Appunti Applicazioni Mobili

Contents

1 Hello World App

When creating a new project, a low api version implies higher compatibility with older devices, but it also means that you may not be able to use the latest features of the Android platform. Conversely, a high api version allows you to use the latest features but may limit compatibility with older devices.
A project contains Kotlin code in the `java` folder, layouts, images, `.xml` files and translations in the `res` folder, and an `AndroidManifest.xml` file that describes the application.
The manifest is the only file that is exposed to the OS. It contains:

- Application declaration;
- Permissions;
- Intent filters;
- Targets.

1.1 Versioning

Gradle is a build system that manages dependencies and compiles the project. It uses a file called `build.gradle` to define the project structure, dependencies, and build configurations.
compileSdkVersion The version of the Android SDK used to compile the project.
minSdkVersion The minimum version of the Android SDK required to run the application.
targetSdkVersion The version of the Android SDK that the application targets.

$$\text{minSdkVersdion} \leq \text{targetSdkVersion} = \text{compileSdkVersion}$$

1.2 Deploying

Android applications must be signed before being installed on a real device. build → generate signed bundle/apk,
Using the V2 signature you can just transfer the `.apk` file to your phone.

2 Kotlin

Kotlin is a type inference language, meaning that the compiler can deduce the type of a variable based on its value. It is statically typed, which means that types are checked at compile time rather than at runtime.
It compiles to java bytecode, which means that it can run on the Java Virtual Machine (JVM) and can be used in Android development.

```
var x: Int 42 //declaration of a variable with type int
var x = 42 // Declaration of a variable with type inferred as int
val x = 42 // Declaration of a read-only variable (constant) with type inferred as
int
```

```
// Hello World example
fun main() {
  val nickname: String = "stradivarius"
  println("Hello world, my name is $nickname")
}
```

```
// If-else statement
if ( condition ) {
// Then Clause
} else {
// Else Clause
}
```

```
// Shorthand for assigments
var y = if (x == 42) 1 else 0
```

```
when ( x ) {
  in 0..21 -> println("One line clause")
  in 22..42 -> {
  println("Multiple line clause")
}
  else -> println("Default clause")
}
```

With the double dot (`..`) you can specify ranges, which originate Lists.

```
val arr: IntArray = intArrayOf(1, 2, 3) // [1,2,3]
println(arr[0])
```

Arrays are a class and can be instantiated in several ways (they also have their subtypes): Equivalent to their primitive in C: immutable in size, type-invariant

```
// Array of int of size 5 with values [0, 0, 0, 0, 0]
val arr = IntArray(5)
// Array of int of size 5 with values [42, 42, 42, 42, 42]
val arr = IntArray(5) { 42 }
// Array of int of size 5 with values [0, 1, 2, 3, 4]
var arr = IntArray(5) { it * 1 }
```

Lists can be constants or variables:

```
// Immutable List
val myList = listOf<String>("one", "two", "three")
println(myList)
// Mutable List (referenced by a val because it is the pointer)
val myMutableList = mutableListOf<String>("one", "two", "three")
myMutableList.add("four")
```

Iteration constructs

```
// While loop
var counter = 0
while (counter < myMutableList.size) {
println(myMutableList[counter])
counter++
}
// For loop
for(item in myMutableList) // Here we can use ranges as well
println(item)
```

2.1 `null` safety

One of the main features of Kotlin is its null safety. The program doesn't crash because of `null` values. Types are non-nullable, in fact variables are either

- Initialized with a value of the type, or
- Initialized with a value of the type followed by a `?`, which means that the variable can be `null`.
- Explicitly `null`, but they throw an error at compile time

Non nullable types

```
var s: String = "Hello" // Regular initialization means non-null by default
s = null // compilation error
```

Nullable types

```
var s: String? = "Hello" // Nullable initialization means it can be null
s = null // this is ok: e.g. if you print it, it will print "null"
```

Null safety

```
val l = s.length // Compiler error: "s can't be null"
val l = s?.length // If s is null then l is null (if nullable)
val l = if (s != null) s.length else -1 // Custom workaround
```

Can be used in more complex expressions:

```
val name: String? = department?.head?.getName()
```

If anything in the chain is `null`, the function is not called.

2.2 Functions

```
fun isEven(number: Int = 0): Boolean { // number is set to 0 if not passed
return number % 2 == 0
}
isEven(14)
```

```
fun Int.isEven(): Boolean { // Extend the class Int
return this % 2 == 0
}
14.isEven()
```

Higher order functions take functions as parameters or return functions

```
// Function that counts members in a List of strings that respect a certain
condition
fun List<String>.customCount(function: (String) -> Boolean): Int {
  var counter = 0
  for (str in this) {
    if (function(str))
    counter++
  }
  return counter
}
```

Lambdas Lambdas are undeclared functions that are passed directly as they are and used once.

```
val myList = listOf<String>("one", "two", "three")
val x: Int = myList.customCount { str -> str.length == 3 }
val x: Int = myList.customCountAllTypes { str -> str.length == 3 }
```

2.3 Classes

Classes are pretty much like in Java, however they typically have a primary constructor.

```
class Animal ( // Constructor is within round brackets
  val name: String,
  val legCount: Int = 4 // Default value if not passed
) {
  var sound: String = "Hey" // Property not initialized by the constructor
  init {
    println("Hello I am a $name") // Function executed at instantiation time
  }
}
val dog = Animal("dog") // Instantiation of a class into an object
val duck = Animal("duck", 2)
```

Attributes can have default access modifiers (getters and setters), or custom and private ones.

```
// Equivalent notation
var sound: String = "Hey"
    get() = field
    set(value) { field = value } // Keyword field refers to the property
// Custom notation
var sound: String = "Hey"
    get() = this.name
    private set // Setter is private

val dog = Animal("dog")
dog.sound // Will access the getter, not the property
```

A class can be extended if it's `open`. If the class is not specified as open, then it's `final` by default and cannot be extended.

```
class Dog: Animal("dog") {
    fun bark() {
      println("WOOF")
    }
}
```

```
class Duck: Animal("duck", 2) {
  fun quack() {
      println("QUACK")
  }
}
```

A class can be make `abstract`, and then it can be extended, but it cannot be instantiated. Abstract methods must be implemented and overridden in the subclasses.

```
abstract class AbstractAnimal (
    val name: String,
    val legCount: Int = 4
) {
    abstract fun makeSound()
}
```

```
class Cat: AbstractAnimal("cat") {
    override fun makeSound() {
        println("MEOW")
    }
}
```

Anonymous classes are classes that are defined without a name and are used only once. They can be used to create instances of classes that are not meant to be reused.

```
val bear = object: AbstractAnimal("bear") {
    override fun makeSound() {
        println("GROWL")
    }
}
```

In kotlin every object attribute needs to be initialized when declaring the object. This can be deferred by using `lateinit`.

```
class Animal (
    val name: String,
    val legCount: Int = 4
) {
    lateinit var sound: String
}
```

You are basically telling the compiler that the attribute is going to have a value set before using it, so there is no need to throw an error if it is not initialized at the time of declaration.

