
AwRI
Android
WITH
Kotlin
By Ravivarma
LEARN

Android Architecture

1. Android Architecture contains different number of components to support any device needs.
2. There are 5 major android components (Fig 1.1)
    - Applications
    - Application Framework
    - Android Runtime
    - Platform Libraries
    - Linux kernel

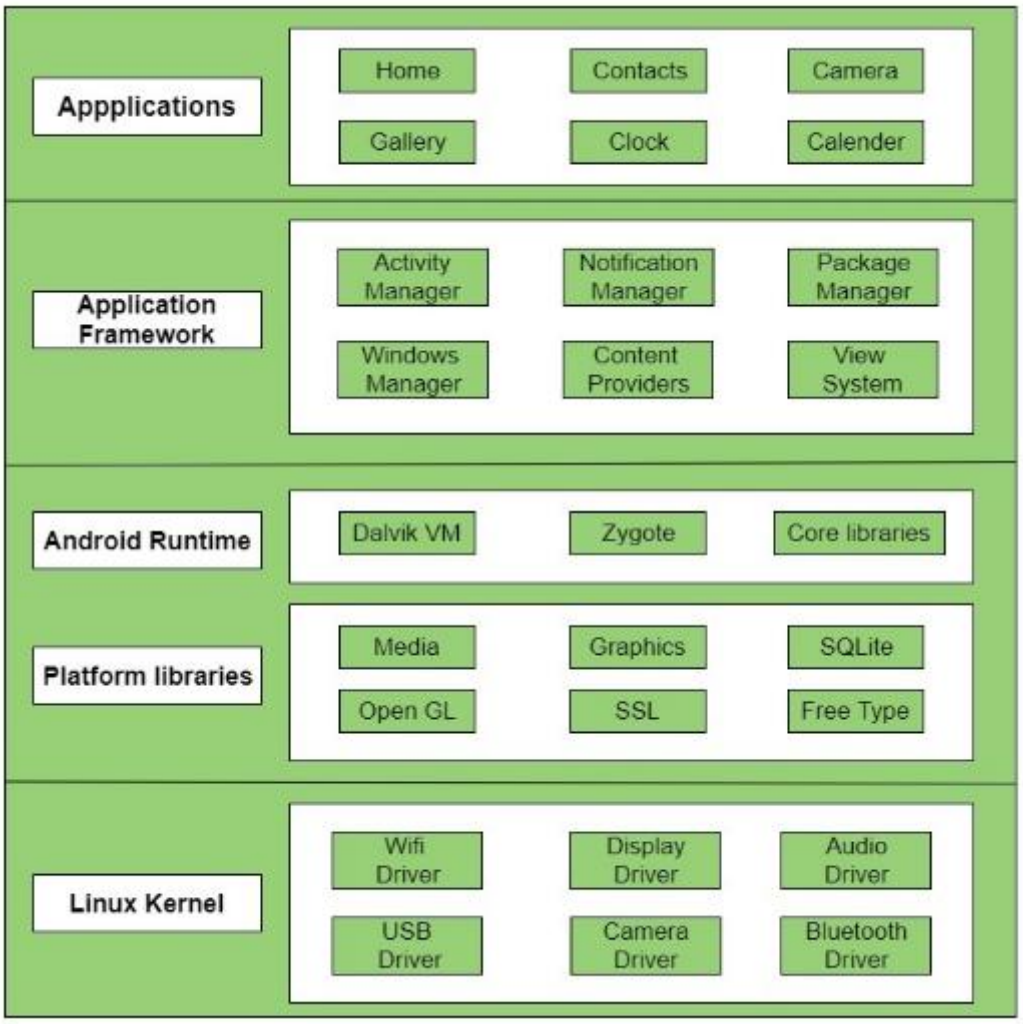


Figure 1.1

1.1 Applications

1. Applications are the top level components of android components it can be preinstalled or user installed application.
   Ex. Camera, Gallery, games, chat applications etc.

1.2 Application framework

1. Application framework provides several classes to build android apps and it provides abstraction between applications and android hardware.
2. It includes package mangers, activity mangers, view system, notification managers etc.

1.3 Android runtime

1. Android runtime contains core libraries and Dalvik virtual machine **(DVM)** to run android applications
2. DVM is virtual machine specially designed and optimized to run multiple android applications
3. Core libraries enables us to implements android apps using Java and Kotlin.

1.4 Platform libraries

1. Platform Libraries contains C/C++ and java based libraries such as sqlite, media, openGL, surface manager, SSL

1.5 Linux Kernel

1. 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.
2. The linux kernel provides abstraction between hardware and other components of android applications.
3. And linux kernel is responsible for security, memory management, power management, process management, driver model and network stack.

Android Components

Android components are the basic building blocks of android applications, there are 7 android application components

1. Activity
2. Services
3. Broadcast managers
4. Content providers
5. Widgets
6. Notifications
7. Intents

Android Activity

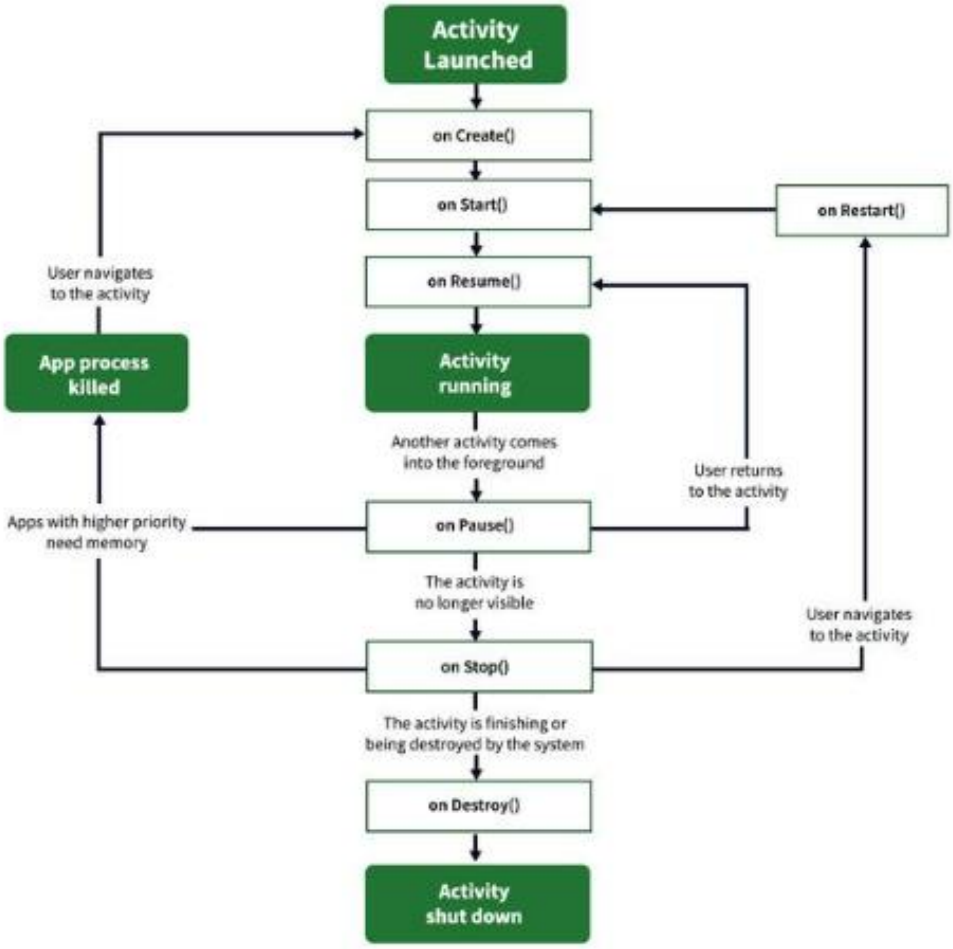
- Introduction
- Activity Life Cycle
- Use cases
- Save state of an activity
- Restore data on configuration change
- Cross questions

