Introduction to Android

Android is composed of an Operating System and a software platform to create apps and games. It is designed to be robust: it is based on the Linux OS Kernel. Each application runs with its own process, in its own instance of the virtual machine. When an application is installed, a new user is created. In this way, each application will be able to R/W only in its own part of the system. Furthermore, processes are liquid: the OS manages resource consumption, and it can free memory if needed, by alerting processes of termination. Each application contains a manifest file in which will be declared all the components, the permissions needed and other configurations. Each application consists of:

- **Activity**: it handles the user interaction and for this reason it has a GUI.
- **Service**: it can perform long lasting tasks and runs on the background.
- **Content Provider**: it manages application data, and it can send data to other applications that are requesting it.
- **Broadcast Receiver**: it waits for messages coming from the OS and can generate notifications

When the application is launched, an Intent is created and sent to the Zygote, which is a pre-warmed VM that will perform a fork so that only the component of the application needs to be instantiated. From the forked process the Intent is used to locate and load the APK and the manifest. This will instantiate the application object, which is the first to be instantiated and the last to be destroyed. An intent is an abstract description of an operation, and it consists of several parts: **action** to be performed, **data** to operate upon, **category** which are further information about the component that should handle the intent. If an application wants to handle Intent, it declares them in the manifest with filters. When an Intent is launched, OS will look through the filters for a match.

Activities

Activity is the base class that provides a GUI from which a user can interact. The OS creates the activity and handles its life cycle. An activity must:

- Acquire the required resources
- Build and configure GUI
- React to events from the user interactions
- Manage notifications about its life cycle

Each activity shows a single user interface, so it can handle a specific task. For this reason, an application may contain several activities. The first one to be shown is marked in the manifest file. If an activity wants to launch another activity it will use an Intent, which can be of the same application or another one. The OS handles activities by creating an **activity stack**. The one that is shown goes on top of it, while all the previous are shifted down of one position. When the activity is removed (e.g., back press of the user) the previous activity goes back on top of the stack. The OS sends different notifications to track the status of an application and the programmer must react to these, since activities can be destroyed and recreated. The lifecycle goes through some methods:

- **onCreate()**: called when the activity is run for the first time (and the Bundle is null) or when the activity is launched again after being terminated (Bundle will contain status information). When the activity stops it can save its state in a bundle. The activity exists but it is not visible.
- **onStart():** called when the activity becomes visible to the user. User cannot interact directly with it.
- **onRestart():** called when an activity that was paused is restarted again.
- **onResume()**: called when the activity reaches the top of the stack. It is in the foreground and user can interact with it.

- **onPause():** called when the activity has to be moved in second position of the stack. Here all non-needed resources are released (e.g., persisting data is committed, listeners unregistered).
- **onStop()**: called when the activity is no more visible to the user. Next possible notification can be onRestart() or onDestroy().
- **onDestroy()**: activity terminated and removed from memory.

In the onCreate() method it is needed to prepare a View and make it visible. It is in charge of presenting content to the user and is usually made of elementary widgets connected together to form a visual tree. Android uses the **composite pattern** to build the hierarchy. Among the view subclasses there are the elementary views (which are leaves in the tree) and view containers which can internally host other views. It is possible to create the hierarchy in three ways:

- **Programmatically** by direct instantiation of elements. This gives flexibility because they are built based on actual data, but also control since everything is created explicitly from the programmer. The problem is that maintenance is difficult and does not support internationalization.
- Via **XML files** by naming each file with a unique ID and then by inflating it from the code. Each element of the tree will have a unique ID too, to be managed also via code. It is easy to maintain, and the visual editor helps building the XML files. The problem is that each element needs an ID, and they must be kept in sync between the various XML representations.
- Using **Jetpack Compose** library, which is inspired from React and it can describe declaratively the content and the behavior of the view. Each view is represented as a function labelled "@Composable". The problem is that it requires programming skills and designer find the XML files more intuitive.