A companion object is an object that is associated with a class and can be used to create static methods and properties. It is similar to a static class in Java.

```
class Animal (
    val name: String,
    val legCount: Int = 4
) {
    companion object {
        const val Kingdom: String = "Animalia"
    }
}
println(Animal.Kingdom)
```

The example shows a constant property that can be accessed without creating an instance of the class, but it can also be an object like a factory.

2.4 Scope Functions

Scope functions are used to simplify multiple interactions with the same object.
Apply context object is the receiver `this`, returns the object itself.

```
val snake = Animal("snake") // Without
"apply"
snake.legCount = 0
snake.sound = "Hiss"
```

```
val snake = Animal("snake").apply { //
With "apply"
    legCount = 0
    sound = "Hiss"
}
```

Let context object is the lambda argument `it`.

```
val numbers = mutableListOf("one",
"two", "three", "four", "five")
// Without Let
val resultList = numbers.map
{ it.length }.filter { it > 3 }
println(resultList)
```

```
val numbers = mutableListOf("one",
"two", "three", "four", "five")
// With Let
numbers.map { it.length }.filter { it >
3 }.let {
    println(it)
// and more function calls if needed
without using a result variable
}
```

With context object passed, but the receiver is `this`.

```
val snake  = Animal("snake")
// Without with
snake.makeSound()
```

```
val snake = Animal("snake")
// With with
snake.with {
    makeSound()
}
```

Run context object is the receiver `this`, but returns the lambda result.

```
val snake = Animal("snake")
// Without run
snake.legCount = 0
val legNumbers = snake.howManyLegs()
```

```
val snake = Animal("snake")
// With run
val legNumbers = snake.run() {
legCount = 0
howManyLegs()
}
```

Also context object is the lambda argument `it`, but returns the object.

```
val numbers = mutableListOf("one",
"two", "three", "four", "five")
// Without also
numbers.add("six")
println(numbers)
```

```
val numbers = mutableListOf("one",
"two", "three", "four", "five")
// With also
numbers.also {
    it.add("six")
    println(it)
}
```

2.5 Delegation

delegation is a way to reuse behavior from another object instead of writing it yourself. Basically, you "delegate" part of your class's responsibilities to another object.

Class delegation

```
class MyService(logger: Logger) : Logger by logger {
    fun doWork() {
        log("Doing work")
    }
}

fun main() {
    val service = MyService(ConsoleLogger())
    service.doWork()  // prints: LOG: Doing work
}
```

`MyService` implements `Logger`, but all calls to `log()` are automatically forwarded to the `logger` object.

Property delegation Kotlin also lets you delegate how a property is stored or computed. The main examples are:

- `lazy` (initialize only on first use)
- `observable` (run code whenever value changes)
- custom delegates using getValue and setValue

3 Android System Architechture

The Android operating system is based on a modified version of the Linux kernel, which provides a stable and secure foundation for the platform.

It has numerous advantages:

- Portability: Android can run on a wide range of devices, from smartphones and tablets to smart TVs and wearables.

Security: Android has a robust security model that includes features such as app sandboxing, permissions, and encryption.

- Power Management: Android includes a range of power management features that help to extend battery life on mobile devices;
- Android Runtime (ART): relies on the kernel for threads and memory management;
- Manufacturers build drivers on top of a reliable kerne;

- User based permission model;
- Processes are isolated;
- Inter-process communication (IPC);
- Resources are protected from other processes;
- Each application has its own User ID (UID); this means that in android, each app is a different Linux user;
- Verified boot.

Android 5.0 Mandatory access control (MAC) between system and apps, all third party apps run in a restricted environment;

Android 8.0 Limited system calls are available to user-level apps.

Android 9.0 all non privileged apps with SDK version 28+ must run in individual SELinux sandboxes.

Android 10.0 apps have a limited raw view of the filesystem, with no direct access to paths like `/sdcard/DCIM`. However, apps retain full raw access to their package specific directories, such as `/sdcard/Android/data/<package_name>` and `/sdcard/Android/obb/<package_name>`.

3.1 Hardware Abstraction Layer (HAL)

The Hardware Abstraction Layer (HAL) is a layer of software that provides a standard interface for hardware components. It allows the Android operating system to communicate with hardware components without needing to know the details of how the hardware works.
It comes with numerous advantages:

- Shadows the real device;
- Manages different devices of the same type;
- Standard interface to expose lower level capabilities to higher level APIs

Android is agnostic about lower level driver implementations.

Android has native libraries written in C/C++ that are used by various components of the Android system.

- Graphics: Surface Manager;
- Multimedia: Media Framework;
- Database: SQLite;
- Web: WebKit;
- Font Management: FreeType;
- C libraries: bionic;

The Native Development Kit (NDK) is a set of tools that allows developers to write parts of their applications in native code, such as C or C++.
Java APIs are provided for most used libraries. The NDK can be used to squeeze extra performance out of a device to achieve lower latency or run computationally intensive tasks.

