BAKI ALİ NEFT MƏKTƏBİ
BAKU HIGHER OIL SCHOOL

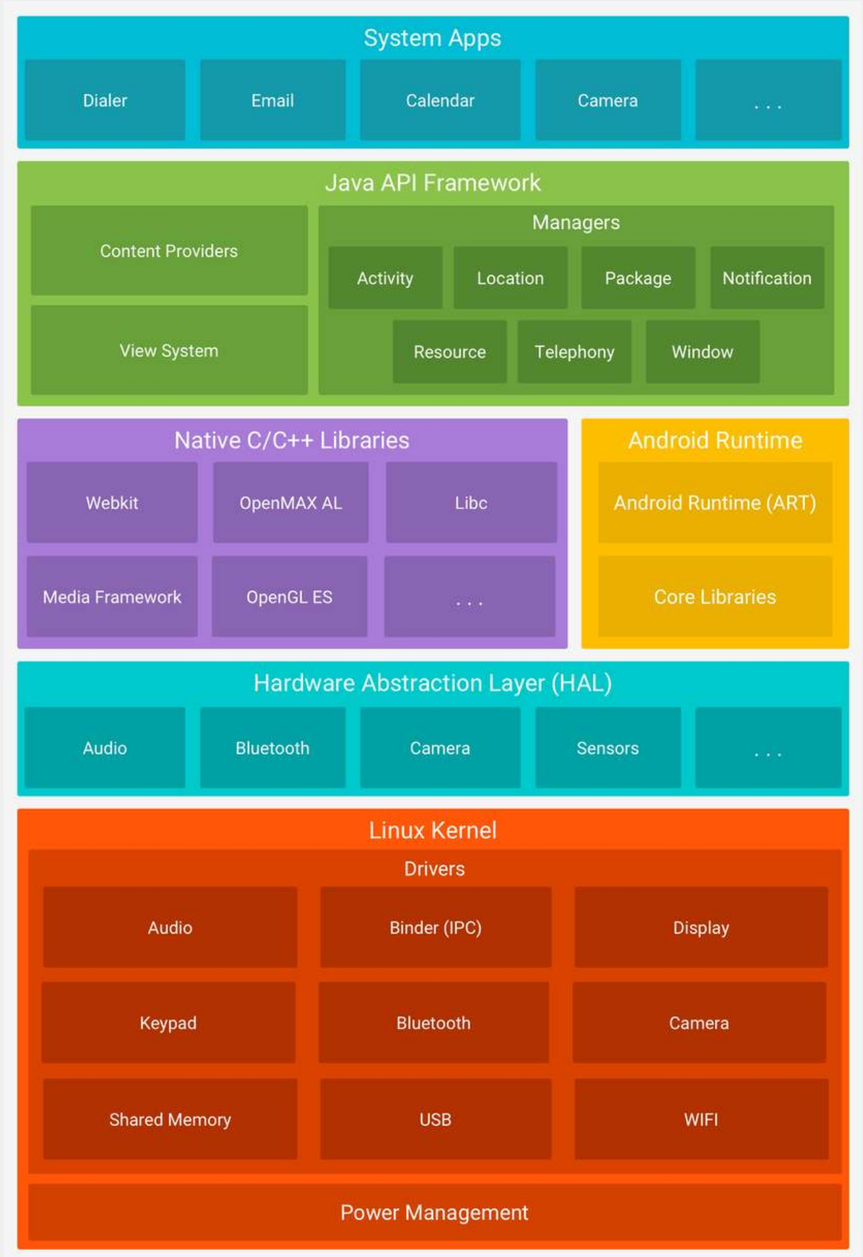
# Mobile development and security

Session 2

**Karim Karimov**
Lecturer

# Android

Android is a mobile operating system based on a modified version of the Linux kernel and other open-source software, designed primarily for touchscreen mobile devices such as smartphones and tablets. Android is developed by a consortium of developers known as the Open Handset Alliance, though its most widely used version is primarily developed by Google. It was unveiled in November 2007. Over 70 percent of smartphones based on **Android Open Source Project** run Google's ecosystem (which is known as simply Android), some with vendor-customized user interfaces and software suites, such as TouchWiz and later One UI by Samsung and HTC Sense. Competing ecosystems and forks of AOSP include Fire OS (developed by Amazon), ColorOS by OPPO, OriginOS by Vivo, MagicUI by Honor, or custom ROMs such as LineageOS