➢ Introduction

1. Activity is important component of android application every application must have at least one activity.
2. Using activity user can interact with application through layouts.
3. Layouts are the XML files contains views and these views will be set to activity using setContentView() method
4. setContentView() takes layout id as the argument.
   ex. setContentView(R.layout.activity)

➢ Activity Life Cycle

1. In android application we can have N number of activity each activity has its own life cycle, there are 7 life cycle methods
2. onCreate(), onStart(), onResume(), onPause(), onStop(), onDestroy(), onRestart().
3. **onCreate()** will be called once activity is created and in onCreate() most of the static work will done like creating view, binding data to list etc.
4. **onStart()** will be called once activity is visible to the user but user cannot interact with activity UI.
5. **onResume()** indicates user that activity is running in foreground so user can interact with application.
6. **onPause()** indicates that activity is going to background, and we should avoid heavy processing in this method because until call back returns from the onPause() new Activity will not be created on top of existing activity.
7. **onStop()** indicates activity is in background state and user cannot interact with the UI of that activity, this method may never be called in low memory situation.
8. **onDestroy()** is the final call to indicate that activity is completely killed and activity instance is destroyed from memory.
9. **onRestart()** is called whenever user is moving back to previous activity.



Activity Lifecycle in Android

➢ Restore Data on configuration change

1. To Restore Data on configuration change we have to override two methods onSaveInstanceState(), onRestoreInstanceState().
2. Data will be stored in the form of Bundle which store data in Key and Value pair
3. On save instance state will called after onStop() and that data will restored from onCreate() method of the activity.
4. On save instance state will not be called if user explicitly closes the activity

➢ Cross questions

1. Each life cycle method must call super class method otherwise compiler will through an error
2. Among all 7 life cycle methods onCreate() is the only method has bundle arguments
3. In android 12 exported must be set in activity tag inside manifest file **android:exported = “false”**
4. Log.d() key value must be 23 charecters
5. Activity can be started without layout file also
6. On configuration change activity on stop will be call then onRestart() onDestroy will not be called

Android Context

- **Introduction**
- **Application context**
- **Activity context**
- **Usage of Context**

Introduction

1. Context tells information about surroundings
2. It allows us to access resources and interact with other Android components by sending messages.
3. Context gives you information about your app environment.
4. By using context we access resources like drawebles, shared preferences, database
5. Intents are used to send information from one component to other components
6. There are two type of context one is application context and activity context

Application Context

1. Application context are used to get the information about the activity
2. **getApplicationContext()** is the method by which we can the app context
3. Application context is singleton object only one object will have created throughout the application.

Activity Context

1. Activity context are get the information about the activities and state of an activity
2. **getContext()** is the method to get activity context

Usage of Context

1. Load Resource Values
2. Layout Inflation
3. Start an Activity
4. Show a Dialog
5. Start a Service
6. Bind to a Service
7. Send a Broadcast
8. Register BroadcastReceiver

Android Bundles

- **Introduction**

Introduction

1. Bundles are used to send information from one component to other component in the application.
2. Information will be send in the form of Key and Value pair.
3. In bundle we can transfer all type of data ex int, string, Boolean, char, string array, Double, float.
4. In Receiving component we must use default values in getter methods to avoid NullPointerException.
5. **putInt**(String key, int value), **getInt**(String key, int value)
6. **putString**(String key, String value), **getString**(String key, String value)
7. **putStringArray**(String key, String[] value), **getStringArray**(String key, String[] value)
8. **putChar**(String key, char value), **getChar**(String key, char value)
9. **putBoolean**(String key, boolean value), **getBoolean**(String key, boolean value)

```
val bundle : Bundle? = intent.extras
var name  = bundle!!.getString( key: "NAME",  defaultValue: "default")
```

10. example of receiving component

11. Example of sending components

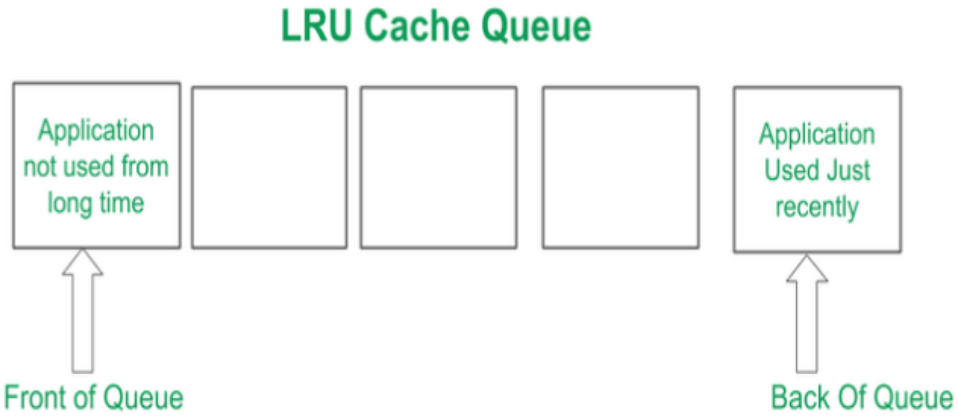
```
fun startActivity(){
    val bundle : Bundle = Bundle();
    bundle.putString("NAME","Ravivarma")
    val intent : Intent = Intent( packageContext: this@FirstActivity,SecondActivity::class.java)
    intent.putExtras(bundle);
    startActivity(intent)
}
```

Process and App life cycle

- **LRU (Least recently Used)**
- **Foreground process**
- **Visible process**
- **Service process**
- **Background process**

➢ LRU (Least recently Used

1. LRU is the cache Queue which hold list of process based on priority
2. Recently used application are place in the Back of the queue and app which is not used for long period of time will placed in Front of the queue.



➢ Foreground process

1. Foreground service are the process with it user is currently interacting or the process is waiting for the system broadcast.
2. These process are given higher priority

➢ Visible process

1. Visible processes are nothing but processes which are in **onPause()** state
2. Example of visible process are alert popup and permission pop up on top of the process

➢ Service process

1. The processes which are doing the task in background are called service process.
2. System will kill such process if it is not able to execute foreground and visible processes.

➢ Background process

1. Processes which are in **onStop()** state are called as Background processes for example if user presses home button process goes to onStop state.

