Introduction to Android Development

⇒ About Android :-

→ Android History:

Android was created in 2003 and was intended as an OS for digital cameras.

In 2005 Google bought Android.inc. and in 2007 Android OS was announced.

HTC dream was the first Android phone launched in 2008.

→ API Levels: [API Level is an value, mostly used by developers to address an OS version.]

Code Name	Version Number	API Level
Jelly Bean	4.1	16
	4.2	17
	4.3	18
KitKat	4.4	19
	4.4W	20
Lollipop	5.0	21
	5.1	22
Marshmallow	6.0	23
Nougat	7.0	24
	7.1	25
Oreo	8.0	26
	8.1	27
Pie	9	28
Android 10	10	29
Android 11	11	30

⇒ Why Android ?:-

1. Android has a huge ecosystem. So, Android is not only available in Phones and Tablets, but it is also available in Watches, TVs, Automobiles etc. and also using Android we can create smart created IOT devices. So, Android gives us a vast scope of employability, also creativity.

2. Android is Open Source. OS. It's managed under Android Open Source Project (AOSP) led by Google. Developers from all over the world contribute to this open source.

3. Market share of Android is nearly 74%. & for IOS, its nearly 22% as of 2021.

4. Android has a vast Developer Community.

⇒ Notes :-

① Android is now Kotlin first. Kotlin was released on 15th Feb, 2016. At Google I/O 2017, Google announced first-class support for Kotlin on Android. At Google I/O 2019, Google announced Kotlin to be the official language of Android.

② Android Jetpack is a suite of libraries, tools, and guidance to help developers write high-quality apps with ease. These components help one follow the best practices, free them from writing boiler-plate code, and simplify complex tasks, so one can focus on the code they care about. Jetpack comprises the "androidx.*" package libraries, unbundled from the platform APIs. This means that it offers backward compatibility i.e, the code is compatible with the previous versions as well, and is updated more frequently than the Android Platform,

making sure you always have ~~the~~ access to the latest and greatest versions of the Jetpack components.

These Jetpack components cover lot of aspects of the Android APPs. such as, Foundation, Architecture, Behavior, and UI.

Type of Mobile Apps

⇒ Mobile Platforms :-

→ Mobile Platforms :

There are 2 major platforms / OS for mobile devices :

1. Android by Google
2. iOS by Apple.

Android runs on various devices - Pixel, Samsung, LG, Sony etc.

iOS runs only on apple devices.

→ Android App Development stack :

- Programming Language : Kotlin / Java
- Integrated Development Environment (IDE) : Android studio
- Software Development Kit (SDK) : Android SDK.

→ iOS App Development stack :

- Programming Language : Swift / Objective - C
- IDE : Xcode
- SDK : iOS SDK.

→ Custom skins designed by companies on top of Android as their core :

1. OxygenOS (~~OnePlus~~) (Samsung)
2. One UI (OnePlus)

these are only custom skins and not an OS. The core platform for these are Android only.

⇒ Native Apps :-

They are specifically targeted to a particular mobile platform in a specific language.

Android apps - Kotlin / Java - Android studio IDE

iOS apps - Swift / Objective-C - Xcode IDE

The development framework provides maximum feature access to the mobile platform and high performance.

Advantages: Full feature access, Performance.

Disadvantage: High development cost, higher learning curve.

* Native apps can run as standalone application.

⇒ Hybrid Apps :-

Hybrid apps are developed using web technologies like HTML, JavaScript, CSS etc.) and are run inside a native wrapper which uses the WebView object of mobile devices. ~~these~~ this let them run the same codebase on multiple platforms.

Tools : Ionic, Apache Cordova, Adobe PhoneGap etc.

Advantage : Single codebase, Lower development cost, Faster development.

Disadvantage : Performance, Feature accessibility.

⇒ Cross-platform Apps :-

Cross-platform apps like Hybrid apps take advantage of single codebase. The major difference is the way they render the layout using native components. This helps them provide near native UX.

