Developer Student Clubs

Intro to Android development

Android OS, Gradle, Kotlin, UI Frameworks & more

Stelios Papamichail
Android Engineer @Threenitas
@MikePapamichail

Google Developers

Agenda

Here’s what we’re covering today

- What is Android?
- Mobile app development workflows
- Android’s build system, Gradle
- Java & Kotlin for Android development
- UI frameworks (View System & Jetpack Compose)
- Android Studio overview
- (Bonus) Development Lifecycle **@Threenitas**

What is Android?

Developer Student Clubs

Android OS

An open-source operating system

Android is an **open-source** operating system that is the largest installed base among various mobile platforms across the globe. Hundreds of millions of mobile devices are powered by Android in more than 190 countries of the world. It conquered around **71%** of the global market share by the end of 2021.

Source : [Global mobile OS market share 2022 | Statista]

Mobile app development workflows

Setup ⟶ Code⟶ Build ⟶ Iterate ⟶ Publish

Developer Student Clubs

Workspace setup

This is the first step of the Android app development process. If you followed the prerequisites for this event, you should have already installed all the necessary tools! We will go over Android Studio later on as well, but if you need more details, see the Android Studio installation page and the guide to creating a project.

Complete a walkthrough with Android Studio and learn some Android development fundamentals with the Build your first Android app guide.

Coding Phase

Once you have set up your workspace, you can begin writing your app. Android Studio includes a variety of tools and intelligence to help you work faster, write quality code, design a UI, and create resources for different device types. In this step, you should consider the architecture that you're going to use for your app, as well as the programming language(s) of choice.

For more information about the tools and features available, see Write your app.

Build & Run Phase

During the build and run phase, you build your project into a **debuggable APK** package that you can install and run on the emulator or an Android-powered device.

You can also customize your build in this phase. For example, you can create build variants that produce different versions of your app from the same project, and shrink your code and resources to make your app smaller. All the aforementioned steps can be executed by leveraging Gradle build scripts. For an introduction to custom build configurations, see Configure your build.

Debug, Profile & Test

In this **iterative** phase, you continue developing your app while eliminating bugs and optimizing app performance. For help to debug and optimize your app, test your app in Android Studio.

For more information about debugging, read Debug your app and Write and view logs with Logcat.

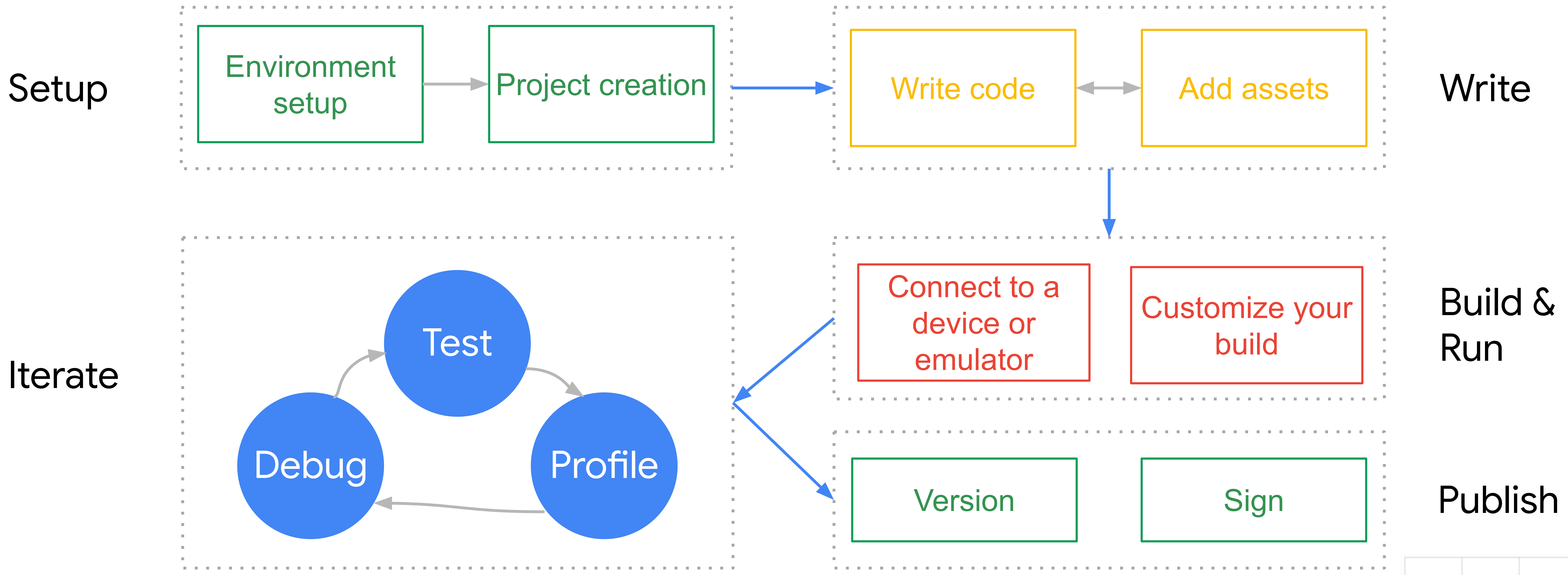
To view and analyze various performance metrics such as memory usage, network traffic, CPU impact, and more, see Profile your app performance.

