeatedly adding geometrical primitives**. The first operation is usually *moveTo(…),* to set the path starting point. Following operations can add simple consecutive primitives or can start a new contour. Constructive geometry is supported via the *op(…)* method, which combines it with another path, by adding, subtracting, intersecting or xoring.

PROF SKIPPED FROM SLIDE 34 TO 52 CAP A11

Handling touch events

Android allows to manage (single or multiple) touch events. The multi-touch management is available from version 2.2 on devices with capacitive touch screens. Use the *onTouchEvent(...)* callback to locate the movements in a view. A **MotionEvent object** is provided as a parameter: it describes a micro-interaction occurred within the view, that may originate from any input device (mouse, pen, finger, trackball).

Alternatively, to detect single and multi-touch gestures you can use the following classes:

- GestureDetectorCompat
- ScaleGestureDetectorCompat

The property "actionMasked" of MotionEvent indicates the evolution of a gesture, hiding details related to the device:

- *ACTION_DOWN*: a pressed gesture did begin (the first finger has come in contact with the screen)
- *ACTION_POINTER_DOWN*: another pointer is added to the gesture (an extra finger is touching the screen)
- *ACTION_MOVE*: a pointer moved while being active (a finger was dragged but is still in contact with the screen) – many events of this kind can be batched
- *ACTION_POINTER_UP*: a non primary pointer is not part of the gesture any more (a finger has lost contact with the screen, but others have not)
- *ACTION_UP*: a pressed gesture has finished (the last finger has lost contact with the screen)
- *ACTION_CANCEL*: the gesture was canceled

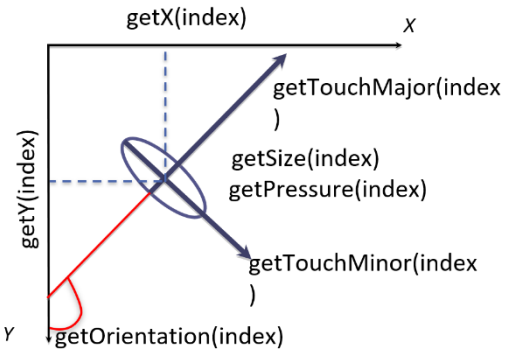
The input subsystem can be slow:

- Several movements of a single finger can happen between two subsequent calls to the onTouchEvent(...) method
- The number of movements that have occurred between the event and the previous one is stored in historySize

The overall number of touches reported in the event is stored in **pointerCount**. The index of the pointer to which the actionMasked is referring to is given by **actionIndex**.

Touch events

Each touch is modelled as an ellipse labelled with a unique ID. The ID is stable as long as the finger touches the screen. *getPointerId(int index)* returns the ID of the indexth active touch. It is possible to query whether a given ID is currently reported via *findPointerIndex(int ID)*. The parameters of a given touch event may be accessed using several methods: getX(), getY(), getOrientation(), getSize(), … These methods require the index of the touch as a parameter.



Consistency guarantees

Motion events are always delivered to views as a consistent stream of events. What constitutes a consistent stream varies depending on the type of device. For touch events, consistency implies that **pointers go down one at a time**, **move around as a group** and then **go up one at a time** or are cancelled.

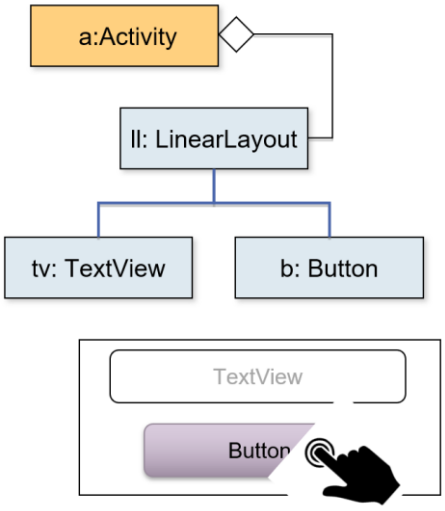
Since events can be dropped or modified by application code, inconsistencies may show in the stream **perceived** by a single view. Views should be prepared to handle *ACTION_CANCEL* and tolerate anomalous situations.

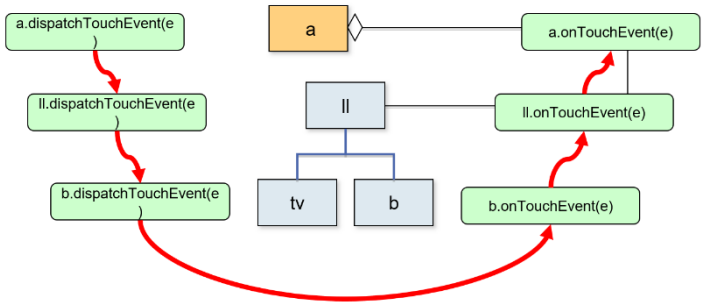
The delivery process

Since views are organized hierarchically in the view tree, Android performs a **depth-first search** to determine if and by whom a given event should be processed based on the value returned by the *onTouchEvent(...)* method when events of type *ACTION_DOWN* are reported.

The event is first notified to the current activity via the *dispatchTouchEvent(...)* method and that forwards it to the root of the view tree.

So, the tree is first visited downwards invoking *dispatchTouchEvent(…)*. Then it is visited upwards calling *onTouchEvent(...)*. If onTouchEvent(...) returns true, the component is selected as the handler of the event and the upward visit is terminated.





If, for an event of type *ACTION_DOWN*, none of the asked components returns true the following events related to the current gesture will be delivered exclusively to the activity and not to the view tree. If a component returns true from the *onTouchEvent(...)* method, its sub-views will not receive any notification. In addition, the following events of type ACTION_MOVE and ACTION_UP will only be delivered to that view.

To allow ViewGroup objects to intercept events aimed at their children, the dispatchTouchEvent(...) method calls onInterceptTouchEvent(...) in order to block the delivery process, the method has to return true. In this case, child views will receive an event marked with ACTION_CANCEL to reset their state.

Jetpack Compose

The limitations of the standard approach

Tree view code is split among several files:

- A layout resource file where the visual hierarchy is defined
- An activity/fragment that hosts it and decorates it with event handlers and state management
- Other resource files (colors.xml, string.xml, themes.xml) that further specify its visual appearance

State ownership and event handling are **intermixed,** which means that each view can choose how to implement these two responsibilities. Furthermore, **data flow is not specified**. How information is propagated from one view where an event is detected to sibling and parent view is a matter of choice of the programmer.

The Jetpack Compose library

A modern toolkit aimed at simplifying UI development inspired by React, Flutter, and Swift and based on a reactive programming model. It is based on a set of **@Composable** functions. These are functions that can call other plain- and / or @Composable-functions, in order to create the visual tree. The library tracks the arguments passed to a composable, and whenever they change, re-invokes it, updating the UI accordingly.

It is **fully declarative** and **data driven**. The programmer is no more responsible of mutating the user interface to track changes in application state. Current state is mapped into component descriptions and the framework takes care of altering the tree-view whenever necessary.

Basic principles

Compose is based on a set of **composable functions.** They represent the building block used to create user interfaces. Each function describes, with a **declarative syntax**, the **aspect** and **behaviour** of the GUI to be built, rather than imperatively drive the construction process.

A composable function is a Kotlin function labelled with the **@Composable annotation**. They use the same syntax as standard functions and can receive parameters of all kinds. They **do not return a value** (their result must be Unit), but they emit **user interface blocks** to be rendered by the Compose runtime, by invoking other composable functions. The library provides a set of built-in elementary functions that emit either basic widgets or layouts, that allows to group other composables.

Composable can be **stateless** or **stateful:**

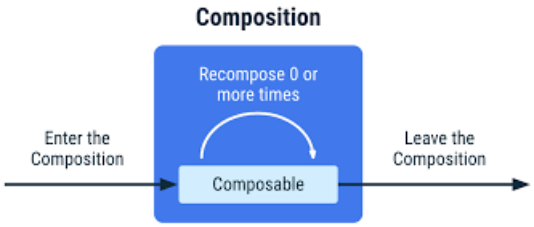
- Stateless composables emissisons are driven only by the parameters they receive: if the function is invoked with the same set of parameters, it will produce the same result.
- Stateful composables encapsulate mutable values which affect both themselves and the composables they invoke: when the values change, a process named recomposition is triggered, possibly re-emitting different interface blocks.

Built-in composables are categorized as either **Foundation**, **Layout**, or **Material design** ones. User-defined composable leverages on them to build higher-level building blocks, up to defining the entire view tree.

Composition and recomposition

In order to create the UI, the Compose framework invokes a composable function and records the set of built-in composables that it invokes, together with their nesting. A tree view is then built according to the outcoming hierarchy. This phase is called **initial composition**. When a value that was passed as an argument to a composable function changes, the composable function is executed again. The effects of its re-invokation replace the portion of view tree that it previously generated. This phase is called **recomposition.**

Recomposition skips as many composable functions and lambdas as possible, based on what parameter changed. Moreover, it is **optimistic**, expecting input parameters not to change before the recomposition phase ends. In case a change occur while a recomposition is ongoing, the recomposition is **cancelled**, the provisional view tree is discarded and the process is started again.



Composable are pure functions

Although composables look like "**standard**" Kotlin functions, in fact they **are not**. Composable functions are transformed by the compiler into an **expanded version**, providing an explicit composition context. Composable functions can execute in **any order**:

- Compose may take advantage of available cores by **running in parallel** sibling composables
- If a composable function calls a function on a ViewModel, the framework might call that function from several threads at the same time
- To ensure that an application behaves correctly, all composable functions should have **no side-effects.**

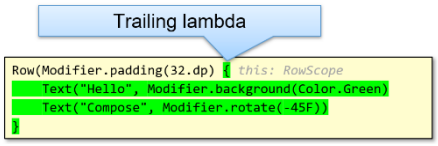
Side-effects can only be produced by call-back functions, like onClick(…), that always execute in the UI thread.