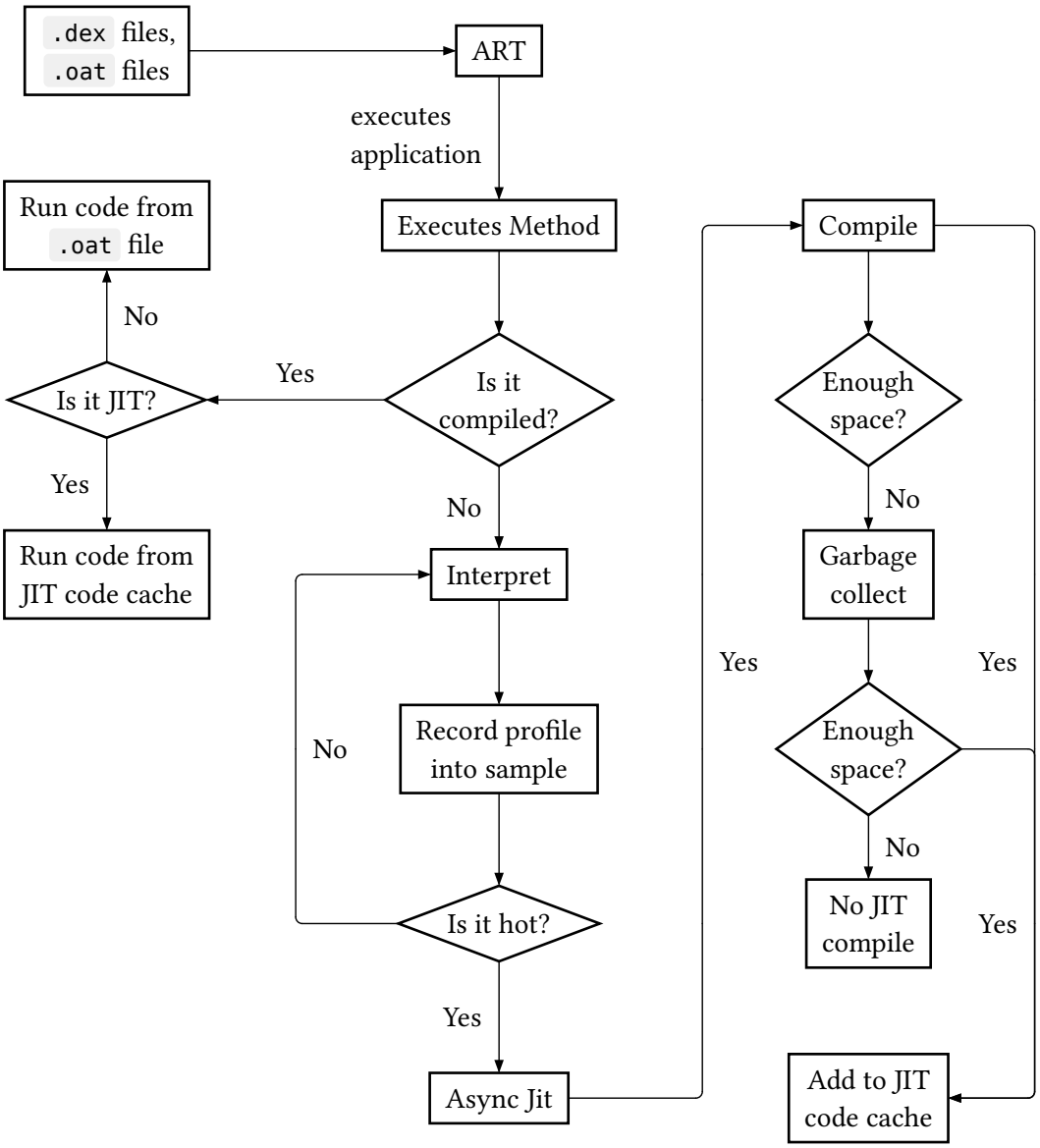
3.2 Android Runtime (ART)

It's a Java Virtual Machine (JVM) implementation optimized for memory-constrained devices.
Like the ordinary JVM it can interpret bytecode and compile it in the SDK.
ART enhances stack size, error handling and optimizes grabage collection.
Ahead-of-time (AOT) compilation is used to compile the entire application into native code when the app is installed.
Just-in-time (JIT) compilation is used to compile parts of the application into native code at runtime.

- Dex files need to be interpreted by the VM (or be JIT compiled);

- OAT files are already machine level code
- A daemon looks for uncompiled apps when the device is idle and compiles them in the background.

AOT and JIT replace the code interpretation that was classic for JAVA.



3.3 APIs

- Activity Manager
- Packet Manager
- Telephony Manager
- Location Manager
- Contents Provider
- Notification Manager
- View System, trough which you build the app UI
- Resource Manager
- Notification Manager
- Activity Manager, handles the activity lifecycle and provides a back stack
- Content Providers, share data among apps.

System Apps are pre-installed applications that come with the Android operating system and have enhanced privileges.

3.4 App Components

Activities An activity is a single screen with a user interface. It is the entry point for interacting with the user. An application can be composed of multiple screens (activities) that are loosely bound together. The home activity is shown when the user launches the application. Different activities can exchange information using intents. Each activity is composed of a list of graphic components. Some of these components (called views) can interact with the user by handling events (e.g. buttons).
The interface can be build programmatically or using XML files (declarative).

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

Listing 1: Programmatic Approach

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

Listing 2: Declarative Approach

XML layouts can be defined for different screen sizes and orientations. At runtime, android detects the current device configuration and loads the appropriate layout and resources.

Events Views can generate events when the user interacts with them. They must be managed by the developer trough the use of callbacks. A callback is a function that is passed as an argument to another function and is executed when a specific event occurs.

The main difference between Android and Java programming, is that mobile devices have limited resource capabilities. The activity lifecycle depends on the user's choices and the system's constraints. The developer must implement lifecycle methods to account for state changes in each activity: there is no main function (reactive programming).

3.5 Intents

Intents are asynchronous messages that activate core android components.
Explicit intents the component `activity1` specifies the destination of the intent (`activity2`).
Implicit intents the component `activity1` specifies the type of intent (e.g. "view a video")

3.6 Services

Services are like activities but they run in the background and do not provide an user interface. They are used for non interactive tasks (e.g. networking)

Android applications run with a distinct system identity (linux user ID and group ID), in an isolated way. Applications must explicitly share resources and data. They do this by declaring the permissions they need for additional capabilities. Applications statically declare the permissions they require. Users must give their consensus upon using the feature. Permission must be asked at runtime too.

`AndroidManifest.xml` :

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

3.7 App distribution

Each android application is contained in a single APK file (Java Bytecode, resources, libraries).

4 Activities

A mobile app experience differs from its desktop counterpart: A user lands on the application non deterministically: you can open your email app from a link in a mail, or open a map app from a link in a message. If you go there from a website, you may land on the "compose message" screen instead. These different contexts are called activities.

An activity is a screen state and the entry point for user interaction: it can be seen as a single screen. It has methods to react to certain events. An application can be composed of multiple activities: it is not an atomic whole. Android maintains a stack of activities.

```
<application ... >
    <activity android:name=".MainActivity" android:exported="true">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
</application>
```

Listing 3: Activities are declared in the manifest before being ran

4.1 Manifest

The `AndroidManifest.xml` file is a required file that describes essential information about your app to the Android build tools, the Android operating system, and Google Play.
Is what the operating system can read about your application.

It tells which activities you have and how a user can access them. `main` and `launcher` means that this activity is acced via the app icon in the home screen.