Publishing your app

To prepare your app for release to users, you will need to build an Android App Bundle, sign it with a security key, and get ready to publish to the Google Play Store. The publishing workflow can be performed manually (i.e., for indie developers) or it can be automated using CI/CD pipelines (i.e., in the case of companies).

For more information, see the Publish your app.

Workflow Summary



Gradle

Android’s advanced & flexible build system

Developer Student Clubs

Android & Gradle

Android Studio uses Gradle as the foundation of the build system, with more Android-specific capabilities provided by the Android Gradle plugin. This build system runs as an integrated tool from the Android Studio menu and independently of the command line. You can use the features of the build system to do the following:

- Customize, configure, and extend the build process.
- Create multiple APKs for your app with different features, using the same project and modules.
- Reuse code and resources across source sets.

By employing the flexibility of Gradle, you can achieve all of this without modifying your app's core source files.

Build files

Android Studio build files are named **build.gradle**. They are plain text files that use Groovy syntax to configure the build with elements provided by the Android Gradle plugin. Each project has one top-level build file for the entire project and separate module-level build files for each module.

When you import an existing project, Android Studio automatically generates the necessary build files.

Starting with Android Studio Giraffe, new projects use the **Kotlin DSL** (build.gradle.kts) by default for build configuration. This offers a better editing experience than the Groovy DSL (build.gradle) with syntax highlighting, code completion, and navigation to declarations. To learn more, see the Gradle Kotlin DSL Primer

What does Gradle do?

The Android build system compiles app resources and source code and packages them into **APKs** or **Android App Bundles** that you can test, deploy, sign, and distribute.
Android Studio uses Gradle, to automate and manage the build process while letting you define flexible, custom build configurations.

Each build configuration can define its own set of code and resources, while reusing the parts common to all versions of your app. The Android Gradle plugin works with the build toolkit to provide processes and configurable settings that are specific to building and testing Android apps.

Java & Kotlin

The primary programming languages for Android

Developer Student Clubs

Java & Kotlin in Android

The backstory

Kotlin, which is developed and maintained by JetBrains, was made the official language for Android Development in Google I/O 2017! Previously, Java was considered the official language for Android Development and is still used in legacy projects or by developers who simply prefer coding in it.

Kotlin

Kotlin

At Google I/O 2019, Google announced that Android development will be increasingly Kotlin-first. Kotlin is an expressive and concise programming language that reduces common code errors and easily integrates into existing apps. If you're looking to build an Android app, it's recommended that you start with Kotlin to take advantage of its best-in-class features.

To support Android development using Kotlin, Google co-founded the Kotlin Foundation and have ongoing investments in improving compiler performance and build speed. To learn more about Android's commitment to being Kotlin-first, see Android's commitment to Kotlin.

What does Kotlin-first mean?

When building new Android development tools and content, such as Jetpack libraries, samples, documentation, and training content, the Android team will design them with Kotlin users in mind while continuing to provide support for using our APIs from the Java programming language.

To see some examples of this, see Kotlin-First in Android!