# The Android software stack

# The Linux Kernel

The foundation of the Android platform is the Linux kernel. For example, the Android Runtime (ART) relies on the Linux kernel for underlying functionalities such as threading and low-level memory management.
Using a Linux kernel allows Android to take advantage of key security features and allows device manufacturers to develop hardware drivers for a well-known kernel.

# Hardware Abstraction Layer (HAL)

The hardware abstraction layer (HAL) provides standard interfaces that expose device hardware capabilities to the higher-level Java API framework. The HAL consists of multiple library modules, each of which implements an interface for a specific type of hardware component, such as the camera or bluetooth module. When a framework API makes a call to access device hardware, the Android system loads the library module for that hardware component.

# Android Runtime

For devices running Android version 5.0 (API level 21) or higher, each app runs in its own process and with its own instance of the **Android Runtime** (**ART**). ART is written to run multiple virtual machines on low-memory devices by executing DEX files, a bytecode format designed specially for Android that's optimized for minimal memory footprint. Build tools, such as d8, compile Java sources into DEX bytecode, which can run on the Android platform.

Prior to Android version 5.0 (API level 21), **Dalvik** was the Android runtime. If your app runs well on ART, then it should work on Dalvik as well, but the reverse may not be true.

# Native C/C++ Libraries

Many core Android system components and services, such as ART and HAL, are built from native code that require native libraries written in C and C++. The Android platform provides Java framework APIs to expose the functionality of some of these native libraries to apps. For example, you can access **OpenGL ES** through the Android framework's **Java OpenGL API** to add support for drawing and manipulating 2D and 3D graphics in your app.
If you are developing an app that requires C or C++ code, you can use the **Android NDK** to access some of these native platform libraries directly from your native code.

# Java API Framework

The entire feature-set of the Android OS is available to you through APIs written in the Java language. These APIs form the building blocks you need to create Android apps by simplifying the reuse of core, modular system components and services:

- A rich and extensible **View System** you can use to build an app's UI, including lists, grids, text boxes, buttons, and even an embeddable web browser
- A **Resource Manager**, providing access to non-code resources such as localized strings, graphics, and layout files
- A **Notification Manager** that enables all apps to display custom alerts in the status bar
- An **Activity Manager** that manages the lifecycle of apps and provides a common navigation back stack
- **Content Providers** that enable apps to access data from other apps, such as the Contacts app, or to share their own data

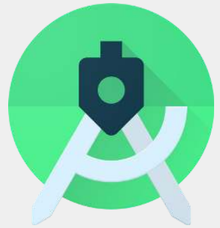
# System Apps

Android comes with a set of core apps for email, SMS messaging, calendars, internet browsing, contacts, and more. Apps included with the platform have no special status among the apps the user chooses to install. So a third-party app can become the user's default web browser, SMS messenger, or even the default keyboard (some exceptions apply, such as the system's Settings app).

The system apps function both as apps for users and to provide key capabilities that developers can access from their own app. For example, if your app would like to deliver an SMS message, you don't need to build that functionality yourself—you can instead invoke whichever SMS app is already installed to deliver a message to the recipient you specify.

# Android studio

To create Android apps, you first need to install Android Studio. Android Studio is a customized IDE based on IntelliJ, an IDE by JetBrains, that provides a powerful set of tools.
You can download Android Studio IDE from the official website https://developer.android.com/studio



# Gradle

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**, an advanced build toolkit, 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.

# Gradle

Gradle and the Android Gradle plugin run **independent** of Android Studio. This means that you can build your Android apps from within Android Studio, the command line on your machine, or on machines where Android Studio is not installed, such as continuous integration servers.