SKIPPED FROM SLIDE 11 TO 16 CAP A12

Compose DSL

```
Text(
    text = "Hello Compose",
    modifier = Modifier.background(Color.Green)
                       .padding(16.dp)
)
```

The body of a composable function typically contains invocations of other composables functions. They cause the emission of the corresponding components, i.e., a new group is inserted in the current composition. Most composable functions accept, amongst their parameters, a **Modifier**. It provides **decorations** and **behaviour** to the element, offering a fluent API which makes it easy to specify several cascading details. The order of modifiers matters!



When a composable offers, amongst its parameters, a **trailing lambda**, this will trigger a **hierarchical composition**. Composable contained in the trailing lambda will become children of the enclosing composable and depending on the enclosing element, some extra functionalities can be available inside the lambda (it may be invoked with a receiver).

Foundation composables

Set of components that provide the basic building blocks to create a graphical user interface:

- **Canvas** – Displays custom drawing on the screen
- **Image** – Display an image, both bitmap and vector
- **LazyColumn** / **LazyRow** – Scrolling lists that only compose and lay out currently visible items
- **LazyHorizontalGrid**, **LazyVerticalGrid** – Similar to above, organized in a grid layout
- **BasicText**, **BasicTextField** - Basic elements that display/edit text

Layout composables

Basic set of composable functions that allow laying out other composables onto the screen, implementing different strategies:

- **Box** – A layout composable with content: similarly to FrameLayout, it lays out contained elements stacked one on top of the other in the composition order

- **BoxWithConstraints** – A composable that defines its own content according to the available space, based on the incoming constraints or the current layout direction
- **Column / Row** – Composables that place children in a vertical / horizontal sequence: similarly to LinearLayout, they can assign the height/width of contained elements according to their weight

Material design composables

This is the higher-level entry point of Compose, designed to provide components that match those described at material.io. It provides a rich set of interactive building blocks that support the creation of rich user interfaces. All elements in this group can be styled via Material Theming, a systematic way to customize colors, typography and shapes to reflect a product's brand. A special composable, **MaterialTheme(…),** allows to set these features, according to the designer's will, providing an easy support for both light and dark variations of the theme.

The Scaffold composable

Component that implements the basic material design visual layout structure, managing four sub-components:

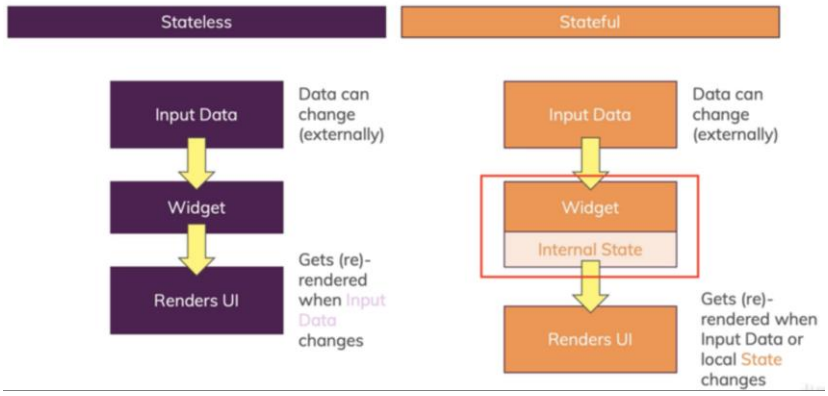
```
Scaffold( topBar = { MyTopBar() }, bottomBar = { MyBottomBar() },
          drawerContent = { MyDrawer() }, floatingActionButton = { MyFAB() } ) {

     //Here goes the main page content
}
```

- Top App Bar (a.k.a. Toolbar)
- Floating Action Button
- Drawer Menu
- Bottom Navigation

Each of these sub-components is expressed via a composable lambda.

Stateless vs stateful



State handling in composables

Composable functions react to changes in parameters or in State<T> values they have subscribed to. The **remember** function is used to **store one or more objects in memory** during the first composition: if a recomposition occurs, the stored value(s) is returned. When the composable that has invoked remember is removed by the composition, the value is forgotten. If the value stored by remember changes, all composable that have accessed it are scheduled for recomposition. This allows creating local state inside a composable, making the composable stateful. There are three equivalent ways to declare a MutableState in a composable:

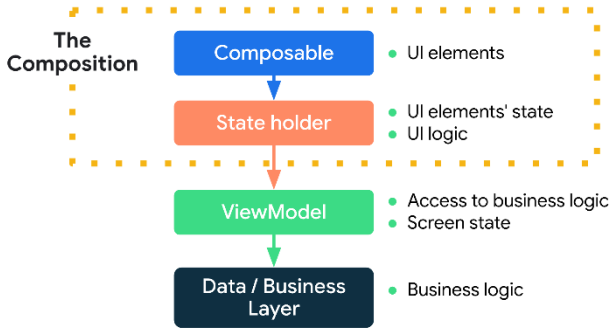
- *val mutableState = remember { mutableStateOf(initialValue) }*
- *var value **by** remember { mutableStateOf(initialValue) }*
- *val (value, setValue) = remember { mutableStateOf(initialValue) }*

The remembered value can be used as a parameter for other composables or can be used to trigger conditional composition. Changes to MutableState should **only occur in the main thread**, typically inside callbacks.

State<T> is an **observable** type part of the Compose runtime that encapsulates a read only, observable, value. Its subtype, MutableState<T> allows the value to change.

```
interface MutableState<T> : State<T> {
    override var value: T
}
```

Other observable data, like *LiveData<T>, StateFlow<T>, Flowable<T>, Single<T>, Maybe<T>, …*, can be converted into State<T> via extension functions. This makes it easy to hoist state outside of composables, making them stateless. ViewModels can hold state and provide access to the business logic.



Navigation

Jetpack Compose is fully integrated with Jetpack Navigation, making the usage of fragments and activities obsolete, since each screen can be described as a composable function. A proper dependency must be added to the project. The built-in *rememberNavController()* function creates a **NavController**. This is a **stateful object** that keeps track of the state of each screen that are part of the application and provides methods to navigate from one screen to another. The composable named **NavHost** is driven by the NavController and wraps a **navigation graph**.

The lambda function passed to a NavHost object is used to build the navigation graph. Each entry in introduced by the *composable(…)* method which identify the destination with a route string. To navigate from a composable within the navigation graph, the NavController's *navigate(...)* method is used. It accepts a **destination route** and its behaviour can be customized by attaching a lambda function. Customizations allow to mimic activity launch behaviors.

Multi-threading in Android

When an application is started, the operating system creates a process and, inside it, a "**main**" thread that manages **all application components**, distributing the events related to their life-cycle. This thread creates a **message queue**, **initializes the objects** defined in the manifest-file and then **begins an infinite loop.** It extracts a message from the queue and delivers it to its recipient. This may cause the insertion of new messages at the end of the queue.

Several other threads are spawned automatically from the main one and they are **managed by the OS**. Users never directly interact with them and hardly notice their presence. Their purpose is to support the **execution abstraction** either of the Dalvik/ART virtual machine or of the overall OS component framework. Some JVM-level threads are:

- GC : low priority garbage collection thread
- JDWP: support debugging and memory inspection
- Compiler: provide just in time compilation of code
- ReferenceQueueD, FinalizerDeamon, FinalizerWatchd: support object finalization and memory cleanup

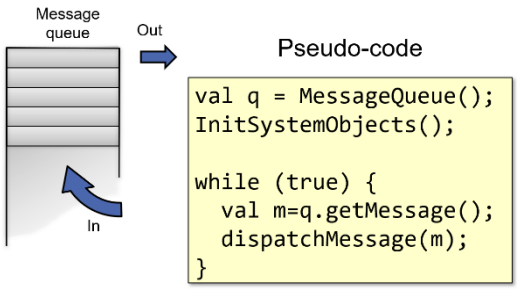
Some android system threads are:

- Binder_X threads: handle incoming intents and other IPC requests
- Signal catcher: handle quit, usr1 and usr2 Linux signals
- GL updater: handle hardware accelerated UI
- hwUITask: handle incoming messages from UI

The Android main thread

The main thread is responsible of:

- **Instantiating the Application and Activity objects**
- **Notifying them the events related to their life cycle**
- **Sending drawing requests to the views**
- **Delivering the events related to user interaction to the corresponding listeners**
- …as well as several other tasks



The access to the queue is **synchronized**. Other threads (including the operating system's ones) may insert new messages **as long as there is no pending dispatching operation**. In this case, the system waits a few seconds, then it forces the termination of the application.

It is necessary that all tasks, carried out in the main thread, terminate in a **short time**, to prevent the displaying of the "Application not responding" dialog box. If you need to perform a long-lasting task, you should either **create a new thread** and delegate to it its execution or use a coroutine, which, in turn, relies on the multi-threading subsystem.

Programming abstractions

User programs may benefit of multithreading as well. Multithreading may be approached using the Java standard abstractions via the **Thread class** and java.util.concurrent.* abstractions (locks, concurrent collections, cyclic barriers, atomic values, …). In order to support the programmer in the development of concurrent programs, Android provides several useful abstractions like **Looper** and **AsyncTask**. The usage of these abstractions is highly encouraged, since they represent the skeleton of the OS infrastructure.

Creating threads

Short running tasks may be implemented creating java.lang.Thread objects by delegating a Runnable object to express the code to be executed. Once started, they run to completion (I.e., until their run() method returns). Thread objects **do not return any value**. In order to know the result of their computation, a value should be stored in some **shared variable**, raising the problem of locking.

