# CONTENTS

**………………………………………………………………………....**

[**CONTENTS**](#_32ydah9btpfe) **2**

[**CHAPTER 1: Introduction to Android and Android Studio**](#_dqj54w2d07rd) **5**

[1.1. The most popular OS](#_406996fup5nu) 5

[1.2. Features of Android](#_41957v4rm8b8) 5

[1.2.1. Interface](#_pkjg05qcnjbd) 5

[1.2.2. Applications](#_qjj98c9igrj6) 6

[1.2.3. Memory management](#_764vl81l7umo) 8

[1.3. Android app development tool: Android Studio](#_6kryq5a3ulbb) 8

[1.3.1. Installation](#_w5pueumxwdgp) 9

[1.3.2. Project Structure](#_nwzwqye1uy5j) 11

[1.3.3. The user interface](#_c3bwc0uhdin4) 12

[1.3.4 Gradle build system](#_3oj125pavv4j) 13

[1.3.5. Code Inspections](#_m052w06t34fr) 14

[1.4. Working in Android Studio:](#_zhm3vyluyme8) 14

[1.4.1. App components](#_ikdh8jrp9kyl) 15

[1.4.1.1. Activities](#_el2hxygyazlh) 15

[1.4.1.2. Services](#_yoo9o7lgnuwe) 17

[1.4.1.3. Broadcast receivers](#_1stmki8sydks) 18

[1.4.2. Activating components](#_jxlkugu0bjjt) 20

[1.4.3. Manifest files](#_20aoftvqpu9) 22

[1.4.4. App resources](#_z553dvfttspp) 22

[1.4.5. Permissions](#_btswjrvdbwa) 24

[1.4.5.1. Permission approval](#_occ11ysqnzlj) 24

[**Chapter 2: XML in Android Development**](#_6j5vst8mnwd7) **25**

[2.1. What is XML?](#_kc8gee45z8xp) 25

[2.2. Layouts using XML](#_h89kdzkui0r3) 26

[2.2.1. Write the XML](#_rsd44zfbk5q2) 27

[2.2.2. Loading the XML resources](#_1b8a3mly656p) 28

[2.2.3. Attributes](#_4ci6kq1lfx3l) 29

[ID](#_psxtwccnrwzw) 29

[Layout Parameters](#_q3s7ee9dlase) 30

[2.2.4. Layout Position](#_tlje3hh8fzrl) 32

[2.2.5. Size, Padding, and Margins](#_7whqjtbqkgzr) 32

[2.2.6. Common Layouts](#_uxgbcj4d4wmf) 33

[2.2.7. Building Layouts with an Adapter](#_j49y3bjxcbbn) 33

[Filling an adapter view with data](#_yuz9862tdtqe) 34

[ArrayAdapter](#_cwnjjmotclf6) 34

[2.3. Styles and Themes](#_bksjnzyr6joz) 35

[2.3.1. Create and apply a style](#_kpmkqp3uj3oj) 36

[2.3.2. Extend and customize a style](#_ta0bc5des0e) 37

[2.3.3. Apply a style as a theme](#_s1y7v1xncd1o) 38

[Now every view in the app or activity applies the styles defined in the given theme. If a view supports only some of the attributes declared in the style, then it applies only those attributes and ignores the ones it does not support.  
  
Beginning with Android 5.0 (API level 21) and Android Support Library v22.1, you can also specify the android:theme attribute to a view in your layout file. This modifies the theme for that view and any child views, which is useful for altering theme color palettes in a specific portion of your interface.  
  
The previous examples show how to apply a theme such as Theme.AppCompat that's supplied by the Android Support Library. But you'll usually want to customize the theme to fit your app's brand. The best way to do so is to extend these styles from the support library and override some of the attributes, as described in the next section.  
  
2.3.4. Style hierarchy](#_nuv07spi2a8g) 40

[2.3.5. Customize the default theme](#_a2kipzehcrbw) 41

[**CHAPTER 3: Kotlin in Android development**](#_vgznzko08ugu) **44**

[3.1. What is Kotlin?](#_nyr4ezd0rczg) 44

[3.2 Why Kotlin over Java for android?](#_thi1y8tsrt30) 44

[3.3. Basics of Kotlin](#_f0v8em1ou30e) 45

[3.3.1. Variables](#_8qr3yf4onrli) 45

[Read-only local variables are defined using the keyword val. They can be assigned a value only once.](#_l9gyqj3pv4q0) 45

[val a: Int = 1 // immediate assignment](#_64q2k919lhgl) 45

[3.3.2. Functions](#_uj50mobtfl2z) 46

[fun sum(a: Int, b: Int): Int {](#_tm34qhagy3gk) 46

[return a + b](#_ftom1bvrtauz) 46

[}](#_f74jsqqvx82i) 46

[fun sum(a: Int, b: Int) = a + b](#_erhh98hbbj7x) 46

[3.3.3. Using Conditional Statements](#_ph2p5f5aaml) 46

[fun maxOf(a: Int, b: Int): Int {](#_nsu3qq610w7c) 47

[if (a > b) {](#_te7pzhdz06w3) 47

[return a](#_jnka0d93298) 47

[} else {](#_zb4zn94j7qhq) 47

[return b](#_j2bomk9w08d0) 47

[}](#_ktyd1lp9ojb) 47

[}](#_hqoiwph7bpam) 47

[3.3.4. Loops](#_76adig495ryx) 47

[3.3.4.1 For Loop](#_eomggh37gjqn) 47

[3.3.4.2 While Loop](#_drhsu0tn2wem) 47

[3.3.5. Type checks and automatic casts](#_w7ag8f83h39q) 48

[3.3.6. ‘when’ expression in Kotlin](#_reapycr8ko4h) 49

[This works as switch statement in C/C++. For e.g:](#_binmj0vh4e7u) 49

[fun describe(obj: Any): String =](#_kpyyu2g9y6q1) 49

[3.3.7. Classes in Kotlin](#_drmfrw52uywj) 49

[3.3.7.1. Constructors in class](#_rkghcljx1884) 50

[3.3.7.2. Initialization block](#_len2dfxrdqev) 50

[3.3.7.3. Creating Instance of Class](#_o6w3ek9mzox7) 50

[3.3.7.4. Inheritance in Kotlin](#_bffj26gehwcz) 51

[3.3.7.5. Overriding methods](#_bf01yrg76qeb) 51

[3.3.7.6. Nested classes](#_u80t9hrrwug6) 52

[3.3.8. Arrays in Kotlin](#_8kqacrjbyv2w) 53

[3.3.9. Packages and Import in Kotlin](#_uflbo78lc5ut) 53

[3.3.10. ArrayLists in Kotlin](#_7vt4c4ivrunw) 54

[3.3.10.1 add( )](#_csa70x1pu9wp) 54

[3.3.10.2 addAll( )](#_x3fjxi3q5p8s) 55

[3.3.10.3 clear( )](#_6xgkon6dovjd) 55

[3.3.10.4 get( index:Int)](#_ohecv65l5dc) 55

[3.3.10.5 indexOf(Element e)](#_yvmsxj34d3) 55

[3.3.10.6 remove(Element e)](#_dvqzkbykgu5g) 55

[**Chapter 4: Echo Music Player App**](#_dhip8l62hd95) **56**

[4.1. Aim:](#_8e4xm7oqv94h) 56

[4.2. Functionalities and features](#_h627cxw0f29e) 57

[4.3. Functionalities in details](#_sdhmdjy9n24j) 58

[4.3.1. Splash screen](#_rhggnrtxww6m) 58

[4.3.2. Navigation Drawer](#_qez05j9qzfhq) 58

[4.3.3. All songs screen](#_k4t5uwcrko4t) 59

[4.3.4. Sort feature](#_uhhwnaygtysb) 60

[4.3.5. Favorites screen](#_lgddrpb0il2d) 61

[4.3.6. Settings screen](#_cwfmjsnihyzs) 62

[4.3.7. The Now playing screen](#_ypjy69uw07yv) 63

[4.3.8. Now playing bar](#_l2ihhp7noj3r) 66

[4.3.9. Notification](#_dql3crhl9s12) 66

# CHAPTER 1: Introduction to Android and Android Studio

## 1.1. The most popular OS

Android is an operating system designed for handheld PCs or mobiles developed by Google, based on a modified Linux kernel and other open source software. It’s primarily designed for touchscreen devices and Google has also developed Android TV for televisions, Android Auto for cars and Wear OS for smartwatches each with a specialized user interface. Variants of Android are also used on game consoles, digital cameras, PCs and other electronics.

Developed by Android Inc., which Google bought in 2005, Android was unveiled in 2007, with the first commercial Android device launched in September 2008. The operating system has since gone through multiple major releases, with the current version being 9 "Pie", released in August 2018. The core Android source code is known as Android Open Source Project (AOSP), and is primarily licensed under the Apache License.

Android has been the best-selling OS worldwide on smartphones since 2011 and on tablets since 2013. As of May 2017, it has over two billion monthly active users, the largest installed base of any operating system, and as of June 2018, the Google Play store features over 3.3 million apps.

## 1.2. Features of Android

### 1.2.1. Interface

Android's default user interface is mainly based on direct manipulation, using touch inputs that loosely correspond to real-world actions, like swiping, tapping, pinching, and reverse pinching to manipulate on-screen objects, along with a virtual keyboard.

The response to user input is designed to be immediate and provides a fluid touch interface, often using the vibration capabilities of the device to provide haptic feedback to the user. Internal hardware, such as accelerometers, gyroscopes and proximity sensors are used by some applications to respond to additional user actions, for example adjusting the screen from portrait to landscape depending on how the device is oriented, or allowing the user to steer a vehicle in a racing game by rotating the device, simulating control of a steering wheel.

Android home screens are typically made up of app icons and widgets, app icons launch the associated app, whereas widgets display live, auto-updating content, such as a weather forecast, the user's email inbox, or a news ticker directly on the home screen. A home-screen may be made up of several pages, between which the user can swipe back and forth. Third-party apps available on Google Play and other app stores can extensively re-theme the home screen, and even mimic the look of other operating systems, such as Windows Phone. Most manufacturers customize the look and features of their Android devices to differentiate themselves from their competitors.

Along the top of the screen is a status bar, showing information about the device and its connectivity. This status bar can be "pulled" down to reveal a notification screen where apps display important information or updates. Notifications are "short, timely, and relevant information about your app when it’s not in use", and when tapped, users are directed to a screen inside the app relating to the notification.

Beginning with Android 4.1 "Jelly Bean", "expandable notifications" allow the user to tap an icon on the notification in order for it to expand and display more information and possible app actions right from the notification.

### 1.2.2. Applications

Applications ("apps"), which extend the functionality of devices, are written using the Android software development kit (SDK) and, often, the Java programming language.Java may be combined with C/C++, together with a choice of non-default runtimes that allow better C++ support. The Go programming language is also supported, although with a limited set of application programming interfaces (API).In May 2017, Google announced support for Android app development in the Kotlin programming language.

The SDK includes a comprehensive set of development tools, including a debugger, software libraries, a handset emulator based on QEMU, documentation, sample code, and tutorials. Initially, Google's supported integrated development environment (IDE) was Eclipse using the Android Development Tools (ADT) plugin; in December 2014, Google released Android Studio, based on IntelliJ IDEA, as its primary IDE for Android application development. Other development tools are available, including a native development kit (NDK) for applications or extensions in C or C++, Google App Inventor, a visual environment for novice programmers, and various cross-platform mobile web applications frameworks. In January 2014, Google unveiled a framework based on Apache Cordova for porting Chrome HTML 5 web applications to Android, wrapped in a native application shell.