Desugaring in Android

- **What is desugaring**
- **Practical implementation of desugaring**
- **How it works**

➢ What is desugaring

1. Consider an example of Time API which is introduced in API level 26 if user try to use this API in lower level of API for example API level 23 app will crash to avoid this crash Desugaring is introduced.

➢ Practical implementation of desugaring

1. Add the dependency
   coreLibraryDesugaring 'com.android.tools:desugar_jdk_libs:1.0.9'

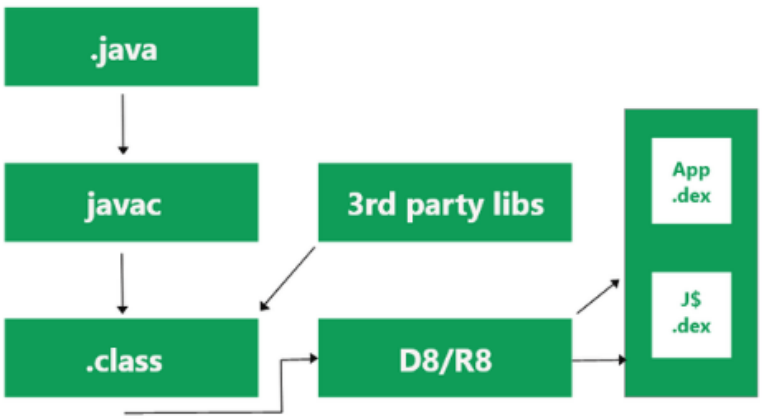
2. Add **corelibraryDesugaringEnabled true** in compileOption block

```
compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
    coreLibraryDesugaringEnabled true
}
```

3. Add **multiDexEnabled true** in defaultConfig block

➢ How it works

1. Previously for converting your app's code in dex code we have to use Proguard but to reduce compile-time and to reduce app size the new R8 tool was introduced.
2. R8 tool provides facility to add missing file to dex file



Intents

- Introduction
- Components of intents

➢ Introduction

1. Intents are the messaging objects used to send the data from application to other application or from one component to other component of the application
2. There are two types of intents
   Implicit intents
   Explicit intents
3. Implicit intents are pre-defined by the system used to communicate outside of the application
4. Explicit intents are defined by the programmer used to communicate inside the application
5. Intents are used to start the activity or services and used to send broadcast etc

➢ Components of intents

1. Intents are having 5 components
   a. Action
   b. Data
   c. Category
   d. Component Name
   e. Extra
2. Action is the String specifies particular Action to be performed ex ACTION_SEND, ACTION_VIEW
3. Data is the information or type of the information on which action will be performed
4. Category is used in explicit intents to specify type of application that will be used to perform the action. **addCategory()**
5. Component specify the name of the component to be started. **setComponent(), setClass().**
6. Extra is addition information in the form of key and value pair. **putExtra()**

ListView

- **Introduction**
- **Creating List view**

Introduction

1. List View is the ViewGroup which display list of data to be displayed on the UI
2. List view uses Adapter which fetch data from array or DB and shows it on each item of the List view
3. setAdapter() method is used to set the adapter to the list view
4. There are 4 types of adapter
    a. Array adapter
    b. Cursor adapter
    c. Simple Adapter
    d. Base Adapter

Creating List view

1. Define array to be displayed on the list view
2. Create layout for item
3. Create an adapter which takes array and adapter as the arguments
4. Set the adapter using setAdapter() method to list view

```
var nameList = arrayOf("Android","JAVA","Kotlin","Activity","BroadCast","Services",
                       "Content provider","Fragment")
var listView : ListView? = null
var adapter : ArrayAdapter<String>? = null
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_list_view)
    listView = findViewById(R.id.listView)
    val adapter = ArrayAdapter( context: this,R.layout.item_list_view,nameList)
    listView?.adapter = adapter
}
```

Recycler View

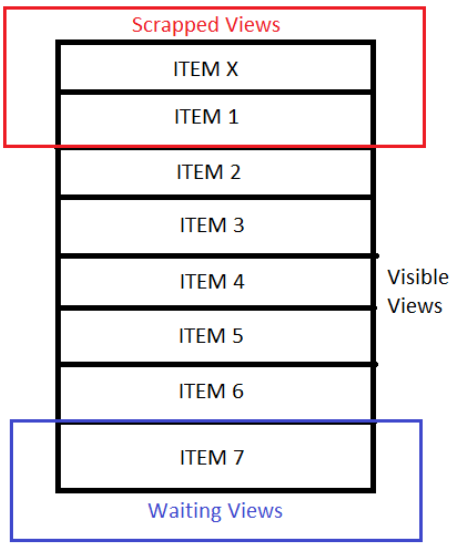
- **Introduction**
- **Working of recycler view**
- **Implementing recycler view**
- **Optimizing recycler view**

Introduction

1. Recycler view is the View group which hold list of data to be displayed on the screen
2. There are 3 major components of recycler view
   a. Adapter
   b. View holder
   c. Layout Manager
3. Adapter is the class which extends **RecyclerView.Adapter** and holds list of data to be displayed on each item of Recycler view
4. View Group is the class which extends **RecyclerView.ViewHolder** and helps to draw UI for each item
5. Layout manager is used to display items in linear view or grid view

Working of recycler view

1. Consider we have Item X to Item 7 in the recycler view and initially item x to item 4 are visible to the user are called as visible view
2. In Step 2 if user scrolls up now item 1 to item 5 are visible to the user and item x moved to scrapped views
3. Scrapped views are collection of views once visible to the user
4. In step 3 user scrolls one more item up now item 2 to 6 are visible to the user and item 7 is waiting to be displayed is called waiting views
5. In step 4 when user tries to load item 7 this item make use of scrapped views are called as dirty views



➢ Implementing recycler view

1. Create a model class which store the data of each item and make a list of type model class
2. Create a View holder class which extends **RecyclerView.ViewHolder** and define all the view for each item in this class
3. Create an adapter class which extends **RecyclerView.Adapter<VH>** of type view holder and override
   a. onCreateViewHolder()
   b. onBindViewHolder()
   c. getItemCount()
4. onCreateViewHolder() method is used to inflate item layout using LayoutinFlater.inflate()
5. onVindViewHolder() method is used to set views to the view holder class
6. getItemCount() returns size of the item list
7. create the object of adapter class by passing list of items and set this adapter object to recycler view using setAdapter() method
8. In the last set the layout manager to the recycler view layout manager can be LinearLayoutManager or GridLayoutManager

