Android App Development using Kotlin

{ AADK }

Kotlin

for Android App Development

6th Semester CSE/CSE-AIML/CSE-DS

Department of CSE
GIET University Gunupur

Ranjit Patnaik
Asst. Professor

February 10, 2025

Table of Content

Syllabus
Android Operating System
Android Architecture
Android Core Building Blocks
What is Kotlin Programming?
Key Features of Kotlin
Applications of Kotlin language
First program in Kotlin
Kotlin Data Types
Kotlin Variables
Scope of a variable
Kotlin Operators
Relational Operators
Kotlin - Control Flow
Kotlin while loop
Kotlin for loop
Kotlin Function
Kotlin user-defined function
Kotlin Default arguments
Setters and Getters
Lambda functions
Higher Order Function in Kotlin
Kotlin Type Conversion

Table of Content II

Object-Oriented Programming

Kotlin Collections

Kotlin generics

Kotlin Exception Handling

Application Structure

Application Architecture

Table of Content III

Syllabus

Kotlin

for Android App Development

UNIT: I (8 Hours)

- Introduction to Kotlin: Kotlin Architecture, Basic syntax, Kotlin Comments, Kotlin Keywords, Defining variable, Use of variables, Data types- (Booleans, Strings, Arrays, Ranges), Operators,
- Control Flow, if. . . else expression, for loops, While loops, Break and Continue, functions in Kotlin, user defining functions. Kotlin Collections (Lists, Sets, Maps),
- Kotlin objects and Classes, Constructors, Inheritance, Interface, Visibility Control, Extension, Data Classes, Sealed Class, Generics, Delegation, Destruction, Exception Handling.

Kotlin

for Android App Development

UNIT:2 (15 Hours)

- Introduction to Android: Introduction to Android Operating System, Android Architecture, Core Building Blocks, Android Emulator, Application Structure, Setting up development environment, Dalvik Virtual Machine and .apk file extension, AndroidManifest.xml, uses-permission and uses-sdk, Resources and R.java, Assets, Layouts and Drawable Resources,
- Fundamentals: Basic Building blocks - Activities, Services, Broadcast, Receivers and Content providers, UI Components - Views and notifications, Components for communication -Intents and Intent Filters, Android API levels (versions and version names), User Interface Architecture: Application context, intents, Activity life cycle, Services,

Kotlin

for Android App Development

- Android widgets, UI widgets, Working with Buttons, Toast, Custom Toast, Toggle Button, Check Button, Custom Check Button, Radio Button, Dynamic Radio button, Alert Dialog, Android Views: Text view, Autocomplete Text view, List View, Custom List View, Rating Bar, WebView Seek Bar, Date Picker, Time Picker, Analog and Digital, Progress Bar,

Kotlin

for Android App Development

- Android Layouts: Linear Layout, Relative Layout, Table Layout, Table Layout with Frame Layout, Search View, Search View on Toolbar, Edit Text with Text Watcher. Switchers: Switch, Dynamic Switch, Text Switcher, Dynamic Text Switcher, Image Switcher, Dynamic Image Switcher. Spinner: Spinner, Dynamic Spinner, Dynamic TextClock, Chronometer, Dynamic Chronometer, Notifications, Android Toast, Dynamic RadioGroup, Android Slide Up/Down in Kotlin, Scroller: Vertical Scroll, View Horizontal Scroll View, Image Switcher, Image Slider, View Stub, XML Parsing using DOM Parser, XML Parsing using SAX Parser,
- Kotlin Android XMLPullParser

Kotlin

for Android App Development

UNIT: III (10 Hours)

- Menu : Option menu, Context menu, Sub menu, menu from xml, menu via code, Examples. Navigation: Create a fragment, Define navigation paths, Start an external activity,
- Kotlin Regex: Regular Expressions, Kotlin Annotations,
- Adapters, Android Recycler View, Testing Android applications, Publishing Android application, Using Android preferences, Managing Application resources in a hierarchy, working with different types of resources.

Kotlin

for Android App Development

UNIT:4 (10 Hours)

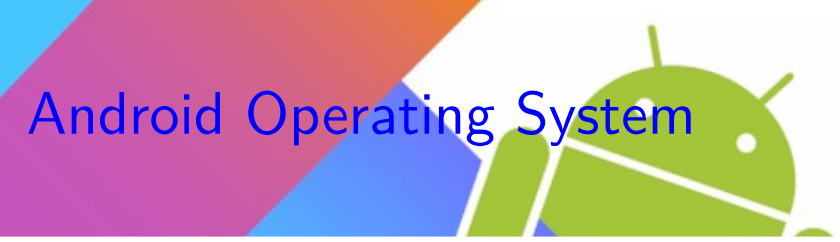
- Data Storage: Shared Preferences, Android File System, Internal storage, External storage, SQLite
- Firebase. Common Android APIs: Using Android Data and Storage APIs etc.

Introduction

Kotlin

for Android App Development

- Android is an operating system and programming platform developed by Google for mobile phones and other mobile devices, such as tablets, smart phones and wearable devises etc. Android includes a software development kit (SDK) that helps you write original code and assemble software modules to create apps for Android users.
- The android app development can be implemented through using programming languages like Java and Kotlin.



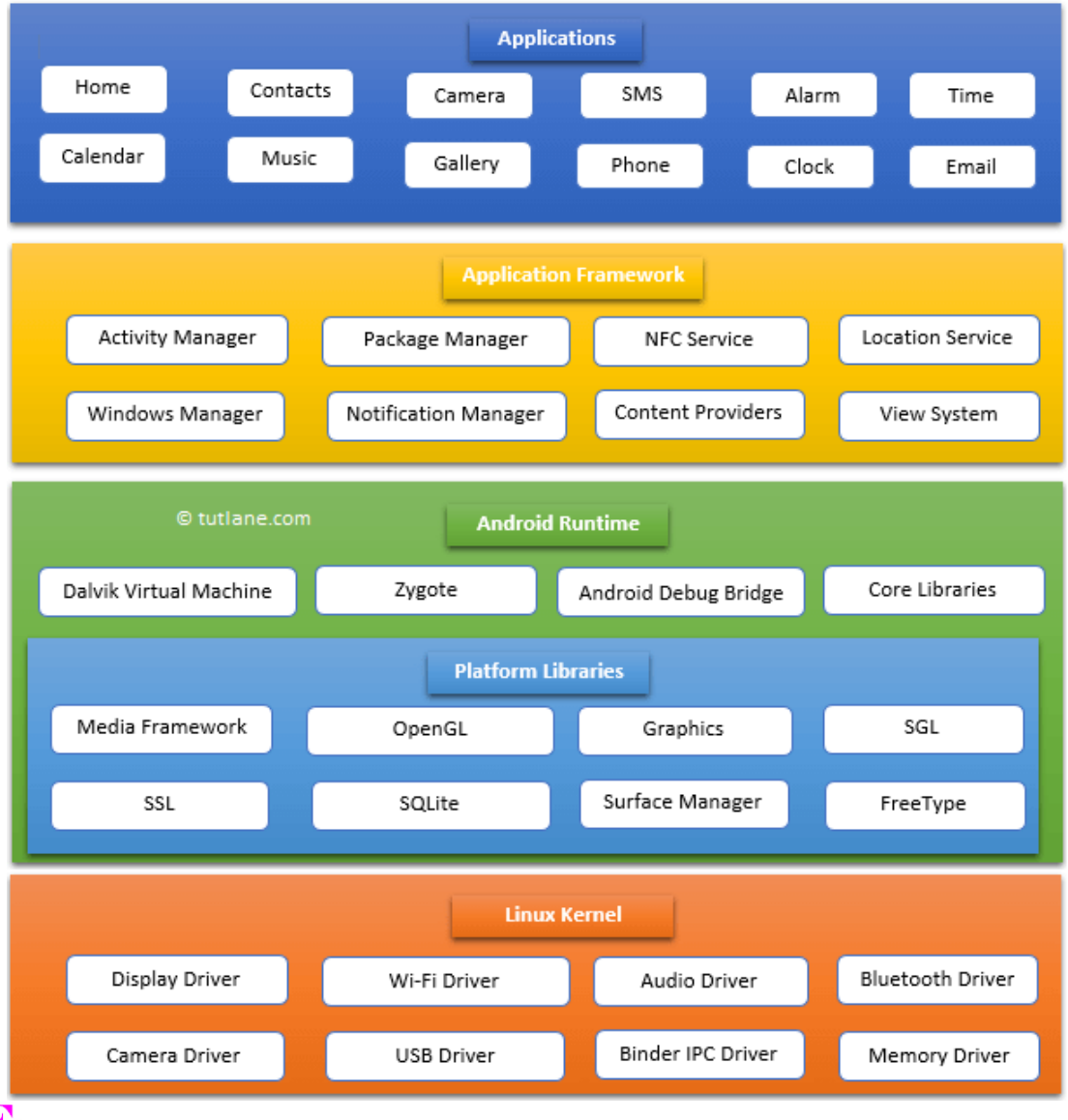
Android Operating System

Android is described as a mobile operating system, initially developed by Android Inc. Android was sold to Google in 2005. Android is based on a modified Linux 2.6 kernel. Google, as well as other members of the Open Handset Alliance (OHA) collaborated on Android (design, development, distribution). Currently, the Android Open Source Project (AOSP) is governing the Android maintenance and development cycle

- An open platform for mobile development
- A hardware reference design for mobile devices
- A system powered by a modified Linux 2.6 kernel
- A run time environment
- An application and user interface (UI) framework

Kotlin
for Android App Development

Figure 1 outlines the current (layered) Android Architecture. The modified Linux kernel operates as the HAL, and provides device driver, memory management, process management, as well as networking functionalities, respectively. The library layer is interfaced through Java (which deviates from the traditional Linux design). It is in this layer that the Android specific libc(Bionic) is located. The surface manager handles the user interface (UI) windows. The Android runtime layer holds the Dalvik Virtual Machine (DVM) and the core libraries (such as Java or IO). Most of the functionalities available in Android are provided via the core libraries.


Applications
Home
Contacts
Camera
SMS
Alarm
Time
Calendar
Music
Gallery
Phone
Clock
Email
Application Framework
Activity Manager
Package Manager
NFC Service
Location Service
Windows Manager
Notification Manager
Content Providers
View System
© tutlane.com
Android Runtime
Dalvik Virtual Machine
Zygote
Android Debug Bridge
Core Libraries
Platform Libraries
Media Framework
OpenGL
Graphics
SGL
SSL
SQLite
Surface Manager
FreeType
Linux Kernel
Display Driver
Wi-Fi Driver
Audio Driver
Bluetooth Driver
Camera Driver
USB Driver
Binder IPC Driver
Memory Driver

The application framework houses the API interface. In this layer, the activity manager governs the application life cycle. The content providers enable applications to either access data from other applications or to share their own data. The resource manager provides access to non-code resources (such as graphics), while the notification manager enables applications to display custom alerts. On top of the application framework are the built-in, as well as the user applications, respectively. It has to be pointed out that a user application can replace a built-in application, and that each Android application runs in its own process space, within its own DVM instance. Most of these major Android components are further discussed (in more detail) in the next few sections of this report.

Kotlin

for Android App Development

The main components of android architecture are following:-

1. Applications
2. Application Framework
3. Android Runtime
4. Platform Libraries
5. Linux Kernel

Kotlin

for Android App Development

Applications – Applications is the top layer of android architecture. The preinstalled applications like home, contacts, camera, gallery etc and third party applications downloaded from the play store like chat applications, games etc. will be installed on this layer only. It runs within the Android run time with the help of the classes and services provided by the application framework.

Application Framework provides several important classes which are used to create an Android application. It provides a generic abstraction for hardware access and also helps in managing the user interface with application resources. Generally, it provides the services with the help of which we can create a particular class and make that class helpful for the Applications creation.
It includes different types of services activity manager, notification manager, view system, package manager etc. which are helpful for the development of our application according to the prerequisite.

Application runtime

Android Runtime environment is one of the most important part of Android. It contains components like core libraries and the Dalvik virtual machine(DVM). Mainly, it provides the base for the application framework and powers our application with the help of the core libraries.
Like Java Virtual Machine (JVM), Dalvik Virtual Machine (DVM) is a register-based virtual machine and specially designed and optimized for android to ensure that a device can run multiple instances efficiently. It depends on the layer Linux kernel for threading and low-level memory management. The core libraries enable us to implement android applications using the standard JAVA or Kotlin programming languages.