Android has a growing selection of third-party applications, which can be acquired by users by downloading and installing the application's APK (Android application package) file, or by downloading them using an application store program that allows users to install, update, and remove applications from their devices. Google Play Store is the primary application store installed on Android devices that comply with Google's compatibility requirements and license the Google Mobile Services software. Google Play Store allows users to browse, download and update applications published by Google and third-party developers; as of July 2013, there are more than one million applications available for Android in Play Store. As of July 2013, 50 billion applications have been installed. Some carriers offer direct carrier billing for Google Play application purchases, where the cost of the application is added to the user's monthly bill. As of May 2017, there are over one billion active users a month for Gmail, Android, Chrome, Google Play, and Maps.

Due to the open nature of Android, a number of third-party application marketplaces also exist for Android, either to provide a substitute for devices that are not allowed to ship with Google Play Store, provide applications that cannot be offered on Google Play Store due to policy violations, or for other reasons. Examples of these third-party stores have included the Amazon Appstore, GetJar, and SlideMe. F-Droid, another alternative marketplace, seeks to only provide applications that are distributed under free and open source licenses.

### 1.2.3. Memory management

Since Android devices are usually battery-powered, Android is designed to manage processes to keep power consumption at a minimum. When an application is not in use the system suspends its operation so that, while available for immediate use rather than closed, it does not use battery power or CPU resources. Android manages the applications stored in memory automatically: when memory is low, the system will begin invisibly and automatically closing inactive processes, starting with those that have been inactive for the longest amount of time.

## 1.3. Android app development tool: Android Studio

Android Studio is the official Integrated Development Environment (IDE) for Android app development, based on IntelliJ IDEA . On top of IntelliJ's powerful code editor and developer tools, 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
* Instant Run to push changes to your running app without building a new APK
* 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

### 1.3.1. Installation

First, be sure you download the latest version of Android Studio.

**Windows:**

To install Android Studio on Windows, proceed as follows:

1. If you downloaded an .exe file (recommended), double-click to launch it.
2. If you downloaded a .zip file, unpack the ZIP, copy the android-studio folder into your Program Files folder, and then open the android-studio > bin folder and launch studio64.exe (for 64-bit machines) or studio.exe (for 32-bit machines).
3. Follow the setup wizard in Android Studio and install any SDK packages that it recommends.

**MAC:**

To install Android Studio on your Mac, proceed as follows:

1. Launch the Android Studio DMG file.
2. Drag and drop Android Studio into the Applications folder, then launch Android Studio.
3. Select whether you want to import previous Android Studio settings, then click OK.
4. The Android Studio Setup Wizard guides you through the rest of the setup, which includes downloading Android SDK components that are required for development.

**LINUX:**

To install Android Studio on Linux, proceed as follows:

1. Unpack the .zip file you downloaded to an appropriate location for your applications, such as within /usr/local/ for your user profile, or /opt/ for shared users.
2. To launch Android Studio, open a terminal, navigate to the android-studio/bin/ directory, and execute studio.sh.
3. Select whether you want to import previous Android Studio settings or not, then click OK.
4. The Android Studio Setup Wizard guides you through the rest of the setup, which includes downloading Android SDK components that are required for development.

### 1.3.2. Project Structure

Each project in Android Studio contains one or more modules with source code files and resource files. Types of modules include:

* Android app modules
* Library modules
* Google App Engine modules

By default, Android Studio displays your project files in the Android project view, as shown in figure 1. This view is organized by modules to provide quick access to your project's key source files.

**Figure 1.** The project files in Android view.

All the build files are visible at the top level under Gradle Scripts and each app module contains the following folders:

* **manifests**: Contains the AndroidManifest.xml file.
* **java**: Contains the Java source code files, including JUnit test code.
* **res**: Contains all non-code resources, such as XML layouts, UI strings, and bitmap images.

The Android project structure on disk differs from this flattened representation. To see the actual file structure of the project, select Project from the Project dropdown (in figure 1, it's showing as Android).

We can also customize the view of the project files to focus on specific aspects of our app development. For example, selecting the Problems view of your project displays links to the source files containing any recognized coding and syntax errors, such as a missing XML element closing tag in a layout file.

### 1.3.3. The user interface

The Android Studio main window is made up of several logical areas identified in figure 3.



**Figure 3.** The Android Studio main window.

1. The **toolbar** lets you carry out a wide range of actions, including running your app and launching Android tools.
2. The **navigation bar** helps you navigate through your project and open files for editing. It provides a more compact view of the structure visible in the **Project** window.
3. The **editor window** is where you create and modify code. Depending on the current file type, the editor can change. For example, when viewing a layout file, the editor displays the Layout Editor.
4. The **tool window bar** runs around the outside of the IDE window and contains the buttons that allow you to expand or collapse individual tool windows.
5. The **tool windows** give you access to specific tasks like project management, search, version control, and more. You can expand them and collapse them.
6. The **status bar** displays the status of your project and the IDE itself, as well as any warnings or messages.

### 1.3.4 Gradle build system

Android Studio uses Gradle as the foundation of the build system, with more Android-specific capabilities provided by the [Android plugin for Gradle](https://developer.android.com/studio/releases/gradle-plugin.html). This build system runs as an integrated tool from the Android Studio menu, and independently from the command line. You can use the features of the build system to do the following:

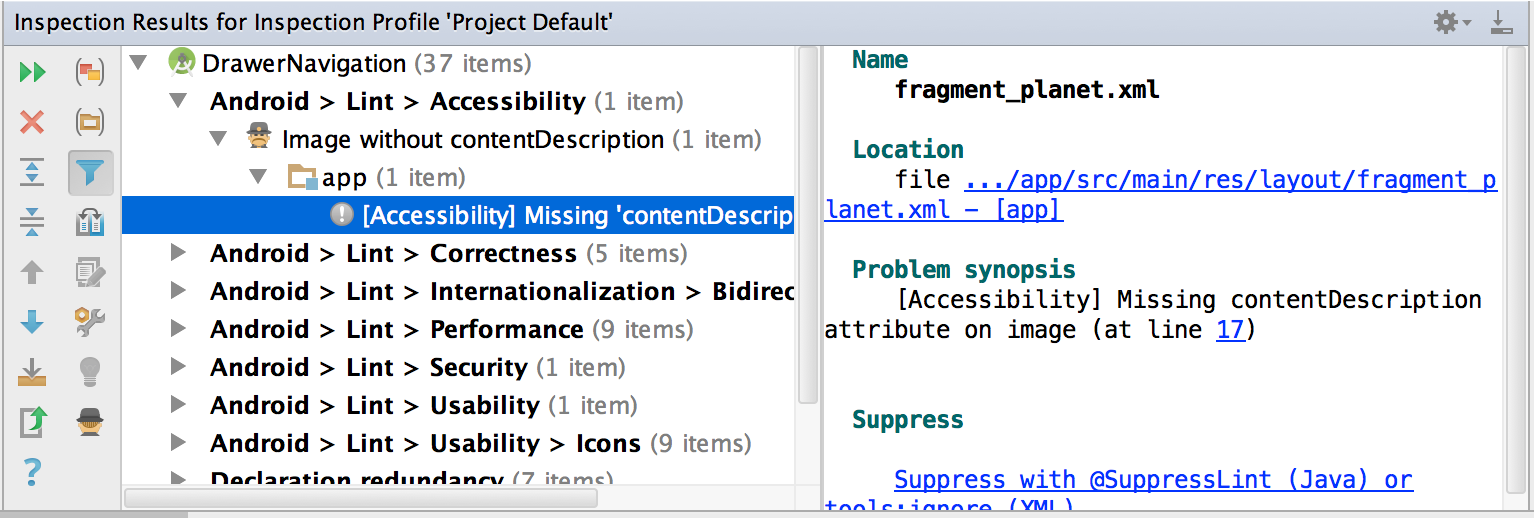
* Customize, configure, and extend the build process.
* Create multiple APKs for your app, with different features using the same project and modules.
* Reuse code and resources across sourcesets.

By employing the flexibility of Gradle, you can achieve all of this without modifying your app's core source files. Android Studio build files are named build.gradle. They are plain text files that use [Groovy](http://groovy-lang.org/) syntax to configure the build with elements provided by the Android plugin for Gradle. 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.

### 1.3.5. Code Inspections

Whenever you compile your program, Android Studio automatically runs configured [Lint](https://developer.android.com/studio/write/lint.html) and other [IDE inspections](https://www.jetbrains.com/help/idea/2018.1/code-inspection.html) to help you easily identify and correct problems with the structural quality of your code.

The Lint tool checks your Android project source files for potential bugs and optimization improvements for correctness, security, performance, usability, accessibility, and internationalization.



**Figure 7.** The results of a Lint inspection in Android Studio.

In addition to Lint checks, Android Studio also performs IntelliJ code inspections and validates annotations to streamline your coding workflow.

## 1.4. Working in Android Studio:

Android apps can be written using Kotlin, Java, and C++ languages. The Android SDK tools compile your code along with any data and resource files into an APK, an Android package, which is an archive file with an .apk suffix. One APK file contains all the contents of an Android app and is the file that Android-powered devices use to install the app.

### 

### 1.4.1. App components

App components are the essential building blocks of an Android app. Each component is an entry point through which the system or a user can enter your app. Some components depend on others.

There are four different types of app components:

* Activities
* Services
* Broadcast receivers
* Content providers

Each type serves a distinct purpose and has a distinct lifecycle that defines how the component is created and destroyed. The following sections describe the four types of app components.

#### 1.4.1.1. Activities

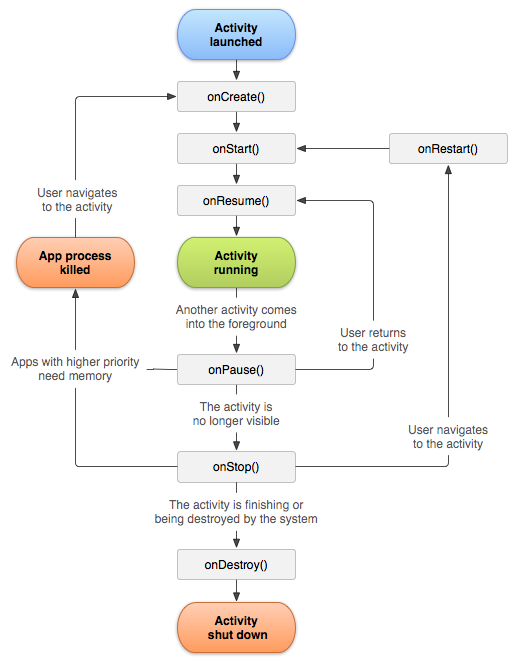
An *activity* is the entry point for interacting with the user. It represents a single screen with a user interface. For example, an email app might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails. Although the activities work together to form a cohesive user experience in the email app, each one is independent of the others. As such, a different app can start any one of these activities if the email app allows it. For example, a camera app can start the activity in the email app that composes new mail to allow the user to share a picture. An activity facilitates the following key interactions between system and app:

* Keeping track of what the user currently cares about (what is on screen) to ensure that the system keeps running the process that is hosting the activity.
* Knowing that previously used processes contain things the user may return to (stopped activities), and thus more highly prioritize keeping those processes around.
* Helping the app handle having its process killed so the user can return to activities with their previous state restored.
* Providing a way for apps to implement user flows between each other, and for the system to coordinate these flows. (The most classic example here being share.)

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.

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.