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): RecyclerViewHolder {
    val layoutInflater = LayoutInflater.from(context)
    val view = layoutInflater.inflate(R.layout.item_recycler_view,parent, attachToRoot: false)

    return RecyclerViewHolder(view)
}

override fun onBindViewHolder(holder: RecyclerViewHolder, position: Int) {
    val programmingLanguageData = list?.get(position)

    holder.iconIV?.setImageResource(programmingLanguageData!!.icon)
    holder.titleTV?.setText(programmingLanguageData!!.title)
    holder.subTitle?.setText(programmingLanguageData!!.subTitle)
}

override fun getItemCount(): Int {
    return list?.size!!
}
```

Adapter class

```
val recyclerView = findViewById<RecyclerView>(R.id.recycler_view)
val adapter = RecyclerAdapter( context: this, list!!)
recyclerView.adapter = adapter
recyclerView.layoutManager = LinearLayoutManager( context: this)
```

Set adapter in activity

➢ Optimizing recycler view

1. Use the image loading libraries like glide or Picasso to avoid un responsive UI
2. Set fixed image height and width to avoid flickering.
3. Do less work on onBindViewHolder method
4. Use notify item api
    a. notifyItemRemoved(position)
    b. notifyItemChanged
    c. notifyItemInserted
    d. notifyItemRangeInserted(from, to)
5. Avoid using of nested view
6. Use setItemViewCacheSized(size) to retain views which are just screlled

Navigation Drawer

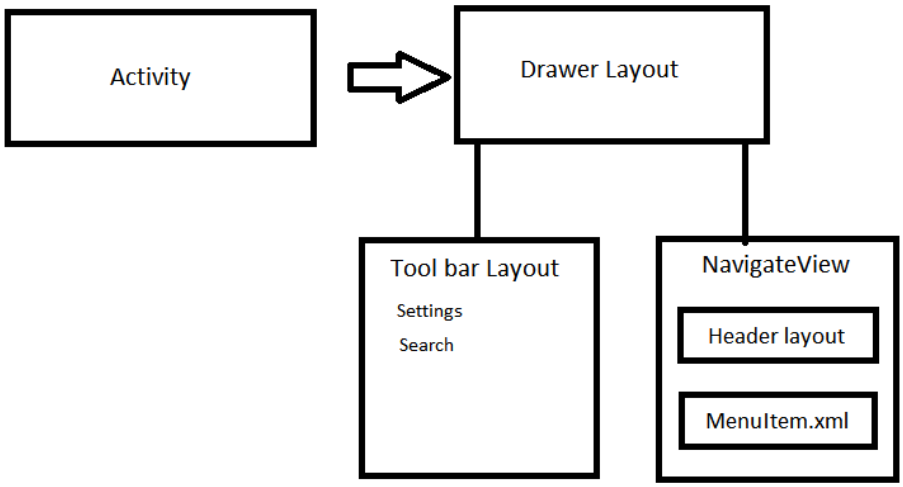
- **Introduction**
- **Implementation**

➢ Introduction

1. Navigation drawer is component which provides user to navigate across various screen in the application using menu items
2. Basically navigation component has two components
    a. Tool bar layout
    b. Drawer layout
3. Tool bar layout is used to hold tools such as settings of application etc
4. Drawer layout is used to show header for example profile information and menu items

➢ Implementation

1. Add dependency for to import google material components **implementation 'com.google.android.material:material:1.7.0-alpha03'.**
2. Create an Activity which holds layout of navigation drawer which overrides
    a. **onCreateOptionMenu(menu : Menu)** this adds item to the action bar
    b. **onSupportNavigationUp()** this handles menu option icon click
    c. **onNavigationItemSelected(MenuItem)** this handles each menu item click
3. create a layout for an activity and the root element of this activity should be DrawerLayout
4. Drawer layout holds 2 items
    a. Tool bar layout
    b. Navigate View
5. Include Tool bar layout and Navigation View in Drawer layout
6. Create Header layout and menu item and add these to NavigateView



7. Set tool bar using setSupprtToolBar(ToolBar)
8. Create action drawer toggle using class ActionDrawerToggle() which takes 4 arguments context, drawer layout, open string and close string and sync the state using syncState() method

```
setContentView(R.layout.activity_nav_drawer)
toolBar = findViewById(R.id.toolbar)
drawerLayout = findViewById(R.id.drawer_layout)
navigationView = findViewById(R.id.nav_view)

setSupportActionBar(toolBar)
val actionBarDrawerToggle = ActionBarDrawerToggle(
    activity: this, drawerLayout,
    "Open", "Close",
)
actionBarDrawerToggle.syncState()

supportActionBar?.setDisplayHomeAsUpEnabled(true);
navigationView.setNavigationItemSelectedListener(this)
```

Fragments

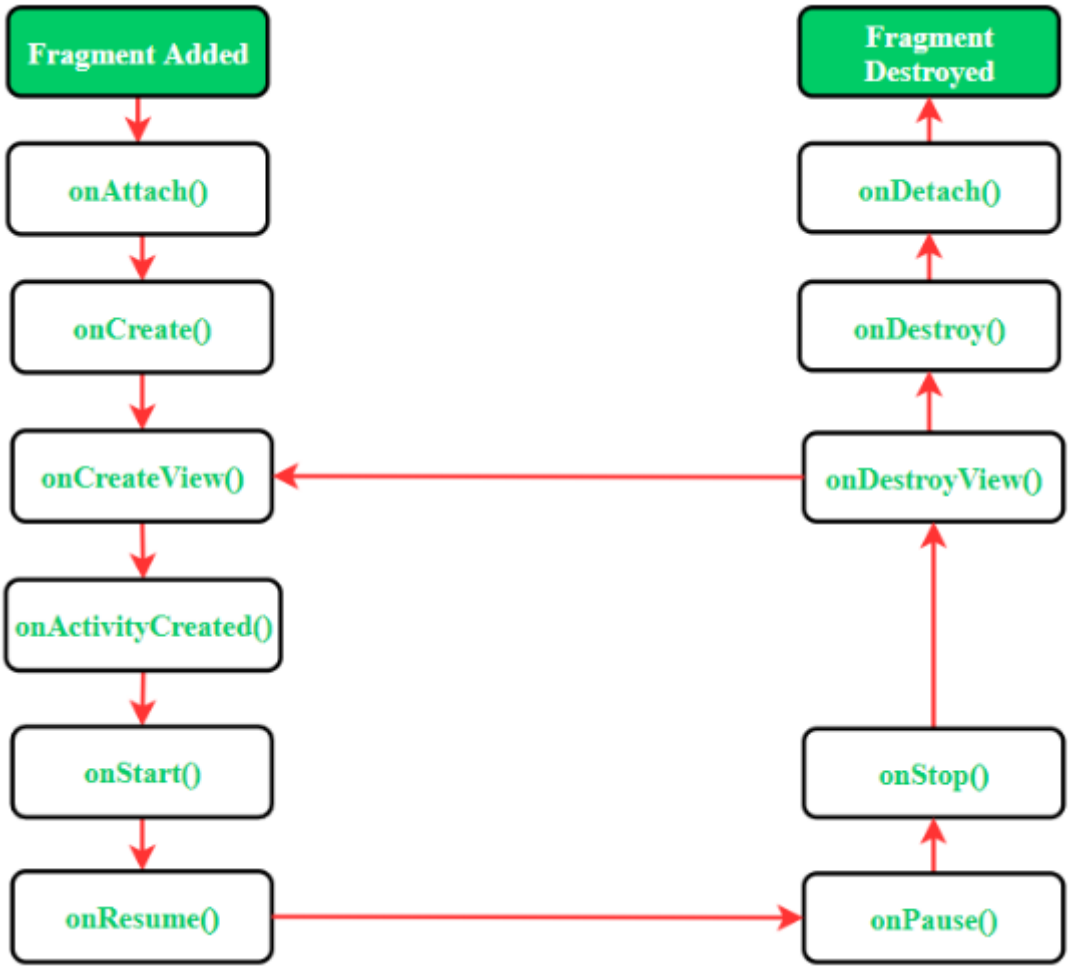
- **Introduction**
- **LifeCycle of Fragments**
- **Implementation**
- **Transaction**
- **Difference between add() and replace()**
- **Difference between activity and fragment**
- **Cross questions**

Introduction

1. Fragments are the part of activity which adds its own UI to the Activity and also called as sub activity
2. Fragments life cycle is dependent on activity life cycle
3. We can add multiple fragment in one activity

Lifecycle of Fragments

1. Fragment has its own life cycle like activity and has 11 lifecycle methods onAttach(), onCreate(), onCreateView(), onActivityCreated(), onStart(), onResume(), onPause(), onStop(), onDestroyView(), onDestroy(), onDetach()
2. onActivityCreated is deprected



➢ Implementation

1. Create an activity and the layout of that activity which holds fragment container
2. Create a fragment and the layout for that fragment and inflate this layout in onCreateView using layoutInflator.inflate()
3. Inflate function take three arguments
   a. fragment layout
   b. view group
   c. Boolean to attach to root