Intent is an asynchronous messaging mechanism used by the OS to associate process requests with activities. They can be:

- **Implicit**: the action to be performed is indicated and the OS will find the component that is able to handle it. In this case it is provided the action, the URI (resource id) and the category.
- **Explicit**: It ask the OS to activate a specific component within the application process.

Intents can be enriched with bundles in the form of key/value pairs. Intents can also be **broadcasted** and anyone who registered a listener will receive it. Typically used to generate system wide notifications.

All the elements of an application (Activity, Service, Content Provider, Broadcast Receiver) have a common root: the **Context**. It provides the functionalities to access resources (identified by unique ID in the project) and all the application-specific classes, but also to interact with the OS.

Graphical User Interface

The usage of graphic tools facilitates in terms of operational level the design of user interfaces, but still there are:

- **Cognitive issues**: building a pleasing, functional and easy to learn interface is challenging. The design of informational flow and graphic contents must be done carefully, otherwise user will get confused and stop using the application.
- **Operational issues**: mobile devices vary widely in terms of physical dimensions, screen resolution and orientation, dot density. A set of possible alternative configurations should be designed by also taking in account the different languages, cultures and metaphors and level of confidence.

General guidelines says that the first step is related to the analysis of the context and the processes to be supported, the second step subdivide complex tasks into smaller ones, while the third step relies on general principles of interaction design.

Android offers a rich family of classes derived from ViewGroup specialized in the way they manage the space assigned to them. By selecting suitable policies, it is possible to mitigate the effects of device variability. Each layout can have some properties (such as orientation, layout_width and layout_height). For width and height it is possible to specify:

- A number plus unit of measure (es. 100dp)
- **Wrap_content**: it will occupy the space needed from the total of its children
- **Match_parent**: it will fill all the available space given by its parent

It is also possible to add white space among widgets by specifying **padding** and/or **margin**.

The possible layouts are:

- **Linear Layout**: child elements arranged one next to the other and, by using the orientation property, it is possible to specify if they are arranged vertically or horizontally.
- **Relative Layout**: the position of children is described in relation to each other or the parent. The alignment constraints are typically expressed using Boolean properties.
- **Frame Layout**: show children superimposed (sovrapposte) on the same area. The most recent is the last to be designed and it appears above the others.
- **Scroll View**: used when a view needs to display more data than what can be displayed on a single screen.
- **Constraint Layout**: extension of mechanisms available in linear and relative layouts. It allows to create flat, fast and effective visual hierarchies supporting almost all needs. It is integrated with the visual editor of Android Studio and it provides:
    - **Constraints**: one-way relationship between two widgets. Controls how they will be positioned within the layout.
    - **Chains**: when a constraint is duplicated in both directions, a chain is built. This allows to share the space between views and control how much of it is given to each view. Chains can be in three modes: **spread** (widgets are evenly separated)**, spread inside** (first and last touch the borders of layout) and **packed** (concentrated to the beginning and end of the chain).

Resource management system provides automatic support to use different layouts based on the actual characteristics of a device. It is needed to add a **suffix** in the folder to specify layouts for different configurations. The alternative resources must be named exactly as the default one and use the same identifiers.

Android Architecture Components

Jetpack is a set of components, libraries, and tool with the aim to reduce development time, reduce boilerplate code and build robust and high-quality apps. It is divided in 4 areas:

- **Foundation**: common to all apps that allows backward compatibility and unit and runtime tests, as well as benchmarks and security.
- **Architecture**: classes and interfaces that help make a robust, testable, and maintainable application (e.g., lifecycles, LiveData, room, ViewModel)
- **Behavior**: components that help the integration with standard Android services such as notifications, permissions, and preferences.
- **UI**: components and classes that allows to improve overall UX (e.g., animations, fragments, emojis)

Components of an application can be launched individually and out-of-order and can be destroyed at any time by the OS or user. For this reason, we need to perform separation of concerns. No component should store inside its own properties, application data or state information. Moreover, a component should not

depend on the existence of another one. Activities and fragments should only deal with UI or OS interactions. The content of UI should reflect a model and user will not lose data if the app gets destroyed. Google proposed the **Model-View-ViewModel** (MVVM) architectural pattern:

- **Model** contains all the data classes, database classes and repository
- **View** is the UI that represent the current state of information visible to user
- **ViewModel** contains the data required by the View and transforms data stored in Model in a way suitable to be presented in the View.