Tools : Xamarin (C#), React Native (Javascript), Flutter (Dart) etc.

Advantage : Single codebase, Lower development cost, Faster development, Native UI look and feel, choice of development language.

Disadvantage : Performance, Feature accessibility.

⇒ Progressive Web Apps:-

Progressive Web Apps provide an installable, app-like experience on desktop and mobile that are built directly via the web. In simple terms they are like websites running in a web browser but unlike Hybrid apps they can work offline.

Tools: Angular, React etc.

Advantage: Single codebase, Lower development cost, Faster development, No need to install from play store/app store., responsive, easily shareable, secure.

Disadvantage: Performance, Feature Accessibility.

* For PWAs all users are on the same version of the app.

⇒ SDK :-

⊛ Different Components from SDK Manager :-

1. SDK Tools : These are some tools that are required for doing Android development. This includes tools like SDK manger & Virtual device manager.

2. SDK Platform Tools : Important tools like adb that are required for debugging a lot of issues related to android devices and software.

3. SDK Build Tools : These tools are required to build, run and test android applications.

⊛ In the SDK manager you will see different installable for different Android API levels. Here are the different packages that you find inside each API Level :-

1. SDK Platform : This is the main package that you need to test your app against a particular API level.

2. System Images : You will find multiple system images for each api. These images

are used to emulate a specific API level on the emulator machine. You would need to install this along with the platform if you want to test your code using an emulator for a particular android api level.

3. Sources for SDK: This has the source code for the particular SDK.

4. Google API's: You will need to install these in case you want your emulator to have google apps like Google maps etc.

5. Samples: Includes sample code files showing example use of different api's from that API level.

(*) Extras :-

1. Android Support Library: This includes some additional API's and it also provides backward compatibility by providing support for newer api's in older android releases.

2. Google Play Services: Allows your app to access Google maps API, Google Wallet etc.

3. HAXM (Hardware Accelerated Execution Manager):

It is an engine created by Intel to accelerate the Android emulation. This tool leads to a considerable improvement in the speed of the emulator and it is the prime factor why we should prefer intel based system images.

4. Android API specific Packages:

In the SDK manager you will see different installable for different Android API levels. Here are the different packages that you find inside each API Level.

Introduction to Kotlin

⇒ Introduction :-

Kotlin is a language developed by Jetbrains. It has been around since 2011 but first stable release was announced in 2016. Kotlin is now preferred by Google for Android App development.

→ Kotlin is Fully Compatible with Java :

Kotlin code compiles to Java Bytecode. Kotlin is interoperable with Java.

⊙ Kotlin is a Null safe object Oriented Programming language.

⇒ Why Kotlin ? :-

→ Mutability :

Good programming practice says that a variable should change its value less often and preferably not at all.

In Java we have an optional "final" but in Kotlin its much more robust. We need to clarify the mutability of a variable in the declaration itself.

Mutability plays a very important part in multithreaded code, where different codes want to change a variable.

→ Null Safety : [A variable's value is "null" means that it's not pointing to any object]

One of the most common source of app crashes during runtime is Null Pointer Exception. This happens when we try to ~~declare~~ access a method or property of a variable which is null.

In Kotlin we have to explicitly declare if a variable can be null or not.

⇒ Basic Data Types :—

→ Basic Data Types:

1. Double – 64 bit
2. Float – 32 bit
3. Long – 64 bit
4. Int – 32 bit
5. Short – 16 bit
6. Byte – 8 bit
7. Boolean – 8 bit
8. Char – 16 bit
9. String
10. Array

* Primitive data types exist within Kotlin, but they only work under the hood and hence cannot be used to create variable. So, we cannot directly declare a variable of primitive type in Kotlin.

→ Everything is an Object:

In Java we have primitive data types and their counterpart wrapper class. For example: int and Integer. We can't call functions on an "int" variable.

In Kotlin we only have objects. Although primitive types are used but internally.

