

# Kotlin

A decorative horizontal bar composed of various colored segments (black, blue, yellow, light blue, dark blue, teal) is positioned above the text.

Presented by  
VAISHALI TAPASWI

**FANDS INFONET Pvt.Ltd.**

**[www.fandsindia.com](http://www.fandsindia.com)**

**[fands@vsnl.com](mailto:fands@vsnl.com)**

# Ground Rules

- ❑ **Turn off cell phone. If you cannot, please keep it on silent mode. You can go out and attend your call.**
- ❑ **If you have questions or issues, please let me know immediately.**
- ❑ **Let us be punctual.**

A decorative vertical bar on the left side of the slide, composed of numerous horizontal segments in various shades of blue, black, and yellow, creating a striped effect.

# Agenda

# Effectively using Functional Programming



# Functional Programming

- ❑ Functional programming languages are specially designed to handle symbolic computation and list processing applications. Functional programming is based on mathematical functions. Some of the popular functional programming languages include: Lisp, Python, Erlang, Haskell, Clojure, etc.
- ❑ Two Categories
  - Pure Functional Languages – These types of functional languages support only the functional paradigms. For example – Haskell.
  - Impure Functional Languages – These types of functional languages support the functional paradigms and imperative style programming. For example – LISP.

# Characteristics

- ❑ Functional programming languages are designed on the concept of mathematical functions that use conditional expressions and recursion to perform computation.
- ❑ Functional programming supports higher-order functions and lazy evaluation features.
- ❑ Functional programming languages don't support flow Controls like loop statements and conditional statements like If-Else and Switch Statements. They directly use the functions and functional calls.
- ❑ Like OOP, functional programming languages support popular concepts such as Abstraction, Encapsulation, Inheritance, and Polymorphism.

# Advantages

- ❑ Bugs-Free Code – Functional programming does not support state, so there are no side-effect results and we can write error-free codes.
- ❑ Efficient Parallel Programming – Functional programming languages have NO Mutable state, so there are no state-change issues. One can program "Functions" to work parallel as "instructions". Such codes support easy reusability and testability.
- ❑ Efficiency – Functional programs consist of independent units that can run concurrently. As a result, such programs are more efficient.
- ❑ Supports Nested Functions – Functional programming supports Nested Functions.
- ❑ Lazy Evaluation – Functional programming supports Lazy Functional Constructs like Lazy Lists, Lazy Maps, etc.

# Disadvantages

- ❑ As a downside, functional programming requires a large memory space. As it does not have state, you need to create new objects every time to perform actions.
- ❑ Functional Programming is used in situations where we have to perform lots of different operations on the same set of data.
- ❑ Lisp is used for artificial intelligence applications like Machine learning, language processing, Modeling of speech and vision, etc.
- ❑ Embedded Lisp interpreters add programmability to some systems like Emacs.



# OOP Vs FP

OOP	Functional Programming
Uses Mutable data.	Uses Immutable data.
Follows Imperative Programming Model.	Follows Declarative Programming Model.
Focus is on "How you are doing"	Focus is on: "What you are doing"
Not suitable for Parallel Programming	Supports Parallel Programming
Its methods can produce serious side effects.	Its functions have no-side effects
Flow control is done using loops and conditional statements.	Flow Control is done using function calls & function calls with recursion
It uses "Loop" concept to iterate Collection Data. For example: For-each loop in Java	It uses "Recursion" concept to iterate Collection Data.
Execution order of statements is very important.	Execution order of statements is not so important.
Supports only "Abstraction over Data".	Supports both "Abstraction over Data" and "Abstraction over Behavior".

# Working with Nulls

A decorative horizontal bar with a wavy, undulating shape, composed of various colored segments including black, blue, yellow, and teal, spanning the width of the slide.

## Null Safety

## Programming for null safety

# Null Safety

- ❑ Compiler by default doesn't allow any types to have a value of null at compile-time.
- ❑ For Kotlin, Nullability is a type. At a higher level, a Kotlin type fits in either of the two
  - Nullability Type
    - `var a : String? = null`
  - Non-Nullability Type
- ❑ `var streetName : String? = address?.locality?.street?.addressLine1`

# Operators

## ❑ !! Operator

- `a = null`
- `println(a!!.length)` // runtime error.

## ❑ Elvis Operator (?:0)

- Up until now, whenever the safe call operator returns a null value, we don't perform an action and print null instead. The Elvis operator `?:` allows us to set a default value instead of the null
- `print(newStr?.length?:0)`

## ❑ Safe Casting

- The `?` operator can be used for preventing `ClassCastException` as well
- `var b : String = "2"`
- `var x : Int? = b as? Int`
- `println(x)` //prints null

# Getting Functional

A decorative horizontal bar with a wavy, segmented pattern in various shades of blue, teal, and yellow, spanning the width of the slide.