This mechanism allows the project to be **loosely coupled** (by separating business logic from presentation), it is **easy to maintain** since view is not aware of computation happening behind the scenes and it gives **great structure to the project** making the code easier to navigate. The Model implementation of Google contains the **repository** which provides façade access to data that can be accessed locally (provided by **Room**/SQLite) and remotely (with **Remote Data Source**).

LiveData is an **observable data holder class** that notifies the observer when the data change. It is **lifecycle aware**, which means that it respects the lifecycle of the other components. The advantages are that there are no memory leaks, no crashes due to stopped activities, always up to date data and no manual lifecycle handling. Since the model state evolves continuously and asynchronously, it provides its content using LiveData objects. Room provides an abstraction layer using an ORM-Compliant architecture. It allows to specify tables of a database using entities (classes marked with @Entity) and each variable is a column in the table. Each entity must have a primary key. Relationships can be expressed via the @ForeignKey annotation passed as a property to the entity one. Data can be retrieved as they are or as LiveData objects, but also as Cursor objects.

The ViewModel store and manage Ui-related data in a lifecycle conscious way. It is instantiated inside an activity or fragment using a **delegated property**. This allows to rebind the same VM when the activity/fragment is recreated. For this reason, a VM must never refer a view, or anything related to it, except for the Application Context, which is the only one that doesn't change during the life of the app. VM provides immutable LiveData to the observers and internally uses MutableLiveData to update its own state. A VM can be used also to communicate between different fragments, decoupling the data exchange.

Displaying Collections

It is often needed to manage collection whose size is not know a priori and may be large. Jetpack provides the so-called **recycler view**. It provides a way to manage items efficiently by exploiting the fact that only some of these items are visible on the screen. The recycler view will handle only the number of items that can be shown on the screen plus a **ghost item**. When an item disappears from the screen that will become the new ghost item. This **drastically improve performance**. A recycler view is composed of:

- An **adapter** that provides and fill the views of the data set. It has three responsibilities: it stores the data set and retrieve the total number of items (*getItemCount()*), it creates the ViewHolder and corresponding visual tree (*onCreateViewHolder()*), and it binds the data with the view (*onBindViewHolder()*).
- Each item is represented by a recyclable visual hierarchy managed by the **ViewHolder**
- The visual presentation of items is delegated to the **LayoutManager** (how to display items, grid, linear, etc...)

If any change happens to the data set, an animation needs to be performed. This is typically done through the **diffUtil** class, which computes the difference between the two sets and outputs a list of update operations. RecyclerView may require different views for different type of data, so it is possible to use *getItemViewType()* to give a number to each different item so that different ViewHolders will be used.

Fragments

To help the designer in the creation of usable layouts, the concept of Fragment has been introduced to make GUI more **adaptive**, to facilitate the creation of tabbed views, to support the creation of custom dialog boxes and to allow to design custom navigation graphs. Fragments encapsulate major components in a **reusable way**. Fragments provide:

- **Modularity**: each fragment supports a well-defined interaction with user. A complex activity can be split in several fragments (better organization and maintenance). A complex task done by an activity can be divided into many smaller tasks that are independent and that can work by themselves. The divided element can be called "module" because of that, and each fragment can perform one of them.
- **Reusability:** the behavior of a single fragment may be re-used by multiple activities, or it can appear several times in the same one. Those modules, since they are independent, can be reused in the same activity many times and in other activities, reducing the needs to rewrite from zero the same things.
- **Adaptability**: to simplify the task of adapting interface to different screens, it is possible to divide it in smaller blocks and combine them in several ways, according to characteristics of the device. They are kept together by the activity that hosts them.

A fragment is an object that conceptually stands between Activity and View. As an activity, it has a complex life-cycle but it also owns a hierarchy of views that may become part of the host activity visual tree. Differently from activities, fragments are **manipulated directly by the programmer**.