SKIPPED SLIDES 11-12 CAP A13

Accessing shared state

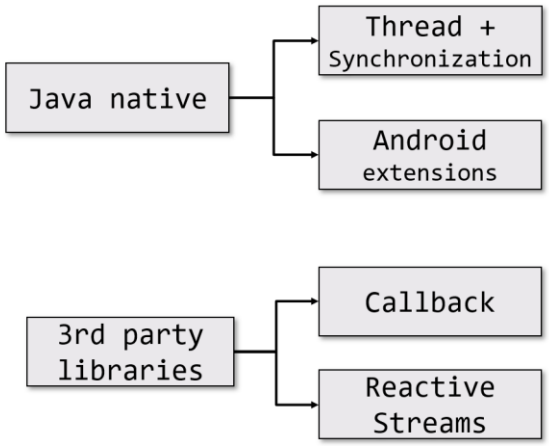
When two (or more) threads need to access the same variable a conflict may arise. Read-Modify-Write operations may lead to **inconsistent values**. A lock can be used to guarantee mutual exclusion between threads but only one thread at a time can hold the lock. If other threads are willing to hold it, they are temporarily suspended, until the one holding it leaves the block.

Problems related to threads

Here is a list of problems related to threads:

- **All threads access the same address space:** in particular, there is only a single heap where objects are allocated.
- If an object reference is known to two or more threads, the object is shared. The threads may communicate storing values in its attributes. This gives rise to **interference**, leading to unexpected results, if no synchronization is performed.
- By default, threads do not provide ways to **asynchronously** communicate among them. Accessing a shared state is the only viable solution. If a thread needs to know whether some state has changed, either polling or blocking waiting is to be used, making it difficult to keep things working efficiently.
- Secondary threads are not aware of the application **lifecycle** nor of any other callbacks received inside the main thread.
- A part of the Android framework is ONLY accessible from the main thread (typically, the View hierarchy). Any attempt to perform operations entailing this kind of objects from a secondary thread will give rise to an exception, terminating the current process.
- Threads are extremely **easy to create**, but **hard to cancel.** The only supported mechanism for thread cancellation relies on **cooperatively polling the interrupted flag** provided by Thread objects, but its semantics is complex and it is subject to a large number of border cases that need manual intervention by the programmer.

Dealing with asynchronous tasks in Android

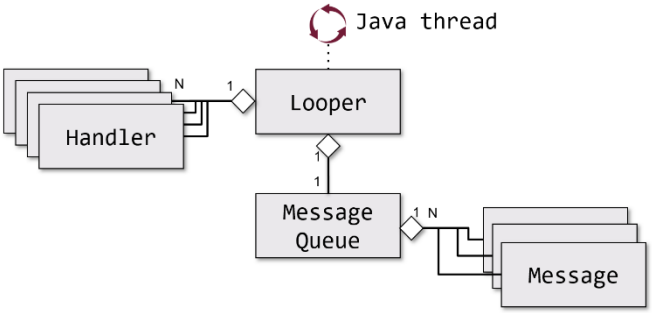


Android threading extensions

- **Looper / Handler / MessageQueue**
  - Mechanism on which all applications are built
  - Robust and powerful
  - Complex
- **AsynchTask**
  - Formerly proposed as a general solution
  - Now deprecated
- **Coroutines**
  - Kotlin-only powerful solution

The low-level Looper API

The **android.os.Looper** is a class that manages a message queue associated with a single thread. The queue is not directly accessible to the programmer, but is it created through the *Looper.prepare()* static method.



The thread that invokes this method becomes the **owner of the queue**. When the Looper.loop() static method is called, an infinite loop starts, waiting for messages from other threads, until a request of quitting is posted. The main thread of an application is built with the following schema.

```
class MyLooper :Thread() {
  private lateinit
     var h : Handler
  private
     val cdl = CountDownLatch(1)
  override fun run(){
      Looper.prepare()
      h = MyHandler()
      cdl.countDown()
      Looper.loop()
  }

  fun getHandler(): Handler {
    cdl.await()
    return h
  }
}
```

Create a queue associated with the current thread

Create a message processor, through which messages and tasks can be sent

Performs a loop that continuously waits (without consuming CPU) for the next message, and then dispatches it

Generally, looper objects are only accessible from their own thread via method *Looper.myLooper()*. Main looper is an exception: it is accessible from **any** thread, via *Looper.getMainLooper()*. Loopers can be terminated via their *quit()* method or via *quitSafely()*.

android.os.MessageQueue

Unbounded linked list of messages to be processed by the consumer thread. Every looper (and thread) has at most **one** MessageQueue. It is possible to retrieve the MessageQueue for the current thread via *Looper.myQueue().*

Handler

The android.os.Handler is a class that provides a **thread-safe interface for inserting and processing of new requests in a looper queue**. An Handler object is bound to the implicit queue associated with the thread in which it was created. The request may be constituted by a **Runnable** or a **Message** object. In the first case, the handler will execute the associated run() method, while in the other case, it will process it via its own *handleMessage(Message m)* method. In both cases, processing will happen inside the associated looper thread.

The requests can be queued immediately (e.g., via post/sendMessage methods) or after a given time interval (e.g., via postDelayed/postAtTime methods). The handler mechanism can be used **to asynchronously communicate with a thread,** provided it is a looper. A looper thread can be associated with many different handlers but they all insert messages in the same queue. When the message will be picked up, it will be processed by the handler through which it was inserted.

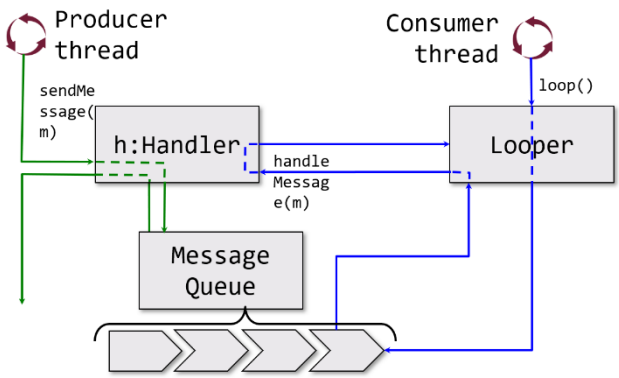
android.os.Message

This class models an event described by an integer value, a pair of (integers) parameters and an eventual object to be delivered to the recipient. The use of integer values makes the use (and reuse) of objects of this type efficient.

Given the allocation and releasing cost of new items, the operating system maintains a queue of messages that are not in use: you get a message via *Message.obtain*(), you can release it via *recycle*().

The producer/consumer pattern

Looper threads naturally lend themselves to implement the producer/consumer pattern. A producer thread **asynchronously generates pieces of information**, storing them in Message objects. Messages are inserted in the **MessageQueue** of the consumer via Handler sendMessage() method. Looper (consumer) thread extracts one message at a time and dispatches it to the handler that was used to post it in order to process it.



SKIPPED SLIDES FROM 32 TO 34 CAP A13

Managing typical problems

Only the main thread can access objects of the graphical interface. If a secondary thread tries to operate on them, the application is terminated with an exception. A mechanism that allows the secondary thread to notify its results to the main thread is required. The main thread will then show them.

Storing a value in an activity attribute is not usually a good solution. The main thread would not notice this change and, unless an external event that causes the redesign of the interface occurs, the view would remain unchanged. It is necessary that a new **event is inserted in the message queue**. When the main thread processes it, it will understand that the secondary thread has finished and what are its results.

Sending messages to the main thread

Android provides five different alternatives, each with its own strengths and weaknesses.

- **activity.runOnUiThread(r: Runnable)**
  - When this method is executed, if the current thread is not the main one, a new message that encapsulates a Runnable object is inserted into the queue. When the message will be processed, the main thread invokes the run() method of its parameter, and the contained code will be executed. It requires the secondary thread to hold a reference to the current ongoing activity.
- **view.postRunnable(r: Runnable)**
  - Inserts the Runnable object in the message queue. It can be called from any thread as long as the view is displayed inside a window
- **view.postDelayed(r: Runnable, l: Long)**

    - Similarly to the previous case, it inserts the object into the queue, but it will not be processed before an interval equal to "l" milliseconds. In this case, too, the view must be connected to a window and be visible.
- Use instances of the Handler class, created in the main thread
    - And **queue messages**
    - Or **forward a Runnable object**

android.os.HandlerThread

It is a simplified looper. It is a Helper class that builds a secondary thread that incorporates a Looper and a MessageQueue.

Handler threads can safely be incorporated into standard components, binding their lifecycle to the component one. The HandlerThread is created in the onCreate() method and quitted in the onDestroy() one.

```
val ht= HandlerThread("HT");
ht.start();
val handler = Handler(ht.looper) {
    when (it.msg) {
       // Process messages here
    }
    return true
}
```

By creating HandlerThread subclasses, it possible to expose convenience methods to simplify custom operations while guaranteeing ordered execution of requested messages.

It is just a “normal” Thread which sets up the internal message passing mechanism automatically. So, there is no need to init a Looper manually. It is applicable to many background execution use cases, where sequential execution and control of the message queue is desired. Using the constructor with the name argument simplifies debugging.

AsyncTask

Abstract and parametric class defining a tasks to be performed in a secondary thread offering at the same time, a set of methods that will be invoked from the main thread in the appropriate time. They are **deprecated** in API level R, in favor of coroutines. Avoid them.

Kotlin coroutines

They are the implementation of a long-time existing concept that allows to simplify writing asynchronous code by **managing long-running tasks that might otherwise block the main thread and cause apps to freeze** and by **providing main-safety, or safely calling network or disk operations from the main thread**.

Coroutines provide:

- Possibility to **express sequential logic**, while coping with asynchronous computation
    - Sequential code is easier to write and to understand: no callback hell, standard semantics for handling exceptions
    - Asynchronous computation allows not to block on the main thread
- **Composability and cancellability**
    - Several cooperating computations may be grouped in task and subtask, easily tracking termination and cancelation
    - Easy to refactor code to dial with more complex situations

Coroutines are part of the standard Kotlin library, but they have some specific implementation for Android.