Higher-Order Functions in Kotlin

Lambda Expressions in Kotlin

Closures in Kotlin

Extension Functions in Kotlin

Operator Overloading

Object Serialization

# Higher-Order Functions (HOF)

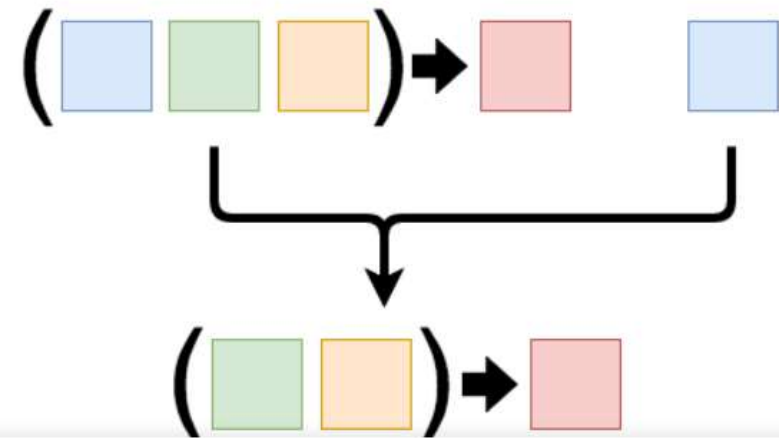
- A function which takes a function as an argument and/or returns a function.
  - `const filter = (predicate, xs) => xs.filter(predicate)`
  - `const is = (type) => (x) => Object(x) instanceof type`
  - `filter(is(Number), [0, '1', 2, null]) // [0, 2]`

# Closure

```
const addTo = (x) => {  
  return (y) => {  
    return x + y  
  }  
}
```

- A closure is a scope which retains variables available to a function when it's created. This is important for partial application to work.
- Usage
  - var addToFive = addTo(5)
  - addToFive(3) // => 8

# Partial Application



- Is a technique of fixing a number of arguments to a function, producing another function of smaller arguments i.e binding values to one or more of those arguments as the chain of function progressed.



# Partial Application

- Partially applying a function means creating a new function by pre-filling some of the arguments to the original function.

```
fun discShm(mymap:Map<String,
Int>):Int{
    return gettotal3(mymap,
discCalc3)
}
```

```
val discCalc3 = { sum: Int,
percent: Int ->.....}
```

```
fun gettotal3(mymap:Map<String,
Int>,disc:(Int,Int)->Double ):Int{
    ....
}
```

# Currying

```
const sum = (a, b) => a + b
```

```
const curriedSum = (a) => (b) => a + b
```

```
curriedSum(40)(2) // 42.
```

```
const add2 = curriedSum(2) // (b) => 2 + b
```

```
add2(10) // 12
```

- The process of converting a function that takes multiple arguments into a function that takes them one at a time.
- Each time the function is called it only accepts one argument and returns a function that takes one argument until all arguments are passed.

# Auto Currying

```
const add = (x, y) => x + y
```

```
const curriedAdd = _.curry(add)
```

```
curriedAdd(1, 2) // 3
```

```
curriedAdd(1) // (y) => 1 + y
```

```
curriedAdd(1)(2) // 3
```

- Transforming a function that takes multiple arguments into one that if given less than its correct number of arguments returns a function that takes the rest. When the function gets the correct number of arguments it is then evaluated.

# Function Composition

- The act of putting two functions together to form a third function where the output of one function is the input of the other.

```
const compose = (f, g) => (a) => f(g(a)) // Definition
const floorAndToString = compose((val) => val.toString(),
Math.floor) // Usage
floorAndToString(121.212121) // '121'
```

# Purity

- A function is pure if the return value is only determined by its input values, and does not produce side effects.

```
const greet = (name) =>  
  `Hi, ${name}`
```

```
greet('Brianne') // 'Hi,  
Brianne'
```

```
window.name = 'Brianne'
```

```
const greet = () => `Hi,  
${window.name}`  
greet() // "Hi, Brianne"
```

# Morphism, Endomorphism

- Morphism - A transformation function
- Endomorphism - A function where the input type is the same as the output

```
// uppercase :: String -> String
const uppercase = (str) => str.toUpperCase()
// decrement :: Number -> Number
const decrement = (x) => x - 1
```

# Isomorphism

- Isomorphism - A pair of transformations between 2 types of objects that is structural in nature and no data is lost.

// Providing functions to convert in both directions makes them isomorphic.

```
const pairToCoords = (pair) => ({x: pair[0], y: pair[1]})  
const coordsToPair = (coords) => [coords.x, coords.y]  
coordsToPair(pairToCoords([1, 2])) // [1, 2]  
pairToCoords(coordsToPair({x: 1, y: 2})) // {x: 1, y: 2}
```

# Extensions

- Similar to C# and Gosu, the ability to extend a class with new functionality without having to inherit from the class or use any type of design pattern such as Decorator
- Done using special declarations called extensions. Kotlin supports extension functions and extension properties.



