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 $\rightarrow$ 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.

```kotlin
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.

```kotlin
// 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.

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

```kotlin
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.

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

```kotlin
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.

```kotlin
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`.

```kotlin
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.

```kotlin
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.

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

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

Let context object is the lambda argument `it`.

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

```kotlin
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`.

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

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

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

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

```kotlin
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.

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

```kotlin
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

```kotlin
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;

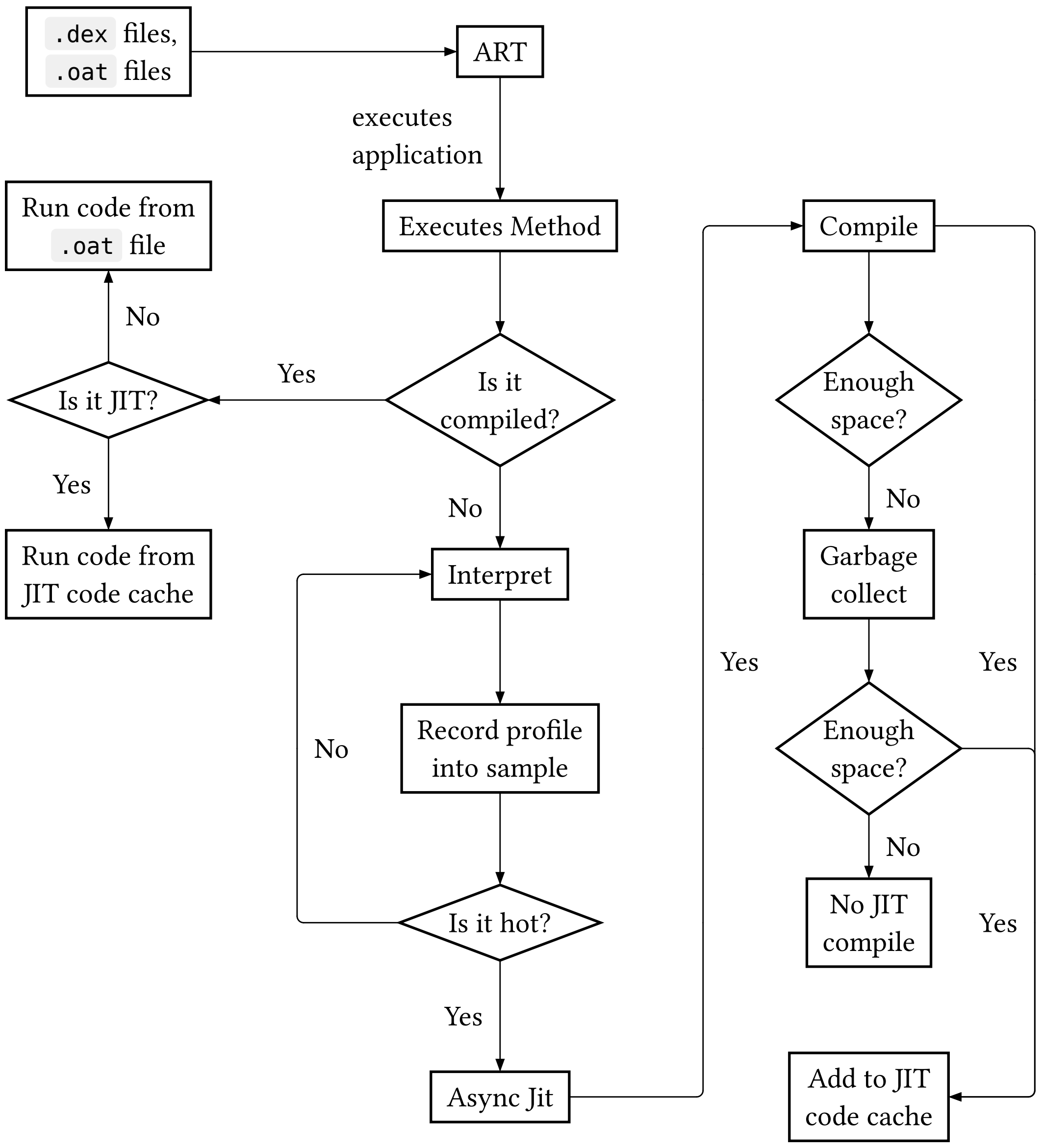
The Native Development Kit (NDK) is a set of tools that allows developers to write parts of their applications in native code, such as C or C++.
Java APIs are provided for most used libraries. The NDK can be used to squeeze extra performance out of a device to achieve lower latency or run computationally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

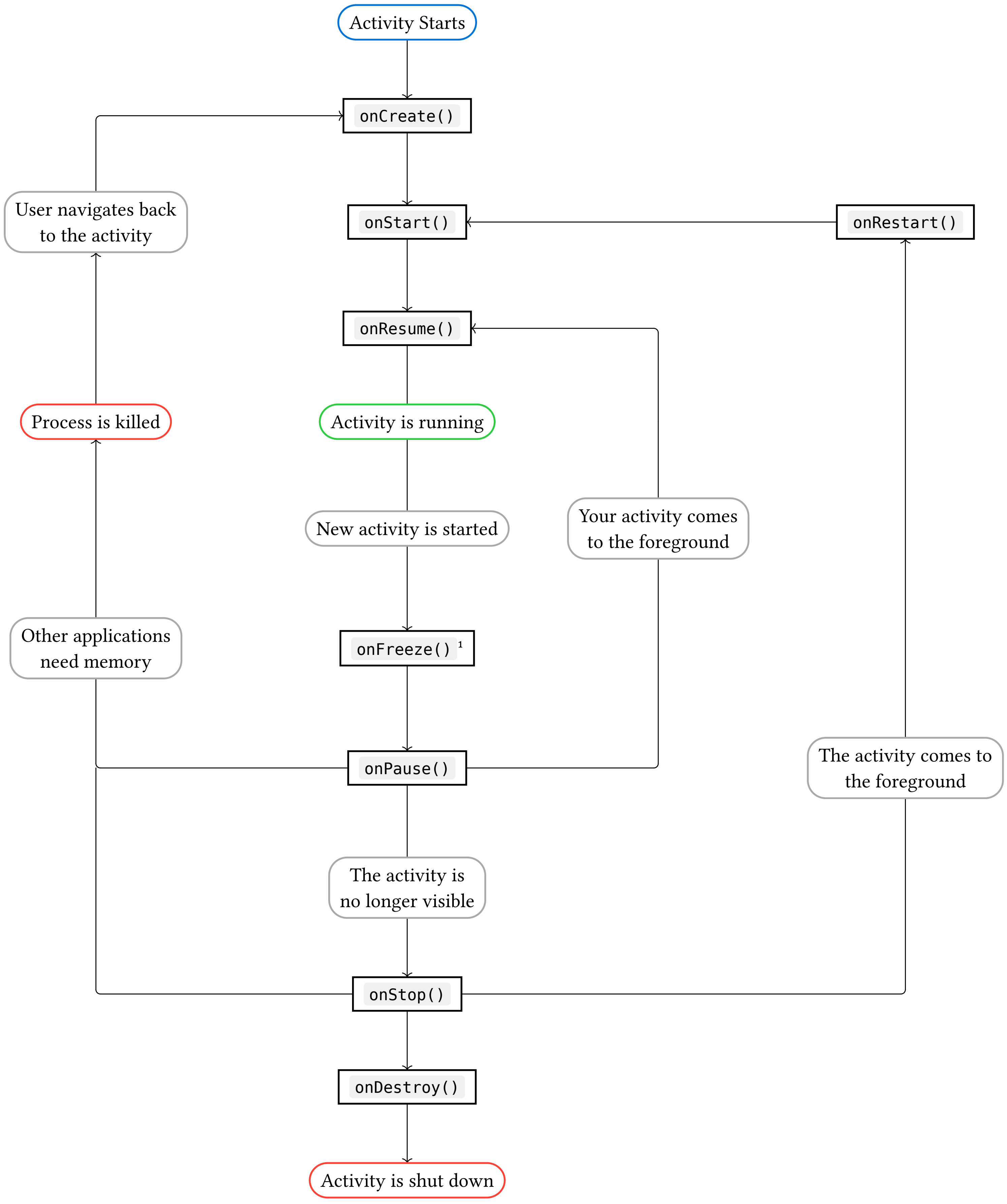
4.2 Activity Lifecycle

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()`