4.2 Activity Lifecycle

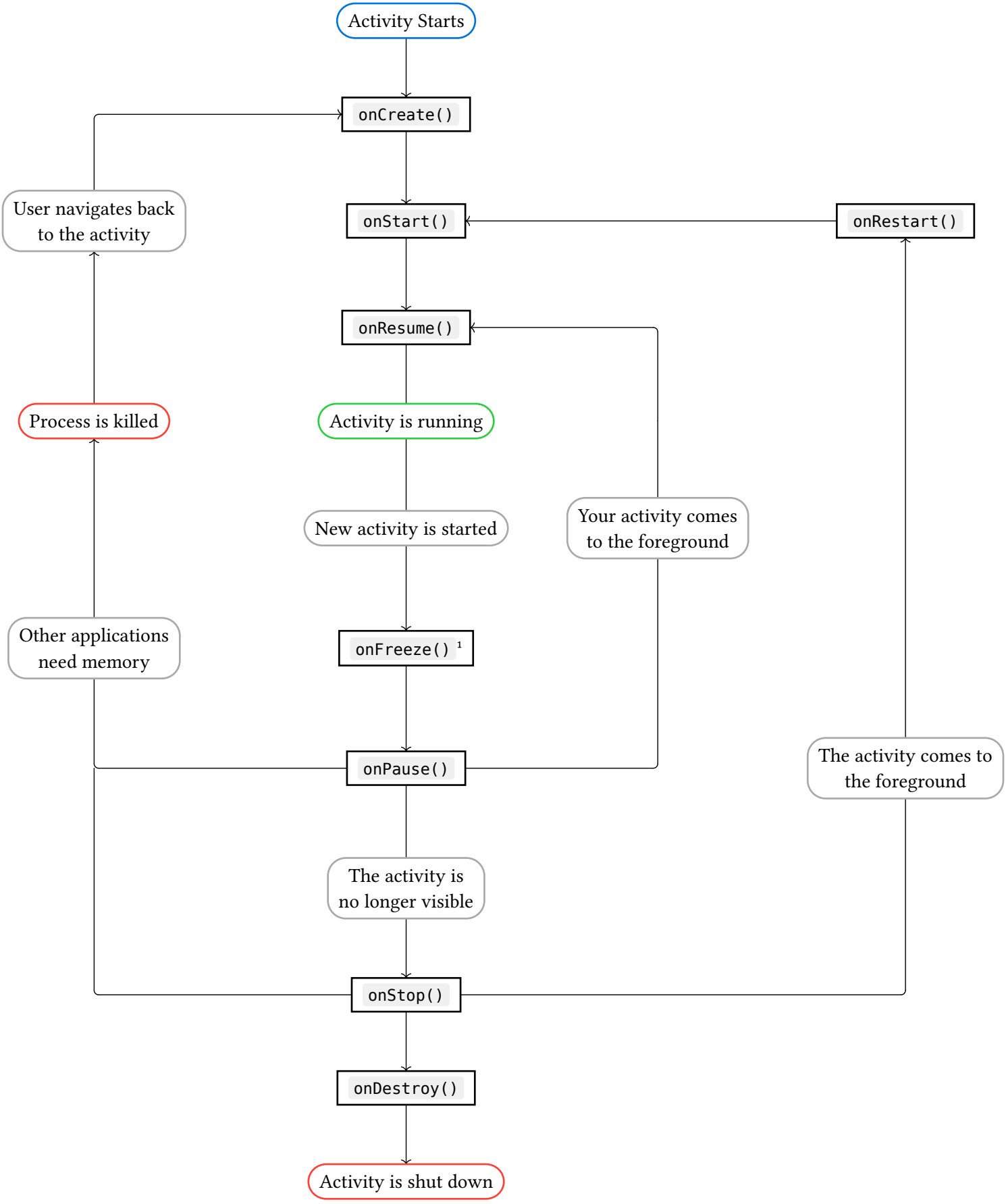
As the user navigates in and out the app, the activity can go trough states.
We use reactive programming since we put our code in callbacks, invoked when the activity transitions from one state to another.

1. Visible and interactable
2. Visibile but not interactable
3. Not visible
4. Not in memory

Resumed the activity is in the foreground and the user can interact with it.
Paused the activity is visible, maybe overlaid by another activity. It cannot execute code or receive direct inputs.
Stopped the activity is hidden in the background. It cannot execute any code.

It's not necessary to implement every method in the lifecycle, it depends on the application's complexity.
Activity lifecycles are important so that

- your application does not crash while the user is running something else on the smartphone.
- your application does not consume unnecessary resources while in the background.
- the user can safely stop your application and return to it later.



[1]Not used since 2008

4.3 Activity Lifecycle methods

`onCreate()` called when the activity is first created. It should contain the startup logic to be executed only once. It has a `Bundle` parameter that contains the activity's previously saved state. When `onCreate()` terminates, `onStart()` is called.
`onCreate()` is responsible for drawing the UI with `setContentView()` and initializing essential components

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

`onStart()` called right before it is visible to the user, where the code that maintains the UI is initialized.

```
override fun onStart() {
    super.onStart()
}
```

`onResume()` if it successfully terminates, then the activity is running. It restores the components that were disposed of during `onPause()`.

```
override fun onResume() {
    super.onResume()
}
```

`onPause()` is called when something interrupts the activity (e.g. a call or a message). It should release resources that may affect battery life and CPU usage while the activity is not in the foreground. It should also save any unsaved data (e.g. in a database) since there is no guarantee that `onStop()` will be called.

```
override fun onPause() {
    super.onPause()
}
```

It's called when
- a component from a different activity requests the foreground;
- a component comes in the foreground partially hiding the activity (e.g. a dialog);
- another window in a multi window application is selected;
- any other event that will also trigger `onStop()`.

`onRestart()` called when the activity is coming back to interact with the user after being stopped. It is followed by `onStart()`.

```
override fun onRestart() {
    super.onRestart()
}
```

`onStop()` called when the activity is no longer visible to the user. It could be called because:
- the activity is about to be destroyed;
- another activity is coming to the foreground
  It's used to perform CPU-intensive shutdown operations.

```
override fun onStop() {
    super.onStop()
}
```

4.4 Lifecycle loops

Entire Lifetime

- between onCreate and onDestroy;
- setup of global state in onCreate;
- release remaining resources in onDestroy;

Visible Lifetime

- between onStart and onStop;
- maintain resources that have to be shown to the user;

Foreground Lifetime

- between onResume and onPause;
- code should be light;

4.5 Logging

The logcat window helps debug your app by displaying logs from your device in real time.

```
Log.v ("LABEL", "message") // VERBOSE
Log.d ("LABEL", "message") // DEBUG
Log.i ("LABEL", "message") // INFORMATION
Log.w ("LABEL", "message") // WARNING
Log.e ("LABEL", "message") // ERROR
Log.wtf ("LABEL", "message") // SHOULD NEVER HAPPEN IN LIFE
```

4.6 Recreating activities

When an activity is destroyed and then navigated back, the system recreates a new instance. Usually we want everything as it was: this state is saved in a bundle called Instance State.
Android keeps the state of each view: each of them should have an unique id and no explicit code is needed for basic behavior.