# Extensions

- To declare an extension function, we need to prefix its name with a *receiver type*, i.e. the type being extended.
- Note : **Extensions are resolved statically**

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {  
    val tmp = this[index1] // 'this' corresponds to the  
    list  
    this[index1] = this[index2]  
    this[index2] = tmp  
}
```

# InFix

```
fun main(args: Array<String>) {
    val s:String = "aaa"
    println(s)
    println(s.shout("ss"))
    println(s shout "AAA")
}

infix fun String.shout(s:String):String
{return this.toUpperCase()+s}
```

# Lambda

- Ways of Instantiating a function type
  - Code Block
    - a lambda expression: `{ a, b -> a + b },`
    - an anonymous function: `fun(s: String): Int { return s.toIntOrNull() ?: 0 }`
    - Function literals with receiver can be used as values of function types with receiver.
  - Callable reference
    - a top-level, local, member, or extension function: `::isOdd,` `String::toInt,`
    - a top-level, member, or extension property: `List<Int>::size`

# Operator Overloading

- To override an operator

```
interface IndexedContainer {
    operator fun get(index: Int)
}
```

- Check operators

<https://kotlinlang.org/docs/operator-overloading.html>

Expression	Translated to
<code>+a</code>	<code>a.unaryPlus()</code>
<code>-a</code>	<code>a.unaryMinus()</code>
<code>!a</code>	<code>a.not()</code>
<code>a++</code>	<code>a.inc()</code>
<code>a--</code>	<code>a.dec()</code>
<code>a..b</code>	<code>a.plus(b)</code>
<code>a-b</code>	<code>a.minus(b)</code>
<code>a * b</code>	<code>a.times(b)</code>
<code>a / b</code>	<code>a.div(b)</code>
<code>a % b</code>	<code>a.rem(b)</code>
<code>a..b</code>	<code>a.rangeTo(b)</code>

# Serialization

## □ Serialization

- process of converting data used by an application to a format that can be transferred over a network or stored in a database or a file.

## □ Deserialization

- opposite process of reading data from an external source and converting it into a runtime object.

## □ Popular Formats

- Json
- XML

# Json Serialization

```
import kotlinx.serialization.Serializable
import kotlinx.serialization.decodeFromString
import kotlinx.serialization.encodeToString
import kotlinx.serialization.json.Json

@Serializable
data class Name(val fname: String, val lname: String)

fun main() {
    val json = Json.encodeToString(Name( fname: "Vaishali", lname: "Tapaswi"))
    println(json)
    val json1 = """"{"fname":"Fands", "lname":"Infonet"}""""

    val data = Json.decodeFromString<Name>(json1)
    println(data)
}
```

# Co-Routines (without KTOR)

What is co-routine  
Managing long running tasks  
Kotlin Flows

# Kotlin and Concurrency

- ❑ Kotlin provides only minimal low-level APIs in its standard library to enable various other libraries to utilize coroutines. Unlike many other languages with similar capabilities, `async` and `await` are not keywords in Kotlin and are not even part of its standard library.
- ❑ Kotlin's concept of suspending function provides a safer and less error-prone abstraction for asynchronous operations than futures and promises.



# Coroutine basics

## □ Modify Gradle

- implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.2.1'

## □ Write code to test

- Lifecycle
- JVM
- main method
- Thread.sleep
- Delay

```
fun main() {  
    GlobalScope.launch {  
        delay(1000L)  
        println("World!")  
    }  
    println("Hello,")  
    Thread.sleep(1500)  
}
```

# What is happening?

- Essentially, coroutines are light-weight threads. They are launched with launch coroutine builder in a context of some CoroutineScope. Here we are launching a new coroutine in the GlobalScope, meaning that the lifetime of the new coroutine is limited only by the lifetime of the whole application.

# Thread.sleep Vs Delay

- Delay is a special suspending function that does not block a thread, but suspends coroutine and it can be only used from a coroutine
- Sleep is blocking

# Blocking Vs non-blocking

```
import kotlinx.coroutines.*
```

```
fun main() {
```

```
    GlobalScope.launch { // launch a new coroutine in background  
    and continue
```

```
        delay(1000L)
```

```
        println("World!")
```

```
    }
```

```
    println("Hello,") // main thread continues here immediately
```

```
    runBlocking { // but this expression blocks the main thread
```

```
        delay(2000L) // ... while we delay for 2 seconds to keep JVM  
alive
```

```
    }
```

```
}
```

# Blocking calls

```
class MyTest {  
    @Test  
    fun testMySuspendingFunction() = runBlocking<Unit>  
    {  
        // here we can use suspending functions using any  
        assertion style that we like  
    }  
}
```

# Waiting for a job

- wait (in a non-blocking way) until the background job that we have launched is complete:

```
val job = GlobalScope.launch { // launch a new coroutine  
    and keep a reference to its Job  
        delay(1000L)  
        println("World!")  
    }  
println("Hello,")  
job.join() // wait until child coroutine completes
```

# Introduction to Async Programming

A decorative horizontal bar with a wavy, torn-edge effect is positioned below the title. It is composed of various colored segments including black, blue, light blue, yellow, and teal.

Scenarios

Implementation Styles

# Asynchronous Programming Techniques

- Threading
- Callbacks
- Futures, promises, and others
- Reactive Extensions
- Co-routines



# Delegation

Delegation pattern Vs inheritance  
Property Delegation  
Overriding a member of an  
interface implemented by  
delegation

# Delegation

- Inheritance Vs Aggregation
- Inheritance
  - Tightly Coupled Relationship
  - Single Class can be inherited
- Delegation in Kotlin
  - Loosely Coupled
  - Dynamic

# Implementation by Delegation

- The Delegation pattern has proven to be a good alternative to implementation inheritance, and Kotlin supports it natively requiring zero boilerplate code. A class Derived can implement an interface Base by delegating all of its public members to a specified object:

# Implementation by Delegation

```
interface Base {  
    fun print()  
}  
  
class BaseImpl(val x: Int) : Base {  
    override fun print() { print(x) }  
}  
  
class Derived(b: Base) : Base by b  
  
fun main() {  
    val b = BaseImpl(10)  
    Derived(b).print()  
}
```

# Delegated Properties

- With some common kinds of properties, even though you can implement them manually every time you need them, it is more helpful to implement them once, add them to a library, and reuse them later. For example:
  - Lazy properties: the value is computed only on first access.
  - Observable properties: listeners are notified about changes to this property.
  - Storing properties in a map instead of a separate field for each property.

# Lazy Properties

```
val CheckLazyValue: String by lazy {  
    println("computed!")  
    "Hello" ^lazy  
}
```

- lazy() is a function that takes a lambda and returns an instance of Lazy<T>, which can serve as a delegate for implementing a lazy property.
- The first call to get() executes the lambda passed to lazy() and remembers the result. Subsequent calls to get() simply return the remembered result.
- By default Synchronized evaluation

```
var name: String by Delegates.observable( initialValue: "<no name>" ) {  
    prop, old, new ->  
        println("$old -> $new")  
}
```

# Observable Properties

- Delegates.observable() takes two arguments: the initial value and a handler for modifications.
- The handler is called every time you assign to the property (after the assignment has been performed). It has three parameters: the property being assigned to, the old value, and the new value:

## Storing properties in a map

```
class User1(val map: MutableMap<String, Any?>) {  
    var name: String by map  
    var age: Int by map  
}
```

- Storing the values of properties in a map is often handy in applications for things like parsing JSON or performing other dynamic tasks. In this case, you can use the map instance itself as the delegate for a delegated property.
- Map or MutableMap depending on requirements



# Reflection

What and why reflection?  
Writing code to find class  
properties

# Reflection

- ❑ Introspecting the structure of program at runtime
- ❑ Dynamic Code
- ❑ Kotlin makes functions and properties first-class citizens in the language, and introspecting them (i.e. learning a name or a type of a property or function at runtime) is closely intertwined with simply using a functional or reactive style.

# Important Note

- ❑ On the Java platform, the runtime component required for using the reflection features is distributed as a separate JAR file (kotlin-reflect.jar). This is done to reduce the required size of the runtime library for applications that do not use reflection features. If you do use reflection, please make sure that the .jar file is added to the classpath of your project.
- ❑ Modify gradle file (module app)
  - implementation "org.jetbrains.kotlin:kotlin-reflect:\$kotlin\_version"

# Reflection

```
class Maths {  
    fun sqr(no : Int):Int {  
        return no*no;  
    }  
}
```

```
val c = Maths::class  
println("List of Constructors ...");  
c.constructors.forEach{ println(it)}  
println("\n\nList of Members ...");  
c.members.forEach{println(it)}  
val inst = Maths();  
val ret = c.members  
                .filter{it.name=="sqr"}.first().call(inst,10);  
println("returned from call = "+ ret)
```


# Reflection and Annotations

```
@Target(AnnotationTarget.CLASS)
@Retention(AnnotationRetention.RUNTIME)
@MustBeDocumented
@Repeatable
annotation class Author(val name : String)
```

```
@Author("Vaishali")
class MyClass {
    fun getHelloString(): String {
        return "Hello World!"
    }
}
```