Benefits of Kotlin

- **Expressive and concise**: You can do more with less. Express your ideas and reduce the amount of boilerplate code. 67% of professional developers who use Kotlin say Kotlin has increased their productivity.
- **Safer code**: Kotlin has many language features to help you avoid common programming mistakes such as null pointer exceptions. *Android apps that contain Kotlin code are 20% less likely to crash.*
- **Interoperable**: Call Java-based code from Kotlin or call Kotlin from Java-based code. *Kotlin is 100% interoperable with the Java programming language*, so you can have as little or as much of Kotlin in your project as you want.
- **Structured Concurrency**: Kotlin coroutines make asynchronous code as easy to work with as blocking code. Coroutines dramatically simplify background task management for everything from network calls to accessing local data.

UI Frameworks in Android

The view system & Jetpack Compose

Developer Student Clubs

UI Frameworks in Android

View system & Jetpack Compose

There are two main UI frameworks used in Android development. The first and “oldest” one is called the “View System” and the newest & recommended one is called “Compose”, and is part of Android's Jetpack libraries. Let's go over these two frameworks with a short introduction and see why Compose is the cool new kid in town.

The View system

Views & ViewGroups

A layout defines the structure for a user interface in your app, such as in an activity. All elements in the layout are built using a hierarchy of View (also called Widgets) and ViewGroup (also called Layouts) objects. **A View usually draws something the user can see and interact with.**

Whereas a **ViewGroup is an invisible container that defines the layout structure** for View and other ViewGroup objects, as shown in figure 1.