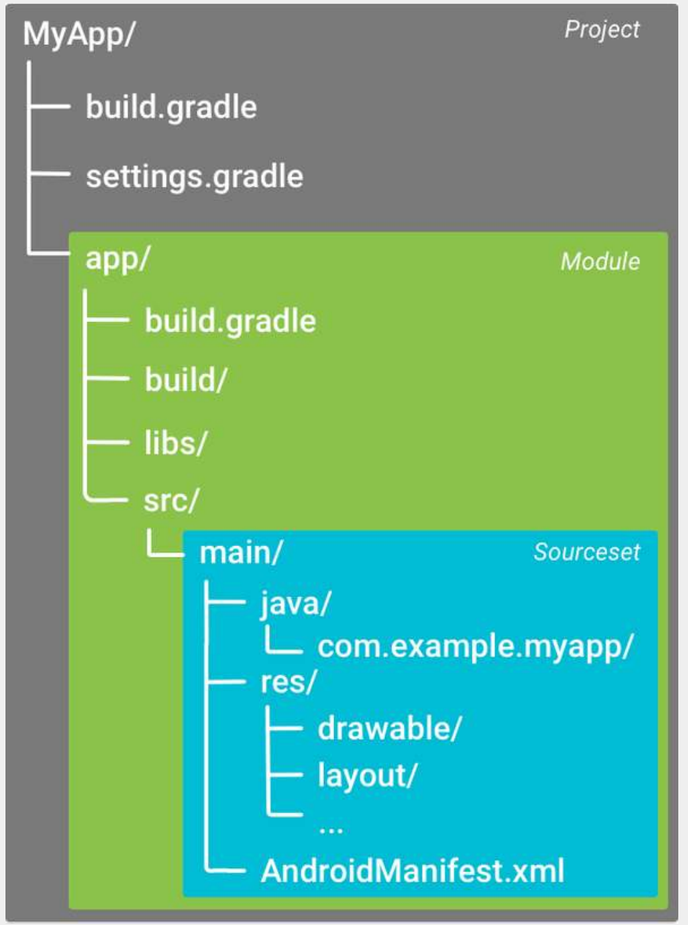
If you aren't using Android Studio, you can learn how to build and run your app from the command line. The output of the build is the same whether you are building a project from the command line, on a remote machine, or using Android Studio.

# Build processes

The build process involves many tools and processes that convert your project into an **Android Application Package (APK)** or **Android App Bundle (AAB)**.
The Android Gradle plugin does much of the build process for you, but it can be useful to understand certain aspects of the build process so you can adjust the build to meet your requirements.
Different projects may have different build goals. For example, the build for a third-party library produces AAR or JAR libraries. However, an app is the most common type of project, and the build for an app project produces a debug or release **APK** or **AAB** of your app that you can deploy, test, or release to external users.

# Build configuration files

Creating custom build configurations requires you to make changes to one or more build configuration files or **build.gradle** files. These plain-text files use **Domain Specific Language (DSL)** to describe and manipulate the build logic using either Groovy, which is a dynamic language for the Java Virtual Machine (JVM), or Kotlin script, which is a flavor of the Kotlin language. You don't need to know Groovy or Kotlin script to start configuring your build, because the Android Gradle plugin introduces most of the DSL elements you need.



# App manifest

Every app project must have an **AndroidManifest.xml** file (with precisely that name) at the root of the project source set. The manifest file describes essential information about your app to the Android build tools, the Android operating system, and Google Play.
Among many other things, the manifest file is required to declare the following:

- The components of the app, which include all **activities**, **services**, **broadcast receivers**, and **content providers**. Each component must define basic properties such as the name of its Kotlin or Java class. It can also declare capabilities such as which device configurations it can handle, and intent filters that describe how the component can be started.
- (continued)

# App manifest

- The **permissions** that the app needs in order to access protected parts of the system or other apps. It also declares any permissions that other apps must have if they want to access content from this app. This information does not limit the user to install application.
- The **hardware and software features** the app requires, which affects which devices can install the app from Google Play. Example: Camera, NFC features can be required to allow users for installation.