Platform libraries

The Platform Libraries includes various C/C++ core libraries and Java based libraries such as Media, Graphics, Surface Manager, OpenGL etc. to provide a support for android development.
Media library provides support to play and record an audio and video formats. Surface manager responsible for managing access to the display subsystem. SGL and OpenGL both cross-language, cross-platform application program interface (API) are used for 2D and 3D computer graphics. SQLite provides database support and FreeType provides font support. Web-Kit This open source web browser engine provides all the functionality to display web content and to simplify page loading. SSL (Secure Sockets Layer) is security technology to establish an encrypted link between a web server and a web browser.

Kotlin

for Android App Development

Linux Kernel is heart of the android architecture. It manages all the available drivers such as display drivers, camera drivers, Bluetooth drivers, audio drivers, memory drivers, etc. which are required during the runtime.
The Linux Kernel will provide an abstraction layer between the device hardware and the other components of android architecture. It is responsible for management of memory, power, devices etc.

Security: The Linux kernel handles the security between the application and the system. Memory Management: It efficiently handles the memory management thereby providing the freedom to develop our apps. Process Management: It manages the process well, allocates resources to processes whenever they need them. Network Stack: It effectively handles the network communication. Driver Model: It ensures that the application works properly on the device and hardware manufacturers responsible for building their drivers into the Linux build.

An android component is simply a piece of code that has a well defined life cycle e.g. Activity, Receiver, Service etc.
The core building blocks or fundamental components of android are activities, views, intents, services, content providers, fragments and AndroidManifest.xml.

- Activity:
  Being similar to a Frame in AWT, an activity as a class represents a single screen. It is a core building block, i.e., the fundamental component of android.
- View: The UI element including button, label, text field, etc, and anything that one can see is a view.

Kotlin

for Android App Development

Intent: Along with invoking the components, the Intent is used for many purposes including:

- Start the service
- Launch an activity
- Display a web page
- Display a list of contacts
- Broadcast a message
- Dial a phone call etc.

- Service: Being a background process the service can run for a long time. The local and remote are the two types of services. Within the application, the local service is accessed. While from other applications running on the same device the remote service can be accessed remotely.
- Content Provider: To share the data between the applications, the Content Providers are used.
- Fragment: Being a part of an activity, one or more fragments can be displayed on the screen at the same time by the activity.

Kotlin

for Android App Development

AndroidManifest.xml: Information about activities, content providers, permissions, etc is in the AndroidManifest.xml which is like the web.xml file in Java EE.
Android Virtual Device (AVD): To test an android application without using a mobile or a tablet, the Android Virtual Device or AVD is used. To emulate different types of real devices, an AVD can be created in different configurations.

Android Versions, Codename and API

Version	Code name	API Level
1.5	Cupcake	3
1.6	Donut	4
2.1	Eclair	7
2.2	Froyo	8
2.3	Gingerbread	9 and 10
3.1 and 3.3	Honeycomb	12 and 13
4.0	Ice Cream Sandwitch	15
4.1, 4.2 and 4.3	Jelly Bean	16, 17 and 18
4.4	KitKat	19
5.0	Lollipop	21
6.0	Marshmallow	23
7.0	Nougat	24-25
8.0	Oreo	26-27
9.0	Pie	27
10.0	Android Q	29
11.0	Android 11	30

What is Kotlin Programming?

- Kotlin is a statically typed, general-purpose programming language developed by JetBrains, that has built world-class IDEs like IntelliJ IDEA, PhpStorm, Appcode, etc.
- It was first introduced by JetBrains in 2011 and a new language for the JVM. Kotlin is object-oriented language, and a "better language" than Java, but still be fully interoperable with Java code.

Key Features of Kotlin

- Statically typed
- Data Classes
- Concise
- Safe
- Interoperable with Java
- Functional and Object Oriented Capabilities
- Smart Cast
- Compilation time
- Tool- Friendly

Kotlin

for Android App Development

Statically typed

Statically typed is a programming language characteristic that means the type of every variable and expression is known at compile time. Although it is statically typed language, it does not require you to explicitly specify the type of every variable you declare.

Example:

var a = 10

var b = "String"

var c = 12.34567

Data Classes

In Kotlin, there are Data Classes which lead to auto-generation of boilerplate like equals, hashCode, toString, getters/setters and much more. Consider the following example/*

Kotlin Code */ data class Book(var title:String, var author:Author)

Kotlin

for Android App Development

Concise

It drastically reduces the extra code written in other object oriented programming languages.

Safe

It provides the safety from most annoying and irritating NullPointerExceptions by supporting nullability as part of its system. Every variable in Kotlin is non-null by default.

Strings="Hello KOTLIN" // Non-null

If we try to assign s null value then it gives compile time error.So,

s=null // compile time error

To assign null value to any string string it should be declared as nullable.

StringnullableStr?=null // compiles successfully

length() function also disabled on the nullable strings.

Interoperable with Java

Kotlin runs on Java Virtual Machine(JVM) so it is totally interoperable with java. We can easily access use java code from kotlin and kotlin code from java.

Kotlin

for Android App Development

Functional and Object Oriented Capabilities

Kotlin has rich set of many useful methods which includes higher-order functions, lambda expressions, operator overloading, lazy evaluation, operator overloading and much more.

Higher order function is a function which accepts function as a parameter or returns a function or can do both.

Kotlin

for Android App Development

Example of higher-order function

```
fun myFun(company: String,product: String, fn: (String,String) -> String): Unit {
        val result = fn(company,product)
        println(result)
}

fun main(args: Array){
        val fn:(String,String)->String={org,portal->"$org develops $portal"}
        myFun("JetBrains","Kotlin",fn)
}
```

Output:

JetBrains develops Kotlin

Smart Cast

It explicitly typecasts the immutable values and inserts the value in its safe cast automatically.

If we try to access a nullable type of String (String? = “BYE”) without safe cast it will generate a compile error.

```
        fun main(args: Array){
                var string: String? = "BYE"
                print(string.length)        // compile time error
        }
}
```

```
fun main(args: Array){
        var string: String? = "BYE"
        if(string != null) {                // smart cast
                print(string.length)
        }
}
```

Kotlin

for Android App Development

Compilation time and Tool-Friendly

Compilation time: It has higher performance and fast compilation time. Tool-Friendly: It has excellent tooling support. Any of the Java IDEs – IntelliJ IDEA, Eclipse and Android Studio can be used for Kotlin. We can also be run Kotlin program from command line.

Kotlin

for Android App Development

Advantages of Kotlin language

- Easy to learn - Basic is almost similar to java.If anybody worked in java then easily understand in no time.
- Kotlin is mult-platform - Kotlin is supported by all IDEs of java so you can write your program and execute them on any machine which supports JVM.
- It's much safer than Java.
- It allows using the Java frameworks and libraries in your new Kotlin projects by using advanced frameworks without any need to change the whole project in Java.
- Kotlin programming language, including the compiler, libraries and all the tooling is completely free and open source and available on github. Here is the link for Github https://github.com/JetBrains/kotlin

Applications of Kotlin language

- You can use Kotlin to build Android Application.
- Kotlin can also compile to JavaScript, and making it available for the frontend.
- It is also designed to work well for web development and server-side development.

First program in Kotlin

Hello, World! is the first basic program in any programming language. Let's write the first program in Kotlin programming language.

- Open your favorite editor notepad or notepad++ and create a file named firstapp.kt with the following code.

```
// Kotlin Hello World Program
fun main(args: Array<String>) {
        println("Hello, World!")
}
```

You can compile the program in command line compiler.
$ kotlinc firstapp.kt
Now Run the program to see the output in command line compiler.

$kotlin firstapp.kt

Hello, World!

Kotlin Data Types

The most fundamental data type in Kotlin is Primitive data type and all others are reference types like array and string. Java needs to use wrappers (java.lang.Integer) for primitive data types to behave like objects but Kotlin already has all data types as objects.
There are different data types in Kotlin-

1. Integer Data type
2. Floating-point Data Type
3. Boolean Data Type
4. Character Data Type

These data types contain the contain integer values.

Data Type	Bits	Min Value	Max Value
byte	8 bits	-128	127
short	16 bits	-32768	32767
int	32 bits	-2147483648	2147483647
long	64 bits	-9223372036854775808	9223372036854775807

Let's write a program to represent all the integer data types and their min and max value.

Kotlin

for Android App Development

```
// Kotlin code
fun main(args : Array<String>) {
        var myint = 35
        //add suffix L for long integer
        var mylong = 23L
        println("My integer ${myint}")
        println("My long integer ${mylong}")
        var b1: Byte = Byte.MIN_VALUE
        var b2: Byte = Byte.MAX_VALUE
        println("Smallest byte value: " +b1)
        println("Largest byte value: " +b2)
        var S1: Short = Short.MIN_VALUE
        var S2: Short = Short.MAX_VALUE
```

Kotlin

for Android App Development

```
        println("Smallest short value: " +S1)
        println("Largest short value: " +S2)
        var I1: Int = Int.MIN_VALUE
        var I2: Int = Int.MAX_VALUE
        println("Smallest integer value: " +I1)
        println("Largest integer value: " +I2)
        var L1: Long = Long.MIN_VALUE
        var L2: Long = Long.MAX_VALUE
        println("Smallest long integer value: " +L1)
        println("Largest long integer value: " +L2)
}
```

Output:
My integer 35
My long integer 23
Smallest byte value: -128
Largest byte value: 127
Smallest short value: -32768
Largest short value: 32767
Smallest integer value: -2147483648
Largest integer value: 2147483647
Smallest long integer value: -9223372036854775808
Largest long integer value: 9223372036854775807

Floating-Point Data Type

These data type used to store decimal value or fractional part.

Data Type	Bits	Min Value	Max Value
float	32 bits	$1.4012984643248170 7e^{-45}$	$3.4028234663852886 0e^{+38}$
double	64 bits	$4.9406564584124654 4e^{-324}$	$1.7976931348623157 0e^{+308}$

Let's write a program to represent both the floating-point data type and their min and max value.

```kotlin
// Kotlin code
fun main(args : Array<String>) {
        var myfloat = 54F                    // add suffix F for float
        println("My float value ${myfloat}")

        var F1: Float = Float.MIN_VALUE
        var F2: Float = Float.MAX_VALUE
        println("Smallest Float value: " +F1)
        println("Largest Float value: " + F2)

        var D1: Double = Double.MIN_VALUE
        var D2: Double = Double.MAX_VALUE
        println("Smallest Double value: " + D1)
        println("Largest Double value: " + D2)
}
```

Output:
My float value 54.0
Smallest Float value: 1.4E-45
Largest Float value: 3.4028235E38
Smallest Double value: 4.9E-324
Largest Double value: 1.79769313486231570E308

Boolean Data Type

Boolean data type represents only one bit of information either true or false. The Boolean type in Kotlin is the same as in Java. These operations disjunction (||) or conjunction (&&) can be performed on boolean types.

```
// Kotlin code
fun main(args : Array<String>){
        if (true is Boolean){
                print("Yes,true is a boolean value")
        }
}
```

Output:

Yes, true is a boolean value

Character Data Type

Character data type represents the small letters(a-z), Capital letters(A-Z), digits(0-9) and other symbols.

```kotlin
// Kotlin code
fun main(args : Array<String>){
        var alphabet: Char = 'C'
        println("C is a character : ${alphabet is Char}")
}
```

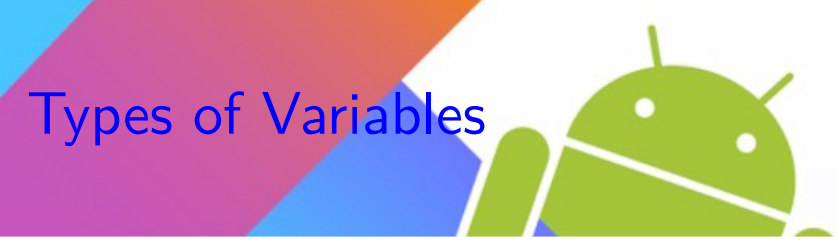
Output:

C is a character : true

Kotlin
for Android App Development

In Katlin, every variable should be declared before it's used. Without declaring a variable, an attempt to use the variable gives a syntax error. Declaration of the variable type also decides the kind of data you are allowed to store in the memory location. In case of local variables, the type of variable can be inferred from the initialized value.

```
var rollno = 21
var name= "Aryamaan"
println(rollno)
println(name)
```



In Katlin, variables are declared using two types

1. Immutable using val keyword
2. Mutable using var keyword

Immutable Variables Immutable is also called read-only variables. Hence, we can not change the value of the variable declared using val keyword.

```
val myName =  "Gaurav"
myName =  "Praveen"    // compile  time  error

// It gives  error  Kotlin  Val  cannot  be  reassigned
```

Note: Immutable variable is not a constant because it can be initialized with the value of a variable. It means the value of immutable variable doesn't need to be known at compile-time, and if it is declared inside a construct that is called repeatedly, it can take on a different value on each function call.