* Primitive data types can hold the value directly instead of pointing to an object. That makes it very fast to work with.
* Under the hood, the objects act as primitive data types, so performance in Kotlin. So, performance is not an issue.

⇒ Variables - Mutability :-

[val name : String = "John"]

"val" keyword declares the variable.
"name" is the name of the variable.
":" signifies what data type is associated with the variable.
"String" is the data type associated with the variable.
"John" is the string value assigned to the variable.

Now, Kotlin compilers are smart enough to identify the data type that is being assigned to the variables. Which is in this case, a 'String'. So, we don't need to explicitly define the data type here. We can write the above statement as,

[val name = "John"]

So, Kotlin can infer the data type of the variable from it's value.

Any variable that is declared using the "val" keyword, once assigned, cannot be reassigned. This is how Kotlin protects mutability of a variable.

In ~~there~~ Kotlin, there are two ways to declare a variable -

① val → immutable.

② var → mutable

This is how Kotlin maintains mutability in its code. A good practice is to use "val" as much as possible as in a good code, we should change the value of a variable as less as possible.

⇒ Variables - Null Safety :-

If a variable holds a null value, i.e., it's pointing to no object in memory heap, for such a variable if we try to access its function or property, we get a ~~null~~ NullPointerException.

This NullPointerException is one of the biggest causes of app crashes during runtime. Kotlin tries to avoid it by implementing Null Safety.

In Kotlin, there are two kinds of types -

① Non Nullable : String, Int

② Nullable : String?, Int?

var name: String = "John" ← It is valid as name is of type "String" and it's Non Nullable.

var name: String = null ← It is invalid as name is of type "String" and it's Non Nullable. So, it can't have "null" as its value.

var name: String? = null ← It is valid, as name is of type "String?" and its Nullable. So, it can have only "null" value at the time of declaration.

We use "var" instead of "val" for Nullable variables, because so that we can assign a proper value to the variable in our code, as Nullable variables are assigned null at the time of declaration.

Kotlin implements Null Safety by not letting us access a nullable type without checking it for null. So, if there is a nullable type, we can't access the variable directly. Because in runtime if the nullable variable stays null, we will get a NullPointerException. So, we first need to check whether a variable is null or not, and then we can access it.

For null checks, Kotlin has a safe call operator (?.). For example:-

```
//Declaration
var name: String? = null

//safe call operator ?.
print(name?.length)
```

In the above code, name is a nullable type. And when printing the value of name's length, by accessing the 'length' property of the variable, we are putting the safe call operator (?.) at the end of "name", so that NullPointerException is not thrown. What (?.) does is that it checks if the variable is "null". If it's null then it returns a "null" value without accessing it value or property and if it's not null then it executes the code.

For better taking care of the NullPointerException, we can use the Elvis Operator (?:) along with the Null Safety check Operator (?.). For example:-

```
// Declaration
var name : String? = null

// Safe Call Operator ?. & Elvis operator ?:
print (name?.length ?: 0)
```

In the above code, name is a nullable type. If ~~it~~ ~~is~~ "name" is null then due to Safe Call Operator, "name?.length" will return null and then due to Elvis Operator, as the operand "name?.length" returns null; "0" will be printed. If "name?.length" did not return null, then their original value will be printed. So, Elvis operator evaluates the operand for null.

There is another operator in context to null. It is called "~~not~~ Null assertion operator(!!.)". In Kotlin only safe(?.) or not-null asserted(!!.) calls are allowed on a nullable receiver of type String. For example :-

```
var firstName: String? = "John"
var lastName: String? = null

if (firstName != null){
    //Not-Null assertion operator (!!.)
    println("Name ${firstName.length}
                ${lastName!!.length}")
}
```

What Not-Null assertion operator (!!.) does is that it tells the Kotlin compiler not to check that specific nullable variable for null as we will explicitly handle it ourselves in the code. And if we did not handle it properly, it will throw ~~or error~~ NullPointerException.