```
fun main() {
    val myClass = MyClass::class
    val annotation =
        myClass.findAnnotation
            <Author>()
    println( annotation?.name)
}
```

# Kotlin Library creation

A decorative horizontal bar composed of various colored segments (black, blue, yellow, cyan, light blue, dark blue, grey) is positioned below the title.

---

Kotlin on different platforms  
Creating Multi-platform libraries  
Sharing code and invoking from  
the different application

# Library

- Purpose
  - Code Sharing
  - Widgets
- Supported Platforms
  - Native
  - JVM
  - JS
  - Android
  - iOS

# Library

- Scope
  - Within project
  - Across projects
  - Public publishing
    - Maven Repository
    - Android Library



# Creating Android project with

No activity

Simple activity

Simple UI

# Retrofit

How to invoke http API

How to invoke http API using  
Retrofit

# Retrofit

- REST Recap
  - REST Client is a method or a tool to invoke a REST service API that is exposed for communication by any system or service provider.
- Retrofit is a REST Client for Java and Android.
  - if an API is exposed to get real-time traffic information about a route from Google, the software/tool that invokes the Google traffic API is called the REST client. It makes it relatively easy to retrieve and upload JSON (or other structured data) via a REST-based webservice.

# Invoking Kotlin code

Directly adding function code for button click

Invoking code in Async manner

Adding code to connect to SQLite

Adding code to connect to remote http server  
and render the contents

Adding code to use Room to connect to  
SQLite

# Developing lifecycle aware components

A decorative horizontal bar spans the width of the slide, just below the title. It is composed of a series of colored rectangular segments in various shades of blue, teal, yellow, and black, arranged in a slightly wavy pattern.

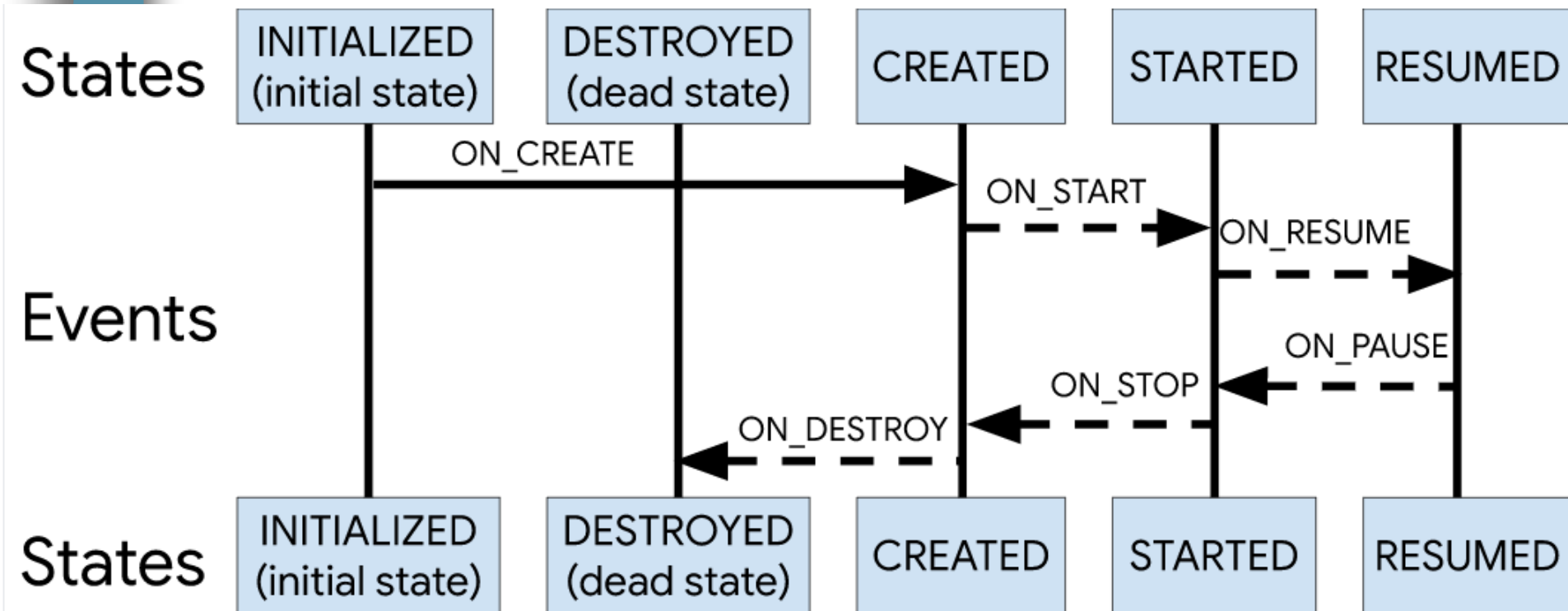
Handle Lifecycle

ViewModel

LiveData

Save UI states

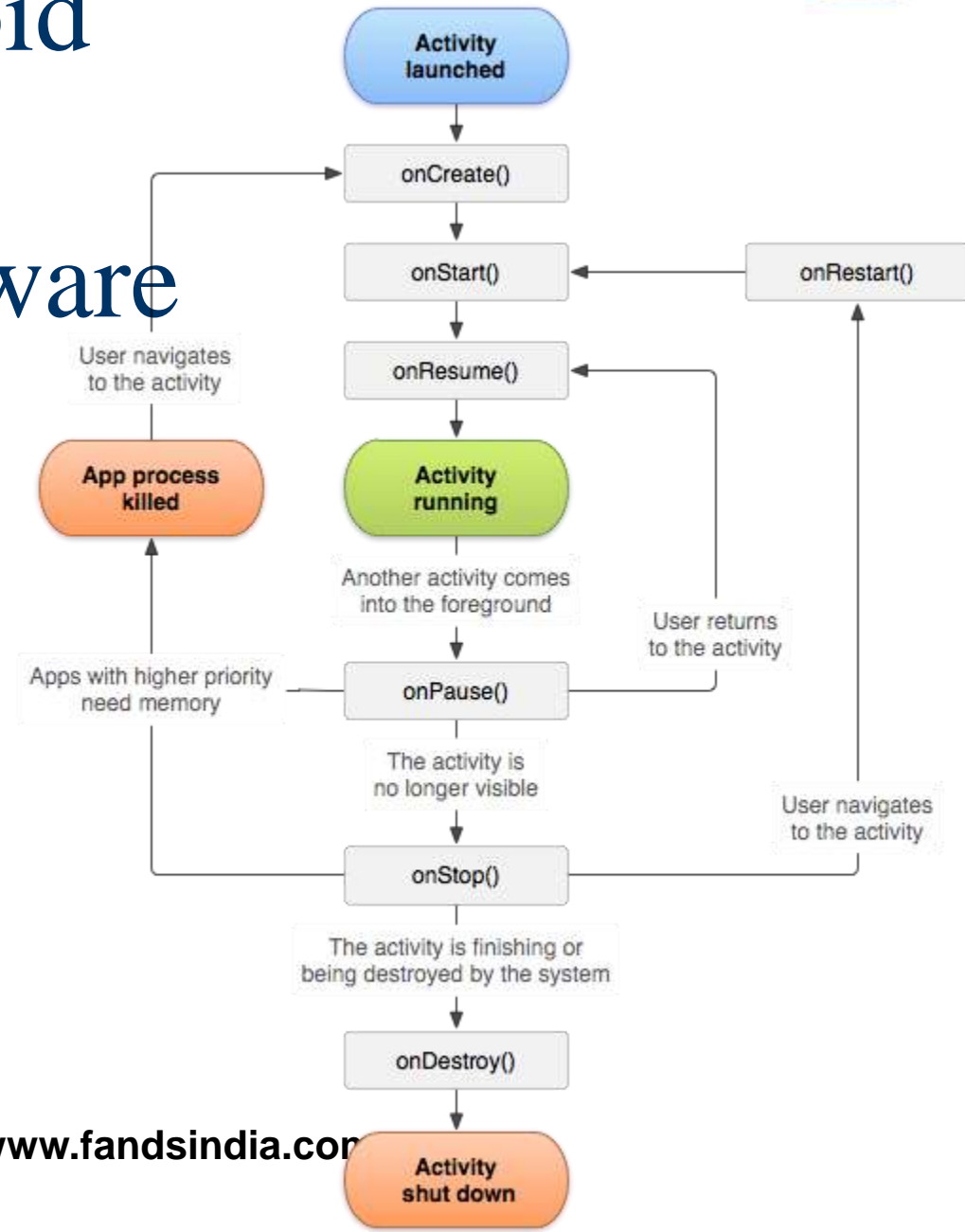
# Handling Lifecycles with Lifecycle-Aware Components



# LifeCycle

- Lifecycle is a class that holds the information about the lifecycle state of a component (like an activity or a fragment) and allows other objects to observe this state.
- Lifecycle uses two main enumerations to track the lifecycle
- Event
  - The lifecycle events that are dispatched from the framework and the Lifecycle class. These events map to the callback events in activities and fragments.
- State
  - The current state of the component tracked by the Lifecycle object.

# Using Android Jetpack Lifecycle-Aware Components

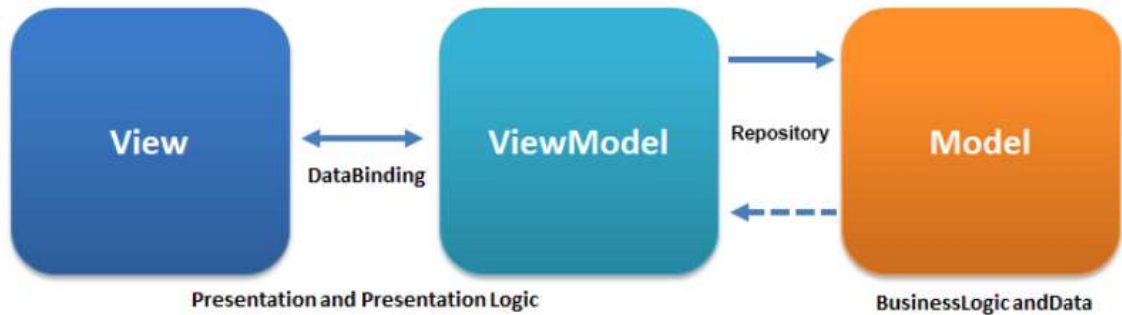




# MVVM

- MVVM stands for Model View ViewModel.
- One of the best design patterns that can be used to develop an Android App but what makes MVVM more powerful are the components

# MVVVM



## □ Model

- contains all the data classes, database classes, API and repository

## □ View

- is the UI part that represents the current state of information that is visible to the user.

## □ ViewModel

- it contains the data required in the View and translates the data which is stored in Model which then can be present inside a View. ViewModel and View are connected through Databinding and the observable LiveData.

# MVVM Benefits

- ❑ It separates the business logic with the presentation logic
- ❑ The View is not aware of all the computation happening behind the scenes, this makes the ViewModel reusable.