Blocking functions

Some operations may block the execution for an unknown time: **anything concerning I/O and networking**, **delay functions**, **synchronizations**. When they are executed in the main thread, the application may **freeze** due to its impossibility to process further events in the message queue. Executing these operations in a secondary thread is easy but collecting the results back in the main thread is hard. It requires some form of synchronization and may easily conflict with components' lifecycle.

Blocking vs. non-blocking I/O

Keeping a thread blocked while waiting for network data or similar operations, makes the code easier to be written and maintained, since it is possible to adopt an imperative coding approach, based on the fact that the current computation remains "stable" (i.e., variables keep their values, the history of function invocation does not change, …) while waiting. Unfortunately, this also requires a **lot of resources**. Each thread needs to be pre-allocated together with its execution stack, and it largely increases complexity, as well.

The picture first shows a usual blocking code style. With a non-blocking call style the continuation of computation is **moved** into a **call-back**. Function arranges for invoking the call-back when data is ready. But which thread is responsible of invoking the call-back?

```
val v = readData( params )
                //blocks waiting for data
process(v)
… //more code

readDataAsync( params ) {
   v ->
      process(v)
      … //more code
}
```

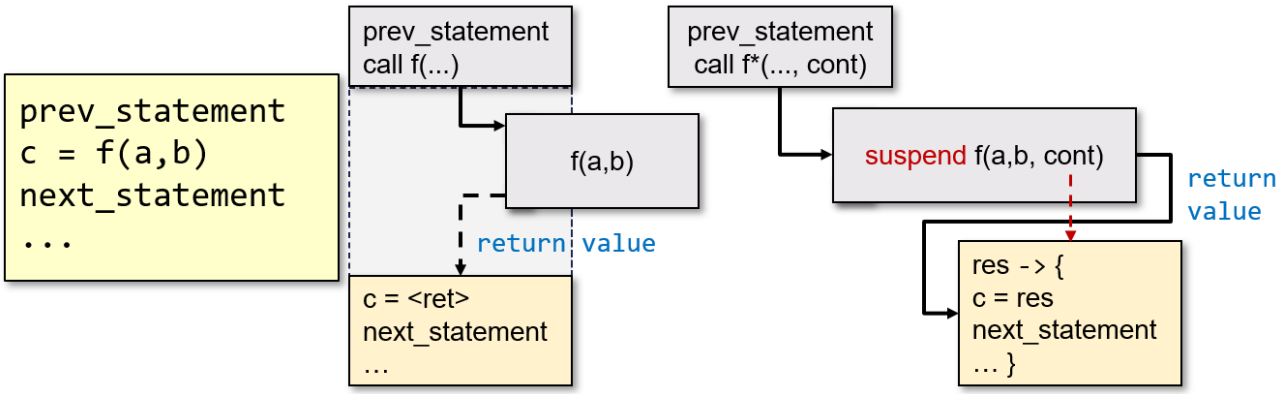
Suspending functions

To support these cases, Kotlin introduces the concept of suspending functions. They are functions syntactically introduced by the keyword "**suspend**". They have the capability to request the OS to suspend the ongoing computation by freezing the current state in a block of memory outside of the stack (the context) and by freeing the current thread, allowing it to perform another task, if available.

Function invocation styles

When a piece of code is compiled, two approaches can be used by the compiler to implement function invocation:

- Direct passing style
- Continuation passing style



Continuation-passing style

Using this style, the code **AFTER** the function is encapsulated into another function, named **continuation**. The continuation is passed as an extra parameter to the invoked function. The invoked function does not RETURN its own result. When it reaches its end, it invokes the continuation, passing its result value as an argument.

CPS is enabled in Kotlin by prefixing a function with the keyword **suspend.** The continuation encloses both the state of the execution inside the function and the callback. The code in the function can control when and **how** the continuation is invoked:

- Synchronously, in the same thread
- Asynchronously, in another thread

A suspend function may invoke, in its body, both regular and suspend functions. The latter will introduce a suspension point, while a regular function **cannot invoke** a suspending one. The compiler rewrites the body of a suspend function, transforming it into a **Finite State Machine**. If there are N suspension points, the FSM will have **N+1** states (the beginning of the function, the end of the function).

The continuation will contain:

- The current state of the FSM
- The set of local variables
- The result of the previous computation
- The callback to invoke to return the value
- The coroutine context that defines
    - the **Dispatcher** responsible to invoke the continuation
    - the **Job** that conveys information about how the current coroutine relates to other (parent, siblings, children) coroutines
    - the **ExceptionHandler** that deals with failures

Coroutines

A coroutine is an instance of a suspendable computation that is, a computation that can suspend at a given point, freeze its own state in a context object, and later resume, possibly in another thread. Coroutines are designed to support:

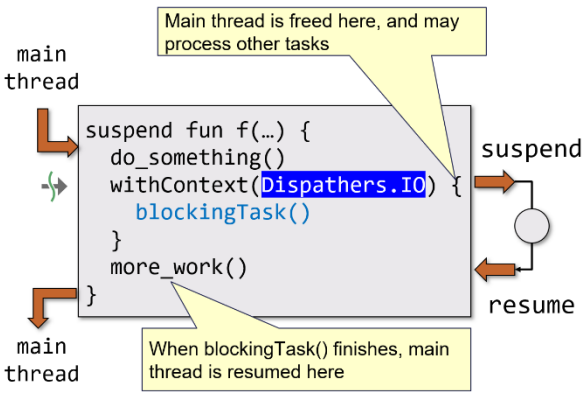
- Asynchronous computations
- Futures
- Generators
- Channels

Like normal functions, coroutines are executed by a thread. When they reach a suspension point, the state is frozen in the the context and the function returns thus the thread is detached from the coroutine, and it can perform other computations. If a suspended coroutine must be resumed, a thread will be picked to invoke it again. The coroutine context contains the information necessary to pick such a thread and jump to the proper code label.

A suspending function makes it easier to switch from a thread to another and back to the previous one. The suspending function may be seen as a four gate block.

The Suspend operation captures the current continuation (with all local variables) and makes it available as the only argument to a call-back routine. This has the opportunity to arrange the execution of the blocking operation in a different thread, which will – later on – invoke the continuation. When the call-back returns, the computation is actually suspended (the invoking function returns), freeing the current thread.

When the other thread that was blocked performing the I/O will proceed and invoke the continuation, the execution of the suspend function can resume, restoring the same state it had when it was suspended.



Execution flow

Not every suspend fun invocation inside a coroutine will actually suspend. The compiler takes care of that by introducing a special return value to mark a suspension. Two typical cases where suspension occurs:

- **delay(milliseconds: Int)** – causes the current thread to be detached from the coroutine and process other tasks: after the given amount of time, the coroutine will be eligible for being resumed by any thread of the current context
- **withContext(ctx: CoroutineContext) { λ }** – delegates the execution to any thread inside the given context and frees the current thread which can pick other tasks; current thread will later resume the coroutine execution, when the lambda returns and it will be able to access the returned value/exception

Coroutine contexts

Coroutines require a context in order to be executed, since it defines:

- **Dispatcher** – the set of threads used to execute it
- **Job** – the state of the computation of the coroutine
- **CoroutineExceptionHandler** – defines how exceptions thrown inside the coroutine are handled
- **CoroutineName** – the name of the coroutine for debugging purposes

Contexts are typically created adding the elements they are made of:

$$val\ c: CoroutineContext\ =\ Dispatchers.IO\ +\ SupervisorJob()$$

Dispatchers

They are objects that define the thread pool used to invoke the coroutine. A custom dispatcher may be created from an ExecutorService. Class **Dispatchers** provide several ready made ones:

- **Dispatcher.Main** which contains only the main thread and it is only available in Android and for GUI applications
- **Dispatchers.IO** which contains a large thread pool, optimized for disk and network IO and used for reading/writing files, accessing databases, networking
- **Dispatchers.Default** which is optimized for CPU intensive work off the main thread. It is used for task like sorting, parsing JSON, encrypting and decrypting, …

Invoking coroutines

Because of their particular code structure, a suspend fun can only be invoked from another suspend fun or from a **coroutine builder function**. Android provides several coroutine builder functions as extension of a CoroutineScope:

- fun **launch**(…): launches a new coroutine without blocking the current thread and returns a reference to the coroutine as a Job
- suspend fun **async**(…): creates a coroutine and returns its future result as a Deferred object

A special coroutine builder function is available as a regular function: **fun runBlocking(...)**. It launches a new coroutine, blocking the current thread until it and all its sub-coroutines finish.

CoroutineScope

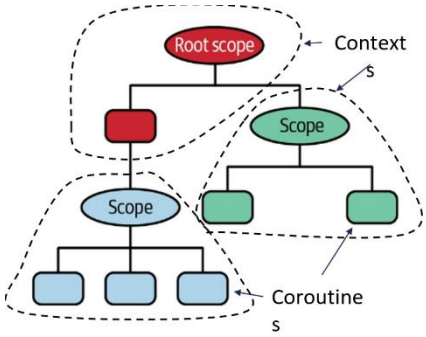
It is an interface that encapsulates a **CoroutineContext** and defines a set of extension functions useful for managing (creating, destroying) a group of related coroutines. Coroutines execution is confined inside the scope in which they were created. A specific scope is typically bound to those entities that have a well-defined lifecycle, so that when the entity is destroyed, the scope is cancelled, actually terminating all contained coroutines that are still alive. Function **CoroutineScope**(...) creates a new scope, allowing to customize the

encapsulated coroutine context. Kotlin defines also a couple of ready made object scopes, namely **MainScope** and **GlobalScope**, which have some restrictions on their usage.