*Fig. Activity lifecycle*

#### 1.4.1.2. Services

A *service* is a general-purpose entry point for keeping an app running in the background for all kinds of reasons. It is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface. For example, a service might play music in the background while the user is in a different app, or it might fetch data over the network without blocking user interaction with an activity. Another component, such as an activity, can start the service and let it run or bind to it in order to interact with it. There are actually two very distinct semantics services tell the system about how to manage an app: Started services tell the system to keep them running until their work is completed. This could be to sync some data in the background or play music even after the user leaves the app. Syncing data in the background or playing music also represent two different types of started services that modify how the system handles them:

* Music playback is something the user is directly aware of, so the app tells the system this by saying it wants to be foreground with a notification to tell the user about it; in this case the system knows that it should try really hard to keep that service's process running, because the user will be unhappy if it goes away.
* A regular background service is not something the user is directly aware as running, so the system has more freedom in managing its process. It may allow it to be killed (and then restarting the service sometime later) if it needs RAM for things that are of more immediate concern to the user.

#### 1.4.1.3. Broadcast receivers

A broadcast receiver is a component that enables the system to deliver events to the app outside of a regular user flow, allowing the app to respond to system-wide broadcast announcements. Because broadcast receivers are another well-defined entry into the app, the system can deliver broadcasts even to apps that aren't currently running. So, for example, an app can schedule an alarm to post a notification to tell the user about an upcoming event... and by delivering that alarm to a BroadcastReceiver of the app, there is no need for the app to remain running until the alarm goes off. Many broadcasts originate from the system—for example, a broadcast announcing that the screen has turned off, the battery is low, or a picture was captured. Apps can also initiate broadcasts—for example, to let other apps know that some data has been downloaded to the device and is available for them to use. Although broadcast receivers don't display a user interface, they may create a status bar notification to alert the user when a broadcast event occurs.

A broadcast receiver is implemented as a subclass of BroadcastReceiver and each broadcast is delivered as an Intent object.

**Content providers**

A *content provider* manages a shared set of app data that you can store in the file system, in a SQLite database, on the web, or on any other persistent storage location that your app can access. Through the content provider, other apps can query or modify the data if the content provider allows it. For example, the Android system provides a content provider that manages the user's contact information.

To the system, a content provider is an entry point into an app for publishing named data items, identified by a URI scheme. Thus an app can decide how it wants to map the data it contains to a URI namespace, handing out those URIs to other entities which can in turn use them to access the data. There are a few particular things this allows the system to do in managing an app:

* Assigning a URI doesn't require that the app remain running, so URIs can persist after their owning apps have exited. The system only needs to make sure that an owning app is still running when it has to retrieve the app's data from the corresponding URI.
* These URIs also provide an important fine-grained security model. For example, an app can place the URI for an image it has on the clipboard, but leave its content provider locked up so that other apps cannot freely access it. When a second app attempts to access that URI on the clipboard,the system can allow that app to access the data via a temporary *URI permission grant* so that it is allowed to access the data only behind that URI, but nothing else in the second app.

Content providers are also useful for reading and writing data that is private to your app and not shared.

### 1.4.2. Activating components

Three of the four component types—activities, services, and broadcast receivers—are activated by an asynchronous message called an *intent*. Intents bind individual components to each other at runtime. You can think of them as the messengers that request an action from other components, whether the component belongs to your app or another.

An intent is created with an Intent object, which defines a message to activate either a specific component (explicit intent) or a specific *type* of component (implicit intent).

For activities and services, an intent defines the action to perform (for example, to *view* or *send* something) and may specify the URI of the data to act on, among other things that the component being started might need to know. For example, an intent might convey a request for an activity to show an image or to open a web page. In some cases, you can start an activity to receive a result, in which case the activity also returns the result in an Intent. For example, you can issue an intent to let the user pick a personal contact and have it returned to you. The return intent includes a URI pointing to the chosen contact.

For broadcast receivers, the intent simply defines the announcement being broadcast. For example, a broadcast to indicate the device battery is low includes only a known action string that indicates *battery is low*.

Unlike activities, services, and broadcast receivers, content providers are not activated by intents. Rather, they are activated when targeted by a request from a ContentResolver.

The content resolver handles all direct transactions with the content provider so that the component that's performing transactions with the provider doesn't need to and instead calls methods on the ContentResolver object.

**Starting an activity**

* An Activity represents a single screen in an app. You can start a new instance of an Activity by passing an Intent to startActivity(). The Intent describes the activity to start and carries any necessary data.
* If you want to receive a result from the activity when it finishes, call startActivityForResult(). Your activity receives the result as a separate Intent object in your activity's onActivityResult() callback. For more information, see the Activities guide.

**Starting a service**

* A Service is a component that performs operations in the background without a user interface. With Android 5.0 (API level 21) and later, you can start a service with JobScheduler. For more information about JobScheduler, see its API-reference documentation.
* For versions earlier than Android 5.0 (API level 21), you can start a service by using methods of the Service class. You can start a service to perform a one-time operation (such as downloading a file) by passing an Intent to startService(). The Intent describes the service to start and carries any necessary data.
* If the service is designed with a client-server interface, you can bind to the service from another component by passing an Intent to bindService(). For more information, see the Services guide.

**Delivering a broadcast**

* A broadcast is a message that any app can receive. The system delivers various broadcasts for system events, such as when the system boots up or the device starts charging. You can deliver a broadcast to other apps by passing an Intent to sendBroadcast() or sendOrderedBroadcast().

### 1.4.3. Manifest files

Before the Android system can start an app component, the system must know that the component exists by reading the app's *manifest file*, AndroidManifest.xml. Your app must declare all its components in this file, which must be at the root of the app project directory.

The manifest does a number of things in addition to declaring the app's components, such as the following:

* Identifies any user permissions the app requires, such as Internet access or read-access to the user's contacts.
* Declares the minimum API Level required by the app, based on which APIs the app uses.
* Declares hardware and software features used or required by the app, such as a camera, bluetooth services, or a multitouch screen.
* Declares API libraries the app needs to be linked against (other than the Android framework APIs), such as the Google Maps library.

You must declare all app components using the following elements:

* <activity> elements for activities.
* <service> elements for services.
* <receiver> elements for broadcast receivers.
* <provider> elements for content providers.

### 1.4.4. App resources

An Android app is composed of more than just code—it requires resources that are separate from the source code, such as images, audio files, and anything relating to the visual presentation of the app. For example, you can define animations, menus, styles, colors, and the layout of activity user interfaces with XML files. Using app resources makes it easy to update various characteristics of your app without modifying code. Providing sets of alternative resources enables you to optimize your app for a variety of device configurations, such as different languages and screen sizes.

For every resource that you include in your Android project, the SDK build tools define a unique integer ID, which you can use to reference the resource from your app code or from other resources defined in XML. For example, if your app contains an image file named logo.png (saved in the res/drawable/ directory), the SDK tools generate a resource ID named R.drawable.logo. This ID maps to an app-specific integer, which you can use to reference the image and insert it in your user interface.

You should place each type of resource in a specific subdirectory of your project's res/ directory. For example, here's the file hierarchy for a simple project:

MyProject/  
 src/  
 MyActivity.java  
 res/  
 drawable/  
 graphic.png  
 layout/  
 main.xml  
 info.xml  
 mipmap/  
 icon.png  
 values/  
 strings.xml

### 1.4.5. Permissions

The purpose of a *permission* is to protect the privacy of an Android user. Android apps must request permission to access sensitive user data (such as contacts and SMS), as well as certain system features (such as camera and internet). Depending on the feature, the system might grant the permission automatically or might prompt the user to approve the request.

A central design point of the Android security architecture is that no app, by default, has permission to perform any operations that would adversely impact other apps, the operating system, or the user. This includes reading or writing the user's private data (such as contacts or emails), reading or writing another app's files, performing network access, keeping the device awake, and so on.

#### 1.4.5.1. Permission approval

An app must publicize the permissions it requires by including <uses-permission> tags in the app manifest. For example, an app that needs to send SMS messages would have this line in the manifest:

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.snazzyapp">

**<uses-permission android:name="android.permission.SEND\_SMS"/>**

<application ...>

... </application>

</manifest>

If your app lists *normal* permissions in its manifest (that is, permissions that don't pose much risk to the user's privacy or the device's operation), the system automatically grants those permissions to your app.

# Chapter 2: XML in Android Development

## 2.1. What is XML?

Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML (ISO 8879). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. XML, stores data in predefined forms from where it may easily be retrieved and shared. While alternative means of representing structured data as text have made inroads, XML is still the most commonly used format.

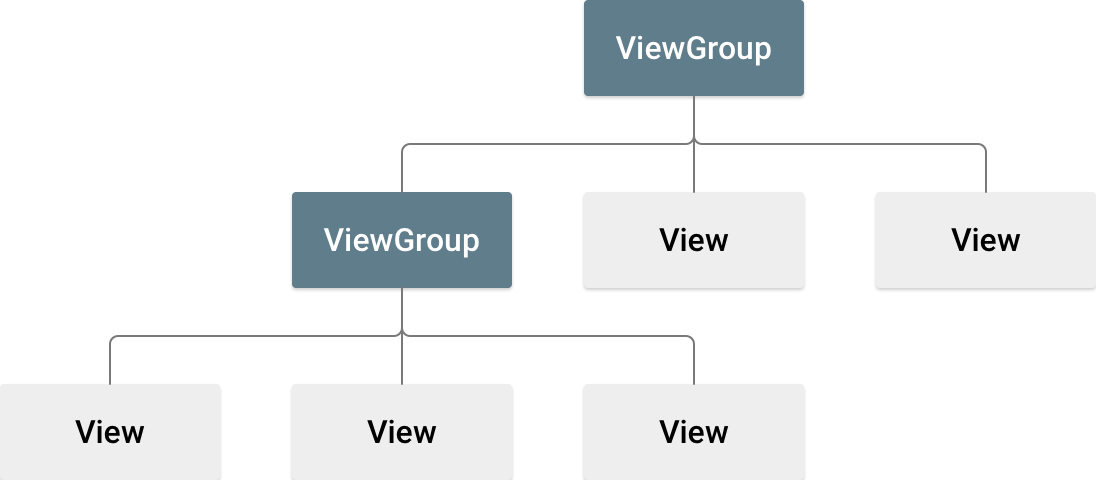
XML is a markup language that often does not get sufficient attention from students and newer developers. However, the significance of XML must not be ignored. Some of the many uses that XML is put to are listed below:

* A significant number of enterprises use XML as a data exchange format. XML is the de facto standard for exchanging messages between enterprise applications in a Services Oriented Architecture. Messages that conform to the canonical model are converted back and forth to XML.
* Business processes between enterprises are more than ever inter-connected at a global scale. B2B data hubs often standardize on XML as their data exchange format.
* Many industry standards have evolved over the years that are based on XML. Years of work and expertise have gone into these standards. In particular, this is the case in finance (ESMA TRACE, MIFID, XBRL) retail, healthcare (HL7), life sciences (CDISC), and public sector (EU) just to name a few.
* XML is used as a serialization format for RDFs (RDF/XML) in a semantic web context.
* In the publishing industry, XML is used throughout the document processing workflow. It is also the standard for Office file formats such as Word, Excel, PowerPoint or the Google Docs equivalents.
* XML finds use for the development of layouts, visual or otherwise in many applications. In web apps, XML may be used for semantically describing the data in a page.

Besides the above-mentioned uses, one use currently is of interest to us- XML in Android development. The User Interface for Android applications and the layout of the elements on the screen are described using XML.

## 2.2. Layouts using XML

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 and ViewGroup 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 below.



*Figure n. Illustration of a view hierarchy, which defines a UI layout*

The View objects are usually called "widgets" and can be one of many subclasses, such as Button or TextView. The ViewGroup objects are usually called "layouts" can be one of many types that provide a different layout structure, such as LinearLayout or ConstraintLayout.

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 (discussed further in Supporting Different Screen Sizes).

The Android framework gives you the flexibility to use either or both of these methods to build your app's UI. For example, you can declare your app's default layouts in XML, and then modify the layout at runtime.