If you want to save more date, you override `onSaveInstanceState()` and `onRestoreInstanceState()` and/or use a `ViewModel`.
`onSaveInstanceState()` is called right before `onStop()`.

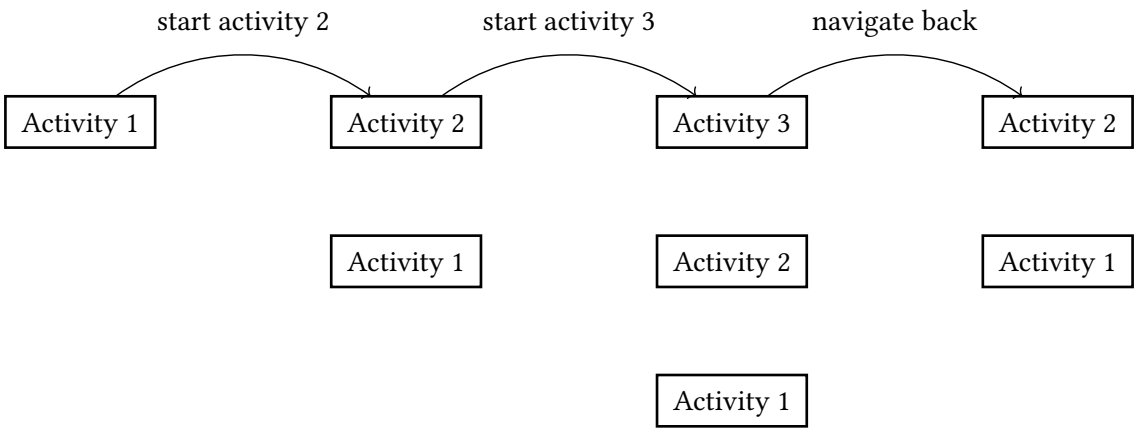
```
override fun onSaveInstanceState(savedInstanceState : Bundle) {
    super.onSaveInstanceState(savedInstanceState)
    outstate.putInt(
        STATE_SCORE, myCurrentScore
    )
}
companion object { val STATE_SCORE = "playerScore" }
```

`onRestoreInstanceState()` is called right after `onStart()`

```
override fun onRestoreInstanceState(savedInstanceState: Bundle) {
// Call the superclass to restore the views
super.onRestoreInstanceState(savedInstanceState)
myCurrentScore = savedInstanceState.getInt(STATE_SCORE)
}
```

4.7 Tasks and BackStack

Activities in the same app can occur on top of each other. In these cases, the previous activity is stopped and put in the backstack. By navigating back, the user pops the current activity from the backstack and destroys it, restoring the one on top.



Launching the same activity in two different phases of the same storyline causes the creation of two separate instances by default. This can be avoided by using flags in the calling Intent.

Navigating back to the root activity causes the app to terminate (android 11-) or brings the current task in the background (android 12+). You need intents in order to navigate trough activities.

A task is a cohesive unit that contains a storyline (a BackStack) and can be in the foreground (if the top activity is running), or in the background if all activities are stopped. A task in the background can be seen in the "recent activities" UI. An app can be made of multiple tasks (and therefore multiple BackStacks).

Launching an app from the home screen by default lands you in the same task. When you open an activity in Android, you can decide how it should start and behave. You can set this either:
- in the manifest: fixed rules written once for that activity.
- with flags in the Intent: temporary rules you choose at the moment you launch it.

4.8 The main thread

Normally, each application runs on its own linux process, called the main thread. Activities are running and keeping the main thread alive, but other components may influence it so we should be careful.

4.9 Contexts

A class activity or AppCompatActivity (like others) implement the abstract class Context.
Context is a handle to the system, providing environment references and used for:
- loading a resource;
- launching a new activity;
- creating views;
- obtaining system services.

4.10 Toasts

They are tiny messages displayed over an activity, they take the context as input. They are used to signal user confirmation or little errors. Their duration can be controlled.

```
Toast.makeText(this, "Hello world, I am a toast.", Toast.LENGTH_SHORT).show()
```

5 Views

Android views are the standard for building responsive UIs. They are built on the concept of view: any self-contained object on the screen (including containers of other views). They are the basic building blocks for user interface components.

Views occupy a rectangular area of the screen, they are responsible for drawing and event handling.

Declarative Views declared in an XML layout file:

```
<TextView
android:id = "@+id/myTextView"
android:layout_width = "match_parent"
android:layout_height = "wrap_content"
android:text = "Hello World"
android:textAlignment = "center"
/>
 <!-- This is in res/layout/activity_main.xml -->
```

They can be accessed in Java/Kotlin trough findViewById

```
lateinit var textView : TextView
textView = findViewById(R.id.myTextView)
```

Programmatic Views are created directly in Java/Kotlin by giving them a context. However, they must also be given all their visual properties in the code.

```
lateinit var textView : TextView
textView = TextView(this);
```

5.1 Handling Events

```
<Button
android:id= "@+id/button1"
android:text= "First Button"
/>
```

```
lateinit var button: Button
button = findViewById(R.id.button1)
```

Views are interactive components: upon certain actions, an approrpiate event will be fired (click, long click, focus, items selected, items checked, drag...) These events can be handled by

XML Callbacks this is possible only for a limited set of components.

```
<Button
    android:id = "@+id/button1"
    android:text= "First Button"
    android:onClick="doSomething"
/>
```

Event Handlers we extend the view class and override the call (e.g. `onTouchEvent()`). This is impractical, its much better to have a separate class that handles all the logic.