When the activity loads its own layout, the **FragmentContainerView** will act as a placeholder where fragments may be appended. Then, the activity is responsible to **instantiate** the required fragments and **displaying** them. To do so, the FragmentManager is used. FragmentManager is a component that allows to dynamically change the fragments shown inside an activity. It provides some method that can be performed inside a **transaction** (which means that it commits if everything goes right, otherwise rollbacks). FragmentManager offers some methods:

- **Add()** method adds the fragment to the activity state. A tag can be optionally attached to a fragment to easy retrieve it.
- **Remove()** method removes a fragment from the view hierarchy and detaches it from the activity
- **Replace()** removes all fragments currently present in the view and inserts the new fragment in the same view

During its lifecycle, a fragment can be in any of these 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**

When a fragment is **statically** added to an activity, his lifecycle is strongly interconnected to the activity one. In fact, the activity state changes are reflect on the fragment state (*onCreate()*, *onStop()*, etc…) and typically the onAttach() method is called right after a call to onCreate() happened. If, instead, a fragment is **dynamically** added or removed from an activity, the onAttach() method can be called even if the activity is in onResume() state. In this case, the two life cycles are only weakly interconnected: the cycle of construction and destruction of a fragment can be contained in one single phase of the life cycle of the activity.

The creation of a fragment means that the user perceives a new screen, so it is reasonable to expect that back button will bring back the previous screen, but this normally does not happen. For this reason, it is possible to tie a transaction on the fragment to an event on the back stack using *addToBackStack()* method.

Android supports different kind of navigation. The back navigation is the one obtained by default, thanks to activity stack. The navigation can be specified in a **declarative way** in an XML file. Each screen can be a fragment or an activity. Activities hosting internal navigation via fragments rely on *NavHostFragment* contained in their visual hierarchy. It offers a *NavController* which is in charge of reading the graph and dynamically instantiate and show screen fragments as navigation occurs. The navigation file can contain arrows that goes from one fragment to another, and they have a unique ID and they are called **action**. These action can happen by using the *NavController* and the method *navigate()*.

Introduction to Cloud Firestore

Cloud Firestore Database is a **schema-less document-based, NoSQL** database which is provided by Google and allows to scale both in terms of number of records and in terms of concurrent users. Due to CAP theorem, **consistency is eventual**. The database is organized in **documents** and **collections**. Each collection has a **name,** and it can contain zero or more documents. Each document has a unique ID. Firestore allows to retrieve individual documents inside a collection through some queries. Thanks to indexing, performance depends on the result set size. Firestore opens a data synchronization channel with each device and any data change is noticed. Queries can be **one-time fetch only** or it is possible to add a *SnapshotListener* in order to receive the newest data and to execute a lambda on any update. Queried data is **cached automatically** and read/write is possible even when device is offline. It will be synchronized again when online. To model relationships, it is possible to:

- **Reference data**: inside a document there will be a reference (URI) to another. Since joins are not supported by Firestore this means that 2 queries need to be performed. Furthermore, it does not support cascading. If a document is deleting, the other keeps on referencing it.
- **De-normalize data**: data duplication is introduced. In this case redundancy and possible conflicts are introduced.

All reading and writing operations are **asynchronous** and callbacks can be set to react on results/failures. The JSON object is mapped to kotlin class. The method *toObject()* can be used to perform this operation.

Queries on firebase have some limitations: it does not support range filters or != clause. Logical OR queries have limited support. The MVVM can be simplified if Firestore is used, since it automatically performs caching and supports offline data persistence.