Custom scopes

Android defines a set of scopes for several of its abstractions:

- *viewModelScope* - used for invoking suspend fun inside a ViewModel
- *lifecycleScope* - used for invoking suspend fun bound to any lifecycle-owning object (activity, fragment, …)
- *mainScope* - used for creating a scope based on Dispatchers.Main

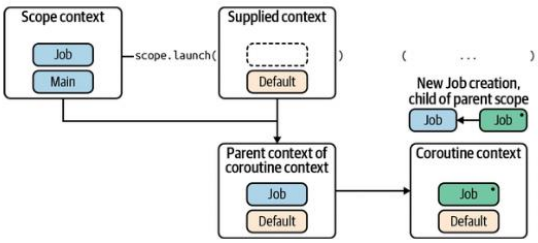


Whenever the object bound to a custom scope reaches the end of its lifecycle, the scope is cancelled. Thus, all the coroutines that might still be running on behalf of it are cancelled as well.

Scopes and coroutines are organized in tree-like structure. The root of the tree is always represented by a scope. Building children scopes allows to group coroutines that share a common lifecycle.

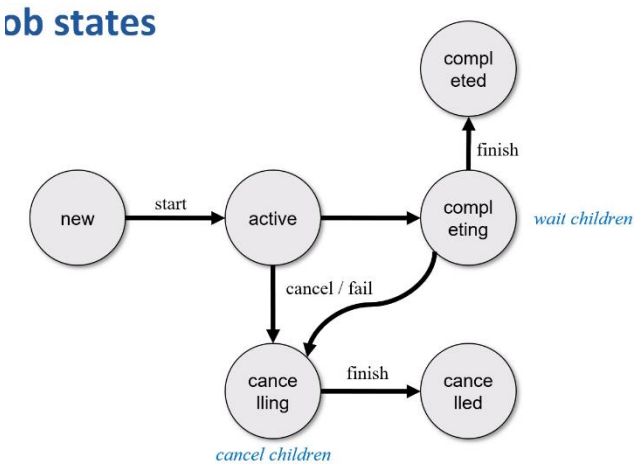
Invoking coroutines

When a coroutine is created inside a scope, a new context context is created joining the scope context with any element supplied as parameter of the creation function. Then, a new Job object is created and marked as a child of the Job contained in the parent context.



Jobs

Jobs are **representations** of background tasks carried out by coroutines. A job may have six states, expressed as a combination of three boolean properties: **isActive, isCompleted, isCancelled**. Jobs may be organized in an hierarchy and each job may have 0 or more children. *CoroutineScope.launch(...)* returns a **job object** representing the currently started coroutine. This job is child of the job contained in the context of the scope: thus cancelling the context causes all ongoing jobs to be cancelled as well, leading to cancellation of ongoing coroutines that are part of the scope.



Jobs offer several methods to control their state:

- **fun start()** – start the coroutine related to the job if not started yet
- **fun cancel(cause**) – cancel the job with an optional cancellation cause
- **suspend fun join()** – suspends the invoking coroutine until the job is complete

Invoking async coroutines

Another interesting coroutine builder extension function is **async(…)**. It allows to start a coroutine that is in charge of computing a value, expressed by its returned value, an instance of *Deferred<T>*. The value may later be obtained by invoking **deferred.await()**. It suspends the caller until the argument is ready and resumes

when the deferred computation is terminated (either successfully or unsuccessfully). It returns the resulting value or throws the corresponding exception if the deferred was cancelled or did fail for any reason.

Typical usage pattern

Coroutines are usually created in a ViewModelScope or LifecycleScope and can switch thread when performing I/O or CPU intensive operations using *withContext(…)*. Coroutine results can be modelled by a generic *Result<T>* class that encapsulates the data or an error.

```
class Result<out T>(
  val value: T?,
  val throwable: Throwable?
)
```

Invoking the Repository

Androidx viewmodel library introduces the **viewModelScope** extension property that provides a dedicated CoroutineScope. It can be used to launch coroutines from the ViewModel instance, all having a SupervisorJob. When the ViewModel is cleared, all pending coroutines are automatically cancelled. The *Dispatchers.Main* is set as the default dispatcher.

Coroutine summary

- The JVM does not provide native support for coroutines
- Kotlin implements coroutines in the compiler via transformation of suspending functions into state machines
- Kotlin uses a single keyword for the implementation, the rest is done in libraries
- Kotlin uses Continuation Passing Style (CPS) to implement coroutines
- By using dispatchers, its easy to switch thread and keep the code simple to read and maintain

Android services

A service is an application component that performs operations in **background**. It does not provide any user interface. A component can connect to a service and interact with it, even if it belongs to another process. There are two types of services:

- **Started services**
- **Bound services**

Started services

A service is "started" when an application component, such as an activity starts it by calling the *startService(…)* method. Once started, a service **can run indefinitely in background** (almost) even if the component that started it has been destroyed, provided it shows its presence in the status bar (ForegroundService). Generally, a "started" service runs a **single operation** and does not return any result to the calling component. For example, a file can be downloaded from the network and saved in the device file system. When the task ends, the service can stop itself.

Bound services

A service is called "bound" when an application component invokes it using the bindService(…) method. This offers a client-server interface. It allows the invoking component to interact with the service, **sending requests**, **obtaining results**, supported by an inter-process communication (IPC) mechanism. The lifecycle of a bound service is tied to the component to which it is connected to. Several components can be associated to the same service at the same time, but when all the associated components are disassociated from it, the service is destroyed.

Using services

Any application component can use a service. It can be declared private in the manifest file, to prevent access from other applications. A service **runs in the main thread** of the hosting process. By default, it does not create a thread of its own nor it runs in a separated process. If the service performs expensive computational operations, they should be carried out in a separate thread.

The android.app.Service class

It is the base class from which all service implementations derive. It requires some methods to be overridden to manage key life cycle aspects and to supply a mechanism be able to associate components.

The system calls **onStartCommand(…)** method whenever another component, such as an activity, request the activation of the service, via the *startService(…)* method. Once this method is executed, the service is started and can run in background indefinitely. To stop it, it is required to call *stopSelf(…)* or *stopService(…),* passing the suitable intent.

The system calls **onBind(…)** method when another component wants to associate with the service, invoking *bindService(…)*. This method should return an object implementing the IBinder interface or null. In the former case, the returned object will be marshalled to the invoker, offering a way to interact with the service. In the latter case, no interaction will be allowed.

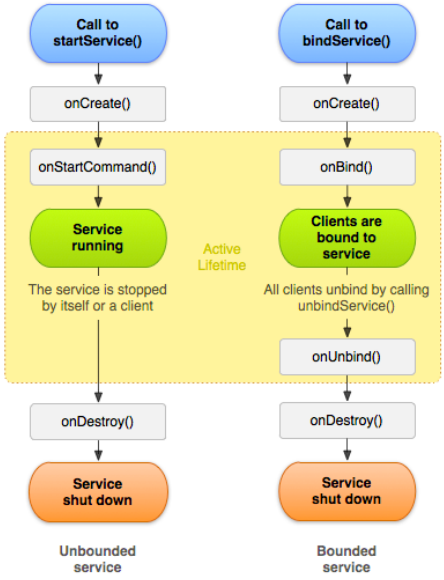
The system calls **onCreate(…)** method when the service object has just been created. Call happens before either *onStartCommand(…)* or *onBind(…)*. If the service has already been started, this method is not called.

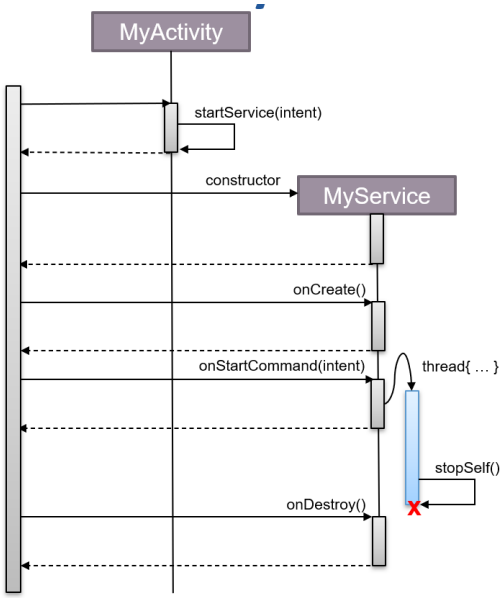
The system calls onDestroy(…) method when the service is no more used and it is to be destroyed. Services should implement this method in order to release resources: thread, registered listener, receiver, … It is the last call received by the service.

Declaring a service in the manifest file

It is necessary to declare all services in the application manifest file by adding an element of type <service> as a child of the <application> one. The attribute android:name is mandatory, but it can define other properties such as service permissions, …

Services lifecycle (left) + started service lifecycle (right)





Extending the Service class

A service may be implemented extending the Service class. In the onCreate(…) method, is it possible to create and start an HandlerThread to manage requests e.g., services message app that receives attachment and on top download a file. The constructor of this object takes a Looper as input and it is used to manage messages queued inside the handler.

In the method onStartCommand(…) for each incoming request, a message is sent to the handler. The service ID is forwarded, so that it can be used when the service will have to be stopped.

The onStartCommand(…) method must return an integer (flag), that indicates how the system will manage the service, in the case it had to stop it:

- *START_NOT_STICKY*: the service is not re-created, unless there are pending Intent requests. It is used when the service is no longer necessary. This refers to an operation that is not important. If it is needed to be killed, Android does it. It will not be restarted unless required.
- *START_STICKY*: the service is re-created, but the last intent is not returned. The system calls onStartCommand(…) passing a null intent, unless there are pending Intent requests. It is suitable for media player that are not executing any commands, but they run indefinitely waiting for a task.
- *START_REDELIVER_INTENT*: the service is created and the last Intent returned. It is suitable for services that are executing a task that must be recovered immediately, for example a file download.