### 2.2.1. Write the XML

Using Android's XML vocabulary, you can quickly design UI layouts and the screen elements they contain, in the same way as you create web pages in HTML — with a series of nested elements.

Each layout file must contain exactly one root element, which must be a View or ViewGroup object. Once you've defined the root element, you can add additional layout objects or widgets as child elements to gradually build a View hierarchy that defines your layout.

For example, here's an XML layout that uses a vertical LinearLayout to hold a TextView and a Button:

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical" >

<TextView android:id="@+id/text"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Hello, I am a TextView" />

<Button android:id="@+id/button"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Hello, I am a Button" />

</LinearLayout>

After you've declared your layout in XML, save the file with the .xml extension, in your Android project's **res/layout/** directory, so it will properly compile.

### 2.2.2. Loading the XML resources

When you compile your app, each XML layout file is compiled into a View resource. You should load the layout resource from your app code, in your Activity.onCreate() callback implementation. Do so by calling setContentView(), passing it the reference to your layout resource in the form of R.layout.layout\_file\_name. For example, if your XML layout is saved as main\_layout.xml, you would load it for your Activity like so (Kotlin):

fun onCreate(savedInstanceState: Bundle) {

super.onCreate(savedInstanceState)

setContentView(R.layout.main\_layout)

}

The onCreate() callback method in your Activity is called by the Android framework when your Activity is launched (see the discussion about lifecycles, in the Activities document).

### 2.2.3. Attributes

Every View and ViewGroup object supports their own variety of XML attributes. Some attributes are specific to a View object (for example, TextView supports the textSize attribute), but these attributes are also inherited by any View objects that may extend this class. Some are common to all View objects because they are inherited from the root View class (like the id attribute). And, other attributes are considered "layout parameters," which are attributes that describe certain layout orientations of the View object, as defined by that object's parent ViewGroup object.

#### ID

Any View object may have an integer ID associated with it, to uniquely identify the View within the tree. When the app is compiled, this ID is referenced as an integer, but the ID is typically assigned in the layout XML file as a string, in the id attribute. This is an XML attribute common to all View objects (defined by the View class) and you will use it very often. The syntax for an ID, inside an XML tag is:

android:id="@+id/my\_button"

The at symbol (@) at the beginning of the string indicates that the XML parser should parse and expand the rest of the ID string and identify it as an ID resource. The plus-symbol (+) means that this is a new resource name that must be created and added to our resources (in the R.java file). There are a number of other ID resources that are offered by the Android framework. When referencing an Android resource ID, you do not need the plus-symbol, but must add the android package namespace, like so:

android:id="@android:id/empty"

With the android package namespace in place, we're now referencing an ID from the android.R resources class, rather than the local resources class.

In order to create views and reference them from the app, a common pattern is to:

1. Define a view/widget in the layout file and assign it a unique ID:

<Button android:id="@+id/my\_button"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="@string/my\_button\_text"/>

1. Then create an instance of the view object and capture it from the layout (typically in the onCreate() method) (Kotlin):

val myButton: Button = findViewById(R.id.my\_button)

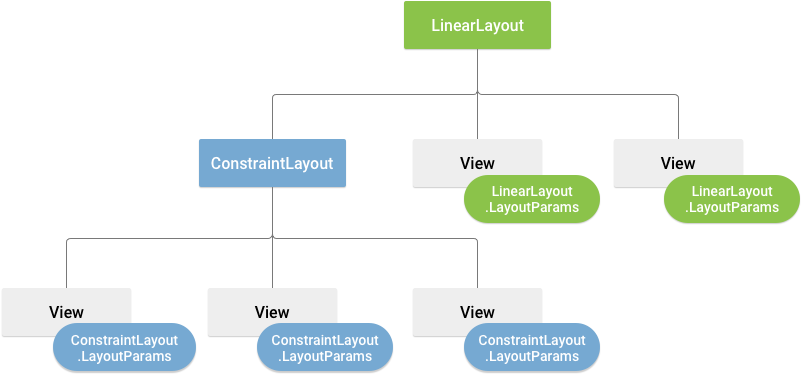
Defining IDs for view objects is important when creating a RelativeLayout. In a relative layout, sibling views can define their layout relative to another sibling view, which is referenced by the unique ID.

An ID need not be unique throughout the entire tree, but it should be unique within the part of the tree you are searching (which may often be the entire tree, so it's best to be completely unique when possible).

#### Layout Parameters

XML layout attributes named layout\_something define layout parameters for the View that are appropriate for the ViewGroup in which it resides.

Every ViewGroup class implements a nested class that extends ViewGroup.LayoutParams. This subclass contains property types that define the size and position for each child view, as appropriate for the view group. As you can see in figure 2, the parent view group defines layout parameters for each child view (including the child view group).



*Figure n. Visualization of a view hierarchy with layout parameters associated with each view*

Note that every LayoutParams subclass has its own syntax for setting values. Each child element must define LayoutParams that are appropriate for its parent, though it may also define different LayoutParams for its own children.

All view groups include a width and height (layout\_width and layout\_height), and each view is required to define them. Many LayoutParams also include optional margins and borders.

You can specify width and height with exact measurements, though you probably won't want to do this often. More often, you will use one of these constants to set the width or height:

* **wrap\_content** tells your view to size itself to the dimensions required by its content.
* **match\_parent** tells your view to become as big as its parent view group will allow.

In general, specifying a layout width and height using absolute units such as pixels is not recommended. Instead, using relative measurements such as density-independent pixel units (**dp**), wrap\_content, or match\_parent, is a better approach, because it helps ensure that your app will display properly across a variety of device screen sizes. The accepted measurement types are defined in the Available Resources document.

### 2.2.4. Layout Position

The geometry of a view is that of a rectangle. A view has a location, expressed as a pair of left and top coordinates, and two dimensions, expressed as a width and a height. The unit for location and dimensions is the pixel.

It is possible to retrieve the location of a view by invoking the methods getLeft() and getTop(). The former returns the left, or X, coordinate of the rectangle representing the view. The latter returns the top, or Y, coordinate of the rectangle representing the view. These methods both return the location of the view relative to its parent. For instance, when getLeft() returns 20, that means the view is located 20 pixels to the right of the left edge of its direct parent.

In addition, several convenience methods are offered to avoid unnecessary computations, namely getRight() and getBottom(). These methods return the coordinates of the right and bottom edges of the rectangle representing the view. For instance, calling getRight() is similar to the following computation: getLeft() + getWidth().

### 2.2.5. Size, Padding, and Margins

The size of a view is expressed with a width and a height. A view actually possesses two pairs of width and height values.

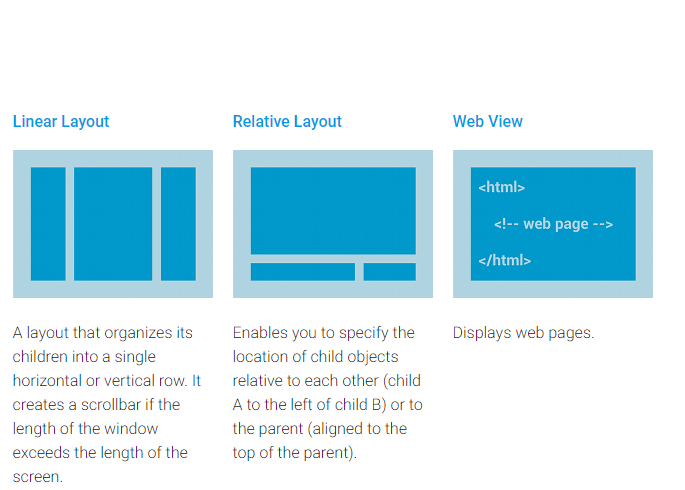
The first pair is known as measured width and measured height. These dimensions define how big a view wants to be within its parent. The measured dimensions can be obtained by calling getMeasuredWidth() and getMeasuredHeight().

The second pair is simply known as width and height, or sometimes drawing width and drawing height. These dimensions define the actual size of the view on screen, at drawing time and after layout. These values may, but do not have to, be different from the measured width and height. The width and height can be obtained by calling getWidth() and getHeight().

To measure its dimensions, a view takes into account its padding. The padding is expressed in pixels for the left, top, right and bottom parts of the view. Padding can be used to offset the content of the view by a specific number of pixels. For instance, a left padding of 2 will push the view's content by 2 pixels to the right of the left edge. Padding can be set using the setPadding(int, int, int, int) method and queried by calling getPaddingLeft(), getPaddingTop(), getPaddingRight() and getPaddingBottom().

### 2.2.6. Common Layouts

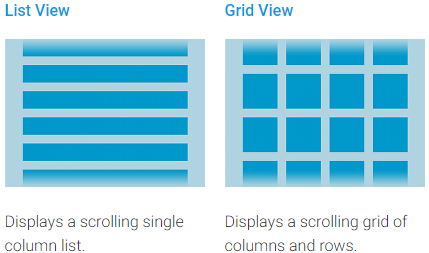
Each subclass of the [ViewGroup](https://developer.android.com/reference/android/view/ViewGroup.html) class provides a unique way to display the views you nest within it. Below are some of the more common layout types that are built into the Android platform.



### 2.2.7. Building Layouts with an Adapter

When the content for your layout is dynamic or not predetermined, you can use a layout that subclasses AdapterView to populate the layout with views at runtime. A subclass of the AdapterView class uses an Adapter to bind data to its layout. The Adapter behaves as a middleman between the data source and the AdapterView layout—the Adapter retrieves the data (from a source such as an array or a database query) and converts each entry into a view that can be added into the AdapterView layout.

Common layouts backed by an adapter include:



#### Filling an adapter view with data

You can populate an AdapterView such as ListView or GridView by binding the AdapterView instance to an Adapter, which retrieves data from an external source and creates a View that represents each data entry.

Android provides several subclasses of Adapter that are useful for retrieving different kinds of data and building views for an AdapterView. One of the most common adapters is the ArrayAdapter:

##### ArrayAdapter

Use this adapter when your data source is an array. By default, ArrayAdapter creates a view for each array item by calling toString() on each item and placing the contents in a TextView.

For example, if you have an array of strings you want to display in a ListView, initialize a new ArrayAdapter using a constructor to specify the layout for each string and the string array (Kotlin):

val adapter = ArrayAdapter<String>(this, android.R.layout.simple\_list\_item\_1, myStringArray)

The arguments for this constructor are:

* Your app Context
* The layout that contains a TextView for each string in the array
* The string array

Then simply call setAdapter() on your ListView (Kotlin):

val listView: ListView = findViewById(R.id.listview)

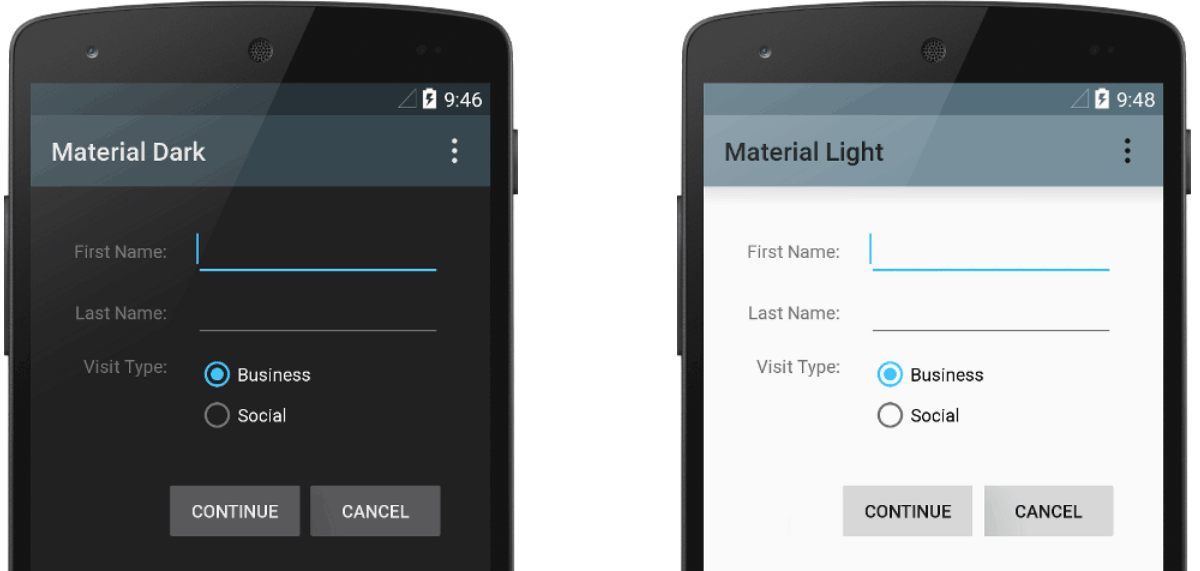
listView.adapter = adapter

To customize the appearance of each item you can override the toString() method for the objects in your array. Or, to create a view for each item that's something other than a TextView (for example, if you want an ImageView for each array item), extend the ArrayAdapter class and override getView() to return the type of view you want for each item.