- ❑ When using repository between ViewModel and Model, ViewModel only knows how to make data request and repository will take care of how the data will be fetched, whether it may be from APIs or the local DB.
- ❑ Since we are using ViewModel and LiveData which are Activity lifecycle aware, it will lead to fewer crashes and memory leaks

# LiveData

- ❑ LiveData is an observable data holder class. Unlike other observables, it is lifecycle aware i.e. it is aware of the lifecycle of the other app components such as activities, fragments or services. This means it only updates app component observers when they are in an active lifecycle state.

# ViewModel

- The ViewModel class is designed to store and manage UI-related data in a lifecycle conscious way. The ViewModel class allows data to survive configuration changes such as screen rotations.

# Saving UI States

- Preserving and restoring an activity's UI state in a timely fashion across system-initiated activity or application destruction is a crucial part of the user experience. In these cases the user expects the UI state to remain the same, but the system destroys the activity and any state stored in it.

# Saving UI States

- ❑ To bridge the gap between user expectation and system behavior, use a combination of ViewModel objects, the `onSaveInstanceState()` method, and/or local storage to persist the UI state across such application and activity instance transitions.
- ❑ Deciding how to combine these options depends on the complexity of your UI data use cases for your app and

	ViewModel	Saved instance state	Persistent storage
Storage location	in memory	serialized to disk	on disk or network
Survives configuration change	Yes	Yes	Yes
Survives system-initiated process death	No	Yes	Yes
Survives user complete activity dismissal/onFinish()	No	No	Yes
Data limitations	complex objects are fine, but space is limited by available memory	only for primitive types and simple, small objects such as String	only limited by disk space or cost / time of retrieval from the network resource
Read/write time	quick (memory access only)	slow (requires serialization/deserialization and disk access)	slow (requires disk access or network transaction)



# Developing Data layer Libraries

A decorative horizontal bar with a wavy, torn-edge effect is positioned below the title. It is composed of various colored segments including black, blue, light blue, yellow, and teal.

DataStore

WorkManager

Dependency Injection (Koin)

MVI architecture

# DataStore

- Jetpack DataStore is a data storage solution that allows you to store key-value pairs or typed objects with protocol buffers. DataStore uses Kotlin coroutines and Flow to store data asynchronously, consistently, and transactionally.

# Two Implementations

- DataStore provides two different implementations
  - Preferences DataStore stores and accesses data using keys. This implementation does not require a predefined schema, and it does not provide type safety.
  - Proto DataStore stores data as instances of a custom data type. This implementation requires you to define a schema using protocol buffers, but it provides type safety.

# WorkManager

- ❑ WorkManager is an API that makes it easy to schedule reliable, asynchronous tasks those can run even if app exits or device restarts.