SKIPPED SLIDES FROM 15 TO 17 CAP A14

Background execution limitations

Starting with Android 8.0 (API level 26) limitations have been imposed on background execution. A distinction is made between **foreground services** and **background ones**:

- A **foreground service** performs some operation that is noticeable to the user by displaying a Notification on the task bar.

- When a **background service** is running the user may not notice its presence. Thus, this mode is only allowed in limited situations.

Foreground apps

An app is considered to be in the foreground if any of the following is true:

- It has a visible activity, even if it is paused
- It has a foreground service
- Another foreground app is connected to it by binding to one of its services or by making use of one of its content providers

In all other cases the app is considered to be in the background.

Background apps

When in background, an app has a window of several minutes in which it is still allowed to create and use services. When the window expires, the app becomes idle. All its background services are (cleanly) stopped by the system.

A background app can renew its run window if it handles a task that is visible to the user: an incoming Firebase Cloud Messaging or a SMS message, executing a PendingIntent from a notification, starting a VPN service.

Handling background tasks

The Jetpack WorkManager API offers a simple way to schedule deferrable, asynchronous tasks. That are guaranteed to run reliably, even if the requesting app or device restarts but the task can be run later on, when the system deems reasonable to do it.

To use the WorkManager API, the following tasks need to be performed:

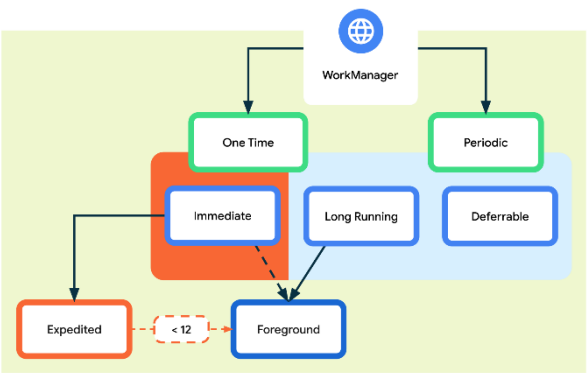
- Create a background task, by extending the androidx.work.Worker class
- Configure how and when it should be run, using a work request Builder instance
- Hand it off to the system, via the WorkManager enqueue(…) method

WorkManager implementations will override method doWork(). This will return a Result object, informing the WorkManager service whether the work succeded and, and in case of failure, whether is should be retried or not.

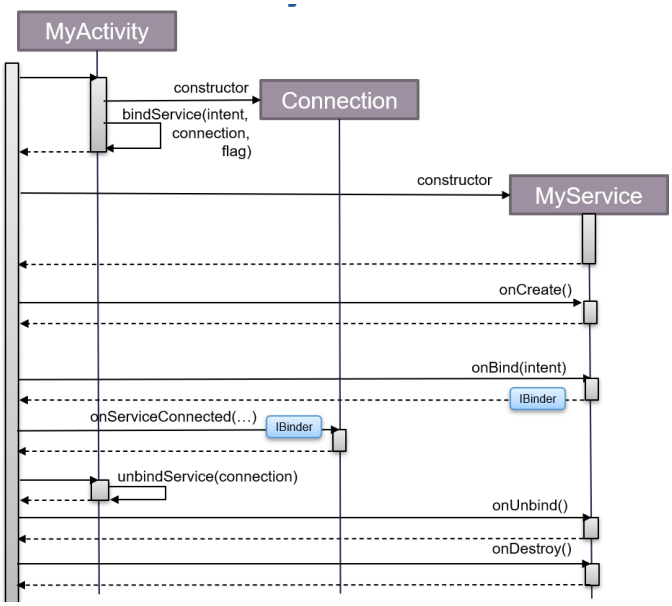
Types of persistent work

WorkManager handles three types of persistent work:

- **Immediate**: Tasks that must begin immediately and complete soon; submitted through a **OneTimeworkRequest** instance, may be expedited.
- **Long Running**: Tasks which might run for longer, potentially longer than 10 minutes; submitted through any WorkRequest instance.
- **Deferrable**: Scheduled tasks that start later and can run periodically; submitted through a PeriodicWorkRequest instance.



Bound service lifecycle



Bound services

A bound service operates as a server in a client-server interface. It allows activities (or other components) to bind to itself in order to send requests and receive responses, possibly across process boundaries. A bound service implements the onBind() method returning an object implementing the IBinder interface. This is a sort of “remote control” that let the client request operations to the service.

The onBind() method

This method is responsible to create and return a IBinder object. The OS guarantees that this method is invoked only once, the first time that a client component invokes bindService(…). The returned object is cached by the OS, and returned to all other clients invoking bindService, as long as the service is running. When the last client disconnects from the IBinder object, the service is destroyed.

Implementing IBinder

This is the API that let clients access the functionality provided by the service. It need to be carefully designed. There exist three possible implementations:

- *Extending the Binder class* – simple, it works only when clients are in the same process as the service
- *Using a Messenger* – medium complexity, requires the creation of a HandlerThread that process messages
- *Creating a remote component via AIDL* – high complexity, it requires to properly deal with multithreading issues

Extending the Binder class

Simple way to let other components to directly access public methods of the service. It is common to provide a getService() method that simply returns the service instance to the client. This lets the client directly invoke methods of the service class.

Connecting to a bound service

A component can connect to a bound service by invoking the *ctx.bindService(…)* method. Providing a ServiceConnection implementation that receives notifications about the binding lifecycle. When binding is created, the forwarded IBinder argument can be used to transparently access the service functionalities.

Cross Platform Applications

The promise

Create a single codebase from which it is possible to build and deploy an application for different platforms:

- **WORA** – Write Once, Run Anywhere
- Old slogan, not always truthful (…**test/debug everywhere**)

The concept of application is often reduced to the user interface layer only relying on a remote service or on a deeply hidden code library for providing the real functionality.

The cross-platform strength

- **The main interest is coming from the customer**: it is possible to reduce development cost and time as well as the skills needed for development (... it does it all by itself!).
- **Developers can find point of interests**: first of all the possibility to dig in the architecture that makes cross platform development possible
- **End Users may get the same functionality on different devices: w**here this is feasible

The cross-platform weakness

In order to provide similar behaviour, only what is available on all target platforms is provided. Other languages/framework need to be learned and mastered by the developer:

- They should be intrinsically neutral w.r.t. the OS
- Hard to integrate existing methodology/libraries
- A dependency on a third party (the framework) is introduced

The need of neutrality makes it difficult to introduce performance optimizations rising the risk of creating an unsatisfactory UX. Ensuring the quality of the result requires multi-platform testing (and debug).

The origin of the differences

Two major approaches in making applications:

- Web-based vs. native
- The web approach is basically limited by the browser sandbox

Native applications are based on operating systems with important differences that emerge on every abstraction level (the structure of the OS process, the adopted language, the set of basic components and their relationships, the UX guidelines…).

Architecture differences

- **Execution model:** based on components hosted in liquid processes in Android, traditional monolithic processes in iOS. Concurrency approach is not perfectly superimposable between the two operating systems
- **Inter-Process Communication:** ApplicationExtension vs Intents & Binder
- **Memory management:** Garbage collector vs ARC and Dynamic libraries handled differently

Sdk differences

Different programming languages:

- Java/Kotlin vs. Objective–C/Swift
- C++ is available on both platforms but with constraints
- Different development environments and toolchains

Similar, but not identical component set

- Elementary widget and layout algorithm
- Touch/keyboard events
- Animations
- Media recording and playback
- Platform related services (location, sensors, battery, networking, …)

Different UXs

Most relevant factor for end users. Supplier companies build on it their identity and market share. Each OS proposes its own "flavour" in an integrated echo-system where applications leave *style and themes*, *interaction patterns, s*et of support services (store, maps & navigation, cloud storage, voice recognition, …).

UX/UI false friends

- NavigationBar vs ActionBar
  - The top-left button goes back or takes to the parent screen?
- Tabs vs Segmented controls
  - Navigation with or without swipe?
- Floating action button vs "call to action" button
  - Different layout
- Hamburger menu vs tabbed menu

Cross-platform technologies

Three product generations:

- WebView-based frameworks: Cordova, Ionic
- Native-widget-based frameworks: Xamarin, React Native
- Custom-widget-based frameworks: Unity, Flutter

WebView

It is a software component witch allow to build an app using web technology: HTML, CSS, JavaScript. It is part of the Webkit library which both Safari and Chrome are based on. The rendering engine provides the view surface as well as the algorithms required to customize the look and feel, to handle the user interaction and media presentation. The Javascript language may be enhanced with custom extensions (plug-ins) allowing to import/export data from and to the native platform.

- **Pros**: Widespread know-how in the development community.
- **Cons**: Render engine performances are far from the one that can be obtained with native components. There are also dependencies from third-party plugins.

Apache Cordova

Framework for building WebView-based applications. For each supported platform, it provides an application skeleton that loads a set of plugins and shows a WebView. The extension mechanism allows the application code (JavaScript) to access native functionalities: device sensors, file system, camera….

Ionic

Open-source development environment based on both Cordova and the Angular framework. It allows to describe the app structure using self-contained modular web components. Each component has its own UI template and its own application logic, defined by a Typescript class that can access system services. It allows to build both progressive web apps and desktop stand-alone ones integrating with the Electron framework

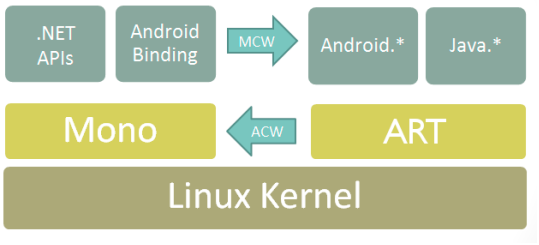
Native widgets

App structure can be divided in an **application logic layer** and **user interface** built with abstract components. While webview-based approaches tried to emulate the look & feel and behaviour of native components, the results are far from optimal. Is possible to let the platform create and manage the UI, driven by an external application layer. The latter may be written in any language and compiled ahead-of-time for the target OS, thus producing portable applications.