## 2.3. Styles and Themes

Styles and themes on Android allow you to separate the details of your app design from the UI structure and behavior, similar to stylesheets in web design.  
  
A style is a collection of attributes that specify the appearance for a single View. A style can specify attributes such as font color, font size, background color, and much more.

A theme is a type of style that's applied to an entire app, activity, or view hierarchy—not just an individual view. When you apply your style as a theme, every view in the app or activity applies each style attribute that it supports. Themes can also apply styles to non-view elements, such as the status bar and window background.  
  
Styles and themes are declared in a style resource file in res/values/, usually named **styles.xml**.



*Figure n. Two themes applied to the same activity: Theme.AppCompat (left) and Theme.AppCompat.Light (right)*

### 2.3.1. Create and apply a style

To create a new style or theme, open your project's res/values/styles.xml file. For each style you want to create, follow these steps:

1. Add a <style> element with a name that uniquely identifies the style.
2. Add an <item> element for each style attribute you want to define.

The name in each item specifies an attribute you would otherwise use as an XML attribute in your layout. The value in the <item> element is the value for that attribute.  
  
For example, if you define the following style:  
  
<?xml version="1.0" encoding="utf-8"?>  
<resources>  
 <style name="GreenText" parent="TextAppearance.AppCompat">  
 <item name="android:textColor">#00FF00</item>  
 </style>  
</resources>  
You can apply the style to a view as follows:  
  
<TextView  
 style="@style/GreenText"  
 ... />

Each attribute specified in the style is applied to that view if the view accepts it. The view simply ignores any attributes that it does not accept.

However, instead of applying a style to individual views, you'll usually apply styles as a theme for your entire app, activity, or collection of views.

### 2.3.2. Extend and customize a style

When creating your own styles, you should always extend an existing style from the framework or support library so that you maintain compatibility with platform UI styles. To extend a style, specify the style you want to extend with the parent attribute. You can then override the inherited style attributes and add new ones.  
  
For example, you can inherit the Android platform's default text appearance and modify it as follows:  
  
<style name="GreenText" parent="@android:style/TextAppearance">  
 <item name="android:textColor">#00FF00</item>  
</style>

However, you should always inherit your core app styles from the Android Support Library. The styles in the support library provide compatibility with Android 4.0 (API level 14) and higher by optimizing each style for the UI attributes available in each version. The support library styles often have a name similar to the style from the platform, but with AppCompat included.  
  
To inherit styles from a library or your own project, declare the parent style name without the @android:style/ part shown above. For example, the following example inherits text appearance styles from the support library:  
  
<style name="GreenText" parent="TextAppearance.AppCompat">  
 <item name="android:textColor">#00FF00</item>  
</style>

You can also inherit styles (except those from the platform) by extending a style's name with a dot notation, instead of using the parent attribute. That is, prefix the name of your style with the name of the style you want to inherit, separated by a period. You should usually do this only when extending your own styles, not styles from other libraries. For example, the following style inherits all styles from the GreenText style above and then increases the text size:  
  
<style name="GreenText.Large">  
 <item name="android:textSize">22dp</item>  
</style>

You can continue inheriting styles like this as many times as you'd like by chaining on more names.  
  
To find which attributes you can declare with an <item> tag, refer to the "XML attributes" table in the various class references. All views support XML attributes from the base View class, and many views add their own special attributes. For example, the TextView XML attributes include the android:inputType attribute that you can apply to a text view that receives input, such as an EditText widget.

### 2.3.3. Apply a style as a theme

You can create a theme in the same way you create styles. The difference is how you apply it: instead of applying a style with the style attribute on a view, you apply a theme with the android:theme attribute on either the <application> tag or an <activity> tag in the AndroidManifest.xml file.  
  
For example, here's how to apply the Android Support Library's material design "dark" theme to the whole app:  
  
<manifest ... >  
 <application android:theme="@style/Theme.AppCompat" ... >  
 </application>  
</manifest>  
And here's how to apply the "light" theme to just one activity:  
  
<manifest ... >  
 <application ... >  
 <activity android:theme="@style/Theme.AppCompat.Light" ... >  
 </activity>  
 </application>  
</manifest>

### Now every view in the app or activity applies the styles defined in the given theme. If a view supports only some of the attributes declared in the style, then it applies only those attributes and ignores the ones it does not support. Beginning with Android 5.0 (API level 21) and Android Support Library v22.1, you can also specify the android:theme attribute to a view in your layout file. This modifies the theme for that view and any child views, which is useful for altering theme color palettes in a specific portion of your interface. The previous examples show how to apply a theme such as Theme.AppCompat that's supplied by the Android Support Library. But you'll usually want to customize the theme to fit your app's brand. The best way to do so is to extend these styles from the support library and override some of the attributes, as described in the next section. 2.3.4. Style hierarchy

Android provides a variety of ways to set attributes throughout your Android app. For example, you can set attributes directly in a layout, you can apply a style to a view, you can apply a theme to a layout, and you can even set attributes programmatically.  
  
When choosing how to style your app, be mindful of Android's style hierarchy. In general, you should use themes and styles as much as possible for consistency. If you've specified the same attributes in multiple places, the list below determines which attributes are ultimately applied. The list is ordered from highest precedence to lowest:

1. Applying character- or paragraph-level styling via text spans to TextView-derived classes
2. Applying attributes programmatically
3. Applying individual attributes directly to a View
4. Applying a style to a View
5. Default styling
6. Applying a theme to a collection of Views, an activity, or your entire app
7. Applying certain View-specific styling, such as setting a TextAppearance on a TextView

### 2.3.5. Customize the default theme

When you create a project with Android Studio, it applies a material design theme to your app by default, as defined in your project's styles.xml file. This AppTheme style extends a theme from the support library and includes overrides for color attributes that are used by key UI elements, such as the app bar and the floating action button (if used). So you can quickly customize your app's color design by updating the provided colors.  
  
For example, your styles.xml file should look similar to this:

<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">  
 <!-- Customize your theme here. -->  
 <item name="colorPrimary">@color/colorPrimary</item>  
 <item name="colorPrimaryDark">@color/colorPrimaryDark</item>  
 <item name="colorAccent">@color/colorAccent</item>  
</style>

Notice that the style values are actually references to other color resources, defined in the project's res/values/colors.xml file. Once you know your colors, update the values in res/values/colors.xml:

<?xml version="1.0" encoding="utf-8"?>  
<resources>  
 <!-- color for the app bar and other primary UI elements -->  
 <color name="colorPrimary">#3F51B5</color>  
  
 <!-- a darker variant of the primary color, used for  
 the status bar (on Android 5.0+) and contextual app bars -->  
 <color name="colorPrimaryDark">#303F9F</color>  
  
 <!-- a secondary color for controls like checkboxes and text fields -->  
 <color name="colorAccent">#FF4081</color>  
</resources>

And then you can override whatever other styles you want. For example, you can change the activity background color as follows:  
  
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">  
 ...  
 <item name="android:windowBackground">@color/activityBackground</item>  
</style>

For a list of attributes which you can use in your theme, see the table of attributes at R.styleable.Theme. And when adding styles for the views in your layout, you can also find attributes by looking at the "XML attributes" table in the view class references. For example, all views support XML attributes from the base View class.  
  
Most attributes are applied to specific types of views, and some apply to all views. However, some theme attributes listed at R.styleable.Theme apply to the activity window, not the views in the layout. For example, windowBackground changes the window background and windowEnterTransition defines a transition animation to use when the activity starts (for details, see Start an Activity with an Animation).  
  
The Android Support Library also provides other attributes you can use to customize your theme extended from Theme.AppCompat (such as the colorPrimary attribute shown above). These are best viewed in the library's attrs.xml file  
  
*Note: Attribute names from the support library do not use the android: prefix. That's used only for attributes from the Android framework.*  
There are also different themes available from the support library that you might want to extend instead of the ones shown above. The best place to see the available themes is the library's themes.xml file.

# CHAPTER 3: Kotlin in Android development

## 3.1. What is Kotlin?

**Kotlin** is a [statically typed](https://en.wikipedia.org/wiki/Statically_typed) [programming language](https://en.wikipedia.org/wiki/Programming_language) that runs on the [Java virtual machine](https://en.wikipedia.org/wiki/Java_virtual_machine) and also can be compiled to [JavaScript](https://en.wikipedia.org/wiki/JavaScript) source code or use the [LLVM](https://en.wikipedia.org/wiki/LLVM) compiler infrastructure. Its primary development is from a team of [JetBrains](https://en.wikipedia.org/wiki/JetBrains) programmers based in [Saint Petersburg](https://en.wikipedia.org/wiki/Saint_Petersburg), Russia. While the syntax is not compatible with Java, the JVM implementation of the Kotlin standard library is designed to interoperate with [Java](https://en.wikipedia.org/wiki/Java_(programming_language)) code and relies on Java code from the existing [Java Class Library](https://en.wikipedia.org/wiki/Java_Class_Library), such as the collections framework. Kotlin uses aggressive [type inference](https://en.wikipedia.org/wiki/Type_inference) to determine the types of values and expressions for which type has been left unstated. This reduces language verbosity relative to Java, which demands often entirely redundant type specifications prior to [version 10](https://en.wikipedia.org/wiki/Java_version_history#Java_SE_10). Kotlin code can run on JVM up to latest Java 11.

Kotlin files have extension ‘.kt’.

As of [Android Studio 3.0](https://en.wikipedia.org/wiki/Android_Studio#Version_history), Kotlin is fully supported by [Google](https://en.wikipedia.org/wiki/Google) for use with their [Android](https://en.wikipedia.org/wiki/Android_(operating_system)) operating system, and is directly included in the IDE's installation package as an alternative to the standard Java compiler. The Android Kotlin compiler lets the user choose between targeting [Java](https://en.wikipedia.org/wiki/Java_language) 6, or Java 8-compatible bytecode.

## 3.2 Why Kotlin over Java for android?

* Easy to set up. User simply need to tick the ‘Include Kotlin support’ box to use a plugin to add Kotlin into android project
* Kotlin is a java-based programming language and is interoperable. User can call Java code using Kotlin and vice-versa. This makes it easier for user to use Kotlin in existing project with old java code.
* Kotlin is more readable or reader-friendly as compare to java which make it more suitable as developers (or someone else) may need to read the whole code again and again.
* Syntax is not as strict as in case of java like there is no use of ‘;’ at end of each line in Kotlin.
* While using java for android, developers face NullPointException. In Kotlin there is no NullPointException.
* Kotlin was first created with JVM in mind, so it can virtually be used on any devices that can run JVM.
* Kotlin follows the principle ‘less code, more functionality’. In Kotlin we can add more functionality in less number of code lines as compare to Java.
* There are other features in Kotlin which are not present in Java like smart casting, null-safety, companion objects, string templates, primary constructors, extension functions, type inference for variables and property types.