```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                          savedInstanceState: Bundle?): View? {
    val view = layoutInflater.inflate(R.layout.fragment_two, container,  attachToRoot: false)
```

➢ Fragment Transaction

1. Create Fragment manager using support fragment manager
2. Create Fragment Transaction from fragment manager using begin transaction method
3. Create an object of Fragment and add this object to transaction using add() or replace() methods
4. Add the transaction to back stack and commit the transaction

```
val fragmentManager : FragmentManager? = activity?.supportFragmentManager
val fragmentTransaction : FragmentTransaction? = fragmentManager?.beginTransaction()
val fragmentOne = FragmentOne()
fragmentTransaction?.add(R.id.fragmentContainer,fragmentOne, tag: "fragment_two")
fragmentTransaction?.addToBackStack( name: "fragment_two_to_one_transaction")
fragmentTransaction?.commit()
```

➢ Difference between add() and replace()

	Add()	Replace()
1	Add is used to add one fragment on top of other fragment	Replace is used to replace top fragment and add new one
2	In add transaction previous fragment view will not be recreated	In replace transaction previous fragment view will be recreated
3	In Transition from fragment A to B fragment A's onPause() Will not called	In Transition from fragment A to B fragment A's onDestroyView() will get called
4	Fragment A will be visible to user below the fragment B	Fragment A will not be visible to the user below the fragment B

➢ Difference between activity and fragment

	Activity	Fragment
1	Activity is a single screen which has its own UI	Fragment adds its UI to the Activity
2	Activity is Single Screen	Fragment can be multiple screen
3	Activity is independent component	Fragment is dependent on activity
4	Activity must be declared in manifest file	Fragment no need to be declared in manifest file
5	Activity takes lot of memory	Fragments are light weighted components

➢ Cross Questions

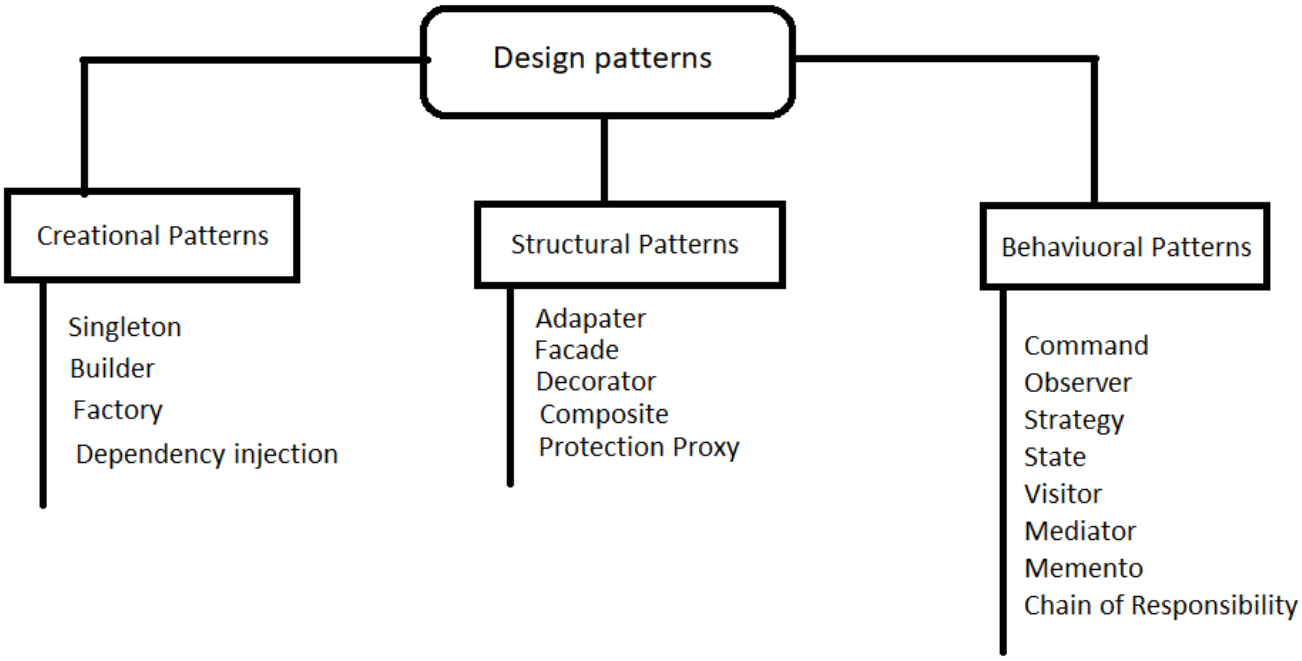
1. If we start fragment from activity buttons of the activity will overlapped on fragment container view, activity buttons get higher priority than other view.
2. To avoid overlapping of button on fragment container view wrap buttons in any of the view group ex constraint layouts

Design Patterns

- **Introduction**
- **Creational pattern**
- **Structural Pattern**
- **Behavioral Pattern**

Introduction

1. Design are blueprint for the solution in programming
2. There are 3 design patterns
   a. Creational pattern
   b. Structural pattern
   c. Behavioral pattern



➢ Singleton

1. Single ton is creational design pattern which restrict instantiation of class to only one object
2. Singleton objects are used in costly resources like database only one instance of the database should be created throughout the app
3. In singleton we can create classic singleton, thread safe singleton, eager singleton.