→ Notes:

① So, Nullables can only be accessed through safe call operator (?.) or not-null assertion operator (!!.) and elvis operator (?:) is the ternary operator for null.

② Nullable is ~~for~~ only for compile time. There is no nullable during runtime. There is only String at runtime. So, implementation of Nullable has no overhead during runtime.

⇒ Type Conversion / casting :—

→ Type Conversion :

```
val a: Int = 10
val b: Long = a
```

In the above code, "a" is a variable of type Int and have a value "10". We are trying to assign it to a variable "b" which is of type "Long". In other programming languages we can implicitly convert a Integer data type to a Long data type. But this is not the case for Kotlin. Kotlin does not support implicit type conversion.

We need to explicitly convert "a" to a Long data type to assign it to "b" which is of Long data type. For that, the code should be —

```
val a: Int = 10
val b: Long = a.toLong()
```

Rather than Typecasting, it is Type Conversion.

→ Type Casting:

There is a data type in Kotlin, which is called "Any". "Any" is a super class of all the classes. "Any" is the base class. (It's like the "Object" class of Kotlin. Just like, "Object" class is the super class of all classes in Java, "Any" is the super class of all classes in Kotlin.). So, any variable having data type "Any" can hold any type of object. (As everything in Kotlin is an Object & there are no primitives).

We check the type of a variable using the "is" operator. For example:-

```
val name: Any = "John"
println(name is String)
```

The above code will print "true" as the value assigned to "name" is of type "String". But "name" itself is of type "Any" that can hold any type of object.

To typecast a variable of "Any" type, use the keyword to the type of the value it holds, we use "as" keyword. For example -

```
val name: Any = "John"
val name1: String = name as String
```

In the above code, "name" is of type "Any" having value of type "String". But in the second line, we convert "name" to a "String" type and assign it to "name1" which is of type "String". Using "as" does not permanently convert "name" to type "String". It just gives us a converted version to assign to "name1", "name" is still of type "Any".

In the above example, we could easily cast "name" to type "String" as it was

But as, "name" is assigned a "String" value, it is inferred as type "String" at compile

time. That's why all three println() statements given below will print "true".

```
val name: Any = "John"
println(name is Any) //true
println(name is String) //true
println(name as String is String) //true
```

Now, we could easily cast "name" to type "String", as it was inferred as "String" type by the compiler, by using the unsafe cast operator "as". "as" does not check whether the cast is possible or not. That's why the code below will throw an error —

```
val x: Int = 7
val y: Long = x as Long //error
```

To cast it successfully without errors/exceptions, we need to use the safe cast operator (as?). So, the correct code should be —

```
val x: Int = 7
val z: Long? = x as? Long
println(z)
```

This will not throw an error. Rather, it will assign x to z if the type cast was successful or it will assign null to z if the type cast was unsuccessful. That's why we have made z of type nullable, so that it can store null if the cast fails. In this case, the cast will fail and println() will print null.

⇒ Functions :—

→ Function Basics:

Functions are through which we implement features of classes and objects. Syntax of a function declaration in Kotlin is given below —

```
fun functionName(parameters): returnType {
    //Code
}
```

"fun" is the keyword that is used to declare a function. An example of a function that takes two Integers as parameter and returns a Integer value that is the sum of the two numbers —

```
fun addNumbers(x: Int, y: Int): Int {
    return x+y
}
```

In Kotlin, it is mandatory that a function returns something. If the function is not returning anything, then it is by default returning a "Unit" object. We do not need to explicitly define it. But if we want,

we can write "Unit" in place of returnType.

Below is an example of a function that takes a String as parameter and prints it. But it does not return anything, So, it will by default return a "Unit" Object.

```
fun printUserName(name: String): Unit {
    println(name)
}
```

we don't need to explicitly define it.