Kotlin

for Android App Development

Mutable Variables In Mutable variable we can change the value of the variable.

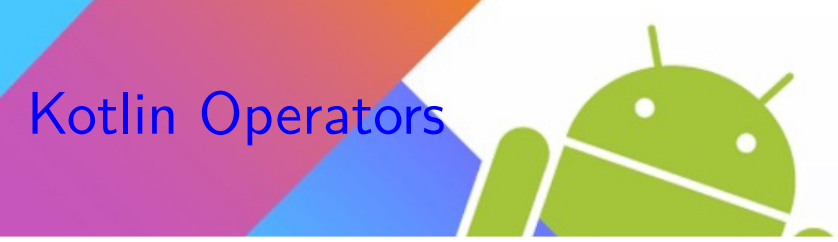
```
var myAge    =  25
myAge    =  26 // compiles    successfully
println( "My new    Age    is ${myAge}")
```

Output:
My new Age is 26

Scope of a variable

- In Mutable variable we can change the value of the variable. A variable exists only inside the block of code() where it has been declared.
- You can not access the variable outside the loop. Same variable can be declared inside the nested loop - so if a function contains an argument x and we declare a new variable x inside the same loop, then x inside the loop is different then the argument. Naming Convention - Every variable should be named using lowerCamelCase.

val myBirthDate = "02/12/1994"



Operators are the special symbols that perform different operation on operands. For example + and − are operators that perform addition and subtraction respectively. Like Java, Kotlin contains different kinds of operators.

- Arithmetic operator
- Relation operator
- Assignment operator
- Unary operator
- Logical operator
- Bitwise operator

Arithmetic Operators

Operators	Meaning	Expression	Translate to
+	Addition	a + b	a.plus(b)
−	Subtraction	a − b	a.minus(b)
*	Multiplication	a * b	a.times(b)
/	Division	a / b	a.div(b)
%	Modulus	a % b	a.rem(b)

```kotlin
// Kotlin code
fun main(args: Array<String>)
{
        var a = 20
        var b = 4
        println("a + b = " + (a + b))
        println("a - b = " + (a - b))
        println("a * b = " + (a.times(b)))
        println("a / b = " + (a / b))
        println("a % b = " + (a.rem(b)))
}
```

Output:

a + b = 24

a - b = 16

a * b = 80

a / b = 5

a % b = 0

Relational Operators

Op	Meaning	Expression	Translate to
>	greater than	a > b	a.compareTo(b) > 0
<	less than	a < b	a.compareTo(b) < 0
>=	greater than or equal to	a >= b	a.compareTo(b) >= 0
<=	less than or equal to	a <= b	a.compareTo(b) <= 0
==	is equal to	a == b	a?.equals(b) ?: (b === null)
!=	not equal to	a != b	!(a?.equals(b) ?: (b === null)) > 0

```
// Kotlin code
fun main(args: Array<String>)
{
        var c = 30
        var d = 40
        println("c > d = "+(c>d))
        println("c < d = "+(c.compareTo(d) < 0))
        println("c >= d = "+(c>=d))
        println("c <= d = "+(c.compareTo(d) <= 0))
        println("c == d = "+(c==d))
        println("c != d = "+(!(c?.equals(d) ?: (d === null))))
}
```

Output:

```
c > d = false
c < d = true
c >= d = false
c <= d = true
c == d = false
c != d = true
```

Bitwise Operators

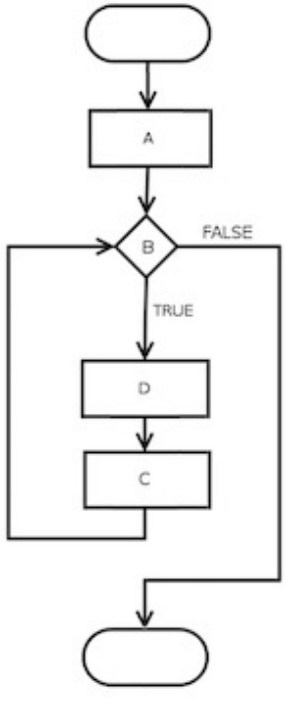
Operators	Meaning	Expression
shl	signed shift left	a.shl(b)
shr	signed shift right	a.shr(b)
ushr	unsigned shift right	a.ushr()
and	bitwise and	a.and(b)
or	bitwise or	a.or()
xor	bitwise xor	a.xor()
inv	bitwise inverse	a.inv()

```
// Kotlin code
fun main(args: Array<String>)
{
        println("5 signed shift left by 1
        ↪  bit: " + 5.shl(1))
        println("10 signed shift right by 2
        ↪  bits: : " + 10.shr(2))
        println("12 unsigned shift right by 2
        ↪  bits:  " + 12.ushr(2))
        println("36 bitwise and 22: " +
        ↪  36.and(22))
        println("36 bitwise or 22: " +
        ↪  36.or(22))
        println("36 bitwise xor 22: " +
        ↪  36.xor(22))
        println("14 bitwise inverse is: " +
        ↪  14.inv())
}
```

Output:
5 signed shift left by 1 bit: 10
10 signed shift right by 2 bits: : 2
12 unsigned shift right by 2 bits: 3
36 bitwise and 22: 4
36 bitwise or 22: 54
36 bitwise xor 22: 50
14 bitwise inverse is: -15

Kotlin for Android App Development

Kotlin flow control statements determine the next statement to be executed. For example, the statements if-else, if, when, while, for, and do are flow control statements. Flow Chart for Control Flow Control flow can be depicted using the following Flow Chart:



Kotlin Control Flow Statements

- Kotlin If...Else Expression
- Kotlin When Expression
- Kotlin For Loop
- Kotlin While Loop
- Kotlin Break and Continue

Kotlin if-else expression

Decision Making in programming is similar to decision making in real life. In programming too, a certain block of code needs to be executed when some condition is fulfilled.A programming language uses control statements to control the flow of execution of program based on certain conditions.If condition is true then it enters into the conditional block and executes the instructions.
There are different types of if-else expressions in Kotlin:

- if expression
- if-else expression
- if-else-if ladder expression
- nested if expression

if statement/expression

It is used to specify a block of statements to be executed or not i.e if a certain condition is true then the statement or block of statements to be executed otherwise fails to execute.

Syntax:

```
// Kotlin code
if(condition) {

        // code to run if
        ↪ condition is
        ↪ true
}
```

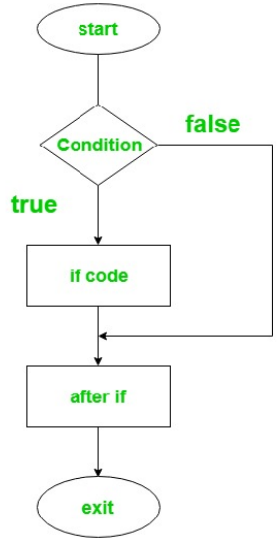


Figure: Flow Chart

Kotlin

for Android App Development

Example

```
fun main(args: Array<String>) {
        var a = 3
        if(a > 0){
                print("Yes,number is positive")
        }
}
```

Output:

Yes, number is positive

if-else statement/expression

if-else statement contains two blocks of statements. 'if' statement is used to execute the block of code when the condition becomes true and 'else' statement is used to execute a block of code when the condition becomes false.

Syntax:

```
if(condition) {
        // code to run if
        ↪ condition is
        ↪ true
}
else {
        // code to run if
        ↪ condition is
        ↪ false
}
```

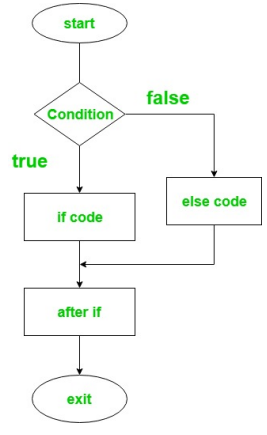


Figure: Flow Chart

Here is the Kotlin program to find the larger value of two numbers.

Syntax:

```
fun main(args:
↪  Array<String>) {
        var a = 5
        var b = 10
        if(a > b){
                print("Number
                ↪  5 is
                ↪  larger
                ↪  than
                ↪  10")
        }
        else{
                println("Number
                ↪  10 is
                ↪  larger
                ↪  than
                ↪  5")
        }
}
```

Output:

Number 10 is larger than 5

Kotlin if-else expression as ternary operator

In Kotlin, if-else can be used as an expression because it returns a value. Unlike java, there is no ternary operator in Kotlin because if-else return the value according to the condition and works exactly similar to ternary. Below is the Kotlin program to find the greater value between two numbers using if-else expression.

Kotlin

for Android App Development

Syntax:

```
fun main(args:
↪  Array<String>) {
        var a = 50
        var b = 40

        // here if-else
        ↪  returns a
        ↪  value which
        // is to be stored
        ↪  in max
        ↪  variable
        var max = if(a >
        ↪  b){
                print("Greater
                ↪  number
                ↪  is:
                ↪  ")
                a
        }
        else{
                print("Greater
                ↪  number
                ↪  is:")
                b
        }
        print(max)
}
```

Output:

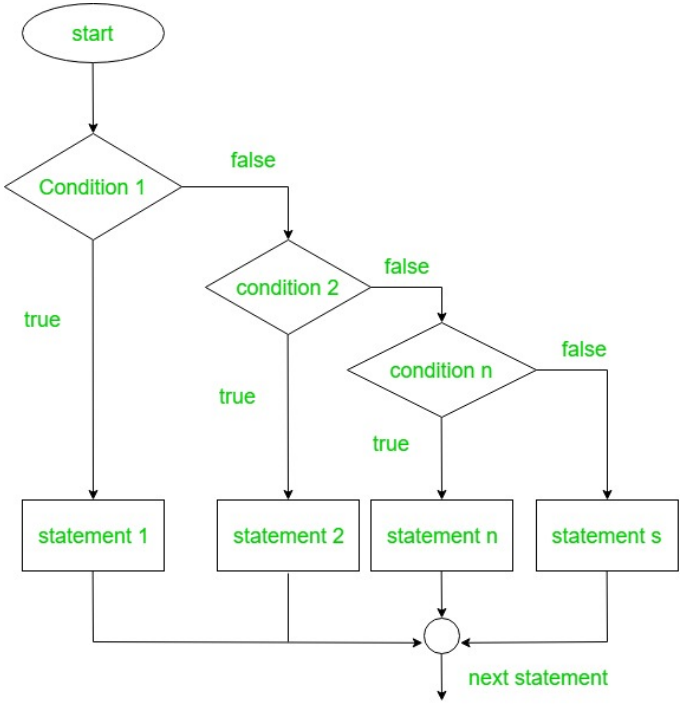
Greater number is: 50

if-else-if ladder expression

Here, a user can put multiple conditions. All the 'if' statements are executed from the top to bottom. One by one all the conditions are checked and if any of the condition found to be true then the code associated with the if statement will be executed and all other statements bypassed to the end of the block. If none of the conditions is true, then by default the final else statement will be executed.