```kotlin
object KotlinSingleTon {

}
```

Kotlin Singleton

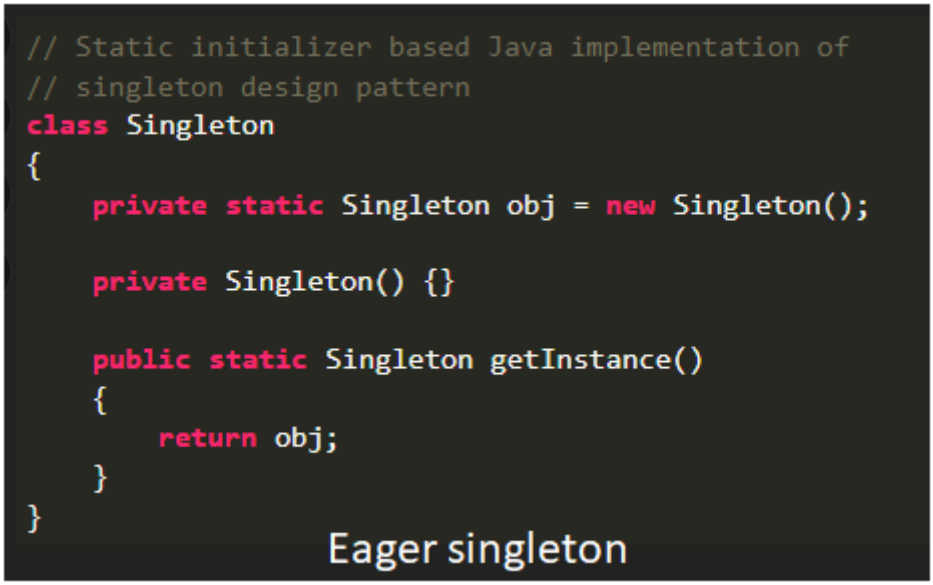
4. Classic singleton are not thread safe if we start two threads at same time different objects may get created for singleton
5. Using CountDownLatch we can demonstrate classic singleton are not safe

```java
/** Classic singleton are not thread safe*/
public class ClassicSingleTon {
    private static ClassicSingleTon obj =null;

    private ClassicSingleTon(){}

    public static ClassicSingleTon getInstance() {
        if(obj == null){
            obj = new ClassicSingleTon();
        }
        return obj;
    }
    public static void destroyObject(){
        obj = null;
    }
}
```

Classic singleton

6. Thread safe singleton

7. Eager singleton will create object in static initializer, these are thread safe as JVM creates the objects



1. Builder Pattern

2. Builder pattern is used to create complex objects step by step and the final step will return the object of product class
3. Advantage of using builder pattern are
   a. readability
   b. reduces parameters in the constructors
   c. Objects will always have instantiated in complete state
4. Disadvantage of using builder pattern is
   a. More number of lines while building objects
   b. Need separate concrete builder class for each product
5. In java builder class will be static and in kotlin builder class can be companion object.
6. **Java Example**

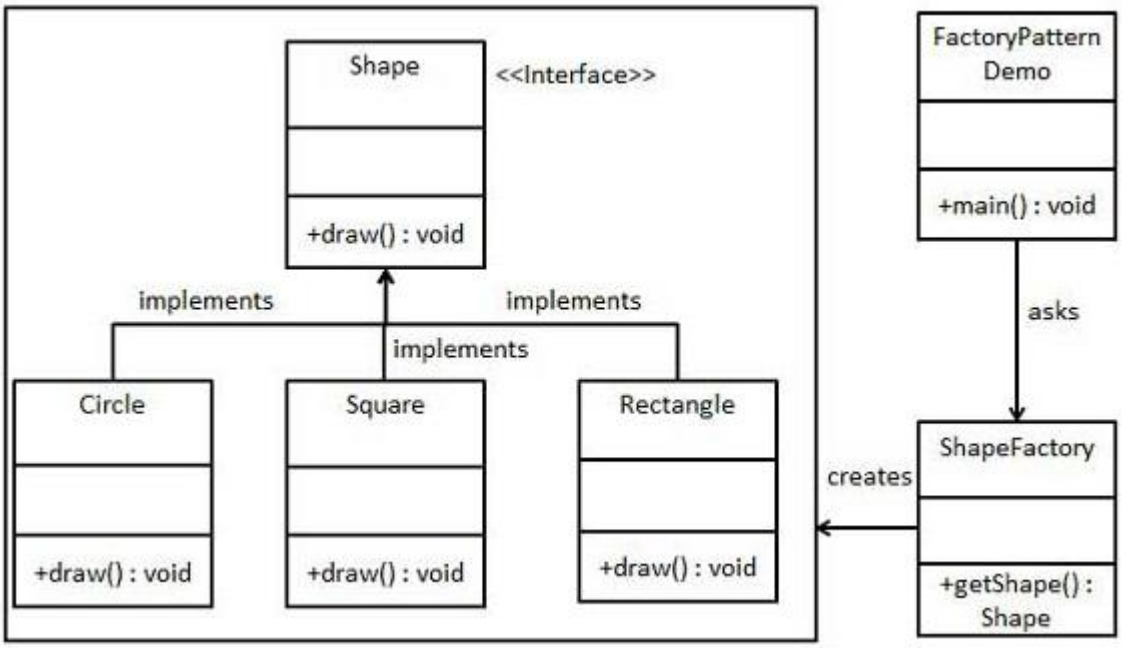
```
public class JavaCarBuilder {
    private int carModel;
    private String brand;
    private String color;

    JavaCarBuilder(Builder builder){
        this.carModel = builder.carModel;
        this.brand = builder.brand;
        this.color = builder.color;
    }
    public static class Builder{
        private int carModel;
        private String brand;
        private String color;
        public Builder(){}

        public Builder setCarModel(int carModel){
            this.carModel = carModel;
            return this;
        }

        public Builder setBrand(String brand){
            this.brand = brand;
            return this;
        }

        public Builder setColor(String color){
            this.color = color;
            return this;
        }

        public JavaCarBuilder build(){
            return new JavaCarBuilder(this);
        }
    }

    @Override
    public String toString() {
        return "JavaCarBuilder{" +
                "Hashcode='"+this.hashCode() + '\''+
                ", carModel='" + carModel+ '\'' +
                ", brand='" + brand + '\'' +
                ", color='" + color + '\'' +
                '}';
    }
}
```

7. Kotlin Example

```
class KotlinCarBuilder(private var carModel: Int, private  var carBrand: String,
private var carColor: String) {

    /**Using kotlin nested class*/
    class Builder {
        private var carModel = 0
        private var carBrand = ""
        private var carColor = ""
        public fun setCarModel(carModel: Int): Builder {
            this.carModel = carModel
            return this
        }

        public fun setCarBrand(carBrand: String): Builder {
            this.carBrand = carBrand
            return this
        }

        public fun setCarColor(color: String): Builder {
            this.carColor = color
            return this
        }

        public fun build(): KotlinCarBuilder {
            return KotlinCarBuilder(this.carModel, this.carBrand, this.carColor)
        }
    }
    /**Using Kotlin companion object*/
    companion object Builder2 {
        private var carModel = 0
        private var carBrand = ""
        private var carColor = ""
        public fun setCarModel(carModel: Int): Builder2 {
            this.carModel = carModel
            return this
        }

        public fun setCarBrand(carBrand: String): Builder2 {
            this.carBrand = carBrand
            return this
        }

        public fun setCarColor(color: String): Builder2 {
            this.carColor = color
            return this
        }

        public fun build(): KotlinCarBuilder {
            return KotlinCarBuilder(this.carModel, this.carBrand, this.carColor)
        }
    }

    override fun toString(): String {
        super.toString()
        return "KotlinCarBuilder : {Hashcode = '${this.hashCode()}', " +
                "CarModel = '$carModel', " +
                "CarBrand = '$carBrand', " +
                "CarColor = '$carColor'}"
    }
}
```

➢ Factory Pattern

1. Factory pattern is object creational design pattern where object creation is not exposed to user
2. Factory class handles all the object creation and give the object back to user
3. Create a vendor class which ask Factory to create object of need
4. Create a Factory class which is responsible of creating object based on vendor needs and return the product
5. Create an interface which hides Product creation



6. **Example**