# Manifest components

For each app component that you create in your app, you must declare a corresponding XML element in the manifest file:

- **<activity>** for each subclass of Activity.
- **<service>** for each subclass of Service.
- **<receiver>** for each subclass of BroadcastReceiver.
- **<provider>** for each subclass of ContentProvider.

If you subclass any of these components without declaring it in the manifest file, the system cannot start it.

# Manifest components

```
<manifest ... >
  <application ... >
    <activity android:name="com.example.myapp.MainActivity" ... >
    </activity>
  </application>
</manifest>
```

# Manifest permissions

Android apps must request permission to access sensitive user data (such as contacts and SMS) or certain system features (such as the camera and internet access). Each permission is identified by a unique label. For example, an app that needs to send SMS messages must have the following line in the manifest:

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

# Manifest permissions

Beginning with Android 6.0 (API level 23), the user **can approve or reject some app permisions at runtime**. But no matter which Android version your app supports, you must declare all permission requests with a ***<uses-permission>*** element in the manifest. If the permission is granted, the app is able to use the protected features. If not, its attempts to access those features fail.
Your app can also protect its own components with permissions. It can use any of the permissions that are defined by Android, as listed in ***android.Manifest.permission***, or a permission that's declared in another app.
Your app can also define its own permissions. A new permission is declared with the **<permission>** element.

# Manifest compatibility

The manifest file is also where you can declare what types of hardware or software features your app requires, and thus, which types of devices your app is compatible with. Google Play Store does not allow your app to be installed on devices that don't provide the features or system version that your app requires.
There are several manifest tags that define which devices your app is compatible with. The following are just a couple of the most common tags.

- **<uses-feature>**
- **<uses-sdk>**

# <uses-feature>

The **<uses-feature>** element allows you to declare hardware and software features your app needs. For example, if your app cannot achieve basic functionality on a device without a compass sensor, you can declare the compass sensor as required with the following manifest tag.

```
<manifest ... >
    <uses-feature
android:name="android.hardware.sensor.compass"
                  android:required="true" />
    ...
</manifest>
```

# <uses-sdk>

Each successive platform version often adds new APIs not available in the previous version. To indicate the minimum version with which your app is compatible, your manifest must include the **<uses-sdk>** tag and its ***minSdkVersion*** attribute.
However, beware that attributes in the **<uses-sdk>** element are **overridden** by corresponding properties in the **build.gradle** file. So if you're using Android Studio, you must specify the ***minSdkVersion*** and ***targetSdkVersion*** values there instead.

```
android {
    defaultConfig {
        applicationId 'com.example.myapp'
        // Defines the minimum API level
required to run the app.
        minSdkVersion 21
        // Specifies the API level used to test
the app.
        targetSdkVersion 33
        ...
    }
}
```

# Activities

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.

# Activities

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.

# Activities

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 float 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

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.
To use activities in your app, you **must register** information about them in the app's manifest, and you must manage activity lifecycles appropriately.