Event Listeners each view can delegate the reaction to one to an object that implements the dedicated listener interface. Each listener is a Single Abstract Method (SAM) interface. Each listener handles a single type of event and contains a single callback method.

e.g. assign the `clickListener` to the `View` trough `setOnClickListener()`

```
lateinit var button : Button
class MainActivity : AppCompatActivity(), OnClickListener {
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        button = findViewById(R.id.button1)
        button.setOnClickListener(this)
    }
    override fun onClick(v: View?) { /* Behavior */ }
}
```

Or, you can make an anonymous object:

```
button.setOnClickListener(object: OnClickListener {
    override fun onClick(v : View?) { /* Behavior */ }
})
```

The most common implementation is with lambdas:

```
button.setOnClickListener { view ->
    // 'view' is the button (or view) that was clicked
    Toast.makeText(this, "You clicked on view with ID: ${view.id}",
Toast.LENGTH_SHORT).show()
}
```

5.2 View Hierarchy

`View` objects are invisible containers that define the layout for the views declared in them. `ViewGroup` is a subclass of `View`.

A `Layout` must extend a `ViewGroup`. Every view in a `Layout` needs to specify `android:layout_height`, `android:layout_width` and a dimension or one between `match_parent` or `wrap_content`.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android=
"http://schemas.android.com/apk/res/android"
 android:orientation="vertical"
 android:layout_width="match_parent"
 android:layout_height="match_parent">
  <TextView
 android:id="@+id/textView"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:text="TextView"
 android:textAlignment="center" />
  <Button
 android:id="@+id/button1"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:text="First Button" />
  <Button
 android:id="@+id/button2"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:text="Second Button" />
</LinearLayout>
```

Your layout is then compiled into a `View` resourrce that has to be loaded by the `Activity` making use of it.

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

Each `View` can have an ID (`android:id="@+id/button1"`). `@` means “parse and expand the rest of the string as an id resource”; `+` means “this is going to be added as a new id in R.java”

5.3 Layouts

XML layout attributes named `layout_<something>` define layout parameters for the view that are appropriate for the `ViewGroup` in which it resides. Each parent `Layout` specifies `LayoutParams` that each children `View` must implement. Each layout needs the children `View`s to implement `layout_width` and `layout_height`, which can be:

- `match_parent`;
- `wrap_content`;
- `0dp`: take up all available space (a dp is a density independent pixel)
- a custom value

Android also supports `padding` and `margin`. Padding is a `View` property and margin is a `Layout` property. The most common static layouts are

- `LinearLayout`
- `RelativeLayout`
- `TableLayout`
- `FrameLayout`
- `ConstraintLayout`

A `Layout` can be declared inside another `Layout`.

5.3.1 Linear Layout

Organizes views on a single row or column, depending on `android:layout_orientation`: either vertical or horizontal

```
<LinearLayout LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
 android:orientation="vertical"
 android:layout_width="match_parent"
 android:layout_height="match_parent">
  <Button
 android:id="@+id/button1"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:text="First Button" />
  <Button
 android:id="@+id/button2"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:text="Second Button" />
  <Button
 android:id="@+id/button3"
 android:layout_width="wrap_content"
 android:layout_height="match_parent"
 android:layout_gravity="center_horizontal"
 android:text="Third Button" />
</LinearLayout>
```

if one of the views has a weight, then the views will take up the entire dimension. Weights tell us how important that `View` is. The best value is usually `0dp`.

If the elements do not fit, the layout can be wrapped in a `ScrollView` or `HorizontalScrollView`

5.3.2 ConstraintLayout

They define constraints (top/bottom/left/right) for each `View`. Each constraint has to be defined to another previously declared `View`, layout or invisible guideline. It's a flat `View` hierarchy and it's the default one.

Each view needs at least one constraint per plane (either horizontal or vertical). Constraints can only be defined between anchor points sharing the same plane. Each handle can define one constraint. Multiple handles can define a constraint to a single anchor point. Adding two opposite constraints places the view in the middle and can adjust the ratio by setting the `bias`.

A size of `0dp` means "match constraint" or "take all available space".

5.3.3 RelativeLayout

`RelativeLayout` displays child views in relative positions The position of each view can be specified as relative to siblings elements (such as to the left-of or below another view) or in postions relative to the parent `RelativeLayout` area (such as aligned to the bottom, left or center).

5.3.4 TableLayout

`TableLayout` positions its children into rows and columns. TableLayout containers do not display border lines for their rows, columns or cells. The table will have as many columns as the row with the most cells. A table can leave cells empty. Cells can span multiple columns, just like in HTML.

5.4 Dynamic Layouts

Sometimes the layouts need to be populated at runtime with views (`ListView`, `GridView`)... These layouts subclass `AdapterView`: they can use an adapter to retriece data from another source and map it into the elements of the `AdapterView`. `AdapterView` is a `ViewGroup` subclass: its subchildren are determined by an `Adapter`. Some subclasses are:

- `ListView`;
- `GridView`;
- `Spinner`;
- `Gallery`;
- `ExpandableListView`;
- `TabLayout`;

`Adapters` are used to visualize dynamic data (e.g. `ArrayAdapter`)

```
// Create a list adapter for a string list
String[] data = {"First", "Second", "Third"};
ArrayAdapter<String> adapter =
new ArrayAdapter<String>(
    this,
    android.R.layout.simple_list_item_1,
    data
);
ListView listView = findViewById(R.id.listView);
listView.setAdapter(adapter);
```

The `Adapter` takes in input:

- the context;
- A layout to be inflated in the single element of the dynamic layout
- the data structure that holds the actual data

```
// Create a list adapter for a string list
val data: Array<String> = arrayOf("First", "Second", "Third")
val adapter = ArrayAdapter<String>(
    this,
    android.R.layout.simple_list_item_1,
    data
)
val listView: ListView = findViewById(R.id.listView)
listView.adapter = adapter
```

5.5 Widgets

`Views` are organized in a hierarchy of classes. Widgets are `Views` with their own behavior implemented.

5.6 TextView

`TextViews` are not directly editable by users, and display static information.

```
<TextView
    android:text="Hello World!"
    android:id="@+id/textLabel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content”
/>
```

Simple strings can be turned into links automatically

```
Linkify.addLinks(textView,
    Linkify.WEB_URLS or
    Linkify.EMAIL_ADDRESSES or
    Linkify.PHONE_NUMBERS
    )
```

5.7 EditText

Similar to a `TextView` but editable by thie users.

```
<EditText
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:ems="10"
 android:inputType="text"
 android:text="Insert your input here" />
```

5.8 AutocompleteTextView

As soon as the user starts typing, hints are displayed. A list of hints is given trough an `Adapter`.