```
interface Shape {
    fun draw() : String
}
```

```
class Circle : Shape {
    override fun draw() : String{
        return "Circle Drawing"
    }
}
```

```
class Rectangle : Shape{
    override fun draw() : String{
        return "Rectangle Drawing"
    }
}
```

```
class ShapeFactory() {
    public fun getShape(shape : String) : Shape? {
        when(shape){
            "RECTANGLE" -> { return Rectangle() }
            "CIRCLE" -> { return Circle() }
        }
        return null
    }
}
```

```
val shapeFactory = ShapeFactory()
val shape1 : Shape? = shapeFactory.getShape( shape: "CIRCLE")
val shape2 : Shape? = shapeFactory.getShape( shape: "RECTANGLE")
Log.d(FACTORY_PATTERN,  msg: "${shape1?.draw()}")
Log.d(FACTORY_PATTERN,  msg: "${shape2?.draw()}")
```

➢ Structural Pattern

1. Behavioral patterns deal with how class and objects are composed and simplify by identifying relationship
2. Behavioral pattern concerned with how classes are inherited each other

➢ Adapter Pattern

1. Adapter pattern used to convert on interface of class to another interface that client wants
2. Adapter pattern is also called as Wrapper
3. In adapter pattern we have 4 components
   a. Target interface
   b. Adoptee interface
   c. Adapter
   d. client

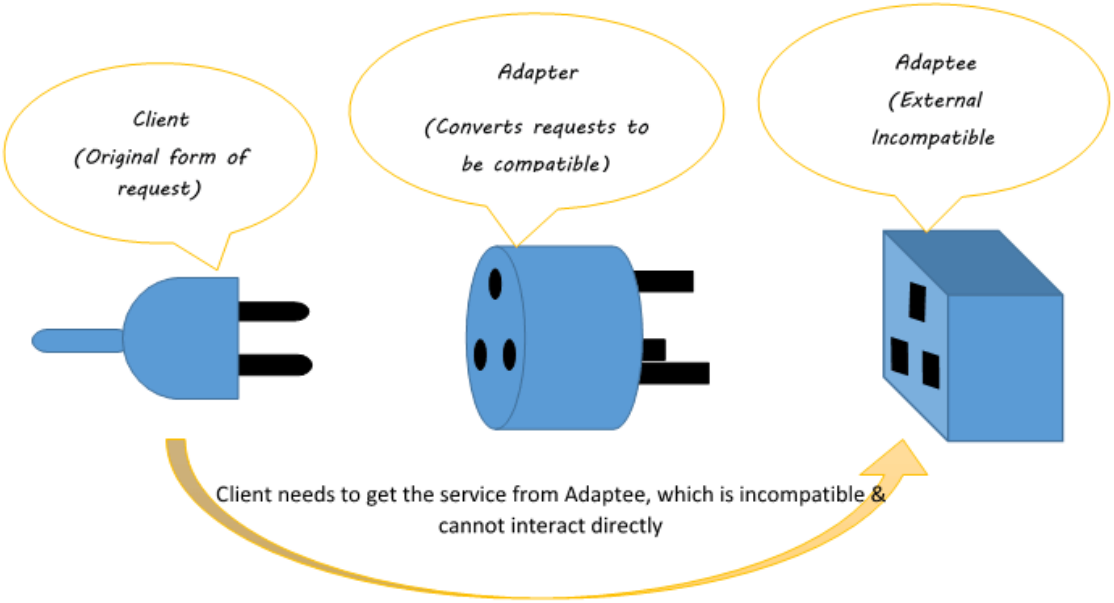


Figure 1-Adapter Pattern Concept

4. **Example**

```
/** target interface */
interface GermanPlug {
    fun provideElectricity() : String
}
```

```
class GermanSockets : GermanPlug{
    override fun provideElectricity() : String{
        return "German Elctricity"
    }
}
```

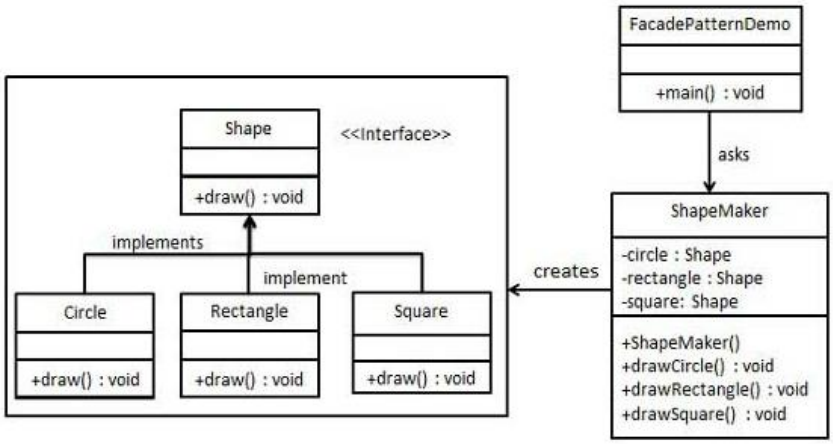
```
/** Adaptee interface */
interface UkPlug {
    fun provideElectricity() : String
}

class UKSockets : UkPlug{
    override fun provideElectricity() : String{
        return "UK Electricity"
    }
}
```

```
/** Adapter converts adaptee to target*/
class UkToGermanPlugConvertorAdapter : UkPlug {
    lateinit var germanPlug: GermanPlug
    constructor(germanPlug: GermanPlug){
        this.germanPlug = germanPlug
    }

    override fun provideElectricity() : String {
        return germanPlug.provideElectricity()
    }
}
```

```
/** Adapter Patterns
 * Client*/
val germanPlug : GermanPlug = GermanSockets()
Log.d(ADAPTER_PATTERN, germanPlug.provideElectricity())

val ukPlug : UkPlug = UKSockets()
Log.d(ADAPTER_PATTERN, ukPlug.provideElectricity())

val adapter : UkPlug = UkToGermanPlugConvertorAdapter(germanPlug)
Log.d(ADAPTER_PATTERN, adapter.provideElectricity())
```

➢ Facade Pattern

1. As the name indicates façade hides how the complex implementation is done
2. In façade pattern client interact with only with façade class which gives required objects to client
3. Façade deals with only interface no implementation
4. Façade pattern used when complex system needs to be hidden to client