Syntax:

```
if(Firstcondition) {
        // code to run if
        ↪ condition is
        ↪ true
}
else if(Secondcondition)
↪ {
        // code to run if
        ↪ condition is
        ↪ true
}
else{
}
```



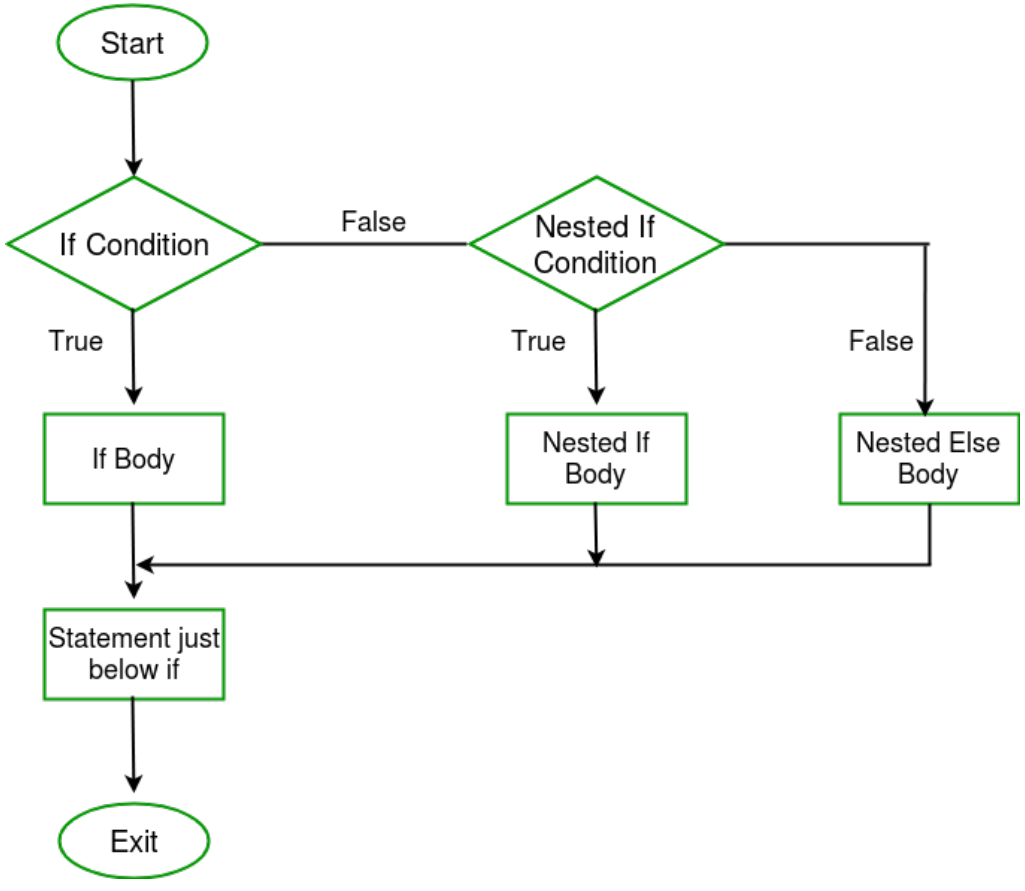

```
import java.util.Scanner

fun main(args: Array<String>) {

        // create an object for scanner
        class
        val reader =
        Scanner(System.`in`)
        print("Enter any number: ")

        // read the next Integer value
        var num = read
        er.nextInt()
        var result = if ( num > 0){
                "$num is positive number"
        }
        else if( num < 0){
                "$num is negative number"
        }
        else{
                "$num is equal to zero"
        }
        println(result)
}
```

Nested if expression

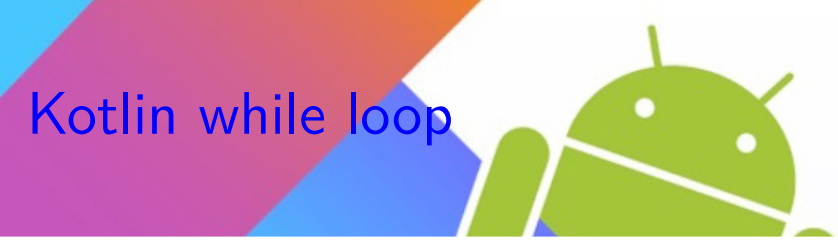
Nested if statements means an if statement inside another if statement.If first condition is true then code the associated block to be executed, and again check for the if condition nested in the first block and if it is also true then execute the code associated with it. It will go on until the last condition is true.

Syntax:

```
if(condition1){
        // code 1
        if(condition2){
                // code2
        }
}
```


Start
If Condition
False
Nested If Condition
True
True
False
If Body
Nested If Body
Nested Else Body
Statement just below if
Exit

```
import java.util.Scanner
fun main(args: Array<String>) {
        // create an object for scanner class
        val reader = Scanner(System.`in`)
        print("Enter three numbers: ")
        var num1 = reader.nextInt()
        var num2 = reader.nextInt()
        var num3 = reader.nextInt()
        var max  = if ( num1 > num2) {
                if (num1 > num3) {
                        "$num1 is the largest number"
                }
        else {
                "$num3 is the largest number"
        }
}
else if( num2 > num3){
        "$num2 is the largest number"
}
else{
        "$num3 is the largest number"
}
println(max)
}
```

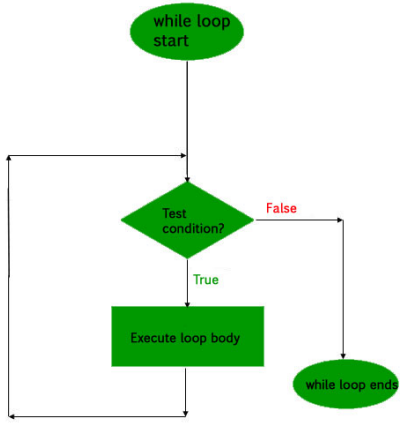


Kotlin while loop

In programming, loop is used to execute a specific block of code repeatedly until certain condition is met. If you have to print counting from 1 to 100 then you have to write the print statement 100 times. But with help of loop you can save time and you need to write only two lines.

It consists of a block of code and a condition. First of all the condition is evaluated and if it is true then execute the code within the block. It repeats until the condition becomes false because every time the condition is checked before entering into the block. The while loop can be thought of as repeating of if statements.
The syntax of while loop-

```
while(condition) {
        // code to run
}
```



Kotlin program to print numbers from 1 to 10 using while loop: In below program, we print the numbers using while loop. First, initialize the variable number by 1. Put the expression (number $<=$ 10) in while loop and checks is it true or not?. If true, enters in the block and execute the print statement and increment the number by 1. This repeats until the condition becomes false.

```
fun main(args:
↪  Array<String>) {
        var number = 1

        while(number <=
        ↪  10) {
                println(number)
                number++;
        }
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

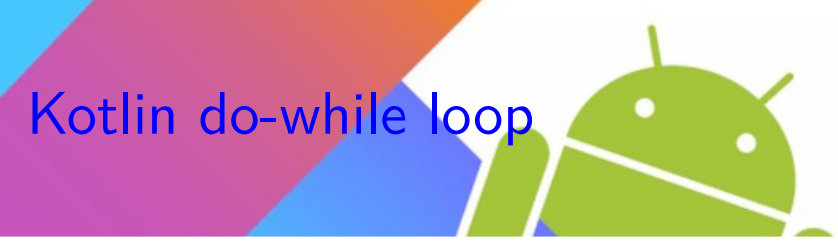
Kotlin

for Android App Development

Kotlin program to print the elements of an array using while loop: In the below program we create an array(names) and initialize with different number of strings and also initialize a variable index by 0. The size of an array can be calculated by using **arrayName.size**. Put the condition (index ⟨ names.size) in the while loop. If index value less than or equal to array size then it enters into the block and print the name stored at the respective index and also increment the index value after each iteration. This repeats until the condition becomes false.

```
fun main(args: Array<String>) {
        var names =
        ↪   arrayOf("Praveen","Gaurav","Akash","Sid
        ↪   hant","Abhi","Mayank")
        var index = 0
        while(index < names.size) {
                println(names[index])
                index++
        }
}
```

Output:
Praveen
Gaurav
Akash
Sidhant
Abhi
Mayank



Kotlin do-while loop

Like Java, do-while loop is a control flow statement which executes a block of code at least once without checking the condition, and then repeatedly executes the block, or not, it totally depends upon a Boolean condition at the end of do-while block. It contrast with the while loop because while loop executes the block only when condition becomes true but do-while loop executes the code first and then the expression or test condition is evaluated.

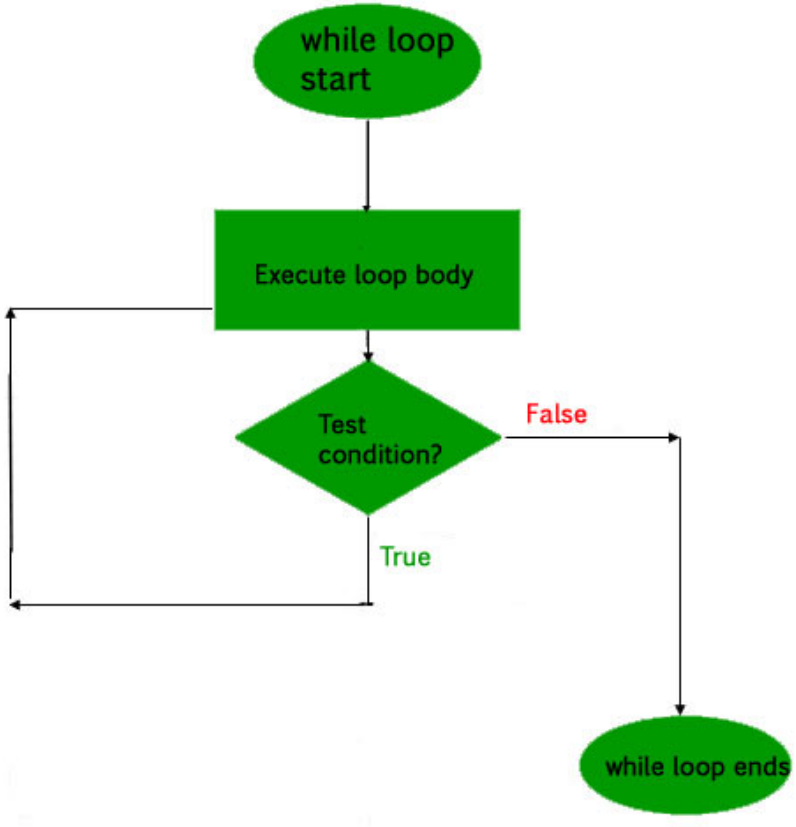
do-while loop working –

First of the all the statements within the block is executed, and then the condition is evaluated. If the condition is true the block of code is executed again. The process of execution of code block repeated as long as the expression evaluates to true. If the expression becomes false, the loop terminates and transfers control to the statement next to do-while loop.

It is also known as post-test loop because it checks the condition after the block is executed.

Syntax of the do-while loop-

```
do {
        // code to run
        {
                while(condition)
        }
```


while loop start
Execute loop body
Test condition?
False
True
while loop ends

Kotlin program to find the factorial of a number using do-while loop -

```
fun main(args:
↪  Array<String>) {
        var number = 6
        var factorial = 1

        do {
                factorial
               ↪  *=
               ↪  number
                number--
        }while(number >
       ↪  0)
        println("Factorial
       ↪  of 6 is
       ↪  $factorial")
}
```

Output:
Factorial of 6 is 720

In Kotlin, for loop is equivalent to foreach loop of other languages like C#. Here for loop is used to traverse through any data structure which provides an iterator. It is used very differently then the for loop of other programming languages like Java or C. The syntax of for loop in Kotlin:

```
for(item in collection) {
        // code to execute
}
```

In Kotlin, for loop is used to iterate through the following because all of them provides iterator.

- Range
- Array
- String
- Collection

Iterate through range using for loop

You can traverse through Range because it provides iterator. There are many ways you can iterate through Range. The in operator used in for loop to check value lies within the Range or not. Below programs are example of traversing the range in different ways and in is the operator to check the value in the range. If value lies in between range then it returns true and prints the value.

Iterate through range to print the values

```
fun main(args:
↪  Array<String>)
{
        for (i in 1..6) {
                print("$i
                ↪  ")
        }
}
```

Output:
1 2 3 4 5 6

Iterate through range to jump using step-3

```
fun main(args:
↪  Array<String>)
{
        for (i in 1..10
        ↪  step 3) {
            print("$i
            ↪  ")
        }
}
```

Output:
1 4 7 10

In Kotlin, when replaces the switch operator of other languages like Java. A certain block of code needs to be executed when some condition is fulfilled. The argument of when expression compares with all the branches one by one until some match is found. After the first match is found, it reaches to end of the when block and executes the code next to the when block. Unlike switch cases in Java or any other programming language, we do not require a break statement at the end of each case. In Kotlin, when can be used in two ways:

- when as a statement
- when as an expression

Using when as a statement with else

```
fun main (args : Array<String>) {
        print("Enter the name of heavenly body: ")
        var name= readLine()!!.toString()
        when(name) {
                "Sun" -> print("Sun is a Star")
                "Moon" -> print("Moon is a Satellite")
                "Earth" -> print("Earth is a planet")
                else -> print("I don't know anything about it")
        }
}
```

Using when as a statement without else

```
fun main (args : Array<String>) {
        print("Enter the name of heavenly body: ")
        var name= readLine()!!.toString()
        when(name) {
                "Sun" -> print("Sun is a Star")
                "Moon" -> print("Moon is a Satellite")
                "Earth" -> print("Earth is a planet")

        }
}
```

Using when as an expression

```
fun main(args : Array<String>) {
        print("Enter number of the Month: ")
        var monthOfYear  = readLine()!!.toInt()
        var month= when(monthOfYear) {
                1->"January"
                2->"February"
                3->"March"
                4->"April"
                5->"May"
                6->"June"
                7->"July"
                8->"August"
                9->"September"
                10->"October"
                11->"November"
                12->"December"
                else-> "Not a month of year"
        }
        print(month)
}
```

A function is a unit of code that performs a special task. In programming, function is used to break the code into smaller modules which makes the program more manageable.
For example: If we have to compute sum of two numbers then define a fun sum().