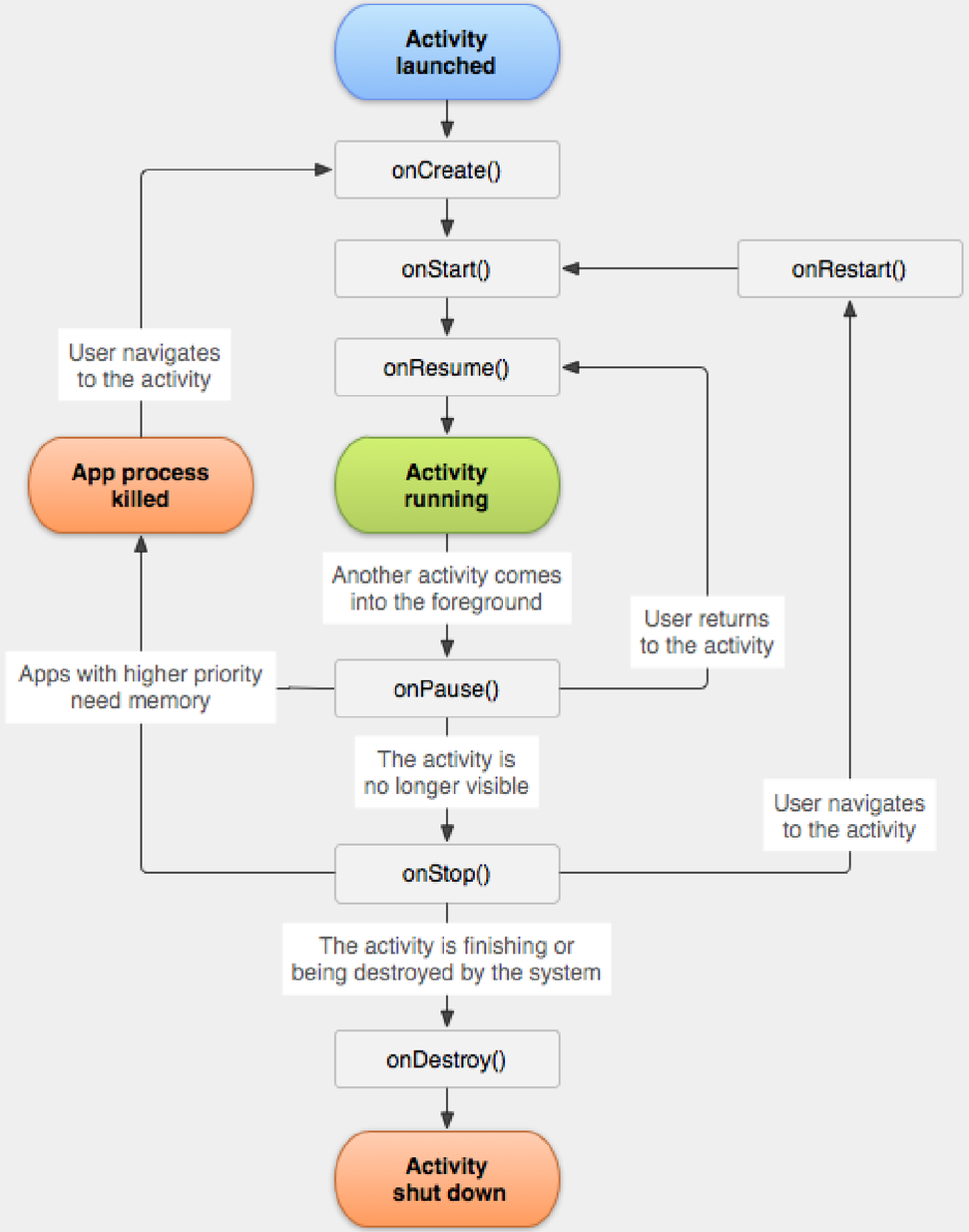
# Activity in manifest

For your app to be able to use activities, you must declare the activities, and certain of their attributes, in the manifest. The only required attribute for this element is **android:name**, which specifies the class name of the activity. You can also add attributes that define activity characteristics such as label, icon, or UI theme.

```
<manifest ... >
  <application ... >
      <activity android:name=".ExampleActivity" />
      ...
  </application ... >
  ...
</manifest >
```

# Activity lifecycle

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 an activity enters a new state.



# Activity lifecycle

**onCreate()**

You must implement this callback, which fires when the system creates your activity. Your implementation should initialize the essential components of your activity: For example, your app should create views and bind data to lists here. Most importantly, this is where you must call setContentView() to define the layout for the activity's user interface.

When onCreate() finishes, the next callback is always onStart().

# Activity lifecycle

**onStart()**
As onCreate() exits, the activity enters the Started state, and the activity becomes visible to the user. This callback contains what amounts to the activity's final preparations for coming to the foreground and becoming interactive.