Developer Student Clubs

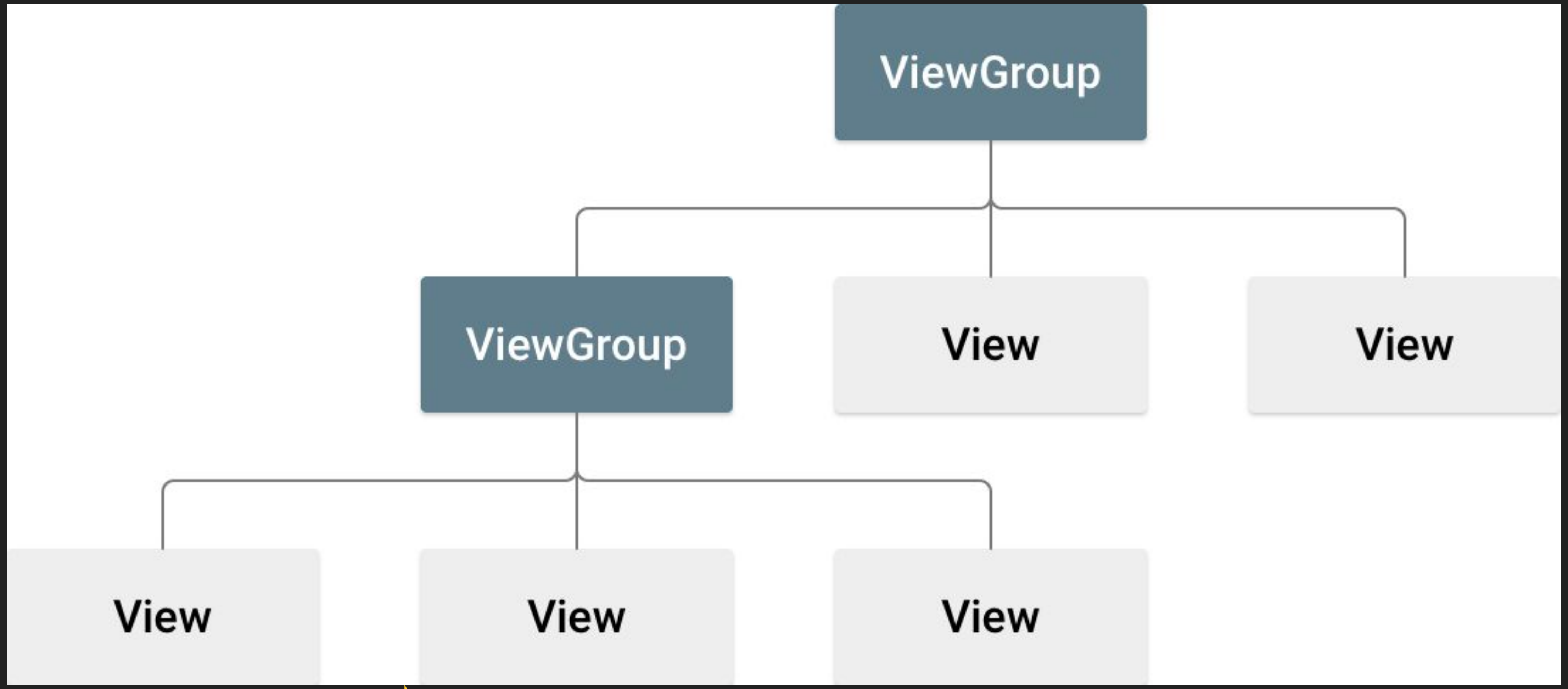


Illustration of a view hierarchy, which defines a UI layout

The View system

Declaring a layout

You can declare a layout in two ways:

- **Declare UI elements in XML**. Android provides a straightforward XML vocabulary that corresponds to the View classes and subclasses, such as those for widgets and layouts. You can also use Android Studio's Layout Editor to build your XML layout using a drag-and-drop interface.
- **Instantiate layout elements at runtime**. Your app can create View and ViewGroup objects (and manipulate their properties) programmatically.

Declaring your UI in XML allows you to separate the presentation of your app from the code that controls its behavior. Using XML files also makes it easy to provide different layouts for different screen sizes and orientations (see Supporting Different Screen Sizes).

Jetpack Compose

A new paradigm for UI development

Jetpack Compose is Android's **modern toolkit for building native UI**. It simplifies and accelerates UI development on Android, bringing your apps to life with less code, powerful tools, and intuitive Kotlin APIs. It makes building Android UI faster and easier. Let's quickly see some of the main benefits of jetpack compose!

Less Code

Writing less code affects all stages of development: as an author, you get to focus on the problem at hand, with less to test and debug and with fewer chances of bugs; as a reviewer or maintainer, you have less code to read, understand, review and maintain.

- Compose allows you to do more with less code, compared to using the Android View system: Buttons, lists or animation - whatever you need to build, now there's less code to write.

- **The code you're writing is written only in Kotlin**, rather than having it split between Kotlin and XML. Code written with Compose is simple and easy to maintain whatever you're building.

Intuitive

Compose uses a declarative API, which means that all you need to do is describe your UI and Compose takes care of the rest. The APIs are intuitive & easy to discover and use.

- With Compose, you build small, **stateless** components that are not tied to a specific activity or fragment. That makes them easy to reuse and test.

- In Compose, state is explicit and passed to the composable. That way, there's one single source of truth for the state, making it encapsulated and decoupled. Then, as app state changes, your UI automatically updates.

Accelerates Development

Compose is compatible with all your existing code: you can call Compose code from Views and Views from Compose. Most common libraries like Navigation, ViewModel and Kotlin coroutines work with Compose, so you can start adopting when and where you want.

- Using the full Android Studio support, with features like live previews, you get to iterate and ship code faster!