```
val autocompleteTextView: AutoCompleteTextView = findViewById(R.id.autoCompleteText)
autocompleteTextView.setAdapter( ArrayAdapter<String> (
 this,
 android.R.layout.simple_dropdown_item_1line,
 arrayOf("Darth Vader", "Darth Sidious", "Darth Tyranus")
))
```

5.9 Spinner

Provides a quick way to select values from a set. The value can be defined in the XML entries tag or trough the `SpinnerAdapter`.

```
val spinner: Spinner = findViewById(R.id.spinner)
val spinnerAdapter = ArrayAdapter<String>(
this, android.R.layout.simple_spinner_item,
    arrayOf("Anakin Skywalker", "Sheev Palpatine", "Count Dooku"))
spinnerAdapter.setDropDownViewResource(
    android.R.layout.simple_spinner_dropdown_item
)
spinner.adapter = spinnerAdapter
```

5.10 CompoundButton

A subclass of `Button`. Represents a button with a state:

- `CheckBox`;
- `ToggleButton`;
- `Switch`;
- `RadioButton`

It responds to `OnCheckedChangeListener()`

If enclosed in a `RadioGroup`, it will imply a mutually exclusive multiple selection.

```
val radioGroup: RadioGroup = findViewById(R.id.radioGroup)
radioGroup.setOnCheckedChangeListener { _, checkedId ->
    when(checkedId) {
        R.id.radioRed -> /* Do your stuff */
        R.id.radioGreen -> /* Do your stuff */
        R.id.radioBlue -> /* Do your stuff */
        else -> /* Do your stuff */
    }
}
```

5.11 RecyclerView

RecyclerView makes it easy to efficiently display large sets of data. You supply the data and define how each item looks, and the `RecyclerView` library dynamically creates the elements when they're needed. When elements go off screen, they (their view) isn't destroyed, it is instead resued for elements that come on the screen.

It's like a highly customizable `ListView`, where you can add, remove and update elements at runtime without redrawing it completely everytime something changes.

1. call `notifyDataSetChanged()`
2. `notifyItemInserted()` or `notifyItemRemoved()`

Step 1 Define the layout of a single element. A `CardView` is a styled container that displays data, using elevation and shadow, sticking to a consistent look across the platform.

```
<androidx.cardview.widget.CardView
  xmlns:android="http://schemas.android.com/apk/res/android"
 android:layout_width="match_parent"
 android:layout_height="wrap_content">
  <LinearLayout
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:orientation="horizontal">
    <TextView
 android:id="@+id/todoTitle"
 android:layout_width="0dp"
 android:layout_height="wrap_content"
 android:layout_weight="1"
 android:textSize="20sp" />
    <CheckBox
 android:id="@+id/todoCheck"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_weight="0"
 android:text="Done?" />
  </LinearLayout>
</androidx.cardview.widget.CardView>
```

Step 2 define a companion class for the element to hold the data. The TODO has a title that goes into the `TextView` and a boolean value for the `CheckBox`.

```
data class Todo(
    var todoTitle: String,
    var done: Boolean = false
)
```

Step 3 define a `ViewHolder` which is the runtime container that will get externally inflated with the element layout and then holds the references to its children Views, so that they can be customized at runtime.

```
class TodoViewHolder(itemView: View): ViewHolder(itemView) {
    val tvTodoTitle: TextView = itemView.findViewById(R.id.todoTitle)
    val cbDone: CheckBox = itemView.findViewById(R.id.todoCheck)
}
```

Step 4 define the `Adapter` which takes in input the data (a list of TODOs) and generates a `ViewHolder` for each entry, overriding `RecyclerView.adapter`

```
ass TodoAdapter (private val todos: MutableList<Todo>): Adapter<TodoViewHolder>() {
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
TodoViewHolder { ... }
    override fun onBindViewHolder(holder: TodoViewHolder, position: Int) { ... }
    override fun getItemCount(): Int { ... }
}
```

Step 4a `onCreateViewGolder` is invoked when a new elements needs to be drawn. It is expected to return the right `ViewHolder`, inflated with the right layout. The parent is the empty container reserved for this element.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): TodoViewHolder {
    return TodoViewHolder(
        LayoutInflater.from(parent.context).inflate(R.layout.todo_card, parent,
false)
    )
}
```

Step 4b `onBindViewHolder` is invoked when then ew element is given a position within the `RecyclerView`. Here we first need to populate the fields of the element.

```
override fun onBindViewHolder(holder: TodoViewHolder, position: Int) {
 holder.apply {
    tvTodoTitle.text = todos[position].todoTitle
    cbDone.apply{
        isChecked = todos[position].done
        setOnCheckedChangeListener { _, b -> todos[position].done = b }
        }
    }
}
```

Step 4c `getItemCount()` needs to output the number of items in our data structure:

```
override fun getItemCount(): Int {
    return todos.size
}
```

Step 5 Assign a `LayoutManager` when creating the `RecyclerView`. It will arrange the items in a defined fashion.

```
val recyclerTodo: RecyclerView = findViewById(R.id.recyclerTodo)
recyclerTodo.adapter = TodoAdapter(mutableListOf())
recyclerTodo.layoutManager = LinearLayoutManager(this)
```

6 Resources

An app must be able to tolerate feature variability and provide a flexible user interface that adapts to different screen configurations. Android separates the code from the application's resources. Resources are defined with a declarative XML-based approach. *Resources are everything that is not code*, including XML layout files, language packs, images, audio/video files

Data presentation is separated from the data management. It provides alternative resources to support specific device configurations. Different resources are used for different device configurations.

1. Define two XML layouts for two different devices;
2. At runtime, android detects the current device configuration and loads the appropriate resources for the app without having to recompile.

```
My Project
├── java
└── res
    ├── layout (application XML layouts)
    ├── values (application lables)
    └── drawable (application images)
```

Each resource has a name/identifier. `values/string.xml` contains all the text that the application uses, like the name of buttons, labels, default text etc.