- **Pros**: Well-known and mature languages and dev. environment can be used (Javascript or C#)
- **Cons**: Some native code is required to support specific functionalities. Furthermore, a bridge between the native and cross-platform part is required: this can potentially limit the performances.

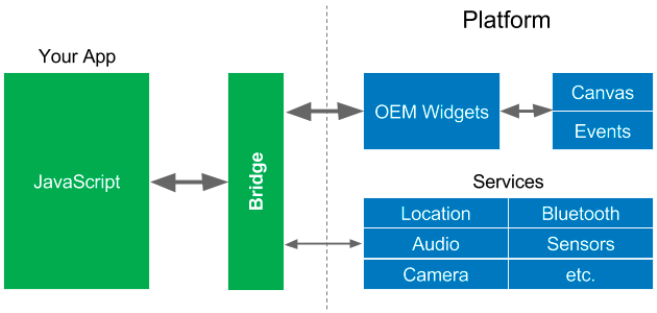
Xamarin

Side by side platform based on the Mono virtual machine. Application logic is written in C# and it relies on the MSIL Mono execution engine. Whenever possible, it directly requests the services to the kernel: for those functionalities that are available only to the native platform, it properly bridges requests to the underlying layer.



React Native

It is a framework developed and maintained by Facebook based on UI component provided by the underlying platform. It is based on the mechanisms offered by the React library (simple components that may be declaratively combined in mainly functional style). Uses an **internal broker** (JS bridge) to coordinate data interchange between the main thread (native app) and the javascript thread.



Custom Widget

The search for higher performances favors the development of innovative solutions: gaming platforms (Unity, Unreal, Cocos2d) provide low-latency and high-reactivity by requiring the native environment to provide only the drawing surface and using GPU support (OpenGL ES) to draw on it. These platforms define their own set of widgets by having full control on their UI/UX.

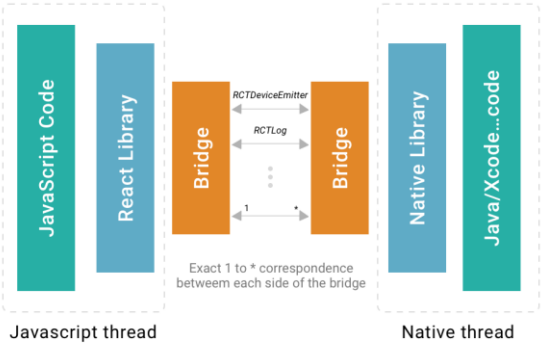
Flutter

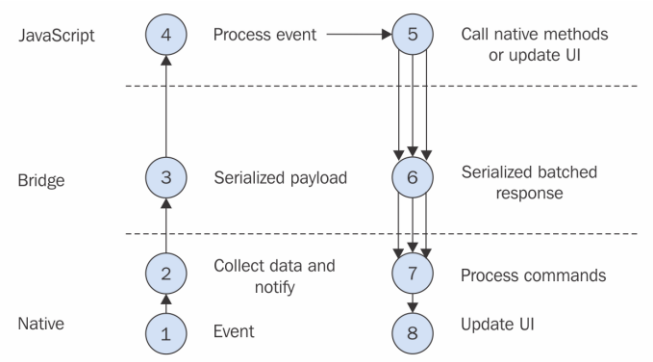
It is an open-source SDK developed by Google. It supports Android, iOS, web, as well as desktop platforms and it is based on the Dart programming language. Applications are AOT-compiled in native code. The Skia graphic library draws widgets directly on a canvas. It relies on channels to allow communication between the native platform and the Dart engine.

React Native

It is a **cross-platform framework** for mobile and desktop app written in JavaScript but rendered with native code. Views are mapped to the corresponding native counterparts at compile time. Components wrap existing native code and interact with native APIs with React declarative paradigm. It is developed and maintained by Facebook, but open source and community driven.

It targets not only iOS and Android, but also Apple TV, VR, AR, Windows and desktop. Projects usually allow a high level (80-95%) of code reuse. It is possible to ship apps over the air bypassing the App store / Play store.





React Native vs React Web

- **Different primitives:** React Native offers 20 core components plus a few dedicated to a single platform (iOS or Android), while HTML offers over 100 different tags
- **Styling:** React Native CSS mimics Web CSS, but they are not identical
- **Most browser APIs do not exists**
- **Navigation:** React Native supports multipage navigation and not generic hyper-texts
- Mobile UI/UX is different from the desktop web UX
- Native IDEs/Configuration are required
- Deployment does not require a web server

RN provides two CLIs (Command Line Interfaces) to develop and deploy mobile applications:

- The *expo-cli*: a fast and out-of-the-box solution to start with React Native
- The *react-native-cli*: a complete and powerful development environment

Both of them are based on NodeJS, a Javascript runtime that can be freely downloaded.

- **Expo Advantages**: easy and fast setting up, easy testing with QR code, support for some native libraries.
- **Expo Disadvantages**: no custom or third-parties native modules and libraries integration, heavy app size even for simple projects (because of the integrated libraries).
- **RN CLI Advantages**: custom or third-parties native modules and libraries integration, complete control over the build phase.
- **RN CLI Disadvantages**: Android Studio and/or XCode needed to run the projects, no iOS development without a Mac, testing on real device needs USB direct connection and setup.

Which one to use? There is no absolute answer. It depends on project purpose and requirements. To learn RN language, Expo is a good starting point. Anyway, it is always possible to start an Expo project, and then "eject" it. By ejecting, Expo advantages will be lost, but it will be possible to have full access to the native environment.

ReactNative basic principles

It is based on the React library:

- Component based & composable
- Mainly declarative programming paradigm
- Uses JavaScript to express the business logic and JSX for the UI description

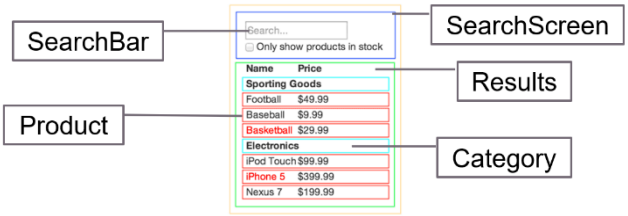
Translates the ReactDOM tree into native component for the specific mobile OS architecture.

React Native is **component-based**. A component is an independent piece of UI. Its main properties are:

- **Encapsulation**: who uses the component doesn't need to know how it works inside
- **Reusability**: they should be designed to work almost everywhere
- **Composability**: basic components can be composed together to make more complicated and specific components

Composition

The UI can be split in a set of **independent** and **reusable** components, similarly to the way we split a program into functions and objects.



Declarativity

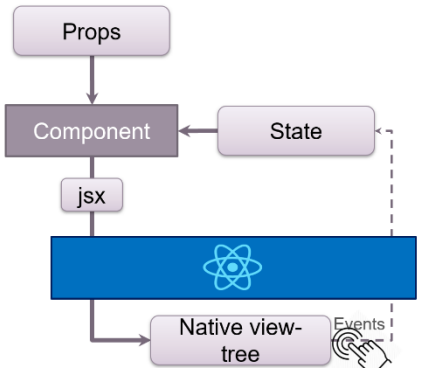
The programmer provides a description of the UI in terms of components, their properties and structure. The toolchain transforms this description into an executable program that creates the required interface and corresponding behaviour.

Components

A React component is a simple **javascript function**. Its return value is a **UI element** expressed via the JSX notation.

Props

Component inputs are called props and they are set by parent component. A prop is an **immutable key/value pair**, where key is the name of the prop, and the value can be any data.



State

A component may rely on some (non-persistent) **state information**. Differently from props, state is **mutable.** Each time it changes, the component function is re-executed. When this happens, the (recursively expanded) function result is 'diffed' with the previous native view-tree representation. Differences are used to update the screen.

JSX

It is a domain-specific language to describe UI. It is based on XML templating model but enriched with Javascript functionalities. A transpiler like Babel is required to convert a JSX piece of code in plain React functions.

RN basic components

React Native provides a basic set of UI components directly linked to native UI elements.

- **View:** the basic and fundamental container component, necessary to build the app layout. It supports styling and can receive touch events. In Android, it is equivalent to LinearLayout. It cannot contain text directly (that must be wrapped in a <Text /> element).
- **Text**: the basic component to display texts. It supports styling, and it can be nested to create complex formatted texts.
- **TouchableNativeFeedback (Android), TouchableOpacity**: not properly UI elements, but wrappers around other components. They expose the *onPress()* callback, to handle user touches events. They include an animation around the content to animate the opacity and give the look and feel of a touch press event.
- **Button**: composed by other components (View, Text, Touchable) to render a basic and simple button.
- **Image:** Native component to render images. Supports different sources (network URIs, local images, static resources, etc.).
- **ScrollView:** layout wrapper of one or more components, implementing native scroll touch events.
- **FlatList**: complex component to render an array of elements. It provides advanced features as custom header and footer UI, refresh indicator and pull to refresh gesture, optional horizontal mode. In Android, it corresponds to RecyclerView.

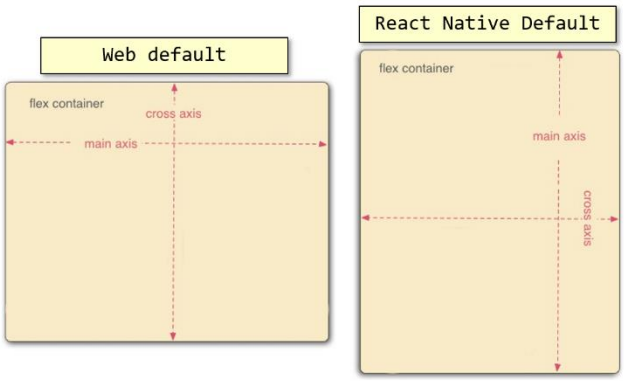
Styling

All React Native core components supports styling through simple Javascript objects passed to the style prop. The object keys and values are similar to the HTML CSS ones, with the main difference that keys are in camelCase. The main styles attributes are: *width, height, color, backgroundColor, fontSize, fontWeight, opacity, textAlign, …* React Native provides a Stylesheet component to create and manage many styles in a better and cleaner way.