Material Design

Material design is a set of **design guidelines** and a **visual language**. It provides a new style for mobile interfaces that is **minimalist, consistent** and **rationale**. It is based on the fact that elements should behave similarly to real world materials: *occupy space*, *cast shadows* and *interact with each other*. It provides guidelines such as to **choose colors deliberately, fill screen edge to edge, use white space intentionally**, and so on… About **imagery** it suggests using images that are **relevant and informative**. About **typography** it suggests adopting a limited set of sizes (since too many sizes is confusing and looks bad). About **colors** it suggests using a **primary color, some shades, and an accent color** by providing the right contrast. About **motion** it should be **responsive, natural, aware, and intentional**. It provides a set of widgets:

- **Toolbar**: generalization of an action bar. It inherits from ViewGroup so it can have children views such as *navigation button, app icon, title/subtitle, one or more custom views, an action menu*.
- **Cards**: contains content and actions about a single object. It is a ViewGroup, so it is possible to add children views.
- **Buttons**: customizable button component with updated visual styles. It is composed of a label, a container, and an optional icon. It comes in multiple flavors: text button, outlined button, contained button, toggle button.

- **Floating Action Button (FAB)**: displays the primary action in an application. It is a **round** icon button that is elevated above other page content. Provide quick access to important or common actions.
- **Text fields**: allows the user to enter and edit some text. It can be **filled** or **outlined**. It is created as a combination of a *TextInputLayout* that wraps a *TextInputEditText*.
- **Snackbars**: temporary widgets to provide quick feedback messages. They disappear automatically after a timeout
- **Chips**: represents a complex entity in a small block. It consists of a label, an optional chip icon and an optional close icon. It can be clicked or toggled.
- **Navigation drawer**: it contains a NavigationView and allow the navigation to different fragments.

Custom Views

To create a custom view, the View class or any of its subclasses should be extended. It has three main responsibilities:

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

A view can also define its custom attributes in order to be configured in the visual layout editor. Since views may be destroyed and recreated, the custom view is **responsible of properly saving and restoring its state**. The lifecycle of a view goes through the following steps:

- **Measure**: onMeasure()
- **Lay it out**: onLayout()
- **Draw it**: onDraw()

Depending on the content that a given view wraps, there may be many possible arrangements of it. For this reason, the parent container starts a **dialogue** with each child view, **proposing** a given amount of **space**. Then, children will compute how large they want to be and then they will store that measure in an attribute.

When a view need some space, it invokes the *requestLayout()* method. The framework puts an event in the event queue and when it is processes, the framework will allow the container to ask to each of his child how much space they need to draw themselves. The process is split in two phases:

- **Measuring all child views:** framework starts the process by invoking *measure()* method of the root in the view tree. Each container will ask to each child view the desired space. Then, the call is propagated to all the descendants.
- **Calculating their sizes and positions:** the *measure()* method will internally call the *onMeasure()* which must be overridden by the views that want to implement their own space management procedure. This method provides as argument how much space the parent view offers to its children. The child view will provide how much space it needs via the method *setMeasuredDimensions()*.

After negotiating the space, the parent lays out its children inside the rectangle it owns. This phases takes place with the *layout()* method call which internally invokes *onLayout()* method. This last method is responsible of **calculating the boundaries of the rectangle** that will be provided to each child view and will **call the layout()** method on each of them.

Finally, the system will call the *draw()* method on the root of view tree. This will first draw the background if present, then it will call the *onDraw()* method and then the *dispatcherDraw()* if the view contains child. Finally, it draws the scrollbars if any.

The onDraw() method receives as a parameter a **Canvas** object, which provides access to the 2D graphics pipeline. Methods of the Canvas class draw different graphical primitives on the drawing surface. Most of the drawXXX() methods require a **Paint** parameter which controls the graphics primitive rendering (color, style, fonts).

Jetpack Compose

The standard approach split the tree view code in several files: layout resource file, activity/fragment codes, several other resource files for appearance. Compose, instead, provide a reactive programming model based on **composable functions**. They are functions that can call other plain and/or composable functions and anytime the arguments passed to a composable change, they are re-invoked. Compose is fully **declarative and data driven**. Programmer is no more required to mutate the GUI to track changes in application state. Composable functions are functions marked with "@Composable" annotation and they do not return a value but typically emit user interface blocks to be rendered by the Compose runtime. Composable can be:

- **Stateless**: emissions are driven only by the parameters they receive.
- **Stateful**: they contain some mutable values which affect both themselves and the composable they invoke. In this case, the **remember** function is used to store one or more objects in memory during the first composition. Remembered value can be used as a parameter for other composables. Changes to the mutable state can occur only in main thread.