- ❑ Suitable and recommended replacement for all previous Android background scheduling APIs, including FirebaseJobDispatcher, GcmNetworkManager, and Job Scheduler.
- ❑ Incorporates the features of its predecessors in a modern, consistent API that works back to API level 14 while also being conscious of battery life.

# Features

- Work Constraints
  - Declaratively define the optimal e.g. run only when the device is on Wi-Fi, when the device is idle
- Robust Scheduling
  - run one-time or repeatedly using flexible scheduling windows. Work can be tagged and named as well, allowing you to schedule unique, replaceable work and monitor or cancel groups of work together.
- Flexible Retry Policy
  - Sometimes work fails, configurable exponential backoff policy.
- Work Chaining
  - chain individual work tasks together using a fluent, natural, interface that allows you to control sequential and parallel executions
- Built-In Threading Interoperability
  - integrates with RxJava and Coroutines and provides the flexibility to plug in your own asynchronous APIs.

# Reliability

- ❑ WorkManager is intended for work that required to run reliably even if the user navigates off a screen, the app exits, or the device restarts. For example:
  - Sending logs or analytics to backend services
  - Periodically syncing application data with a server
- ❑ WorkManager is not intended for in-process background work that can safely be terminated if the app process goes away or for work that requires immediate execution.

# Koin

- Koin is a DI framework for Kotlin developers, completely written in Kotlin.
- It is very light weighted. It supports the Kotlin DSL feature. It is one of the easy DI frameworks which doesn't require a steep learning curve to get hold of it.

# Terminologies

- While working with Koin, there are few terminologies we need to understand before getting started.
  - module - it creates a module in Koin which would be used by Koin to provide all the dependencies.
  - single - it creates a singleton that can be used across the app as a singular instance.
  - factory - it provides a bean definition, which will create a new instance each time it is injected.
  - get() - it is used in the constructor of a class to provide the required dependency.



# What is MVI architecture?

- MVI stands for Model-View-Intent. This pattern has been introduced recently in Android. It works based on the principle of unidirectional and cylindrical flow inspired by the Cycle.js framework.
  - Model: Unlike other patterns, In MVI Model represents the state of the UI. i.e for example UI might have different states like Data Loading, Loaded, Change in UI with user Actions, Errors, User current screen position states.
  - View: The View in the MVI is our Interfaces which can be implemented in Activities and fragments.
  - Intent: Even though this is not an Intent as termed by Android from before. The result of the user actions is passed as an input value to Intents. In turn, we can say we will be sending models as inputs to the Intents which can load it through Views.

# Complete android project using Kotlin

## CRUD Operations

- Create
- Read/Retrieve
- Update
- Delete

# Effectively manage Application

A decorative horizontal bar composed of various colored segments (black, blue, yellow, cyan, light blue, dark blue, grey) is positioned below the main title.

Navigation  
Paging

# Navigation

- ❑ Refers to the interactions that allow users to navigate across, into, and back out from the different pieces of content within your app.