Powerful

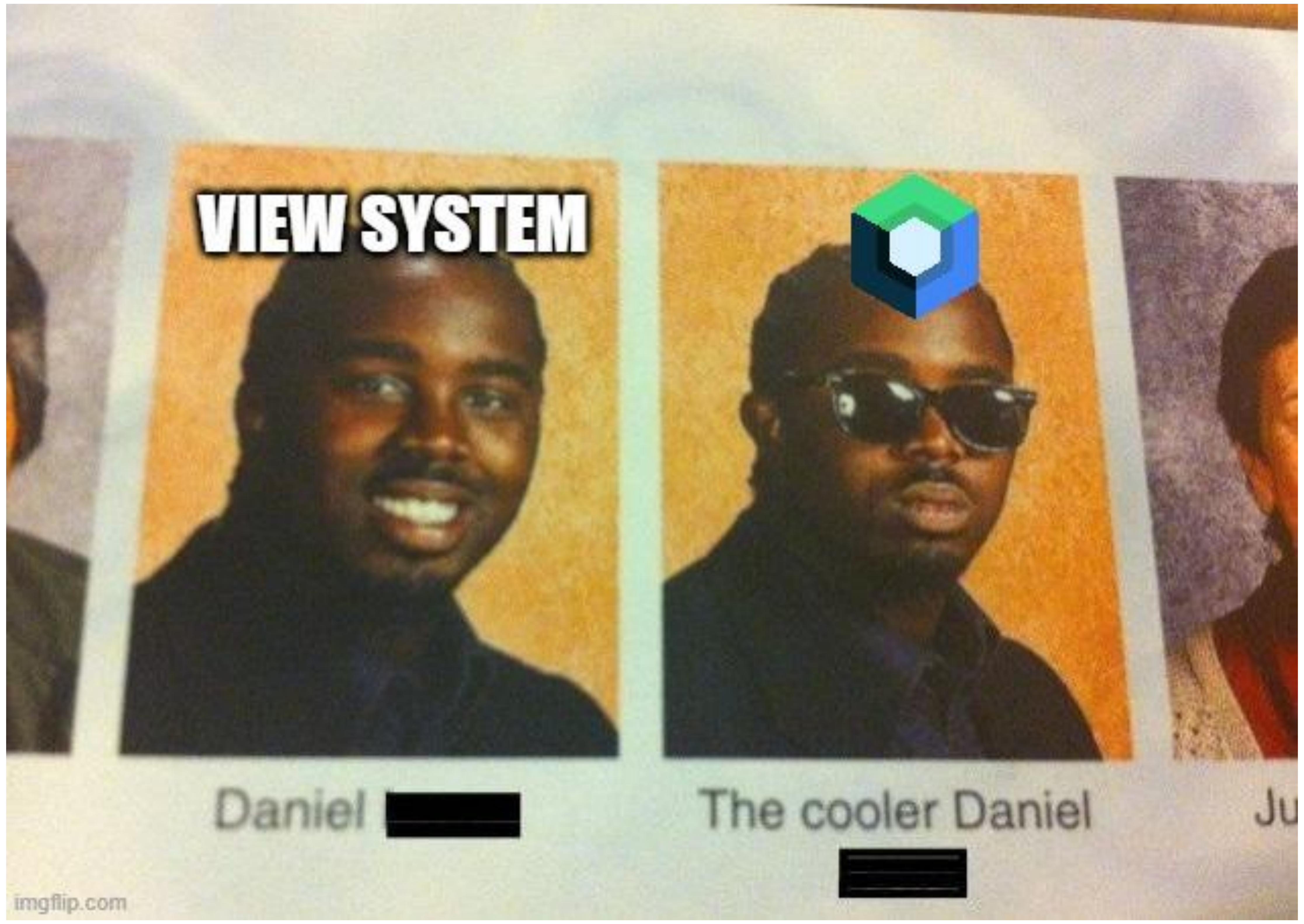
- Compose enables you to create beautiful apps with direct access to the Android platform APIs and built-in support for Material Design, Dark theme, animations, and more.