To create a UI, the compose framework will invoke a composable function that will record the set of composable that it invokes. In this way a **tree view** is built. This is called **initial composition**. When a value passed as argument or a state change, the composable function is executed again. The effects will take place only in those composable that interact with the changed value. This is called **recomposition**. This way of doing things is **optimistic**: if any other changes occur while updating, the composition is cancelled and restarted. Composable functions are **transformed** into an expanded version providing a composition context. They can be executed in **any order**: compose may run in **parallel** sibling composables. Typically, most composable functions accepts among their parameters a **modifier** in order to provide decorations and behavior to the element. When a composable accepts a **lambda**, this will trigger **hierarchical composition**. The composable invoked inside of it will become children of the enclosing composable. Compose provides some foundation composable such as *canvas, image, lazyColumn* but also some layout composables such as *box, row, col*. Material Design is supported and there is also the **Scaffold** composable, which implements basic material design visual layout structure, composed of: top app bar, floating action button, drawer menu, bottom navigation. It is also fully integrate with Jetpack Navigation, providing a NavController and allowing the navigation.

Multi-threading in Android

When an application is started, a process is created, and a **main thread** is attached to it. It creates a **message queue**, **initialize the** objects, and then begin an **infinite loop**. This main thread is responsible of *instantiating the Application and Activity objects, to notify them the events related to their life cycle, send drawing requests to the views, delivering the events related to user interactions to corresponding listeners*. Access to queue is **synchronized**, so other thread may insert new messages if there is no pending dispatching operation. Main thread can perform actions that terminate in a **short time**. Long lasting operation should be performed in a different thread. A possible way is to use a thread coming from the standard Java library, by delegating a runnable object. The problem is that threads do not return values and the result of computation must be put in a shared variable, which raises the problem of **locking**. Moreover, threads are not aware of **application lifecycle,** and they are **hard to cancel**. A possible workaround is to use **android threading extensions**. Finally, only the main thread can access views.

The Looper is a class that **manages a message queue** associated to a single thread. Queue is not directly accessible from the programmer, but it is possible to create it using the *Looper.prepare()* static method. The thread will become the **owner** of the queue. On *Looper.loop()* an infinite loop starts, waiting for messages coming from other threads. The message queue is a linked list of messages which can be processed by a consumer thread. Each looper can have at most **one queue**. The **handler** is a class that provides a thread-safe interface to **insert** and **process** new requests in the looper queue. The handler is bound to the implicit queue associated with the thread in which it was created. Each request can be a **runnable** or **message** object. In the first case, the *run()* method is called, in the second case the *handleMessage(m)* is called. This 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. Implicitly, it implements the **producer/consumer pattern**.

Since main thread is the only one that can perform actions on views, a secondary thread needs to send a message to the main thread, typically by putting an event in the event queue. Android provides 5 alternatives:

- **Activity.runOnUiThread(r: Runnable)**. If the current thread is not the main one, a new message that encapsulate a runnable object is inserted into the queue. The secondary thread needs to have a reference to the current ongoing activity.
- **View.postRunnable(r: Runnable)**. Inserts the runnable object in the message queue. If can be called by any thread as long as the view is displayed inside a window.
- **View**.**postDelayed(r: Runnable, l: Long)**. Inserts the runnable object in the queue, but it will not be processed until the interval "l" (in ms) passed.
- Instances of Handler class, created in the main thread: **queue messages or forward a runnable obj.**

HandlerThread is a simplified Looper. It builds a secondary thread which incorporates a Looper and a MessageQueue. Handler threads can be safely incorporated in standard components, binding their lifecycle to the component one. HandlerTread is created *onCreate()* and destroyed *onDestroy()*.