- ❑ Android Jetpack's Navigation component helps you implement navigation, from simple button clicks to more complex patterns, such as app bars and the navigation drawer.
- ❑ The Navigation component also ensures a consistent and predictable user experience by adhering to an established set of principles.

# Navigation

- Navigation is a framework for navigating between 'destinations' within an Android application that provides a consistent API whether destinations are implemented as Fragments, Activities, or other components.
- The Navigation Component consists of three key parts:
  - 1. Navigation Graph (New XML resource) - contains navigation-related information in one centralized location. This includes all the places known as destinations, and possible paths a user could take through your app.
  - 2. NavHostFragment (Layout XML view) - special widget you add to your layout. It displays different destinations from your Navigation Graph.
  - 3. NavController (Kotlin/Java object) - keeps track of the current position within the navigation graph. It orchestrates swapping destination content in the NavHostFragment as you move through a navigation graph.

# Paging Libraries

- Help you load and display small chunks of data at a time. Loading partial data on demand reduces usage of network bandwidth and system resources.
- Supports different data architectures
  - Served only from a backend server.
  - Stored only in an on-device database.
  - A combination of the other sources, using the on-device database as a cache.

# Android Jetpack

- Jetpack is a suite of libraries to help developers follow best practices, reduce boilerplate code, and write code that works consistently across Android versions and devices so that developers can focus on the code they care about.

# Jetpack libraries (Popular)

<a href="#"><u>activity</u></a> *	Access composable APIs built on top of Activity.
<a href="#"><u>appcompat</u></a> *	Allows access to new APIs on older API versions of the platform (many using Material Design).
<a href="#"><u>appsearch</u></a> *	Build custom in-app search capabilities for your users.
<a href="#"><u>camera</u></a> *	Build mobile camera apps.
<a href="#"><u>compose</u></a> *	Define your UI programmatically with composable functions that describe its shape and data dependencies.
<a href="#"><u>databinding</u></a> *	Bind UI components in your layouts to data sources in your app using a declarative format.
<a href="#"><u>fragment</u></a> *	Segment your app into multiple, independent screens that are hosted within an Activity.
<a href="#"><u>hilt</u></a> *	Extend the functionality of Dagger Hilt to enable dependency injection of certain classes from the androidx libraries.
<a href="#"><u>lifecycle</u></a> *	Build lifecycle-aware components that can adjust behavior based on the current lifecycle state of an activity or fragment.
<a href="#"><u>navigation</u></a> *	Build and structure your in-app UI, handle deep links, and navigate between screens.
<a href="#"><u>paging</u></a> *	Load data in pages, and present it in a RecyclerView.
<a href="#"><u>room</u></a> *	Create, store, and manage persistent data backed by a SQLite database.
<a href="#"><u>test</u></a> *	Testing in Android.
<a href="#"><u>work</u></a> *	Schedule and execute deferrable, constraint-based background tasks.



# QUESTION / ANSWERS



# THANKING YOU !