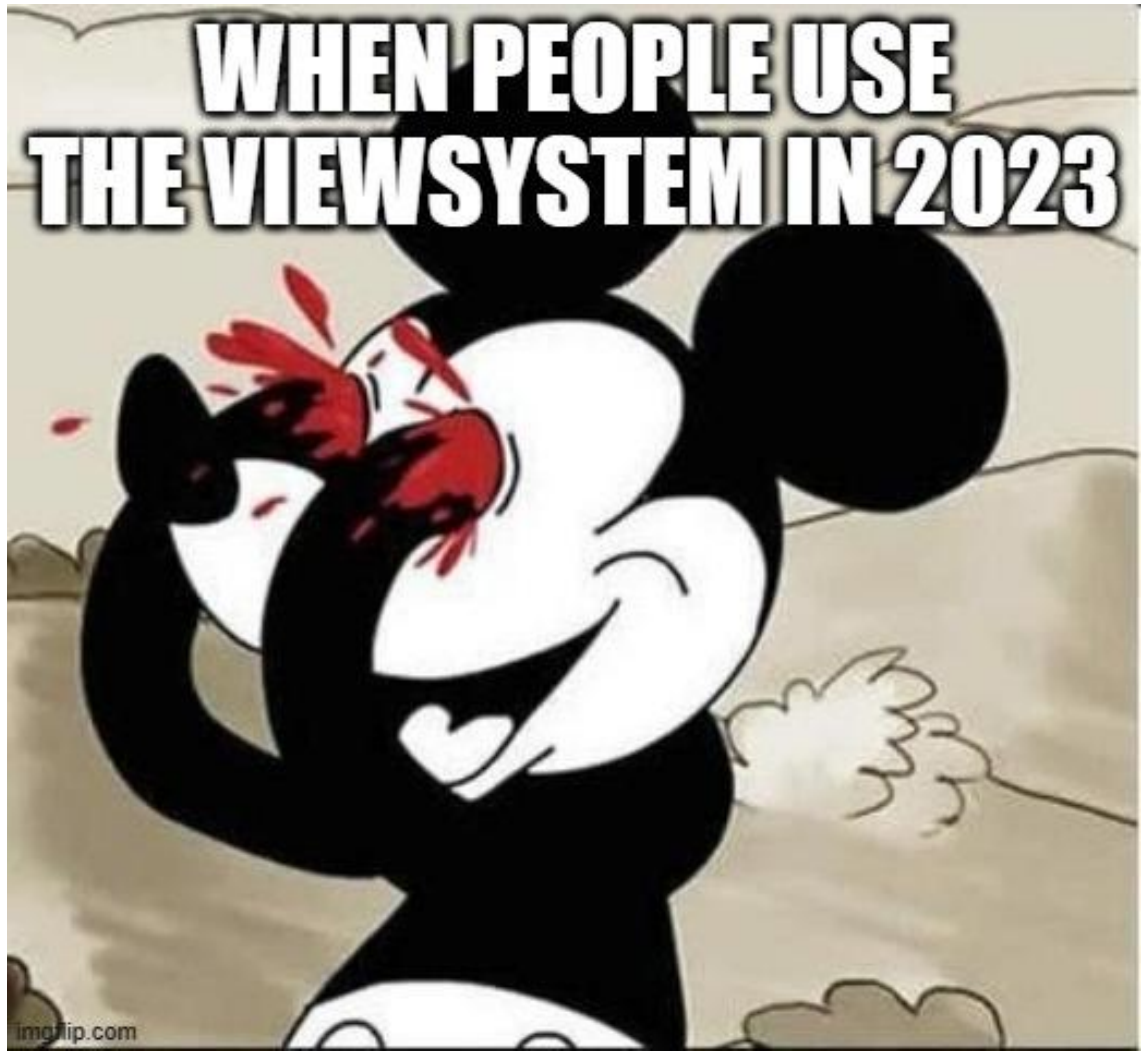
- With Compose, bringing movement and life to your apps through animations is quick and easy to implement.

- Whether you're building with Material Design or your own design system, Compose gives you the flexibility to implement the design you want!

Check out the in-depth case studies to learn more about how Twitter, Square, Monzo and Cuvva are using Compose.

In summary





Developer Student Clubs

Google Developers

Intro to Android Studio

An overview of the most popular Android IDE

Developer Student Clubs

Android Studio IDE

Android Studio is the official Integrated Development Environment (IDE) for Android app development. Based on the powerful code editor and developer tools from IntelliJ IDEA, Android Studio offers even more features that enhance your productivity when building Android apps, such as:

- A flexible Gradle-based build system
- A fast and feature-rich emulator
- A unified environment where you can develop for all Android devices
- Apply Changes to push code and resource changes to your running app without restarting your app
- Code templates and GitHub integration to help you build common app features and import sample code
- Extensive testing tools and frameworks
- Lint tools to catch performance, usability, version compatibility, and other problems
- C++ and NDK support
- Built-in support for Google Cloud Platform, making it easy to integrate Google Cloud Messaging and App Engine

The RecyclerView case-study

A clear win for Jetpack Compose

Developer Student Clubs

Coding a note-taking app live!

Live workshop on mobile app development in Android using Kotlin & the MVVM architecture

Developer Student Clubs

Android Room

A persistence library built on SQLite

Apps that handle non-trivial amounts of structured data can benefit greatly from persisting that data locally. The most common use case is to cache relevant pieces of data so that when the device cannot access the network, the user can still browse that content while they are offline.

The Room persistence library provides an **abstraction layer over SQLite** to allow fluent database access while harnessing the full power of SQLite. In particular, Room provides the following benefits:

- Compile-time verification of SQL queries.
- Convenience annotations that minimize repetitive and error-prone boilerplate code.
- Streamlined database migration paths.

Activities

A crucial component of every Android app

The Activity class is a crucial component of an Android app, and the way activities are launched and put together is a fundamental part of the platform's application model.