→ Default Argument:

In Kotlin we can assign default arguments to the function parameters. If a particular argument for a function is not passed and if that parameter has a default argument then that argument will be used rather than throwing an error. For example-

```
fun printUserName(firstName: String,
                  lastName: String = "Doe") {
    println("$firstName $lastName")
}
```

"Doe" is the default argument for the parameter "lastName". If an argument was not passed for "lastName" at the time of function invokation, then "Doe" will be used.

→ Named Arguments:

In Kotlin we can pass arguments as named arguments. That means we can specifically assign a argument to a specific parameter. When we use named arguments, it is not necessary to keep the order of the arguments passed. For example —

```
fun main() {
    printDetails( width = 10, length = 5)
}
fun printDetails (length : Int, width : Int) {
    println(length) // 5
    println(width) // 10
}
```

→ Single Expression :-

There is a concept called Single Expression in Kotlin. ~~It is called~~ If a function has only a single expression within it, then we can write it in the form of Single Expression. For example —

```
fun addNumbers(x: Int, y: Int): Int {
    return x+y
}
```

↓↓ Single Expression

~~fun~~

```
fun addNumbers(x: Int, y: Int) = x+y
```

→ Note :

① In Kotlin, "Void" is a non-nullable return type. So, a function with the return type "Void" cannot return a null value.

② "Unit" object is actually defined as "Kotlin.Unit".

⇒ String & Array :-

In Kotlin, String & Array are basic data types.

→ String Concatenation :

We can concatenate multiple strings with (+) operator. For example -

"John" + "Doe" → "JohnDoe"

We can concatenate other data types with a string too, But we need to make sure that the concatenation is done in the right order. Otherwise it will give error. For example -

"House no." + 23 → "House no. 23"

23 + ", Block C" → This will give error.

So, for concatenating other data types with string, string needs to be always on the left side of the (+) operator.

In the above code ~~we~~ This is because, other data types do not know how to concatenate. Only strings know concatenation.

→ String Template :-

Another feature that string provides is string Template. We can create a string template using the ($) operator. Using the ($) operator we can embed any variable's value inside a string.

For example -

```
val age: Int = 23
val name: String = "Deepbaran"
val details: String = "$name $age"
println(details) // Deepbaran 23
println("${name.length}") // 9
```

If we want to embed a property of the variable in the string, then we will need to use (${}) operator.

→ Array:-

Arrays are zero indexed in Kotlin. (Arrays in Kotlin can have multiple data types.)

Creating array in Kotlin:

```
val numbers = arrayOf(1, 2, 3, 4)
val numbers1 = arrayOf<Int>(1, 2, 3, 4)
val details = arrayOf("Deepbaran", 23)
```
↗ This works but we should refrain from doing it.

```
val numbers2 = intArrayOf(1, 2, 3, 4)
```

Getting data from array using index:

```
numbers.get(1) // 2
numbers[2] // 3
```
[[ ] → Index operator

Changing values of array using index:

```
numbers.set(3, 5)
numbers[1] = 22
```

Printing the array:

```
println(numbers.joinToString(", ")) // 1, 22, 3, 5
```
↳ Joins the array elements to create a string by using a separator.

Creating a array of specific size and fill it up with a particular value:

```
val numbersValue = Array(3){5} // [5, 5, 5]
```
↑ This array is created.

⊛ This is how we can initialize an array of a specific size.

Size of an array:

```
numbers.size // 4
```

→ Note:

① Strings are immutable in Kotlin.

② Accessing string indexes:

```
str.get(index)
str[index]
```

③ Multidimensional Array:

```
val arr1 = arrayOf(1,2,3)
val arr2 = arrayOf(4,5,6)
val arr = arrayOf(arr1, arr2)

val array = Array(2){Array(3){0}}
```

```
[[0 0 0]
 [0 0 0]]
```

④ Creating an empty string object to fill later:

```
var s = String()
```

⑤ Creating a String object:

```
val s = String("Hello")
```

⑥ substring:

```
val s = "Deepbaran"
println(s.subSequence(0,2)) // De
```

⑦ String Comparison:

~~String~~

```
val s1 = "Deep"
val s2 = "Deep"
val s3 = "baran"

s1.equals(s2) // true
s1.equals(s3) // false

s1.compareTo(s2) // 0 ← zero means, they are equal.
s1.compareTo(s3) // Any numerical value other than zero(0)
```

⑧ As Kotlin runs on JVM, so it uses the same garbage Collector as Java or any other JVM based language. So, Kotlin uses Garbage collection.