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
 <string name="hello”> Hello world! </string>
 <string name="labelButton"> Insert your username </string>
</resources>
```

The `R` class connects code and resources. It's an automatically generated file, recreated in case of changes in the `res/` directory.

Each resource is associated with an unique identifier (ID), that allows its access, composed of
- resource type: string, color, menu, drawable,...;

1. resource name: either the filename without the extension, or the value in the XML `android:attribute` attribute.

When the resource is a `View`, the ID must be specified explicitly and does not use the `type + name` scheme. `android:id="@+id/button1"` means that the view will be seen as an `id` resource.

6.1 Resource Access

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

- `<package_name>` is the name of the package in which the resource is located;
- `<resource_type>` is the name of the resource type;
- `<resource_name>` is the name of the resource filename without the extension or the `android:name` attribute value in the XML element.

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="my_red"> #FF3333 </color>
    <string name="labelButton"> Press Here! </string>
    <string name="labelText"> Hello world! </string>
</resources>
```

From Java/Kotlin Code `[package_name.]R.resource_type.resource_name`

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

```kotlin
// Get a string resource from the string.xml file
// when assigning to a variable use context.getResources()
val hello: String = this.getResources().getString(R.string.hello)
// Get a color resource from the string.xml file
val myRed: Color = getResources().getColor(R.color.my_red)
// Load a custom layout for the current screen
setContentView(R.layout.layout_main)
// Set the text on a TextView object using a resource ID
// keyword as is equivalent to explicit cast
val msgTextView = findViewById(R.id.label1) as TextView
msgTextView.setText(R.string.labelText)
```

Values Example

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_title"> Example Application </string>
  <string name="label"> Hello world! </string>
  <integer name="value"> 53 </integer>
  <string-array name="nameArray">
    <item> John Bonham </item>
    <item> Frank Zappa </item>
  </string-array>
  <integer-array name="valArray">
    <item> 1 </item>
    <item> 2 </item>
  </integer-array>
</resources>
```

px	pixel units
in	inch units
mm	millimeter units
pt	points of 1/72 inch
dp[2]	abstract unit, independent from pixel density
sp	abstract unit, independent from pixel density of a display (font)

6.2 Style

A style is a set of attributes that can be applied to a specific component of the gui or to the whole screen or app (theme) to change their appearance. A style is an XML resource that is referenced using the value provided in the name attribute. Styles can be organized in a hierarchical structure. A style can inherit properties from another style, trough the parent attribute.

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="MyTheme" parent="Theme.Material3.DayNight.NoActionBar">
        <item name="colorPrimary">@color/coin_yellow</item>
        <item name="colorSecondary">@color/hound_grey</item>
    </style>
</resources>
```

Listing 5: Defining a theme

```xml
<Button style="@style/MyTheme"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:text="Push me" />
```

Listing 6: Applying a style to a `View`

6.3 Drawables

A drawable resource is a general concept for a graphic that can be drawn:

- images;
- XML resources with attribute such as `android:drawable` and `android:icon`

An `XMLBitmap` is an XML resource that points to a bitmap file.

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

```
<?xml version="1.0" encoding="utf-8"?>
<bitmap xmlns:android="http://schemas.android.com/apk/res/android"
    android:src="@drawable/tile"
    android:tileMode="repeat" />
```

A BitMap file is a `.png`, `.jpg` or a `gif` file. Android creates a BitMap resource for any of these files saved in the `res/drawable` directory.

```
val drawing: Drawable = theme.resources.getDrawable(R.drawable.AndroidQuestion)
// theme is the property access syntax for getTheme()
// theme.resources uses the resources for the theme associated with the context
// alternative syntax for getDrawable(id, theme), similar to getColor
```

The most common view that displays images is the `ImageView` with XML tag `<ImageView>`

6.4 MipMap

The mipmap directory is dedicated to all images that are used as icons for:

- app launcher;
- app notifications;
- app bar

Icons are retrieved by the manifest file

```
<application ...
android:icon="@mipmap/ic_launcher"
 android:roundIcon="@mipmap/ic_launcher_round"
> ... </ application>
```

Image Asset Studio is a tool to create icons

6.5 Other Resources

Raw is used for resources that have no runtime optimization (audio/video files). They can be accessed as a stream of bytes:
`val inputStream: InputStream = resources.openRawResource(R.raw.videoFile)`

XML `res/xml` contains arbitrary xml files that can be read at runtime trough `R.xml.<filename>`. They can be parsed trough an XML parser:
`val xmlParser: XmlResourceParser = resources.getXml(R.xml.my_repository)`.

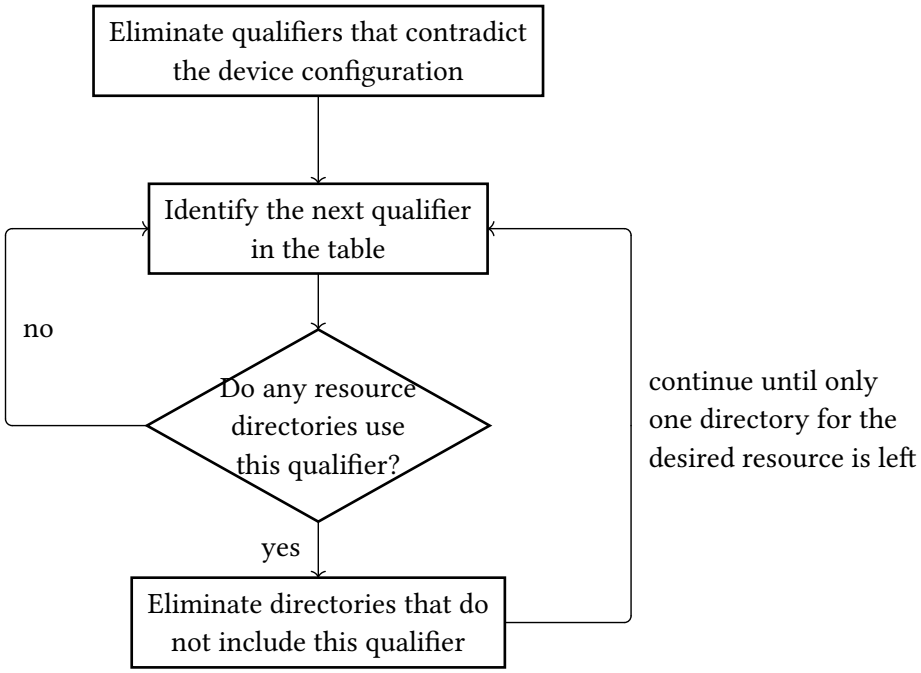
6.6 Resource Alternatives

Android applications should provide alternative resources to support specific device configurations (languages, screen orientations). To specify configuration-specific alternatives you create a new directory in `res/` named like `<resources-name>-<qualifier>`, and save the respective alternative resources in this directory.

```
res
├── values-en
└── values-it
```

`resources_name` is the directory name of the corresponding default resources, `qualifier` is a name that specifies an individual configuration for which these resources are to be used.

When the application requests a resource for which there are multiple altneratives, android selects which alternatve resource to use at runtime, depending on the current device configuration.



Best practices

- provide default resources for your application;
- provide alternative resources based on the target market of your application;
- avoid unnecessary or unused resource alternatives;
- use alias to reduce the duplicated resources;

7 Intents