## 3.3. Basics of Kotlin

### 3.3.1. Variables

### Read-only local variables are defined using the keyword val. They can be assigned a value only once.

### val a: Int = 1 // immediate assignment

val b = 2 // `Int` type is inferred

val c: Int // Type required when no initializer is provided

c = 3 // deferred assignment

Variables that can be reassigned use the var keyword:

var x = 5 // `Int` type is inferred

x += 1

Kotlin is modern language which self-inferred the data-type of the variable. Data-types used inside Kotlin are:

* Byte-8 bits
* Int – 32 bits
* Short – 16 bits
* Long – 64 bits
* Float – 32 bits
* Double – 64 bits
* Boolean
* Strings

### 3.3.2. Functions

Functions in Kotlin are defined using the keyword ‘**fun**’ followed by function’s name and parameters. Return type of function can be explicitly mentioned or can be inferred by the JVM.

Function having two Int parameters with Int return type:

### fun sum(a: Int, b: Int): Int {

### return a + b

### }

Function with an expression body and inferred return type:

### fun sum(a: Int, b: Int) = a + b

### 3.3.3. Using Conditional Statements

### fun maxOf(a: Int, b: Int): Int {

### if (a > b) {

### return a

### } else {

### return b

### }

### }

Using ‘**if**’ as an expression:

fun maxOf(a: Int, b: Int) = if (a > b) a else b

### 3.3.4. Loops

#### 3.3.4.1 For Loop

val items = listOf("apple", "banana", "kiwifruit")

for (item in items) {

println(item)

}

#### 3.3.4.2 While Loop

val items = listOf("apple", "banana", "kiwifruit")

var index = 0

while (index < items.size) {

println("item at $index is ${items[index]}")

index++

}

### 3.3.5. Type checks and automatic casts

The *is* operator checks if an expression is an instance of a type. If an immutable local variable or property is checked for a specific type, there's no need to cast it explicitly:

fun getStringLength(obj: Any): Int? {

if (obj is String) {

// `obj` is automatically cast to `String` in this branch

return obj.length

}

// `obj` is still of type `Any` outside of the type-checked branch

return null

}

**OR**

fun getStringLength(obj: Any): Int? {

if (obj !is String) return null

// `obj` is automatically cast to `String` in this branch

return obj.length

}

Every number type supports the following conversions:

* toByte(): Byte
* toShort():Short
* toInt():Int
* toChar(): Char
* toDouble(): Double
* toFloat(): Float
* toLong(): Long

Absence of implicit conversions is rarely noticeable because the type is inferred from the context, and arithmetical operations are overloaded for appropriate conversions, for example:

val l = 1L + 3 // Long + Int => Long

### 3.3.6. ‘when’ expression in Kotlin

### This works as switch statement in C/C++. For e.g:

### fun describe(obj: Any): String =

when (obj) {

1 -> "One"

"Hello" -> "Greeting"

is Long -> "Long"

!is String -> "Not a string"

else -> "Unknown"

}

### 3.3.7. Classes in Kotlin

Classes in Kotlin are declared using the keyword class. The class declaration consists of the class name, the class header (specifying its type parameters, the primary constructor etc.) and the class body, surrounded by curly braces. Both the header and the body are optional; if the class has no body, curly braces can be omitted.

class Invoice { ... }

**OR**

class Invoice

#### 3.3.7.1. Constructors in class

A class in Kotlin can have a **primary constructor** and one or more **secondary constructors**. The primary constructor is part of the class header: it goes after the class name (and optional type parameters). f the primary constructor does not have any annotations or visibility modifiers, the constructor keyword can be omitted

class Person constructor(firstName: String) { ... }

**OR**

class Person(firstName: String) { ... }

#### 3.3.7.2. Initialization block

The primary constructor cannot contain any code. Initialization code can be placed in **initializer blocks**, which are prefixed with the init keyword. During an instance initialization, the initializer blocks are executed in the same order as they appear in the class body, interleaved with the property initializers:

class InitOrderDemo(name: String) {

val firstProperty = "First property: $name".also(::println)

init {

println("First initializer block that prints ${name}")

}

val secondProperty = "Second property:${name.length}".also(::println)

init {

println("Second initializer block that prints ${name.length}")

}

}

#### 3.3.7.3. Creating Instance of Class

To create an instance of a class, we call the constructor as if it were a regular function:

val invoice = Invoice()

val customer = Customer("Joe Smith")

#### 3.3.7.4. Inheritance in Kotlin

All classes in Kotlin have a common superclass Any, that is the default superclass for a class with no supertypes declared. To declare an explicit supertype, we place the type after a colon in the class header:

open class Base(p: Int)

class Derived(p: Int) : Base(p)

If the derived class has a primary constructor, the base class can (and must) be initialized right there, using the parameters of the primary constructor. If the class has no primary constructor, then each secondary constructor has to initialize the base type using the **super** keyword, or to delegate to another constructor which does that. Note that in this case different secondary constructors can call different constructors of the base type:

class MyView : View {

constructor(ctx: Context) : super(ctx)

constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs)

}

#### 3.3.7.5. Overriding methods

Unlike Java, Kotlin requires explicit modifiers for overridable members (we call them *open*) and for overrides.

open class Base {

open fun v() { ... }

fun nv() { ... }

}

class Derived() : Base() {

override fun v() { ... }

}

#### 3.3.7.6. Nested classes

Nested classes are the classes which are defined inside some other class. Classes can be nested in other classes:

class Outer {

private val bar: Int = 1

class Nested {

fun foo() = 2

}

}

val demo = Outer.Nested().foo() // == 2

A class may be marked as inner to be able to access members of outer class. Inner classes carry a reference to an object of an outer class:

class Outer {

private val bar: Int = 1

inner class Inner {

fun foo() = bar

}

}

val demo = Outer().Inner().foo() // == 1

### 3.3.8. Arrays in Kotlin

Arrays in Kotlin are represented by the Array class, that has get and set functions (that turn into [] by operator overloading conventions), and size property, along with a few other useful member functions:

class Array<T> private constructor() {

val size: Int

operator fun get(index: Int): T

operator fun set(index: Int, value: T): Unit

operator fun iterator(): Iterator<T>

// ... }

To create an array, we can use a library function arrayOf() and pass the item values to it, so that arrayOf(1, 2, 3) creates an array [1, 2, 3]. Alternatively, the arrayOfNulls() library function can be used to create an array of a given size filled with null elements.

### 3.3.9. Packages and Import in Kotlin

**Packages:**

All the contents (such as classes and functions) of the source file are contained by the package declared. So, in the example above, the full name of baz() is foo.bar.baz, and the full name of Goo is foo.bar.Goo.

If the package is not specified, the contents of such a file belong to "default" package that has no name.

A source file may start with a package declaration:

package foo.bar

fun baz() { ... }

class Goo { ... }

// ...

**Import:**

Apart from the default imports, each file may contain its own import directives. Syntax for imports is described in the grammar.

We can import either a single name, e.g

import foo.Bar // Bar is now accessible without qualification

or all the accessible contents of a scope (package, class, object etc):

import foo.\* // everything in 'foo' becomes accessible

### 3.3.10. ArrayLists in Kotlin

Arraylist is extension of array and lets you add, delete and do more operations with the elements of an array. It is commonly referred to as List:

var team = arrayListOf(“player1”,”player2”,”player3”) //creating Arraylist

Arrays are non-resizable. But arraylist are like dynamic arrays whose size can be increased or decreased.

Operations on ArrayLists are:

#### 3.3.10.1 add( )

Adds the specified element to the end of this list.

team.add(“player4”) //add player4 to the end of arraylist team

#### 3.3.10.2 addAll( )

Adds all of the elements of the specified collection to the end of this list. The elements are appended in the order they appear in the elements collection.

team.add(“player5”,”player6”) //add player5 and player6 to team

#### 3.3.10.3 clear( )

Clears all the elements of arraylist.

team.clear() // will delete all the elements of team

#### 3.3.10.4 get( index:Int)

Returns the element with index passed as argument.

team.get(2) // will return the element at index 2 of arraylist team

#### 3.3.10.5 indexOf(Element e)

Returns the index of element ‘e’ inside arraylist, if it exists.

team.indexOf(“player3”) // will return the index of player3 in arraylist

#### 3.3.10.6 remove(Element e)

Removes the element ‘e’ from the arraylist if it exists.

team.remove(“player5”) // will remove player 5 from the list

# Chapter 4: Echo Music Player App

## 4.1. Aim:

The aim of the project is to build a fully functional music player, called **Echo.** The Echo app is able to fulfill all functions expected from a music player- searching for music, playing music with the ability to go forward or backward to any point, favorites, etc.

Not only must the app be capable to perform these functions, it must also perform these efficiently. The app must be lightweight and function well even on low-tier devices to maximize the target audience of the app. This will require efficient use of DB management as well as device resources (memory). App functioning optimizations are an essential part of app development in the modern day, and must be applied to achieve optimal performance.

We must also understand that a working app is not enough. An app must be able to draw in audience and get people to fall in love. This is achieved through an intuitive User Experience and also an appealing User Interface. The UI must allow the user to achieve all the functions that he expects from the app while also being good-looking and appealing. Use of colour schemes, styles, proper placements and layouts, etc. are crucial in the process.

Once an app is created, it must be thoroughly tested and reviewed. Testing is essential to the development process and through testing, we are able to find out weaknesses of our app, as well as bugs. Debugging is an essential skill which must be utilised to ensure that our app functions as we want it to and is able to satisfy a user.

After we make the app, it must not remain limited to the confines of our computer- it must be put out to the world for everyone to use. The default, and the most popular, platform for releasing Android apps is, of course, the Android Play Store. Thus, we also learned how to extract a signed **apk** for a file so that we may be able to offer our product to interested people.

## 4.2. Functionalities and features

Music player with the following functionalities:

1. A Splash screen (gradient background and app logo in center)

2. A Navigation drawer with app logo section at the top along with links to ‘All Songs’,

‘Favorites’, ‘Settings’ and ‘About Us’.

3. An ‘All songs’ screen (where of list all the tracks fetched from offline storage are

displayed and user can sort the tracks by name or recently added). This will the

home screen of the app.

4. The app should be able to fetch and play .mp3 and .wav files.

5. A ‘Favorites’ screen (where list of all the favorite songs are displayed)

6. A ‘Settings’ screen (where the ‘Shake to change song’ feature can be enabled or

disabled)

7. An ‘About us’ screen (where we will display information about the app developer and

the app version)

8. A ‘Now playing’ screen with following features:

a. Track title and track artist

b. Play / Pause button

c. Next button

d. Previous button

e. Shuffle button

f. Loop button

g. Seek bar

h. Mark track as favorite or unfavorite it

i. Third party visualiser in upper half background

j. A ‘Back to list’ button in the header which should take the user to the screen

he came from (kind of like back button behaviour).

k. Shake to change song

9. A ‘Now playing’ bar at the bottom with name of the track playing and play or pause

feature. This would appear if the user has moved from ‘Now playing’ screen to ‘All

songs’ screen or ‘Favorites’ screen without pausing the track.

10. Background play. The app will continue playing the track if the app gets closed (not

killed) without the music being paused.

11. A notification saying "A track is playing in the background" only if the app gets closed

(not killed) without the music being paused.