```
fun sum(a: Int, b: Int): Int {
        return a + b
}
```

We can call sum(x, y) at any number of times and it will return sum of two numbers. So, function avoids the repetition of code and makes the code more reusable.

Types of Function

In Kotlin, there are two types of function-

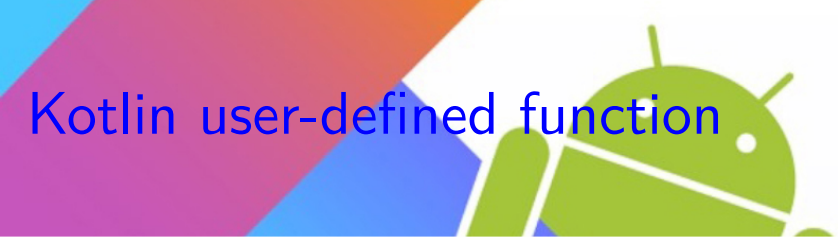
- Standard library function
- User defined function

- In Kotlin, there are different number of built-in functions already defined in standard library and available for use. We can call them by passing arguments according to requirement.
- In below program, we will use built-in functions ***arrayOf()***, ***sum()*** and ***println()***. The function ***arrayOf()*** require some arguments like integers, double etc to create an array and we can find the sum of all elements using ***sum()*** which does not require any argument.

```
fun main(args: Array<String>) {
        var sum = arrayOf(1,2,3,4,5,6,7,8,9,10).sum()

        println("The sum of all the elements of an array is: $sum")
}
```

The list of different standard library functions and their use-

- sqrt() – Used to calculate the square root of a number.
- print() – Used to print a message to standard output.
- rem() – To find the remainder of one number when divided by another.
- toInt() – To convert a number to integer value.
- readline() – Used for standard input.
- compareTo() – To compare two numbers and return boolean value.



Kotlin user-defined function

A function which is defined by the user is called user-defined function. As we know, to divide a large program in small modules we need to define function. Each defined function has its own properties like name of function, return type of a function, number of parameters passed to the function etc.

Creating user-defined function

In Kotlin, function can be declared at the top, and no need to create a class to hold a function, which we are used to do in other languages such as Java or Scala. Generally we define a function as:

```
fun fun_name(a: data_type, b: data_type, ......): return_type  {
        // other codes
        return
}
```

Explanation of the above code:

- fun– Keyword to define a function.
- fun_name – Name of the function which later used to call the function.
- a: data_type – Here, a is an argument passed and data_type specify the data type of argument like integer or string.
- return_type – Specify the type of data value return by the function.
- {. . . .} – Curly braces represent the block of function.

Kotlin

for Android App Development

Kotlin function mul() to multiply two numbers having same type of parameters-

```
fun mul(num1: Int, num2: Int): Int {
        var number = num1.times(num2)
        return number
}
```

Explanation: We have defined a function above starting with fun keyword whose return type is an Integer.

Calling of user-defined function

We create a function to assign a specific task. Whenever a function is called the program leaves the current section of code and begins to execute the function.
The flow-control of a function-

1. The program comes to the line containing a function call.
2. When function is called, control transfers to that function.
3. Executes all the instruction of function one by one.
4. Control is transferred back only when the function reaches closing braces or there any return statement.
5. Any data returned by the function is used in place of the function in the original line of code.

Default and Named argument

The arguments which need not specify explicitly while calling a function are called default arguments. If the function is called without passing arguments then the default arguments are used as function parameters. In other cases, if arguments are passed during a function call then passed arguments are used as function parameters. There are three cases for default arguments-

1. No arguments are passed while calling a function
2. Partial arguments are passed while calling a function
3. All arguments are passed while calling a function

No arguments are passed while calling a function

When no argument is passed while calling a function then the default arguments are used as function parameters. We need to initialize the variables while defining a function. Kotlin program of calling student() function without passing an arguments-

```
// default arguments in function definition name, standard and roll_no
fun student(name: String="Praveen", standard: String="IX" , roll_no: Int=11) {
        println("Name of the student is: $name")
        println("Standard of the student is: $standard")
        println("Roll no of the student is: $roll_no")
}
fun main(args: Array<String>) {
        val name_of_student = "Gaurav"
        val standard_of_student = "VIII"
        val roll_no_of_student = 25
        student()         // passing no arguments while calling student
}
```

Properties are an important part of any programming language. In Kotlin, we can define properties in the same way as we declare another variable. Kotlin properties can be declared either as mutable using the var keyword or as immutable using the val keyword.
Here, the property initializer, getter and setter are optional. We can also omit the property type, if it can be inferred from the initializer. The syntax of a read-only or immutable property declaration differs from a mutable one in two ways:

1. It starts with val instead of var.
2. It does not allow a setter.

A lambda is an expression that makes a function that has no name. Lambdas expression and Anonymous function both are function literals means these functions are not declared but passed immediately as an expression.

Lambda Expression –

The syntax of Kotlin lambdas is similar to Java Lambdas. A function without a name is called an anonymous function. For lambda expression, we can say that it is an anonymous function.

```
fun main(args: Array<String>) {
        val str = { println("Welcome to Kotlin Programming.")}

        // invoking function method1
        str()

        // invoking function method2
        str.invoke()
}
```

Syntax

val lambda_name : Data_type = argument_List -> code_body

- A higher-order function in Kotlin is a function that can accept other functions as arguments and/or return functions as results.
- This concept is a fundamental part of functional programming and provides a powerful mechanism for abstracting and manipulating behavior in your code.
- Higher-order functions are a key feature in Kotlin and enable more concise and expressive programming.

Characteristics of Kotlin Higher Order Functions

- Accepting Functions as Arguments
- Returning Functions
- Lambda Expressions
- Function Types

```
fun operateOnNumbers(a: Int, b: Int, operation: (Int,
↪  Int) -> Int): Int {
        return operation(a, b)
}
```

In this example, operateOnNumbers is a higher-order function that accepts two integers (a and b) and a function (operation) of type (Int, Int) -> Int as its parameters. You can pass different operations as lambdas when calling this function, such as addition or multiplication.

Kotlin Type Conversion

Type conversion (also called as Type casting) refers to changing the entity of one data type variable into another data type. As we know Java supports implicit type conversion from smaller to larger data types. An integer value can be assigned to the long data type.

Example

But, Kotlin does not support implicit type conversion. An integer value can not be assigned to the long data type.

Kotlin

for Android App Development

The following helper function can be used to convert one data type into another:

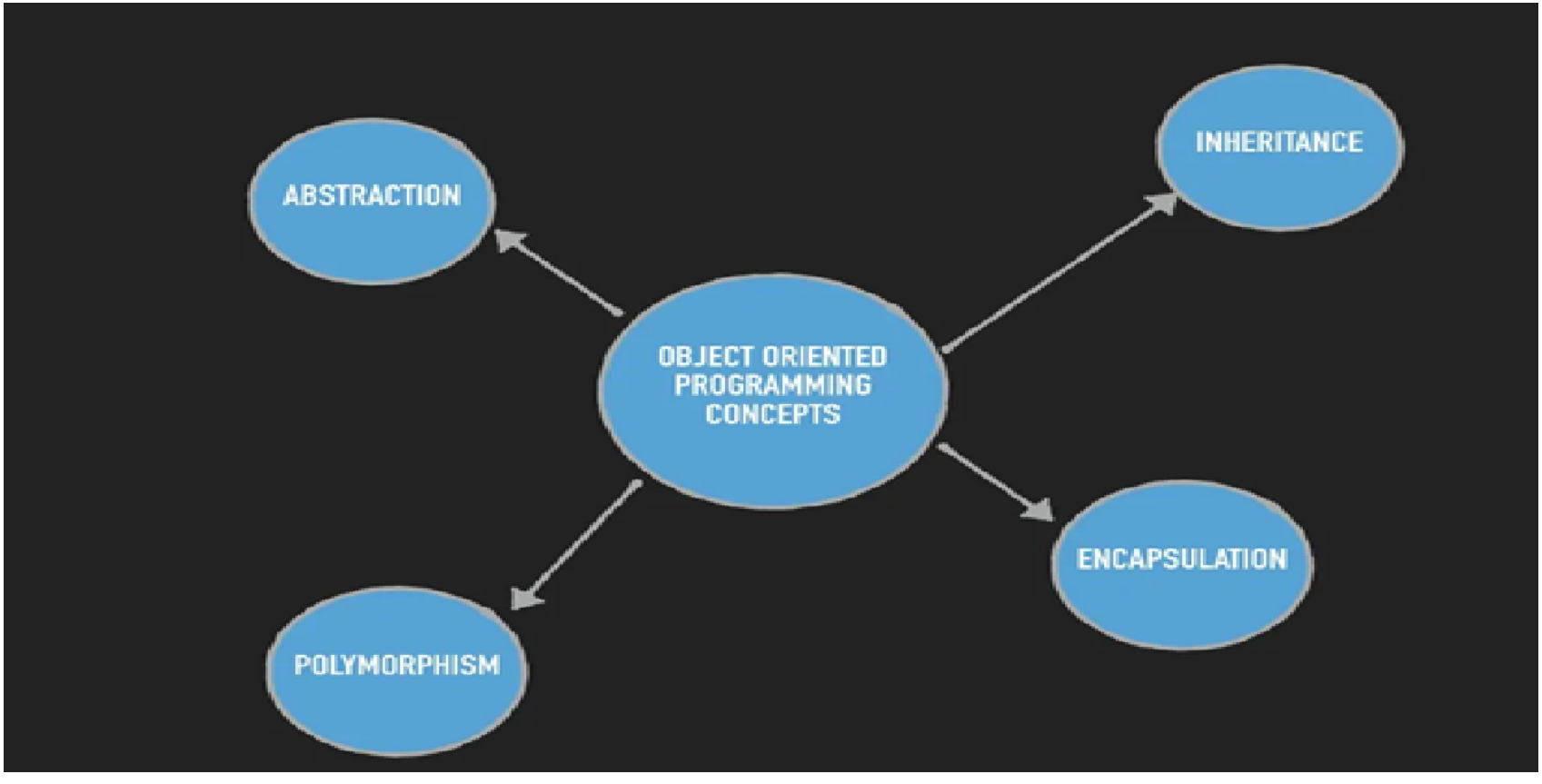
- toByte()
- toShort()
- toInt()
- toLong()
- toFloat()
- toDouble()
- toChar()

Kotlin

for Android App Development

Kotlin Program to convert the one data type into another:

```
fun main(args: Array<String>)
{

        println("259 to byte: " + (259.toByte()))
        println("50000 to short: " + (50000.toShort()))
        println("21474847499 to Int: " + (21474847499.toInt()))
        println("10L to Int: " + (10L.toInt()))
        println("22.54 to Int: " + (22.54.toInt()))
        println("22 to float: " + (22.toFloat()))
        println("65 to char: " + (65.toChar()))
        // Char to Number is deprecated in kotlin
        println("A to Int: " + ('A'.toInt()))
}
```


ABSTRACTION
INHERITANCE
OBJECT ORIENTED PROGRAMMING CONCEPTS
ENCAPSULATION
POLYMORPHISM

Create a Class

To create a class, use the class keyword, and specify the name of the class:

```
class Car {
        var brand = ""
        var model = ""
        var year = 0
}
// Create a c1 object of the Car class
val c1 = Car()

// Access the properties and add some values to it
c1.brand = "Ford"
c1.model = "Mustang"
c1.year = 1969

println(c1.brand)    // Outputs Ford
println(c1.model)    // Outputs Mustang
println(c1.year)     // Outputs 1969
```

Kotlin Program of Default Setter and Getter

In Kotlin, setter is used to set the value of any variable and getter is used to get the value. Getters and Setters are auto-generated in the code. Let's define a property 'name', in a class, 'Company'. The data type of 'name' is String and we shall initialize it with some default value.