**onResume()**
The system invokes this callback just before the activity starts interacting with the user. At this point, the activity is at the top of the activity stack, and captures all user input. Most of an app's core functionality is implemented in the onResume() method.
The onPause() callback always follows onResume().

# Activity lifecycle

**onPause()**

The system calls onPause() when the activity loses focus and enters a Paused state. This state occurs when, for example, the user taps the Back or Recents button. When the system calls onPause() for your activity, it technically means your activity is still partially visible, but most often is an indication that the user is leaving the activity, and the activity will soon enter the Stopped or Resumed state.

**onStop()**

The system calls onStop() when the activity is no longer visible to the user. This may happen because the activity is being destroyed, a new activity is starting, or an existing activity is entering a Resumed state and is covering the stopped activity. In all of these cases, the stopped activity is no longer visible at all.

# Activity lifecycle

**onRestart()**
The system invokes this callback when an activity in the Stopped state is about to restart. onRestart() restores the state of the activity from the time that it was stopped.
This callback is always followed by onStart().
**onDestroy()**
The system invokes this callback before an activity is destroyed.
This callback is the final one that the activity receives. onDestroy() is usually implemented to ensure that all of an activity's resources are released when the activity, or the process containing it, is destroyed.
This section provides only an introduction to this topic. For a more detailed treatment of the activity lifecycle and its callbacks, see The Activity Lifecycle.

# Intent

An intent is an abstract description of an operation to be performed. It can be used with ***startActivity*** to launch an **Activity**, ***broadcastIntent*** to send it to any interested **BroadcastReceiver** components, and **Context.startService(Intent)** or **Context.bindService(Intent, ServiceConnection, int)** to communicate with a background Service.
An Intent provides a facility for performing late runtime binding between the code in different applications. Its most significant use is in the launching of activities, where it can be thought of as the ***glue between activities***. It is basically a passive data structure holding an abstract description of an action to be performed.

# Intent structure

The primary pieces of information in an intent are:

- **action** - The general action to be performed, such as ***ACTION_VIEW***, ***ACTION_EDIT***, ***ACTION_MAIN***, etc.
- **data** - The data to operate on, such as a person record in the contacts database, expressed as a ***Uri***.

In addition to these primary attributes, there are a number of secondary attributes that you can also include with an intent:

- **category** - Gives additional information about the action to execute.
- **type** - Specifies an explicit type (a MIME type) of the intent data.
- **component** - Specifies an explicit name of a component class to use for the intent.
- **extras** - This is a Bundle of any additional information.

# Intent structure

Some examples of action/data pairs are:

- **ACTION_VIEW** - **content://contacts/people/1** - Display information about the person whose identifier is "1".
- **ACTION_DIAL** - **content://contacts/people/1** - Display the phone dialer with the person filled in.
- **ACTION_VIEW - tel:123** - Display the phone dialer with the given number filled in. Note how the VIEW action does what is considered the most reasonable thing for a particular URI.
- **ACTION_DIAL** - **tel:123** - Display the phone dialer with the given number filled in.
- **ACTION_EDIT** - **content://contacts/people/1** - Edit information about the person whose identifier is "1".