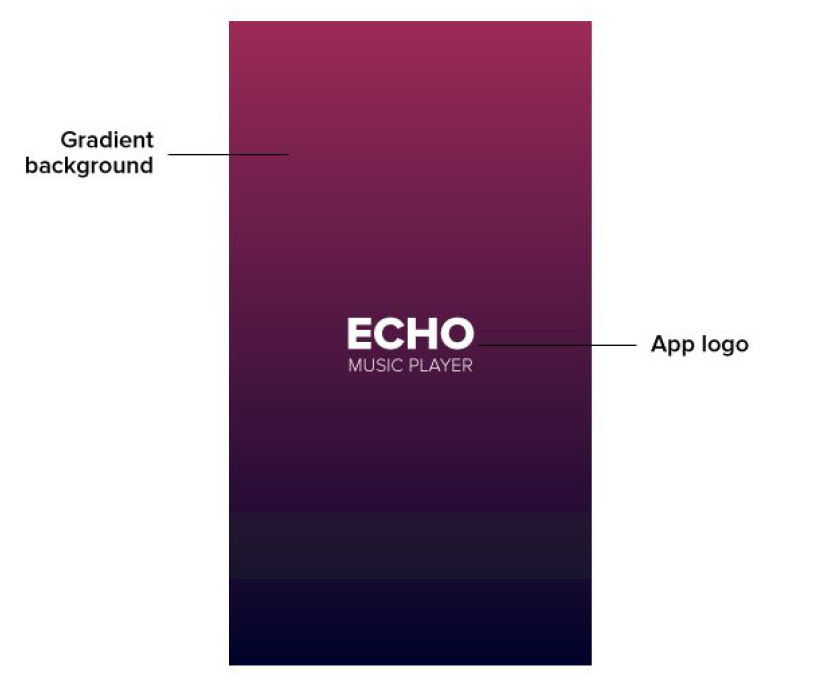
## 4.3. Functionalities in details

### 4.3.1. Splash screen

This would be the first screen that gets displayed when a user opens the app. It would have

a linear gradient background and the app logo in the center of the screen. The user will see

the screen for 1 second and then the home screen will pop up.



### 4.3.2. Navigation Drawer

The navigation drawer is needed so that the user can navigate to different screens within the

app. The user should be able to access the navigation drawer on all the app screens by

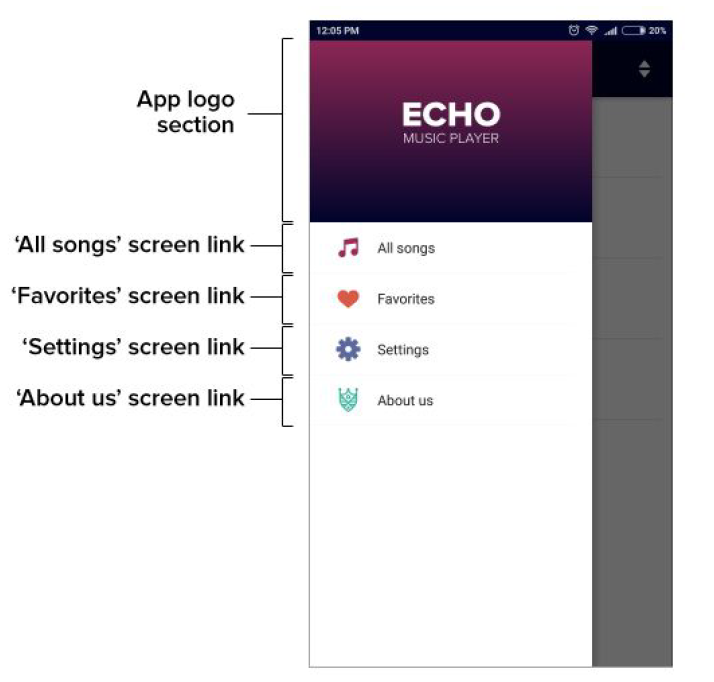
clicking hamburger button on the left side of the header or by swiping right from the left

edge. It would have an app logo section at the top with gradient in the background and app

logo in center (similar to Splash screen but smaller). Below the app logo section, there would

links to ‘All songs’, ‘Favorites’, ‘Settings’ and ‘About us’ screen in a list style. Clicking on any

link will open the corresponding screen.



### 4.3.3. All songs screen

‘All songs’ screen will be the home screen of the app, meaning when the app is launched,

user will see the ‘All songs’ screen after the splash screen.

Once the app is launched, all the tracks (.mp3 and .wav) will be fetched from the offline

storage and will be displayed on the ‘All songs’ screen in a list view. By default, all the tracks

would be sorted by name. For each track in the list, the user should see the title of the track

and the artist of the track. If there is no title, the track name will be displayed in place of the

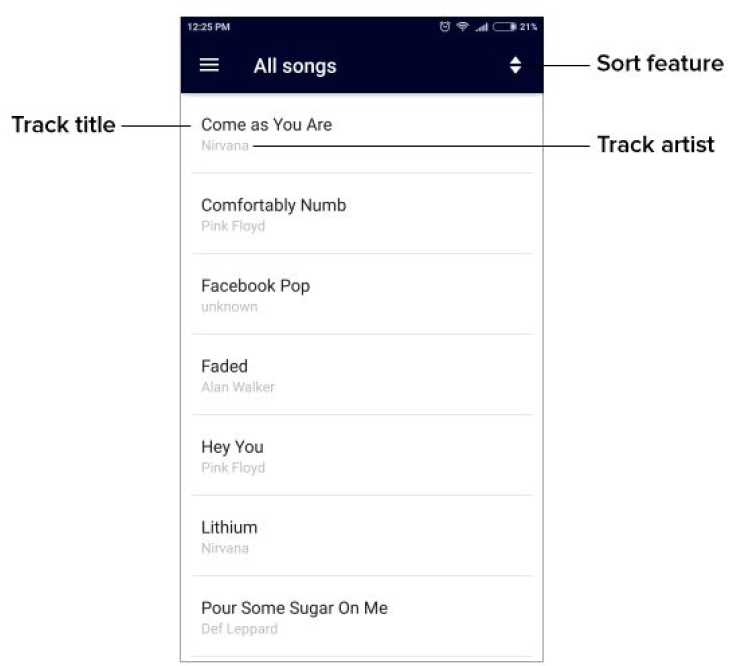
title. If there is no ar

tist, ‘unknown’ will be displayed in place of the artist. There would also be a line separating

each track.

When the user clicks on any track, the ‘Now playing’ screen should open and the track

should start to play.



### 

### 4.3.4. Sort feature

The ‘All songs’ screen will have a Sort feature. Using the Sort feature, the user should be

able to sort the tracks by name or by recently added. A sort icon will be there on the right

side of the header, when clicked on, then a dropdown will appear with two options:

1. By recently added

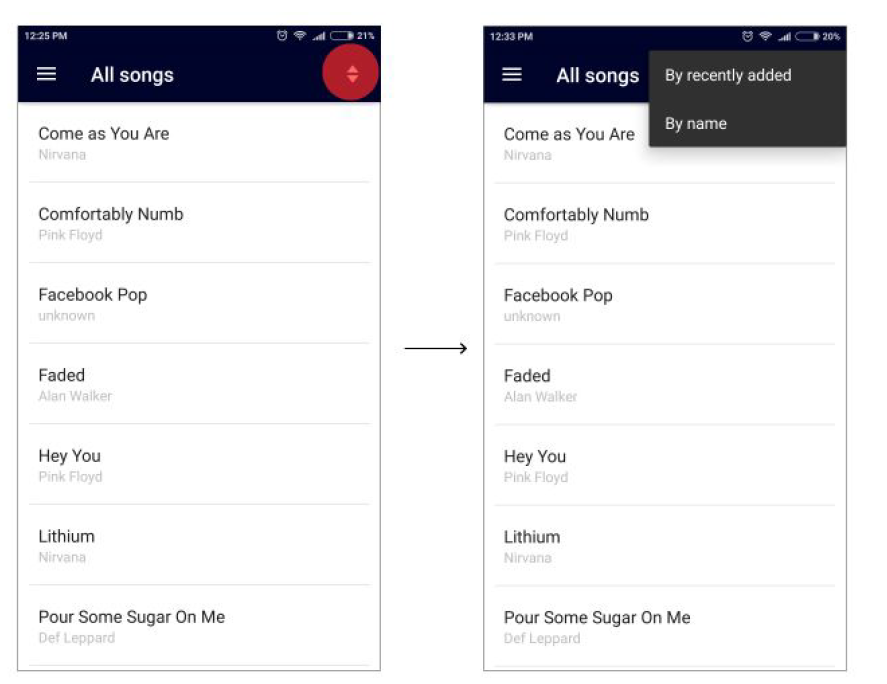
2. By name

If the user clicks on ‘By recently added’, all the tracks would get sorted by recency with most

recently added track on the top. ‘By name’, all the tracks would get sorted in alphabetical

order (or in this order: track title starting with symbols, track title starting with numbers and

then track title starting with letters).



### 4.3.5. Favorites screen

The ‘Favorites’ screen resembles the ‘All songs’ screen. It will display all the tracks that have

been marked favorite by the user. The user would be able to mark a track favorite or

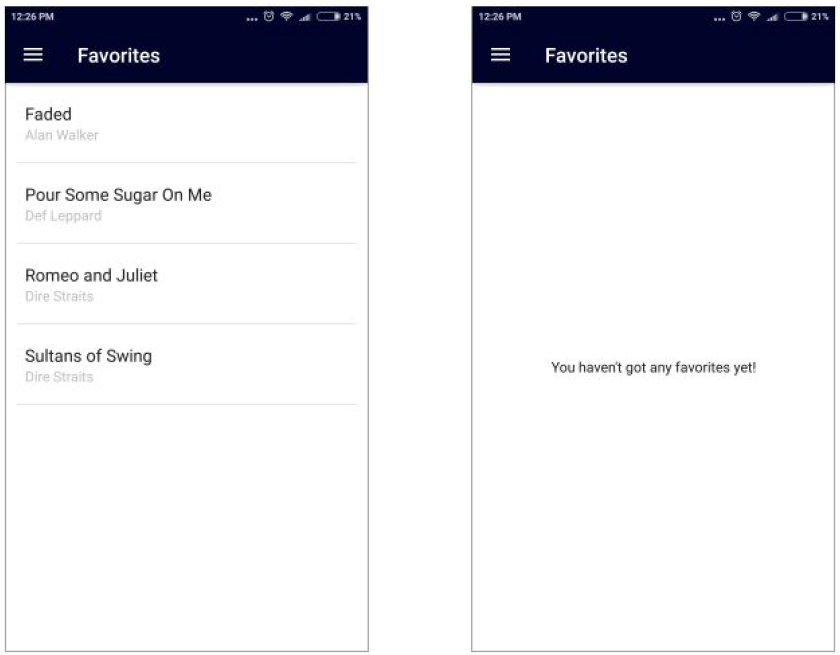
unfavorite it only on the ‘Now playing’ screen. Ensure that if the user marks a track favorite

and then later deletes it from the offline storage, the track shouldn’t appear on the ‘Favorites’

screen.

If the user has no favorite tracks, there would be a message in the center on the screen

saying, “You haven’t got any favorites yet!”.