Kotlin coroutines allows to perform asynchronous computation by using imperative logic. They provide **composability and cancellability**. Coroutines are useful when we must deal with blocking functions, which can be anything concerning I/O and network, delay functions and synchronizations. Keeping a thread blocked while waiting is easier because we can still use an imperative approach, but it consumes **lot of resources,** and the thread needs to be **pre-allocated**. To support these cases, **suspending functions** are used. They can freeze their current state by freeing the current thread. A coroutine is an **instance of a suspending function**. It is built by using a different passing style: **continuation passing style**. The code after the function is encapsulated in a new function called **continuation** and it is passed as an extra parameter to the invoking function. When it reaches the end, the result is the only argument received by the continuation and the execution will restart. The operation can be performed synchronously by the same thread or asynchronously by another thread. Not every suspend function will suspend. It happens in the following situations:

- Delay(milliseconds: Int): causes the current thread to be detached from the coroutine and process other tasks. After the amount of time, another thread is picked, and the computation will start again
- withContext(ctx: CoroutineContext) {}: delegates the execution to any thread inside the given context and frees current thread. When the lambda resumes, current thread will resume the execution.

Coroutines requires a **context**, which defines:

- **Dispatcher**: which thread will execute the coroutine
    - **Dispatcher**.**Main** contains only the main thread
    - **Dispatcher**.**IO** large thread pool optimized for I/O operations
    - **Dispachers.Default** optimized for CPU intensive work
- **Job**: keep the state of computation

- **CoroutineExceptionHandler**: define how exceptions inside coroutines are handled
- **CoroutineName**: the name of coroutine

Coroutines can be invoked from another coroutines or from a builder function. Android provides:

- **Fun launch()**: launches a new coroutine without blocking the current thread and returns a reference to it.
- **Fun async()**: creates a coroutine and returns its future result as a deferred object. The result can be later on obtained by invoking *deferred.await()*. This method will suspend until the result is ready.

CoroutineScope is an interface that encapsulates a CoroutineContext and defines a set of extensions functions to manage a set of coroutines. The function CoroutineScope(…) defines a new scope, allowing to customize the encapsulated context. Android provides also some custom scopes such as *ViewModelScope* or *LifeCycleScope*. Typically, usage is to create a scope inside the ViewModel, which provides a SupervisorJob so that when the ViewModel is destroyed, all related coroutines are cancelled.

Jobs are representations of background tasks carried out by coroutines. A job may have six states, combination of Boolean values: isActive, isCompleted, isCancelled. Jobs offers some methods such as *start(), cancel()* and *join()*.

Android Services

A service is an application component that performs in **background**. A component can connect to a service and interact with it. There are two types of services:

- **Started services**: an application components starts a service using *startService()* method. Once started, it can run indefinitely in background even if the component that started it has been destroyed. It does not provide results to the calling component.
- **Bound services**: an application components starts a service using *bindService()* method. It offers a client-server interface that allows the invoking component to make request and receive answers from the service. In this case, service is **tied** to the component that created it.

Typically, a service runs in the main thread of the hosting process. If it has to perform an expensive computation, it is possible to manually create a separate thread. The **Service** class requires some methods to be overridden to manage life cycle aspects:

- *onStartCommand()* is called after an activity called the *startService()* method. It returns an integer to indicate how system will manage the service, in case it needs to be stopped.
- *onBind()* called after *bindService()* method it can return an IBinder object (used to interact with it) or null.
- *onCreate()* method called when it has just been created (it is called before *onStartCommand()* and *onBind()*).
- *onDestroy()* call when service is no more used and here all resources are released.

It is mandatory to define services in the **manifest file**. Some **restrictions** have been applied to background execution. It is now made a distinction between:

- *Foreground service*: service that performs operation noticeable to the user by displaying a notification in the task bar.
- *Background service*: service runs, and the user does not notice its presence. This is allowed only in **limited situations**.

A service is in foreground only if any of the following is true: it has a visible activity, it has a foreground service, another foreground app is connected to it. In all other cases, the service is considered to be in background.

An app can run services even if in background only for a window of some minutes, after that it will be in idle, and all related services will be stopped.

Jetpack WorkManager API provides a simple way to schedule deferrable, asynchronous tasks. It is needed to create a background task, by extending Worker class, and then to configure how and when it should be run, and then it is hand off to the system. WorkManager can perform three types of work:

- *Immediate:* task needs to be run immediately and should terminate soon.
- *Long running:* task may run longer than 10 minutes
- *Deferrable:* task can be run later and periodically.