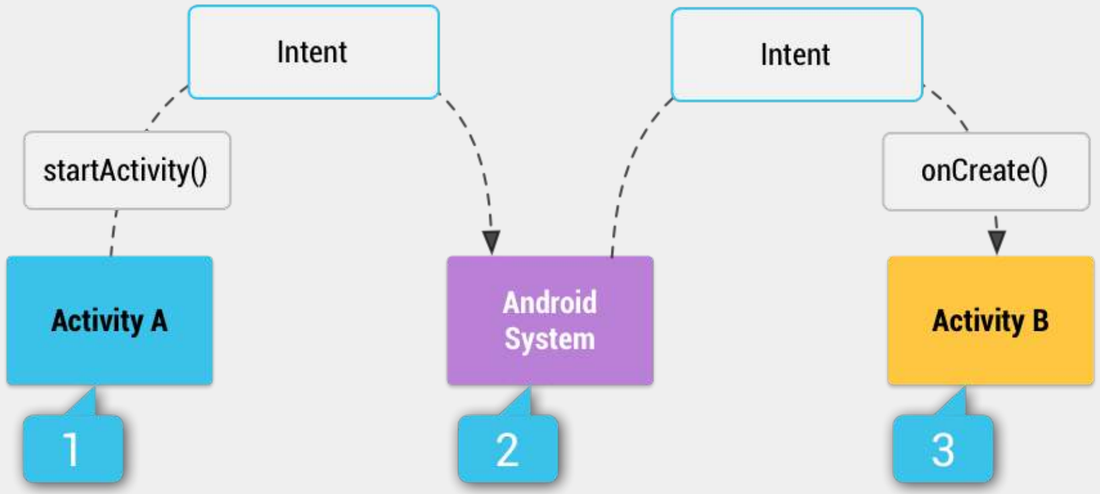
# Intent filters

Intent filters are a very powerful feature of the Android platform. They provide the ability to launch an activity based not only on an explicit request, but also an **implicit** one. For example, an explicit request might tell the system to "***Start the Send Email activity in the Gmail app***". By contrast, an implicit request tells the system to "***Start a Send Email screen in any activity that can do the job***". When the system UI asks a user which app to use in performing a task, that's an intent filter at work.

You can take advantage of this feature by declaring an **<intent-filter>** attribute in the **<activity>** element. The definition of this element includes an **<action>** element and, optionally, a **<category>** element and/or a **<data>** element. These elements combine to specify the type of intent to which your activity can respond.

# Intent filters

How an implicit intent is delivered through the system to start another activity:
[1] Activity A creates an Intent with an action description and passes it to startActivity().
[2] The Android System searches all apps for an intent filter that matches the intent.
[3] When a match is found, the system starts the matching activity (Activity B) by invoking its onCreate() method and passing it the Intent



# Intent filters

For example, the following code snippet shows how to configure an activity that sends text data, and receives requests from other activities to do so:

```
<activity android:name=".ExampleActivity"
android:icon="@drawable/app_icon">
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="text/plain" />
    </intent-filter>
</activity>
```

# Intent filters

In this example, the **<action>** element specifies that this activity sends data. Declaring the **<category>** element as ***DEFAULT*** enables the activity to receive launch requests. The **<data>** element specifies the type of data that this activity can send. The following code snippet shows how to call the activity described above:

```
val sendIntent = Intent().apply {
    action = Intent.ACTION_SEND
    type = "text/plain"
    putExtra(Intent.EXTRA_TEXT, textMessage)
}
startActivity(sendIntent)
```

# Intent filters

In this example, the **<action>** element specifies that this activity sends data. Declaring the **<category>** element as ***DEFAULT*** enables the activity to receive launch requests. The **<data>** element specifies the type of data that this activity can send. The following code snippet shows how to call the activity described above:

```
val sendIntent = Intent().apply {
    action = Intent.ACTION_SEND
    type = "text/plain"
    putExtra(Intent.EXTRA_TEXT, textMessage)
}
startActivity(sendIntent)
```

# References


- **Android** - https://en.wikipedia.org/wiki/Android_(operating_system)
- **Android platform** - https://developer.android.com/guide/platform
- **Gradle** - https://developer.android.com/studio/build
- **Manifest** - https://developer.android.com/guide/topics/manifest/manifest-intro
- **Permissions** - https://developer.android.com/guide/topics/permissions/overview
- **Activity** - https://developer.android.com/guide/components/activities/intro-activities
- **Activity lifecycle** - https://developer.android.com/guide/components/activities/activity-lifecycle
- **Intent** - https://developer.android.com/reference/android/content/Intent
- **Intent filters** - https://developer.android.com/guide/components/intents-filters

# End of the session