Unlike programming paradigms in which apps are launched with a main() method, the Android system initiates code in an Activity instance by invoking specific callback methods that correspond to specific stages of its lifecycle.

The Concept

The mobile-app experience differs from its desktop counterpart in that **a user's interaction with the app doesn't always begin in the same place**. Instead, the user journey often begins non-deterministically. For instance, if you open an email app from your home screen, you might see a list of emails. By contrast, if you are using a social media app that then launches your email app, you might go directly to the email app's screen for composing an email.

The **Activity** class is designed to facilitate this paradigm. When one app invokes another, the calling app invokes an activity in the other app, rather than the app as an atomic whole. In this way, **the activity serves as the entry point for an app's interaction with the user**. You implement an activity as a subclass of the Activity class.

Cont.

An activity provides the window in which the app draws its UI. This window typically fills the screen, but may be smaller than the screen and floats on top of other windows. Generally, one activity implements one screen in an app. For instance, one of an app's activities may implement a *Preferences* screen, while another activity implements a *Select Photo* screen.

Most apps contain multiple screens, which means they comprise multiple activities. Typically, one activity in an app is specified as the **main activity**, which **is the first screen to appear when the user launches the app**. Each activity can then start another activity in order to perform different actions. For example, the main activity in a simple e-mail app may provide the screen that shows an e-mail inbox. From there, the main activity might launch other activities that provide screens for tasks like writing e-mails and opening individual e-mails.

Activities in M.A.D.

Although activities work together to form a cohesive user experience in an app, each activity is only loosely bound to the other activities; there are usually minimal dependencies among the activities in an app. In fact, activities often start up activities belonging to other apps. For example, a browser app might launch the Share activity of a social-media app.

Modern Android Development (M.A.D.), has shifted from using multiple activities to a single activity paradigm. Nowadays, you're more likely to encounter a single activity that consists of multiple fragments or, in the case of compose, multiple composable destinations!

To use activities in your app, you must register information about them in the app's manifest, and you must manage **activity lifecycles** appropriately. We will see more about the importance of Activity lifecycles in a bit!

Activity Lifecycles

As a user navigates through, out of, and back to your app, the Activity instances in your app transition through different states in their lifecycle. **The Activity class provides several callbacks that let the activity know when a state changes** or that the system is creating, stopping, or resuming an activity or destroying the process the activity resides in.

Within the lifecycle callback methods, you can declare how your activity behaves when the user leaves and re-enters the activity. For example, if you're building a streaming video player, you might pause the video and terminate the network connection when the user switches to another app. When the user returns, you can reconnect to the network and let the user resume the video from the same spot.

Activity Lifecycle Callbacks

Each callback lets you perform specific work that's appropriate to a given change of state. Doing the right work at the right time and handling transitions properly make your app more robust and performant. For example, good implementation of the lifecycle callbacks can help your app avoid the following:

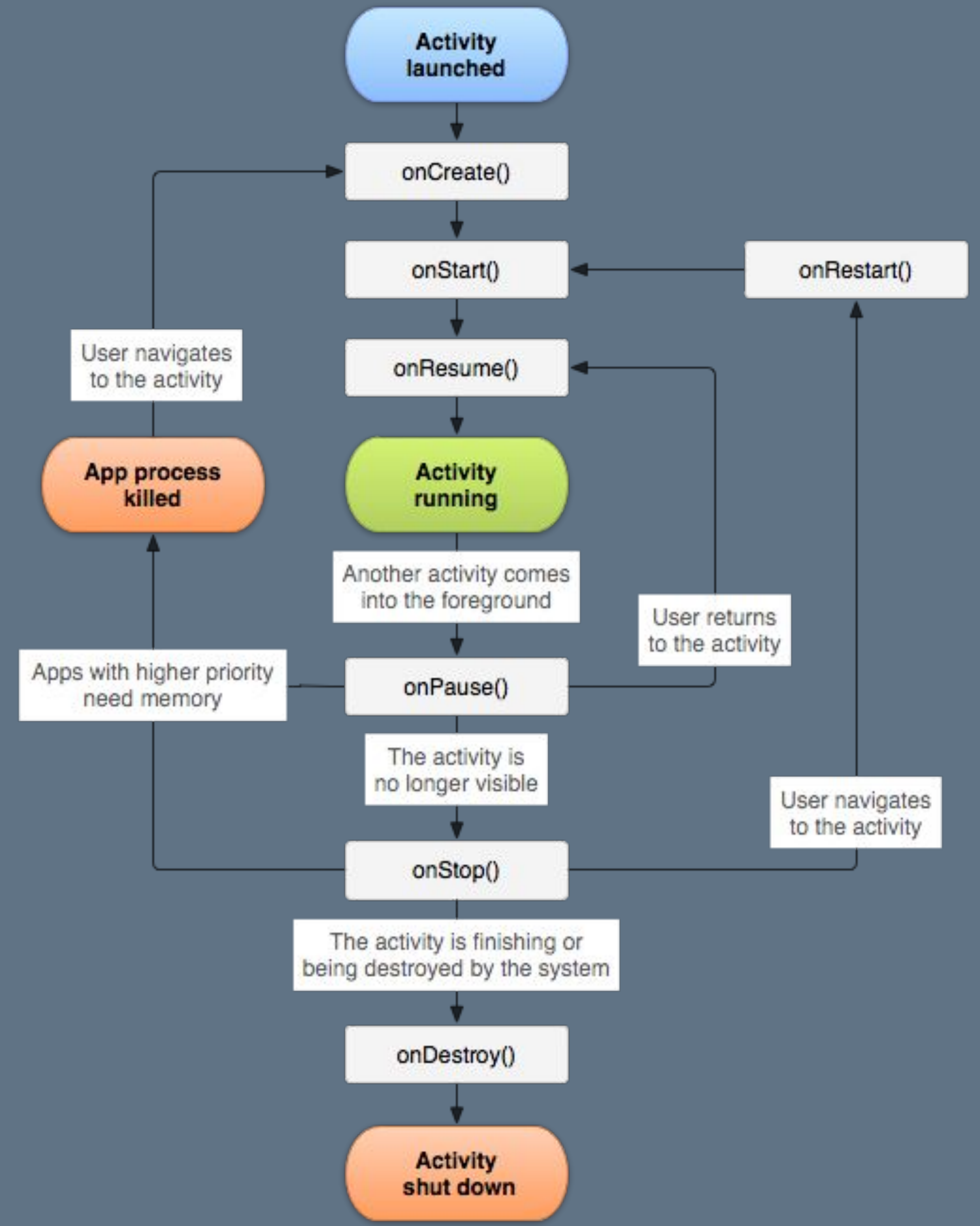
- Crashing if the user receives a phone call or switches to another app while using your app.
- Consuming valuable system resources when the user is not actively using it.
- Losing the user's progress if they leave your app and return to it at a later time.
- Crashing or losing the user's progress when the screen rotates between landscape and portrait orientation (more on this in a bit).

Activity lifecycle concepts

To navigate transitions between stages of the activity lifecycle, the Activity class provides a core set of six callbacks: onCreate(), onStart(), onResume(), onPause(), onStop(), and onDestroy().

The system invokes each of these callbacks as the activity enters a new state.

Developer Student Clubs


Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Cont.

As the user begins to leave the activity, the system calls methods to dismantle the activity. In some cases, the activity is only partially dismantled and still resides in memory, such as when the user switches to another app. In these cases, the activity can still come back to the foreground.

If the user returns to the activity, it resumes from where the user left off. With a few exceptions, apps are restricted from starting activities when running in the background.
The system's likelihood of killing a given process, along with the activities in it, depends on the state of the activity at the time. For more information on the relationship between state and vulnerability to ejection, see the section about activity state and ejection from memory.

Depending on the complexity of your activity, you probably don't need to implement all the lifecycle methods. However, **it's important that you understand each one and implement those that make your app behave the way users expect**.

The ViewModel class

A business logic state holder

The ViewModel class is a business logic or screen level state holder. It exposes state to the UI and encapsulates related business logic. **Its principal advantage is that it caches state and persists it through configuration changes.** This means that your UI doesn't have to fetch data again when navigating between activities, or following configuration changes, such as when rotating the screen. We will explore a simple ViewModel case in our sample project, but you can also see viewmodel benefits for more information!

Developer Student Clubs

Thank You!

Stelios Papamichail
Android Egineer @Threenitas
@MikePapamichail