⑨ Kotlin stores only references to our objects into stack memory. The objects are actually stored in the heap memory.

⑩ In Kotlin, everything is an object (reference type, not primitive type). We don't find primitive types, like the ones we can use in Java. This reduces code complexity. We can call methods and properties on any variable. For example, this is how we can convert the Int variable to a char. Usually (wherever it is possible), under the hood types such as Int, Long, or Char are optimized (stored as primitive types) but we can still call methods on them as on any other objects.

⇒ Conditionals :-

→ if/else :

```
var discount = 0
val price = 85
if (price > 50) {
    discount = 10
else {
    discount = 3
}
println(discount) // 10
```

* In Kotlin, if something is an expression, then we can directly assign it to a variable.

↓ In Kotlin "if" is a expression.

```
val price = 85
var discount = if (price > 50) 10 else 3
println(discount) // 10
```

→ When: [It is similar to switch-case]

```
val rating = 4
val result = when (rating) {
    5 → "High"
    3,4 → "Good"
    1,2 → "Poor"
    else → {
        println("No Rating")
        "Zero"
    }
}
println(result) // Good
```

* "When" is a expression
* "→" is a Kotlin Lambda Higher Order Function.

⇒ Loops :-

→ For Loop:

```
val names = arrayOf("Jack", "John", "Frank")
// Iterate through objects
for(name in names){
    println(name)
}

//Iterate through indices
    for (i in names.indices) {
        println(names[i])
    }
```

→ Range:

```
for(i in 1..5) {
    print(i) //12345
}
```

```
if (4 !in 5..10) {
    print("Not There") //Not There
}
```

```
if (6 in 5..10) {
    print("There") // There
}
```

```
for(i in 1 until 5){
    print(i) //1234
}
```

```
for (i in 1..5 step 2) {
    print(i) //135
}
```

```
for(i in 5 downTo 1) {
    print(i) // 54321
}
```

→ While:

```
var i = 0
while (i < 5) {
    println ("Hello, World!")
    i++
}
```

```
var name: String?
do {
    name = "John"
    println (name)
} while (name == null)
```

→ Note:

① We cannot put ~~uninit~~ uninitialized variables in print() / println()

② do-while loop runs at least one time and it runs one loop more than a while loop.

③ 1 does not mean "true" and 0 does not mean "false" just like Java.

④ There is another loop called "forEach()". It works with collections.

⇒ Collections :—

Collections is a group of Objects which acts as a single unit and gives us easy access to these Objects, which we call elements.

There are three types of collections in Kotlin —

1. List
2. Set
3. Map

→ List :

List is a index based collection which is similar to array but more powerful.

There are two types of lists in Kotlin —

1. Mutable List
2. Immutable List

Creating a Immutable List :

```
val names1 = listOf("John", "Mark", "Finn")
```

Creating a Mutable List :

```
val names2 = mutableListOf("John", "Mark", "Finn")
```

Size of a List :

```
names2.size
```

Getting an item using index :

```
names2.get(index)
names2[index] //using index operator.
```

Getting index of an element:

```
names2.indexOf(element) // -1 will be returned if the element is not present
```

Printing the list:

```
println(names2)
```

Adding elements to the list:

```
names2.add(element)
```

Removing element from the list:

~~names2.remove(index)~~

```
names2.removeAt(index)
names2.remove(element)
```

Replacing elements in the list:

```
names2.set(index, element)
names2[index] = element
```