Style prop accepts also arrays of styles. Each style property is overwritten from the sequent style, in order of appearance in the array. Style may apply conditionally, based on the runtime platform.

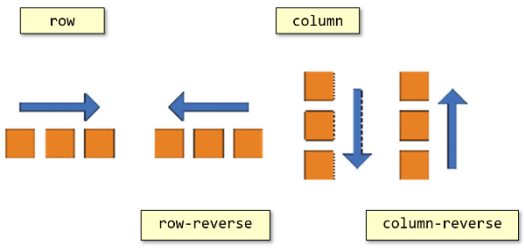
Layout and Flexbox

React Native supports flex technology to create fluid and responsive layouts. It works in a similar way as web CSS flexbox system, except a few differences. The main style properties are: *flex, flexDirection, justifyContent, alignItems*.



Flex Direction

The "flexDirection" property defines how the children components are laid out. All the possible values are: *column, row, column-reverse, row-reverse.* Unlike the web CSS "flex-direction" property, the React Native default value is "column" and not "row". This means that the main axis, by default, is the vertical, and not the horizontal one.



Flex

The “**flex**” property defines how the component will “fill” the space inside its parent container. Unlike the web version, it is simplified to a single number. This number defines the amount of space required by the component, as the ratio of the number itself on the sum of the same property value of all the “sibling” components.

Justify Content

The “**justifyContent**” property defines how to align children within the main axis of their container. It works pretty much the same as the “justify-content" web CSS property. All the possible values are: flex-start (default one), flex-end, center, space-between, space-around, space-evenly

Align Items

The “alignItems” property defines how to align children within the cross axis of their container. It works pretty much the same as the “align-items" web CSS property. Possible values are: stretch (default), flex-start, flex-end, center, baseline. Children may override this setting, via the "alignSelf" style property.

React Native – Hooks

Hooks is a bundle of functions directly connected with the component life cycle to manage in a more efficient way the inner state of the same component. There are different hooks to accomplish different tasks. They are **composable** and **reusable**.

Beyond function behaviour

There are situations in which it is not possible to write a pure functional component: applications need having a state, side effects (storage, networking, caching, …) must be performed. Internally, React Native creates, per each component listed in the VirtualDOM, an object. This is in charge of representing the component to the framework and tracking its lifecycle. Hooks rely on this "hidden" object to perform their behaviour.

The display process

What is inside of the UI, is a consequence of the (recursive) invocation of the component functions. At start-up, the root component function (usually App(…)) is invoked and its returned value is processed, possibly leading to the invocation of other component functions. Per each component that appear in the VirtualDOM tree, a corresponding internal object is created and assigned an unique ID.

SKIPPED SLIDES FROM 6 TO 9 CAP CP03

Component lifecycle

A component goes trough some phases:

- **Mounting** phase: the first time a component function is executed, the corresponding internal object is built and inserted in the VirtualDOM.
- **Updating phase**: if props or state of any component change, the component function is executed again, possibly updating the VirtualDOM
- **Unmounting phase**: if the re-execution of the component function does not return a sub-component that previously existed, it is removed from the VirutalDOM

Hooks – useState

useState is the core React hook to manage **the inner state** of the component. The useState function accepts an **initial value** as argument and returns an array with two parameters. The first one is the variable containing the **current value** of the state variable, while the second one is a function to **modify** the value of the state variable. In other words, a **value** and a **setter**.

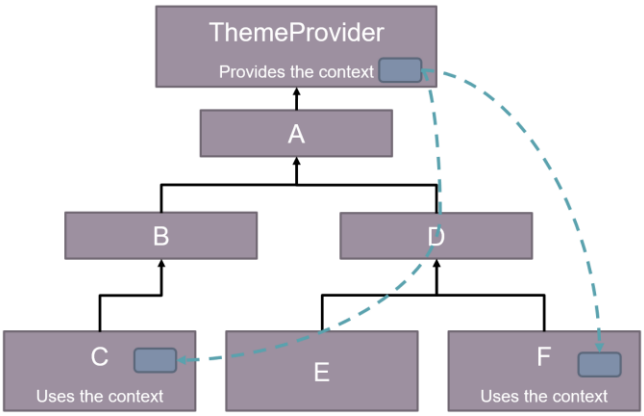
Remember to never change a state variable by assigning a new value to its getter variable. Use always its setter function to update it. Every time that a setter method retrieved by a useState is invoked, a re-render is forced. Remember this to avoid performance issues.

Global State

Applications, sometimes, need to maintain some kind of global information, which may be used by any child node: the theme stylesheet, if themes can be switched or the authentication token for accessing remote resources.

Even if this might be kept in the App node state and propagated downwards, it is often impractical. Thus React allows a root node to provide a READ-ONLY context to all its descendants.

Execution contexts



Creating contexts

A context is an envelope for some data. It is created using the *React.createContext(defaultValue)* function. The envelope is populated using a provider node, that sets the content for all its children.

Accessing context

Hook function *useContext(…)* can be used in children nodes to access the value stored in the corresponding envelope by the nearest provider. Requires the context object as the only argument. If no ancestors are provider for the given context, the default value is returned.

Updating the context

Contexts can store any kind of data, including functions. This may be convenient in order to provide a way to ask the provider to change the content of the context. When the context changes, all nodes that rely on it, by invoking *useContext(…)* are updated.

Handling state complexity

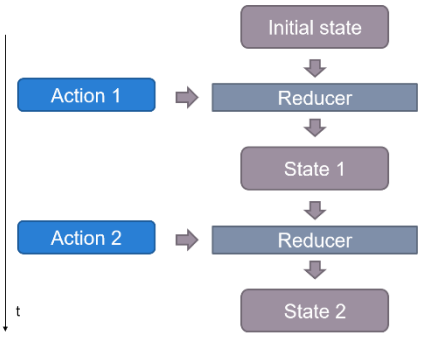
When many interdependent pieces of information need to be kept inside the status, *useState(…)* and *useContext(…)* becomes unsuitable. Several invocations of setter functions need to be performed leading to useless redraw cycles that slow down the application and consume a lot of battery. The hook *useReducer(…)* offers a more structured and solid approach to state handling, improving application consistency and making it easy to govern state dependencies.

Actions and reductions

Application state consists of a large, immutable object with several properties that represents different aspects of our application. An **action** (in reducer terms) is a description of a change that need to be performed on the state. Actions can be coded as plain objects having a type attribute that specify the request and, possibly, more keys to describe further details:

- { type: 'CHECK_OUT' }
- { type: 'ADD_TO_CART', id: 'aSF32PgN' }
- { type: 'USE_PROMO', promoCode: 'XYZ'}

A reducer is a pure function that given the **current state** of the application, and an **incoming action**, decides how to modify the state itself and returns the updated state representation. Reducers MUST treat input state as **immutable**. Returned state must be a new object, possibly containing copies of previous data, if not affected by the action, and new data, because of the action.



useReducer(reducer, initial_state)

This is the hook implementing the reducer pattern. Its parameters provide what is necessary to bootstrap its behaviour. Similarly to useState(…), it returns a list made of two values: current state, a function to dispatch an action to the reducer.

Whenever the dispatch function is invoked, the component that has invoked useReducer(…) is updated thus making the new state available to the component.

Combining contexts and reductions

Both current state and the dispatch function can be of interest of many components along the display tree

- Having the root node (App) setting up the reducer pattern and making it available to any node via context is a widespread pattern
- It makes the state the single source of truth, that each node may access in a read-only way

The reducer, being a pure function, makes it easy to test and debug the application logic. The reducer is access indirectly, via the dispatch function and action objects, preventing misuse.

Hooks – useEffect

useEffect allows the use of side effects inside React function components. It takes 2 arguments:

- The first one is a **callback function** that is executed after every rendering, when the hook triggers
- The second one is an optional array of **dependencies** (state vars, props, etc.). In this case the callback is called only if at least one of the dependencies changed its value before and after the rendering.

Canceling an effect

The hooked function of useEffect(…) may return a value. If such a value is a function, it will be interpreted by React Native as the way to undo the effect. If the effect is to be triggered again, and its previous invocation did return a cancellation function, this will be invoked before re-executing the effect. If the component using the effect is to be removed (unmounted) and there is a cancellation function, this will be run after component removal.

Networking effects

One of the most common effects is contacting an API server:

- React Native offers two major built-in functions for doing this
- They reflect the behaviour that browser provide to JS code

Fetch(…) is function that accepts an URL and some options and returns a Promise object. The Promise represents the ongoing connection. A callback can be attached to the promise via the *then(…)* method. If the promise will be successful, the callback will be invoked passing, as a parameter, the result of the network request. Failure can be detected similarly, using the *catch(…)* method.

Alternatively, React Native also offers the XMLHttpRequest class. This mimics the corresponding browser API that encapsulates a state machine representing the various states of a network request. While more complex and possibly error prone, objects instances of this class allows request cancellation and are used by more advanced (and useful) libraries like Axios.

Hooks – useMemo

useMemo allows the **memoization**, or **caching**, of derived data. It takes 2 arguments:

- The first one is a **callback function** whose return value is the **memoized data**
- The second one is an **array of dependencies** (state vars, props, etc.)

The callback is called only if at least one of the dependencies changed its value, avoiding expensive calculations after every rendering.

SKIPPED SLIDES FROM 38 TO END CAP CP03