```
class Company {
        var name: String = ""
        get() = field        // getter
        set(value) {         // setter
                field = value
        }
}
fun main(args: Array<String>) {
        val c = Company()
        c.name = "GIET University"  // access setter
        println(c.name)           // access getter
}
```

Kotlin

for Android App Development

We instantiate an object ‘c’ of the class ‘Company . . . ’. When we initialize the ‘name’ property, it is passed to the setter’s parameter value and sets the ‘field’ to value. When we are trying to access name property of the object, we get field because of this code get() = field. We can get or set the properties of an object of the class using the dot(.) notation

We have noticed these two identifiers in the above program.

- Value: Conventionally, we choose the name of the setter parameter as value, but we can choose a different name if we want. The value parameter contains the value that a property is assigned to. In the above program, we have initialized the property name as c.name = "GIET University", the value parameter contains the assigned value "GIET University".
- Backing Field (field): It allows storing the property value in memory possible. When we initialize a property with a value, the initialized value is written to the backing field of that property. In the above program, the value is assigned to field, and then, field is assigned to get().

Custom Setter and Getter

```
class Registration( email: String, pwd: String, age: Int , gender: Char) {
        var email_id: String = email
        // Custom Getter
        get() {
                return field.toLowerCase()
        }
        var password: String = pwd
        // Custom Setter
        set(value){
                field = if(value.length > 6) value else throw IllegalArgumentException("Passwords
                ↪  is too small")
        }
        var age: Int = age
        // Custom Setter
        set(value) {
                field = if(value > 18 ) value else throw IllegalArgumentException("Age must be
                ↪  18+")}
```

```
        var gender : Char = gender
        // Custom Setter
        set (value){
                field = if(value == 'M') value else throw IllegalArgumentException("User should be
                ↪  male")}}
fun main(args: Array<String>) {
        val giet = Registration("ranjit.patnaik83@GMAIL.COM","GIET@123",25,'M')
        println("${giet.email_id}")
        giet.email_id = "ranjit@giet.edu"
        println("${giet.email_id}")
        println("${giet.password}")
        println("${giet.age}")
        println("${giet.gender}")
        // throw IllegalArgumentException("Passwords is too small")
        giet.password = "ranjit@1983"
        // throw IllegalArgumentException("Age should be 18+")
        giet.age= 39
        // throw IllegalArgumentException("User should be male")
        giet.gender = 'M'}
```

In Kotlin, there's a faster way of doing this, by using a constructor.
A constructor is like a special function, and it is defined by using two parantheses () after the class name. You can specify the properties inside of the parantheses (like passing parameters into a regular function).
The constructor will initialize the properties when you create an object of a class. Just remember to specify the type of the property/variable:

```
class Car(var brand: String, var model: String, var year: Int)

fun main() {
        val c1 = Car("Ford", "Mustang", 1969)
}
```

Kotlin

for Android App Development

A constructor is a special member function that is invoked when an object of the class is created primarily to initialize variables or properties. A class needs to have a constructor and if we do not declare a constructor, then the compiler generates a default constructor. Kotlin has two types of constructors:

- Primary Constructor
- Secondary Constructor

A class in Kotlin can have at most one primary constructor, and one or more secondary constructors. The primary constructor initializes the class, while the secondary constructor is used to initialize the class and introduce some extra logic.

Primary Constructor The primary constructor is initialized in the class header, and goes after the class name, using the constructor keyword. The parameters are optional in the primary constructor.

```
// main function
fun main(args: Array<String>)
{
        val add = Add(5, 6)
        println("The Sum of numbers 5 and 6 is: ${add.c}")
}

// primary constructor
class Add constructor(a: Int,b:Int)
{
        var c = a+b;
}
```

Output

The Sum of two numbers is: 11

Primary Constructor with Initializer Block

The primary constructor cannot contain any code, the initialization code can be placed in a separate initializer block prefixed with the init keyword.
Kotlin program of primary constructor with initializer block
What is init block?
It acts as an initialiser block where member variables are initialised. This block gets executed whenever an instance of this class is created. There can be multiple init blocks and they are called in the order they are written inside the class.
Note: init blocks gets called before the constructor of this class is called.

Example

```
class Person (val _name: String) {

        // Member Variables
        var name: String

        // Initializer Blocks
        init{
                println("This is first init block")
        }

        init{
                println("This is second init block")
        }

        init{
                println("This is third init block")
        }
        init {
                this.name = _name
                println("Name = $name")
        }
}

fun main(args: Array<String>) {
        val person = Person("Aryan")
}
```

Output:

This is first init block This is second init block

This is third init block Name = Aryan

Kotlin

for Android App Development

Secondary Constructor

Secondary constructors allow initialization of variables and allow to provide some logic to the class as well. They are prefixed with the constructor keyword.
Kotlin program of implementing secondary constructor:

```
// main function
fun main(args: Array<String>)
{
        Add(5, 6)
}

// class with one secondary constructor
class Add
{
        constructor(a: Int, b:Int)
        {
                var c = a + b
                println("The sum of numbers 5 and 6 is: ${c}")
        }
}
```

Output:

The sum of numbers 5 and 6 is: 11

Kotlin supports inheritance, which allows you to define a new class based on an existing class. The existing class is known as the superclass or base class, and the new class is known as the subclass or derived class. The subclass inherits all the properties and functions of the superclass, and can also add new properties and functions or override the properties and functions inherited from the superclass.
Inheritance is one of the more important features in object-oriented programming. Inheritance enables code re-usability, it allows all the features from an existing class(base class) to be inherited by a new class(derived class). In addition, the derived class can also add some features of its own.
Syntax of inheritance:

```
open class baseClass (x:Int ) {
        ..........
}
class derivedClass(x:Int) : baseClass(x) {
        ..........
}
```

```
open class Base(val name: String) {

        init { println("Geeksforgeeks") }

        open val size: Int =
        name.length.also { println("Test code: $it") }
}

class Derived(
name: String,
val lastName: String,
) : Base(name.replaceFirstChar { it.uppercase() }.also { println("Argument for the base class: $it") }) {

        init { println("Geeksforgeeks") }

        override val size: Int =
        (super.size + lastName.length).also { println("Geeksforgeeks: $it") }
}
```

Output Output:

Base Class

Test code

Derived class

Kotlin
for Android App Development

In Kotlin, an interface is a collection of abstract methods and properties that define a common contract for classes that implement the interface. An interface is similar to an abstract class, but it can be implemented by multiple classes, and it cannot have state. Interfaces are custom types provided by Kotlin that cannot be instantiated directly. Instead, these define a form of behavior that the implementing types have to follow. With the interface, you can define a set of properties and methods, that the concrete types must follow and implement.

Creating Interfaces

```
interface Vehicle()
{
        fun start()
        fun stop()
}
```


```
interface Vehicle {
        fun start()
        fun stop()
}

class Car : Vehicle {
        override fun start()
        {
                println("Car started")
        }

        override fun stop()
        {
                println("Car stopped")
        }
}

fun main()
{
        val obj = Car()
        obj.start()
        obj.stop()
}
```

Polymorphism in Kotlin

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.
Real-life Illustration: Polymorphism
A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations. This is called polymorphism. Polymorphism is considered one of the important features of Object-Oriented Programming.
Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word "poly" means many and "morphs" means forms, So it means many forms.
Types of polymorphism Polymorphism is mainly divided into two types:

- Compile-time Polymorphism
- Runtime Polymorphism

Example 1: Compile-time Polymorphism

```
fun main (args: Array<String>) {
        println(doubleof(4))
        println(doubleof(4.3))
        println(doubleof(4.323))
}

fun doubleof(a: Int):Int {
        return 2*a
}

fun doubleOf(a:Float):Float {
        return 2*a
}

fun doubleof(a:Double):Double {
        return 2.00*a
}
```

Output:

8

8.6

8.646

Example 2: Runtime Polymorphism

```
fun main(args: Array<string>){
        var a = Sup()
        a.method1()
        a.method2()

        var b = Sum()
        b.method1()
        b.method2()
}

open class Sup{
        open fun method1(){
                println("printing method 1 from inside Sup")
        }
        fun method2(){
                println("printing method 2 from inside Sup")
        }
}

class Sum:Sup(){
        override fun method1(){
                println("printing method 1 from inside Sum")
        }
}
```

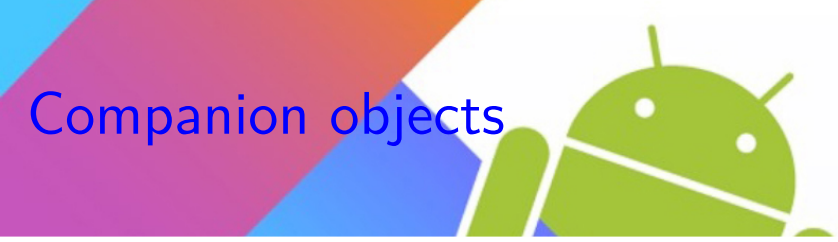
Output:

printing method 1 from inside Sup

printing method 2 from inside Sup

printing method 1 from inside Sum

printing method 2 from inside Sup



Unlike Java, Kotlin doesn't support static methods for a class. Most readers will know that static methods do not belong to the object instance but rather to the type itself. In Kotlin, it is advisable to define methods at the package level to achieve the functionality of static methods. Let's define a new Kotlin file and name it Static. Within this file, we will place the code for a function that will return the first character of the input string (if the input is empty, an exception will be raised).
In Kotlin, if you want to write a function or any member of the class that can be called without having the instance of the class then you can write the same as a member of a companion object inside the class. So, by declaring the companion object, you can access the members of the class by class name only(which is without explicitly creating the instance of the class).
To create a companion object, you need to add the companion keyword in front of the object declaration.


```
class Person {
        companion object Test {
                fun callMe() = println("I'm called.")
        }
}

fun main(args: Array<String>) {
        Person.callMe()
}
```

Singleton Class in Kotlin

A singleton class in Kotlin is designed so only a single instance can be created and used everywhere. Instances of the class that only require one are NetworkService, DatabaseService, etc. It is typically done because creating these objects repeatedly uses system resources. Therefore, creating an object only once and using it repeatedly is better.

Example of a Normal Class

```
class Student(){
        fun run(){
                println("Running")
        }
}
fun main(){
        val t1 = Student()
        val t2 = Student()
        println(t1)
        println(t2)
}
```

Rules for making the Singleton class in Kotlin

To create a Singleton class, the following guidelines must be followed:

- A private constructor
- A static reference of its class
- One static method
- Globally accessible object reference
- Consistency across multiple threads

To create a singleton class, we use the "object" keyword. Yes, it is that simple. There is a long process to make a class singleton in Java, but Kotlin is a lifesaver language. In Kotlin, we must use the "object" keyword to use the Singleton class, as already discussed above. The object class can have properties, functions, and the init method.
Key Point : The constructor method is prohibited in an object, so if some initialization is needed, we can use the init method and the object can be defined inside a class. The object gets instantiated when it is used for the first time.

Example of a Singleton Class

```
object Singleton{
        init{
                println("Singleton method invoked successfully!
        }
        fun run(){
                println("hello Ninja")
        }
}
fun main(){
        val t1 = Singleton
        val t2 = Singleton
        println(t1)
        println(t2)
}
```

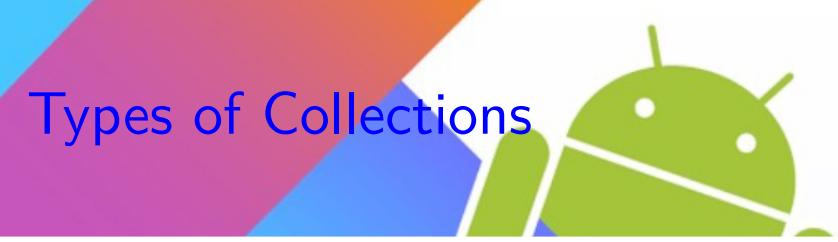
Kotlin Collections

In Kotlin, collections are used to store and manipulate groups of objects or data. There are several types of collections available in Kotlin, including:

- Lists – Ordered collections of elements that allow duplicates.
- Sets – Unordered collections of unique elements.
- Maps – Collections of key-value pairs, where each key is unique.
- Arrays – Fixed-size collections of elements with a specific type.
- Sequences – Lazily evaluated collections of elements that can be processed in a pipeline.

Here's an example of creating and using a list in Kotlin:

```
val fruits = listOf("apple", "banana", "orange", "grape")

// Access elements in the list
println("First fruit: ${fruits[0]}")
println("Last fruit: ${fruits.last()}")

// Iterate over the list
for (fruit in fruits) {
        println(fruit)
}

// Filter the list
val filtered = fruits.filter { it.startsWith("a") }
println("Filtered list: $filtered")
```



In Kotlin collections are categorized into two forms.

- Immutable Collection
- Mutable Collection

Immutable Collection

It means that it supports only read-only functionalities and can not be modified its elements. Immutable Collections and their corresponding methods are:

- List – listOf() and listOf<T>()
- Set – setOf()
- Map – mapOf()

List – It is an ordered collection in which we can access elements or items by using indices – integer numbers that define a position for each element. Elements can be repeated in a list any number of times. We can not perform add or remove operations in the immutable list.

Kotlin program to demonstrate the immutable list:

```
// An example for immutable list
fun main(args: Array<String>) {
        val immutableList = listOf("Mahipal","Nikhil","Rahul")
        // gives compile time error
        // immutableList.add = "Praveen"
        for(item in immutableList){
                println(item)
        }
}
```

Kotlin program to demonstrate the immutable set:

```
fun main(args: Array<String>) {
        // initialize with duplicate values
        // but output with no repetition
        var immutableSet = setOf(6,9,9,0,0,"Mahipal","Nikhil")
        // gives compile time error
        // immutableSet.add(7)
        for(item in immutableSet){
                println(item)
        }
}
```

Kotlin

for Android App Development

Kotlin program to demonstrate the immutable map:

```
// An example for immutable map
fun main(args : Array<String>) {
        var immutableMap = mapOf(9 to "Mahipal",8 to "Nikhil",7 to "Rahul")
        // gives compile time error
        // immutableMap.put(9,"Praveen")
        for(key in immutableMap.keys){
                println(immutableMap[key])
        }
}
```

Mutable Collection It supports both read and write functionalities. Mutable collections and their corresponding methods are:

- List – mutableListOf(),arrayListOf() and ArrayList
- Set – mutableSetOf(), hashSetOf()
- Map – mutableMapOf(), hashMapOf() and HashMap
- List – Since mutable list supports read and write operation, declared elements in the list can either be removed or added.

Kotlin program to demonstrate the mutable list:

```
fun main(args : Array<String>) {
        var mutableList = mutableListOf("Mahipal","Nikhil","Rahul")
        // we can modify the element
        mutableList[0] = "Praveen"
        // add one more element in the list
        mutableList.add("Abhi")
        for(item in mutableList){
                println(item)
        }
}
```

Set – The mutable Set supports both read and write functionality. We can access add or remove elements from the collections easily and it will preserve the order of the elements.

Kotlin program to demonstrate the mutable set:

```
fun main(args: Array<String>) {
        var mutableSet = mutableSetOf<Int>(6,10)
        // adding elements in set
        mutableSet.add(2)
        mutableSet.add(5)
        for(item in mutableSet){
                println(item)
        }
}
```

Map – It is mutable so it supports functionalities like put, remove, clear, etc.
Kotlin program to demonstrate the mutable map.

```
fun main(args : Array<String>) {
        var mutableMap = mutableMapOf<Int,String>(1 to "Mahipal",2 to "Nikhil",3 to "Rahul")
        // we can modify the element
        mutableMap.put(1,"Praveen")
        // add one more element in the list
        mutableMap.put(4,"Abhi")
        for(item in mutableMap.values){
                println(item)
        }
}
```

Kotlin Visibility Modifiers

In Kotlin, visibility modifiers are used to control the visibility of a class, its members (properties, functions, and nested classes), and its constructors. The following are the visibility modifiers available in Kotlin:

- private: The private modifier restricts the visibility of a member to the containing class only. A private member cannot be accessed from outside the class.
- internal: The internal modifier restricts the visibility of a member to the same module. A module is a set of Kotlin files compiled together.
- protected: The protected modifier restricts the visibility of a member to the containing class and its subclasses.
- public: The public modifier makes a member visible to any code. This is the default visibility for members in Kotlin.

In Kotlin, visibility modifiers are used to restrict the accessibility of classes, objects, interfaces, constructors, functions, properties, and their setters to a certain level. No need to set the visibility of getters because they have the same visibility as the property. There are four visibility modifiers in Kotlin.

Generics are the powerful features that allow us to define classes, methods and properties which are accessible using different data types while keeping a check on the compile-time type safety. Creating parameterized classes – A generic type is a class or method that is parameterized over types. We always use angle brackets <> to specify the type parameter in the program. Generic class is defined as follows:

```
class MyClass<T>(text: T) {
        var name = text
}
```

To create an instance of such a class, we need to provide the type of arguments:

```
val my : MyClass<String> = Myclass<String>("Kotlin Generic")
```

If the parameters can be inferred from the arguments of a constructor, one is allowed to omit the type arguments:

```
val my = MyClass("Kotlin Generic")
```

Here, "Kotlin Generic" has type String, so the compiler figures out that we are talking about Myclass<String>
Advantages of generic:
Type casting is evitable- No need to typecast the object. Type safety- Generic allows only a single type of object at a time. Compile time safety- Generics code is checked at compile time for the parameterized type so that it avoids run-time error.

```
class Company<T> (text : T){
        var x = text
        init{
                println(x)
        }
}
fun main(args: Array<String>){
        var name: Company<String> = Company<String>("Kotlin Generic")
        var rank: Company<Int> = Company<Int>(12)
}
```

Output:

Kotlin Generic 1234

Kotlin Exception Handling

An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program's instructions. Exception handling is a technique, using which we can handle errors and prevent run time crashes that can stop our program.
There are two types of Exceptions –

1. Checked Exception – Exceptions that are typically set on methods and checked at the compile time, for example IOException, FileNotFoundException etc
2. UnChecked Exception – Exceptions that are generally due to logical errors and checked at the run time, for example NullPointerException, ArrayIndexOutOfBoundException etc

Kotlin try-catch block –