### 4.3.6. Settings screen

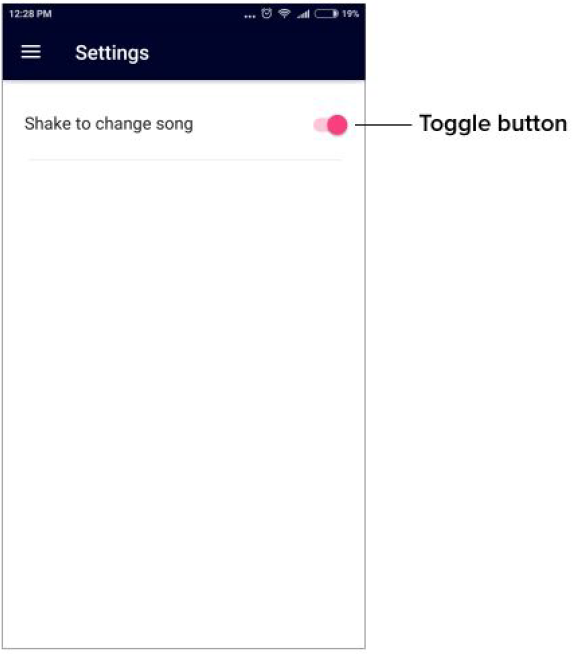
The ‘Settings’ screen will have the option to enable or disable ‘Shake to change song’

feature using the toggle button. The default state of ‘Shake to change song’ feature would be

‘disabled’. If the user enables the ‘Shake to change song’ feature, the app should remember

it, so when the user kills the app and launches it again, the ‘Shake to change song’ should

stay enabled.



### 4.3.7. The Now playing screen

The ‘Now playing’ screen is the actual music player in the app. It houses all the elements

which let the user interact with the current playing track or the next/previous track. It will

open when the user clicks on a track on the ‘All songs’ screen or the ‘Favorites’ screen. It

would have following features and functionalities:

1. Track title and track artist

The track title and track artist would be displayed in the center of bottom half of the

screen. If the track has no title, track name will be displayed. If the track has no artist,

‘unknown’ will be displayed in its place. If the track title or name is too long, an ellipsis

will be added to the title or name.

2. Play/Pause button

As the names suggest, this button will be used to play or pause a track. When a track

is playing in the app, this button will become Pause button. When a track is paused in

the app, this button will become Play button. When a user clicks on the Pause button,

the track will pause and when the Play button is clicked, the track will start playing

from the same place where it was paused.

3. Next button

This button will allow the user to play the next track. When the Next button is clicked,

the consecutive track in the list (from which the ‘Now playing’ screen was triggered)

will start playing.

4. Previous button

This button will allow the user to play the previous track. When the Previous button is

clicked, the consecutive track in the list (from which the ‘Now playing’ screen was

triggered) will start playing.

5. Shuffle button

The default state of Shuffle button would be ‘switched off’ (white). When the Shuffle

button is toggled (switched on) by the user, the button would turn yellow indicating

that Shuffle feature has been turned on. When the Shuffle feature is on, the player

would randomly choose a track (different from the one currently playing) from the list

(from which the ‘Now playing’ screen was triggered) and play it when the next button

is clicked or when the current track ends. When the Shuffle button is toggled again

(switched off), the button would again turn white indicating that Shuffle feature has

been turned off. When Shuffle feature is white or switched off, the player would play

the consecutive track in the list when the next button is clicked or when the current

track ends.

If the user switches the Shuffle feature on, the app should remember it, so when the

user kills the app and launches it again, the Shuffle feature stays on.

If the Shuffle feature is on and the Loop feature is turned on, Shuffle feature would

move back to its default state.

The Shuffle feature and the Loop feature can’t be in ‘switched on’ (yellow) state

simultaneously.

6. Loop button

The default state of the Loop button would be ‘switched off’ (white). When the Loop

button is toggled (switched on) by the user, the button would turn yellow indicating

that Loop feature has been turned on. When the loop feature is on, the player would

play the same track again when the track ends. The loop button won’t affect the

behaviour of Next button. When the Loop button is toggled again (switched off), the

button would again turn white indicating that Loop feature has been turned off.

If the user switches the Loop feature on, the app should remember it, so when the

user kills the app and launches it again, the Loop feature stays on.

If the Loop feature is on and the Shuffle feature is turned on, Loop feature would

move back to its default state.

The Loop feature and the Shuffle feature can’t be in ‘switched on’ (yellow) state

simultaneously.

7. Seek bar

This screen features a seek-bar which displays the track progress throughout the

track’s lifetime.A user can click on the seekbar to skip in between the track or simply

to drag the controller to reach a certain part of the track.

8. ‘Mark as favourite’ button

Clicking this button adds the current track to the favorites list, the button then turns

red indicating that the track has been added to the favorites list. A toast message is

displayed on the screen saying “Added to favorites”.

Clicking this button again will remove the track from the favourites list, the button then

turns back to white indicating that the track has been removed from the favorites list.

A toast message is displayed on the screen saying “Removed from favorites”.

The default state of the ‘Mark as favorite’ button is white, that means, initially there

would be no tracks in the favorites list.

9. Third party visualiser

The ‘Now playing’ screen would have a 4 bar visualiser in the upper half of the

screen. As expected, the visualiser would move in the rhythm of the music. The

visualiser’s motion would be volume sensitive meaning that if you turn down the

volume, the visualiser will also tone down its motion and vice-versa. The visualiser

should start moving once a track is played and should stop moving when a track is

paused.

10. ‘Back to list’ button

The ‘Back to list’ button would take the user to the screen he came from. For ex: if a

user clicked on a track on the ‘All songs’ screen and lands on the ‘Now playing’

screen, the ‘Back to list’ button should take the user back to the ‘All songs’ screen.

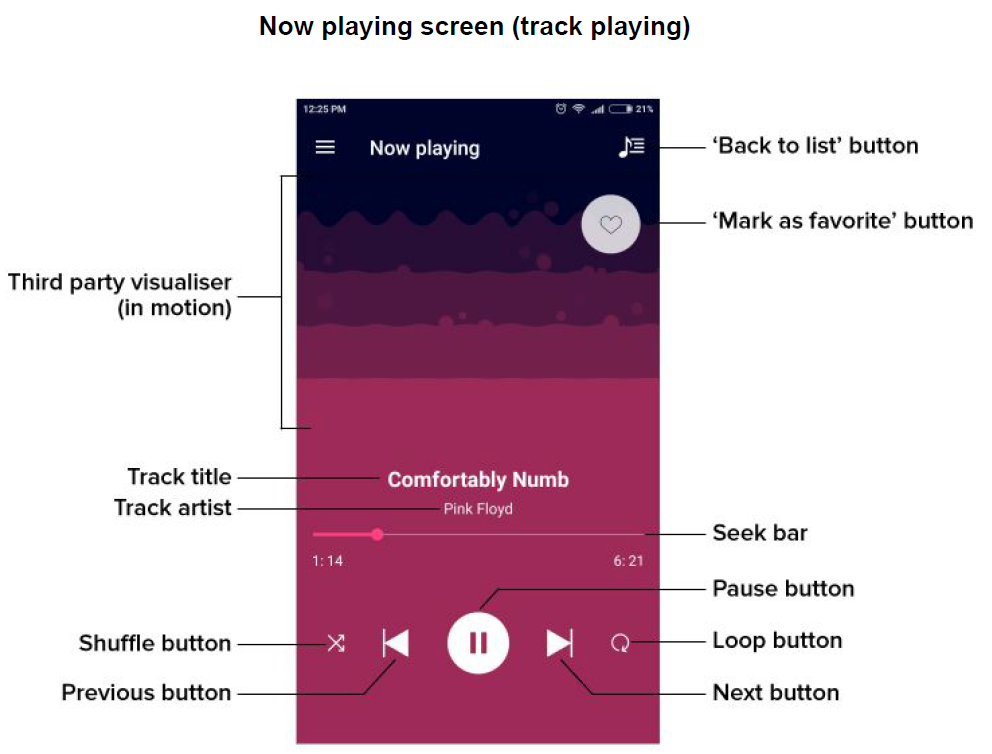
This button would be placed in the right side of the header.

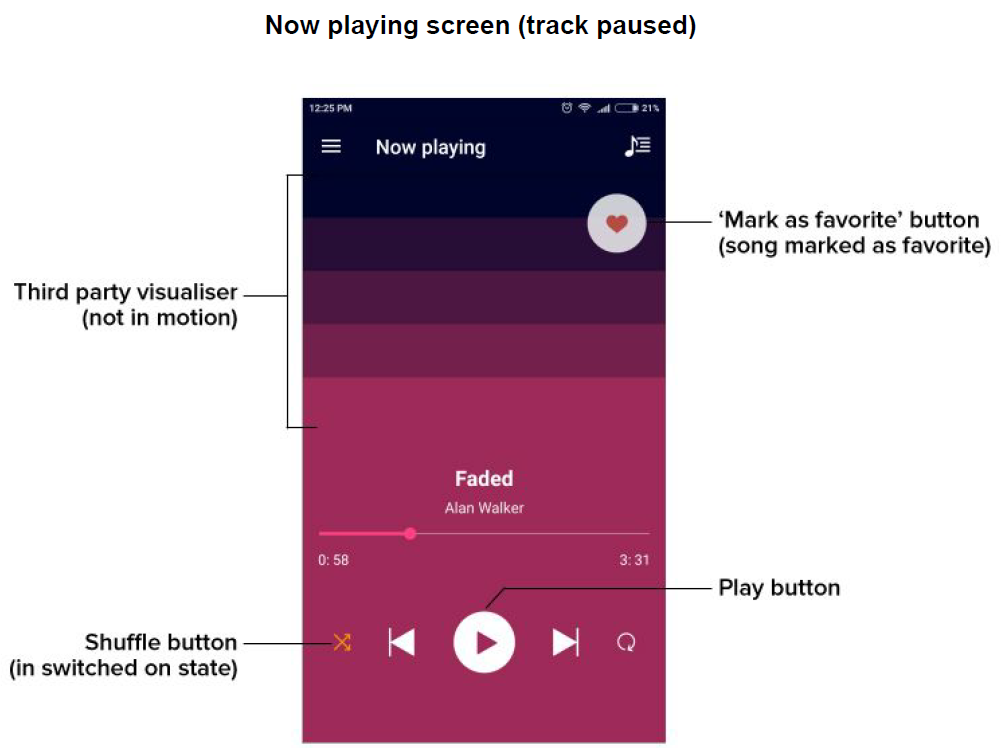
11. Shake to change song

As the name suggests, this feature would allow the user to change the track just by

shaking his/her phone. We would use the accelerometer on the mobile phones to

make this feature work.





### 4.3.8. Now playing bar

The ‘All songs’ screen and the ‘Favorites’ screen will have a ‘Now playing’ bar at the bottom

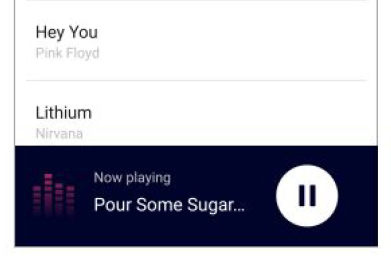
if the app is playing a track. This bar would display the title of the track playing and play or

pause feature. It would also have an image of some equaliser bars in the left side and a

static text above the track title name saying “Now playing”. If the track title is too long to be

displayed in one line, an ellipsis would appear in the track title. The Play/Pause button would

work same as it did on the ‘Now playing’ screen.



### 4.3.9. Notification

With the background play feature in the app, the user can play tracks in the background, but

he would have no way to figure out which app is playing the track (except for opening and

checking all the recent apps). So the app needs to have a notification indicating that this

particular app is playing music in the background.

The notification would say “A track is playing in the background”. This notification would pop

up in the notifications drawer every time the app plays any track in the background. Ensure

that this notification shouldn’t in the notifications drawer when the app is not playing a track

in the background.

Also, the notification would be there only when the app is playing a track in the background

and it won’t be there when the user is interacting with the app (i.e. if he/she is on any of the

screens of the app).