In the case of a bounded service, an **IBinder** object is returned. This object is created and returned after the *bindService()* method call. It 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 object, the service is destroyed. This object allows to access functionalities provided by the service.

Cross Platform Applications

The aim is to create a **single codebase** from which it is possible to build and deploy app for different platforms, based on the WORA (Write Once, Run Anywhere). The strength of this approach is that the deployment costs are reduced and that end user may gate same functionalities on different devices. But this approach has also some weaknesses: it should be **intrinsically neutral** with respect to the OS, it is also hard to integrate existing methodologies, and the dependency on a third-party framework is added. The problem also faces the differences on the execution model of different platforms, as well as different Inter-Process Communication and Memory management. Cross-platform technologies are divided into:

- **WebView-based frameworks:** WebView is a software component that allows to build an app using web technology (HTML, CSS, JS). Low performances w.r.t. native components. Some examples are Cordova, Ionic.
- **Native-widget-based frameworks:** app structure is divided into application logic layer and user interface that is built with abstract components, letting the platform to create and manage UI.
- **Custom-widget-based frameworks:** they define their own set of widgets by having full control on their UI/UX.

React Native

It is a **cross-platform framework** for mobile and desktop applications written in JavaScript but rendered with native code, which means that views are mapped to corresponding native counterpart at compile time. It is different from the React Web since here we do not have browser APIs and we have different primitives. React Native provides two different CLI, one is easier to setup and use, the other one provides better performances. React Native is based on the React library, so it means that it is **composable** and **component-based**, mainly using **declarative programming style**. The properties of a component are:

- **Encapsulation**: who use component does not need to know how it works inside
- **Reusability**: designed to work almost everywhere
- **Composability**: basic components can be composed together to build more sophisticated components

Each component can receive as input some **props**, that are key-value pairs. It works also with **states**, that are some non-persistent state information. Any time the state changes, the component function is re-executed.

React Native's basic components are:

- View: basic container component
- Text: basic component to display text
- Touchable: expose onPress() callback to handle touch of elements
- Button: composed of View+Text+Touchable
- Image
- ScrollView
- FlatList

All core components support **styling** through simple JS objects passed to the style prop. RN provides also a **stylesheet component** to create and manage many styles in a better way. RN supports also **flex** technology to create **fluid and responsive** layouts.

Hooks is a bundle of functions that are directly connected with the component life cycle to manage in a more efficient way the **inner state of the same component**. RN creates, per each component, an **object**. Hooks rely on this object to perform their behavior. A component goes trough the following phases:

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

Some examples of hooks are:

- **useState**: core React hook to manage the inner state of a component. It accepts an initial value and provides an array with two elements: current value and a function to modify it.
- **useEffect**: allows to use side effects inside React function components. It takes two arguments: the callback function that is executed after every rendering and the optional array of dependencies (in this case callback is called only if one of the dependencies changed value)
- **useMemo**: allows memorization (or caching) of derived data.
- **useContext:** applications sometimes need to have some global information, which may be used by any child node. Even if it is possible to propagate downwards this information through props and state, it is typical impractical. For this reason, it is possible to create a **context** which is able to envelope some data. It is created with *React.createContext(defaultValue)* method and any children node can access to it by using the hook function **useContext(contextObj)**.
- **useReducer:** if the information grows in terms of complexity, both useState() and useContext() become unfeasible. It is needed to perform several invocations of setter functions in order to keep everything up to date (and this means to perform lot of redraw cycles which slow down the app and consume battery). For this reason it is possible to use the **reducer pattern** through the hook *useReducer(reducer, initial_state)*. A reducer is a pure function that given the current state of application and incoming **action** (description of change to be performed on the state), it decides how to modify the state itself and returns the updated state representation. The useReducer() hook returns the current state and a function to **dispatch** an action to the reducer.

The combination of contexts and reductions makes possible to handle globally a complex state of the application.