```
try {
        // code that can throw exception
} catch(e: ExceptionName) {
        // catch the exception and handle it
}
```

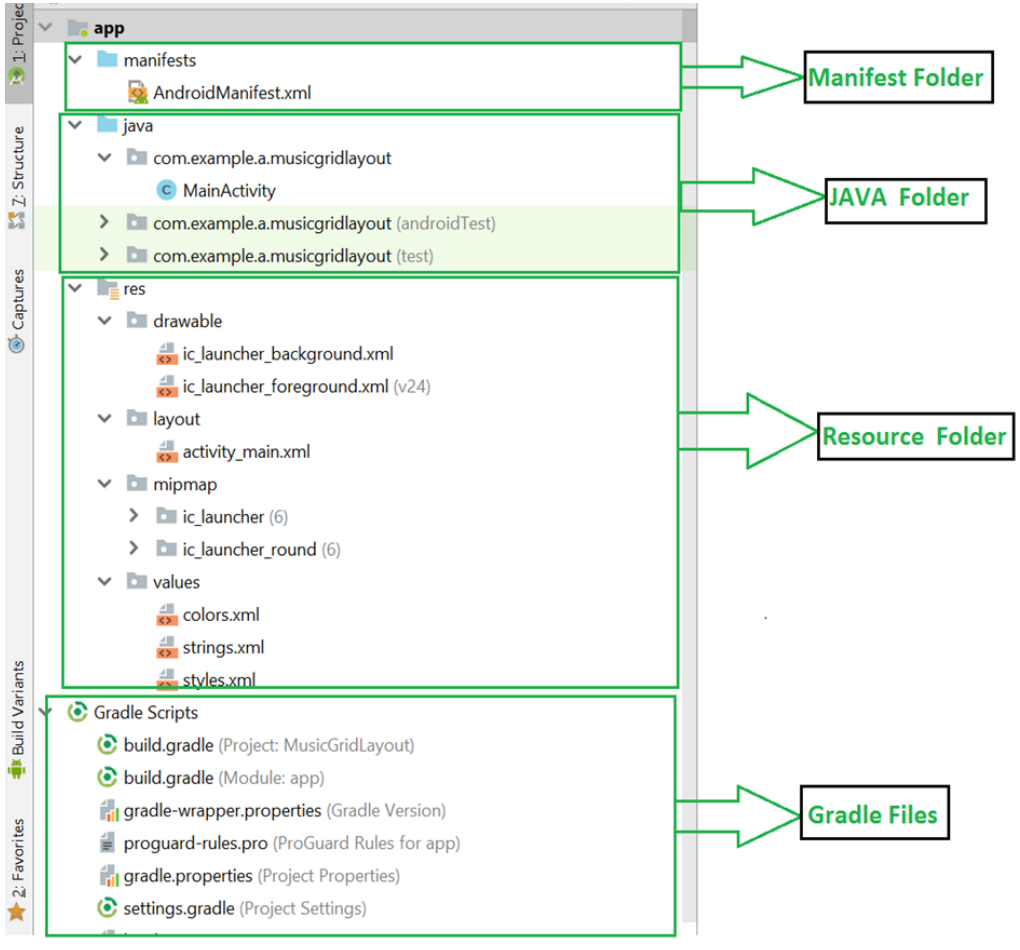
Kotlin

for Android App Development

UNIT II

Application Structure

Kotlin for Android App Development



AndroidManifest.xml

Kotlin for Android App Development

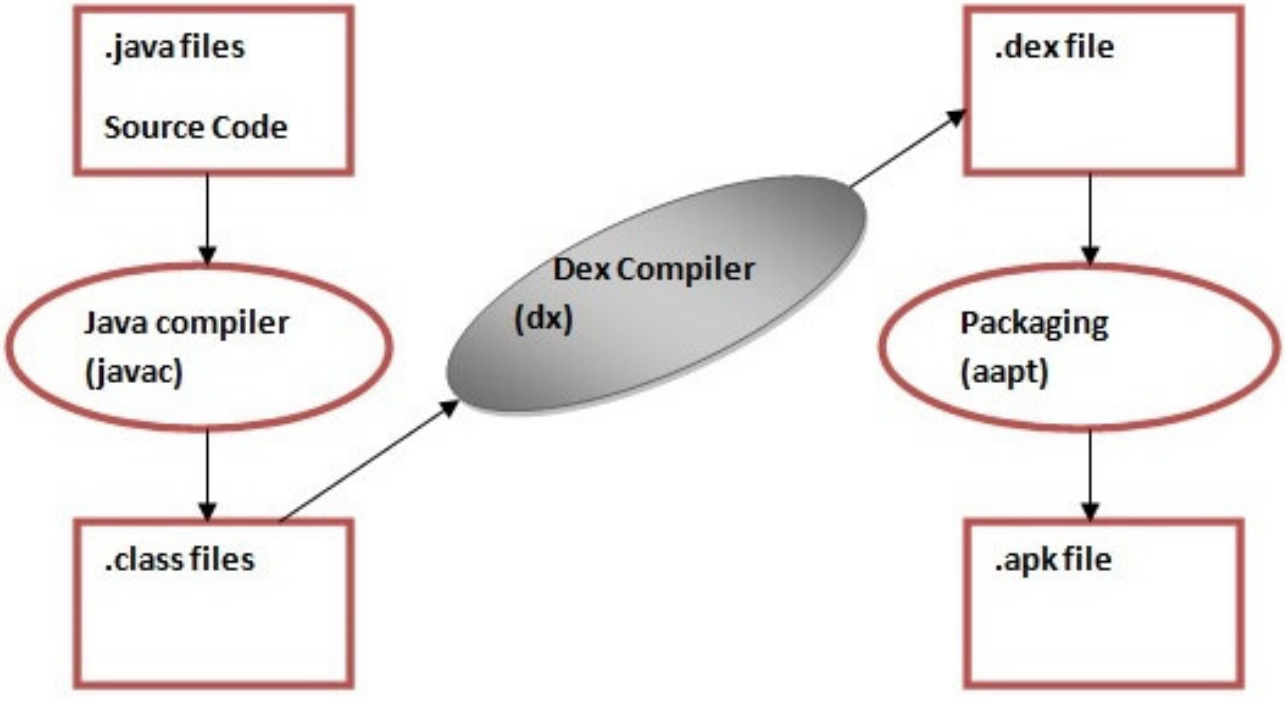
activity_main.xml | MainActivity.kt | AndroidManifest.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="MyAppSec_C_M"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.MyAppSec_C_M"
        tools:targetApi="31">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />


                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Dalvik Virtual Machine

- The Dalvik Virtual Machine (DVM) is an android virtual machine optimized for mobile devices. It optimizes the virtual machine for memory, battery life and performance.
- Dalvik is a name of a town in Iceland. The Dalvik VM was written by Dan Bornstein.
- The Dex compiler converts the class files into the .dex file that run on the Dalvik VM. Multiple class files are converted into one dex file.

Let's see the compiling and packaging process from the source file:



DVM vs. JVM

Kotlin

for Android App Development

DVM (Dalvik Virtual Machine)	JVM (Java Virtual Machine)
It is Register based which is designed to run on low memory.	It is Stack based.
DVM uses its own byte code and runs ".Dex" file. From Android 2.2 SDK Dalvik has got a Just in Time compiler	JVM uses java byte code and runs ".class" file having JIT (Just In Time).
DVM has been designed so that a device can run multiple instances of the VM efficiently. Applications are given their own instance.	Single instance of JVM is shared with multiple applications.
DVM supports Android operating system only.	JVM supports multiple operating systems.
For DVM very few Re-tools are available.	For JVM many Re-tools are available.
There is constant pool for every application.	It has constant pool for every class.
Here the executable is APK.	Here the executable is JAR.

Widgets are an essential aspect of home screen customization. You can think of them as "at-a-glance" views of an app's most important data and functionality that are accessible right on the user's home screen. Users can move widgets across their home screen panels, and, if supported, resize them to tailor the amount of information in the widget to their preference.

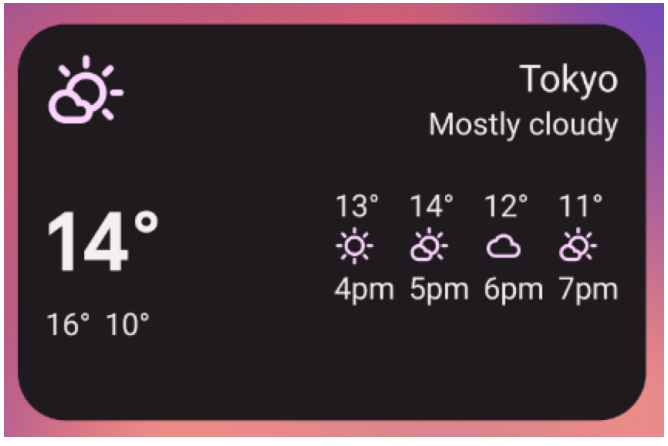
Widget types

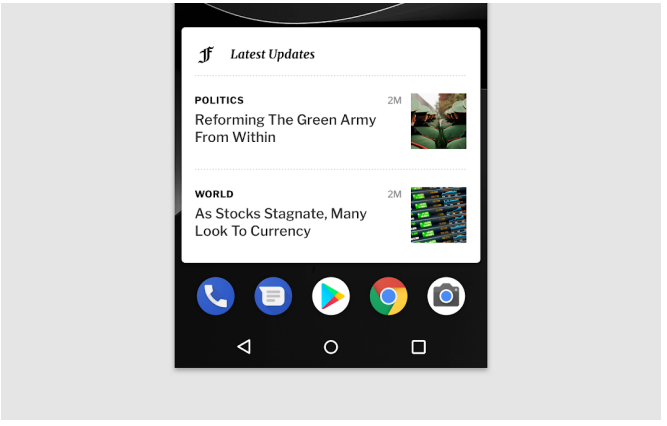
As you plan your widget, think about what kind of widget you want to build. Widgets typically fall into one of the following categories:

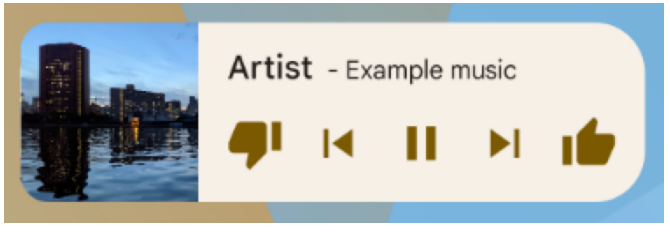
- Information widgets
- Collection widgets
- Control widgets
- Hybrid widgets

Kotlin

for Android App Development







- A widget is a small gadget or control of your android application placed on the home screen. Widgets can be very handy as they allow you to put your favourite applications on your home screen in order to quickly access them. You have probably seen some common widgets, such as music widget, weather widget, clock widget e.t.c
- Widgets could be of many types such as information widgets, collection widgets, control widgets and hybrid widgets. Android provides us a complete framework to develop our own widgets.

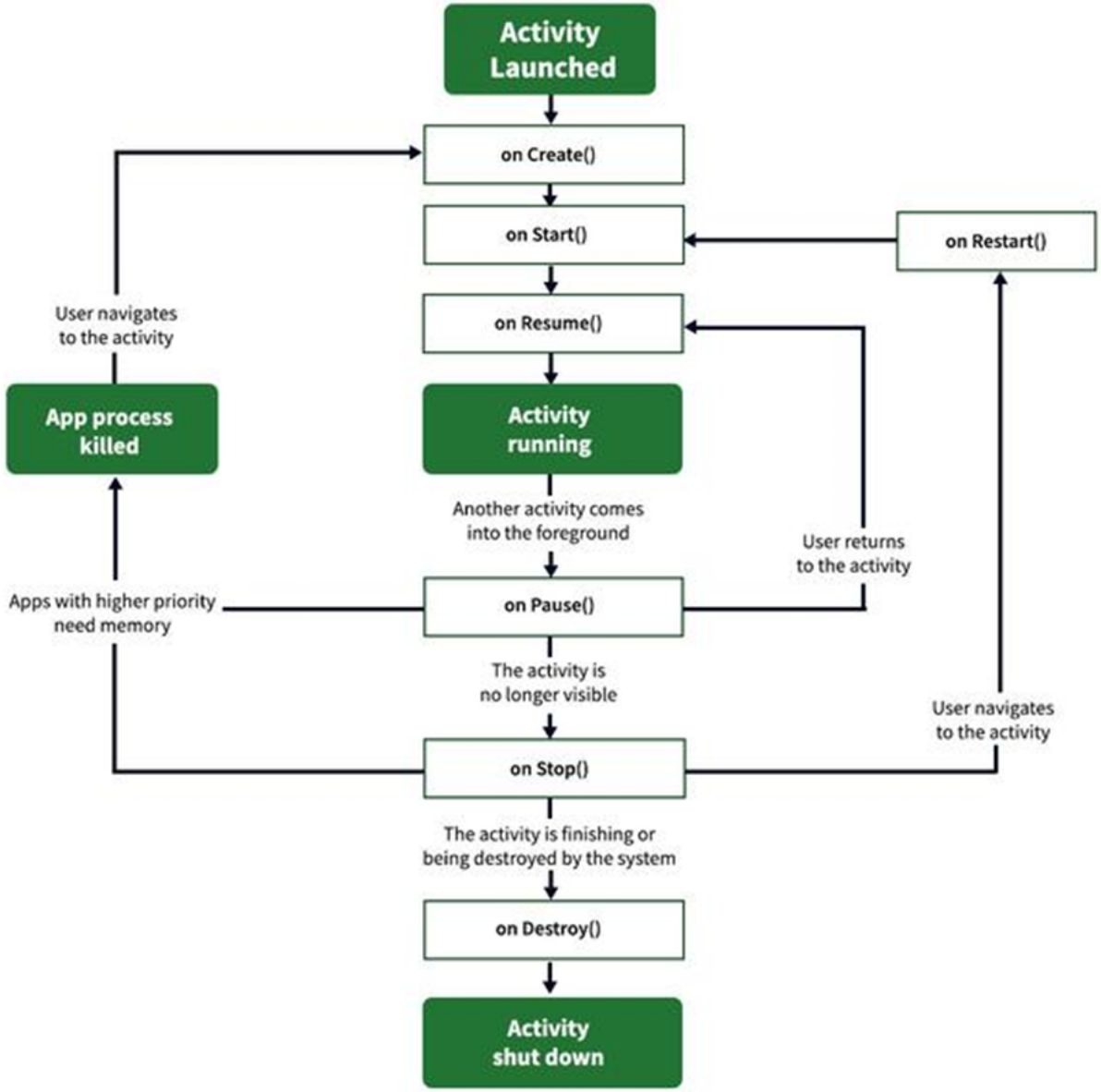
- Android Button
- Android Toast
- Custom Toast
- ToggleButton
- CheckBox
- AlertDialog
- Spinner
- AutoCompleteTextView
- RatingBar
- DatePicker
- TimePicker
- ProgressBar
- TextView
- EditText

etc....

Lifecycle of Activity in Android App

Kotlin

for Android App Development



Activity Lifecycle in Android

States of Android Lifecycle:

- 1. OnCreate: This is called when activity is first created.
- 2. OnStart: This is called when the activity becomes visible to the user.
- 3. OnResume: This is called when the activity starts to interact with the user.
- 4. OnPause: This is called when activity is not visible to the user.
- 5. OnStop: This is called when activity is no longer visible.
- 6. OnRestart: This is called when activity is stopped, and restarted again.
- 7. OnDestroy: This is called when activity is to be closed or destroyed.

Main_Activity.kt

```
import android.os.Bundle
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {

        override fun onCreate(savedInstanceState: Bundle?) {
                super.onCreate(savedInstanceState)
                setContentView(R.layout.activity_main)

                val toast = Toast.makeText(applicationContext, "onCreate Called", Toast.LENGTH_LONG).show()
        }

        override fun onStart() {
                super.onStart()
                val toast = Toast.makeText(applicationContext, "onStart Called", Toast.LENGTH_LONG).show()
        }

        override fun onRestart() {
                super.onRestart()
                val toast = Toast.makeText(applicationContext, "onRestart Called", Toast.LENGTH_LONG).show()
        }
```

```
override fun onPause() {
        super.onPause()
        val toast = Toast.makeText(applicationContext, "onPause Called", Toast.LENGTH_LONG).show()
}

override fun onResume() {
        super.onResume()
        val toast = Toast.makeText(applicationContext, "onResume Called", Toast.LENGTH_LONG).show()
}

override fun onStop() {
        super.onStop()
        val toast = Toast.makeText(applicationContext, "onStop Called", Toast.LENGTH_LONG).show()
}

override fun onDestroy() {
        super.onDestroy()
        val toast = Toast.makeText(applicationContext, "onDestroy Called", Toast.LENGTH_LONG).show()
}
}
```


Kotlin
for Android App Development

1. MVC (Model View Controller)

MVC or Model View Controller breaks the model into three main components Model that stores the application data, View UI layer that holds the component visible on the screen and Controller that establishes the relationship between Model and the View.

2. MVP (Model View Presenter)

To avoid complexities like maintainability, readability, scalability, and refactoring of applications we use MVP model. The basic working of this model relies of the points mentioned below:

- Communication between the View-Presenter and Presenter-Model happens with the help of Interface (also called Contract).
- There is One to One relationship between Presenter and View, One Presenter Class only manages One View at a time.
- Model and View doesn't have any knowledge about each other.

3. MVVM (Model View ViewModel):

MVVM or Model View ViewModel as the name suggest like MVC model it contains three components too Model, View and ViewModel. Features of MVVM model are mentioned below:

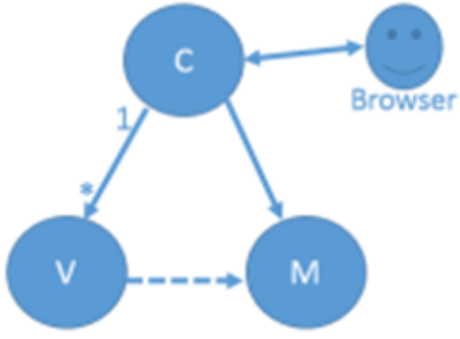
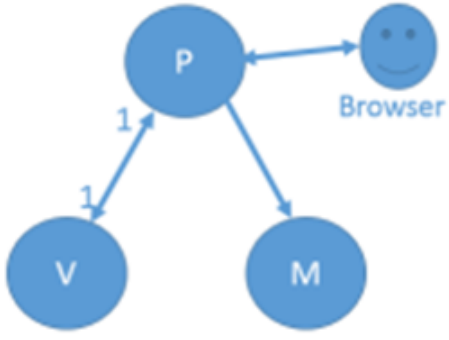
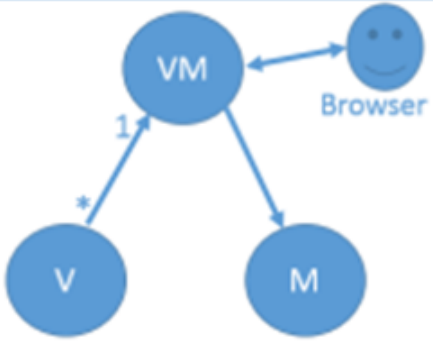
- ViewModel does not hold any kind of reference to the View.
- Many to-1 relationships exist between View and ViewModel.
- No triggering methods to update the View.

And we can achieve this using 2 methods:

- Using DataBinding library of Google.
- Using Tools like RxJava for Data Binding.

Kotlin

for Android App Development

MVC	MVP	MVVM
		
• Controller is the entry point to the application • One to Many relationship between Controller and View • View does not have reference to the Controller • View is very well aware of the Model • Smalltalk, ASP.Net MVC	• View is the entry point to the application • One to One mapping between View and Presenter • View have the reference to the Presenter • View is not aware of the Model • Windows forms	• View is the entry point to the application • One to Many relationship between View and ViewModel • View have the reference to the View Model • View is not aware of the Model • Silverlight, WPF, HTML5 with Knockout/AngularJS