```
interface IShape {
    fun draw() : String
}
```

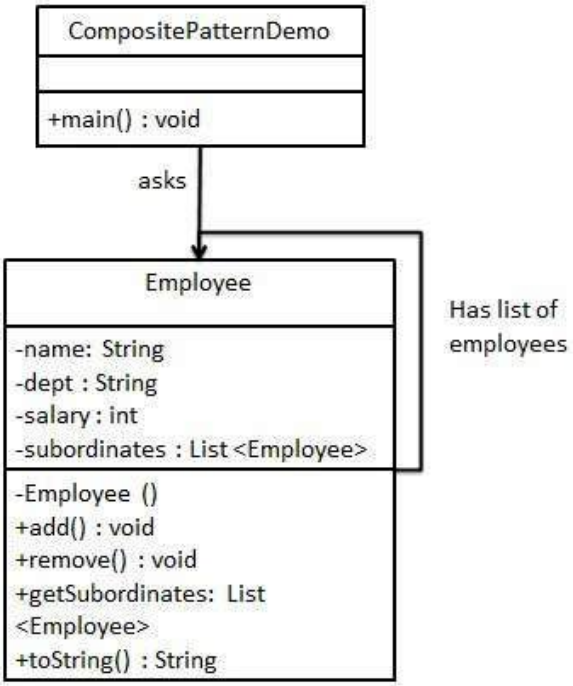
```
class Circle : IShape {
    override fun draw(): String {
        return "Circle"
    }
}
```

```
class Rectangle : IShape {
    override fun draw() : String {
        return "Rectangle"
    }
}
```

```
class ShapeMaker {
    private lateinit var rectangle : IShape
    private lateinit var circle: IShape
    constructor(){
        rectangle = Rectangle()
        circle = Circle()
    }
    fun drawCircle() : String{
        return circle.draw()
    }
    fun drawRectangle() : String{
        return rectangle.draw()
    }
}
/**Facade Patters
* Client*/
val shapeMaker = ShapeMaker()
Log.d(FACADE_PATTERN, shapeMaker.drawRectangle())
Log.d(FACADE_PATTERN, shapeMaker.drawCircle())
```

➢ Composite Pattern

1. In Composite pattern group of objects are treated as similar way as single objects
2. Composite pattern used in hierarchical objects structure using tree representation




```
class Employee(var name : String, var id : Int, var designation : String) {
    private var list = ArrayList<Employee>()
    fun add(employee: Employee){
        list.add(employee)
    }
    fun remove(employee: Employee){
        list.remove(employee)
    }
    fun getEmployees() : ArrayList<Employee>{
        return list
    }

    override fun toString(): String {
        return "EMPLOYEE : [name = '$name', " +
                "id = '$id', " +
                "designation = '$designation']"
    }
}
```

```
 * client*/                                                    ⚠2 ⚠2
val ceo = Employee( name: "Ravi", id: 1, designation: "CEO")
val manager1 = Employee( name: "Rupa", id: 2, designation: "Manager")
val manager2 = Employee( name: "CV", id: 3, designation: "Manager")
ceo.add(manager1)
ceo.add(manager2)

val supervisor = Employee( name: "Manju", id: 4, designation: "Supervisor")
manager1.add(supervisor)

val softwareEngg1 = Employee( name: "Gautem", id: 5, designation: "Software Engineer")
val softwareEngg2 = Employee( name: "Priya", id: 6, designation: "Software Engineer")
supervisor.add(softwareEngg1)
supervisor.add(softwareEngg2)

Log.d(COMPOSITE_PATTERN, ceo.toString())
for (managers in ceo.getEmployees()){
    Log.d(COMPOSITE_PATTERN, managers.toString())
    for(sup in managers.getEmployees()){
        Log.d(COMPOSITE_PATTERN, sup.toString())
        for (sw in sup.getEmployees()){
            Log.d(COMPOSITE_PATTERN, sw.toString())
        }
    }
}
```

```
EMPLOYEE : [name = 'Ravi', id = '1', designation = 'CEO']
EMPLOYEE : [name = 'Rupa', id = '2', designation = 'Manager']
EMPLOYEE : [name = 'Manju', id = '4', designation = 'Supervisor']
EMPLOYEE : [name = 'Gautem', id = '5', designation = 'Software Engineer']
EMPLOYEE : [name = 'Priya', id = '6', designation = 'Software Engineer']
EMPLOYEE : [name = 'CV', id = '3', designation = 'Manager']
```

➢ Behavioral Pattern

1. Behavioral patterns concerned with object interaction and their responsibilities
2. Behavioral pattern provides objects can easily talk with each other and still should be loose coupling

Command:
The command pattern is used to express a request, including the call to be made and all of its required parameters, in a command object. The command may then be executed immediately or held for later use.

Observer:
The observer pattern is used to allow an object to publish changes to its state. Other objects subscribe to be immediately notified of any changes.

Strategy:
The strategy pattern is used to create an interchangeable family of algorithms from which the required process is chosen at run-time.

State:
The state pattern is used to alter the behavior of an object as its internal state changes. The pattern allows the class for an object to apparently change at run-time.

Visitor:
The visitor pattern is used to separate a relatively complex set of structured data classes from the functionality that may be performed upon the data that they hold.

Mediator:
The mediator design pattern is used to provide a centralized communication medium between different objects in a system. This pattern is very helpful in an enterprise application where multiple objects are interacting with each other.

Memento:
The memento pattern is a software design pattern that provides the ability to restore an object to its previous state (undo via rollback).

Chain of Responsibility:
The chain of responsibility pattern is used to process varied requests, each of which may be dealt with by a different handler.

View Model

- **Introduction**
- **Implementation**

➢ Introduction

1. View Model is the Architecture components which is used to hold UI related data in life cycle conscious way.
2. Using view model, we can retain data on configuration changes.
3. View Model holds data until activity is finished or destroyed

➢ Implementation

1. Create a class which extends View Model
2. Use ViewModelProvider().get() to get particular instance of View Model class

```
class ActivityViewModel : ViewModel() {
    var number = 0;
    fun addOne(){
        number++
    }
}
```

```
override fun onCreate(bundle : Bundle?) {
    super.onCreate(bundle)
    setContentView(R.layout.activity_view_model)
    Log.d(TAG, msg: "onCreate: ")
    var viewModel = ViewModelProvider( owner: this)[ActivityViewModel::class.java]

}
```

Live Data

- **Introduction**
- **Implementation**
- **Advantages**

➢ Introduction

1. Live Data is the android architecture component which holds the data
2. This data is observed by other components of application
3. Livadata are life cycle aware components, so live data checks state of an observer before emitting the data
4. If component is in destroyed state, then live data removes the observer thus it avoids memory leaks

➢ Implementation

```
class LiveDataViewModel : ViewModel() {
    public val mutableLiveData = MutableLiveData<Int>()
    var number = 0;
    fun addOne(){
        number++
        mutableLiveData.value = number
    }
}
```

```
button.setOnClickListener(View.OnClickListener{ it: View!
    viewModel.addOne()
})

viewModel.mutableLiveData.observe( owner: this, Observer{ it: Int!
    textView.text = "$it"
})
```

➢ advantages

1. Live data is used with view model to hold the data
2. **No need to update UI every time**, whenever there is change in Data live data automatically emit the data to active observers
3. **Avoid memory leaks,** inactive observers will be remove thus memory leak can be avoided
4. **No more manual lifecycle handling,** live data is life cycle aware component it automatically handles life cycle changes
5. **Handles configuration changes,** data can be retained on configuration change when it is used with view model

Android Architectural Patterns

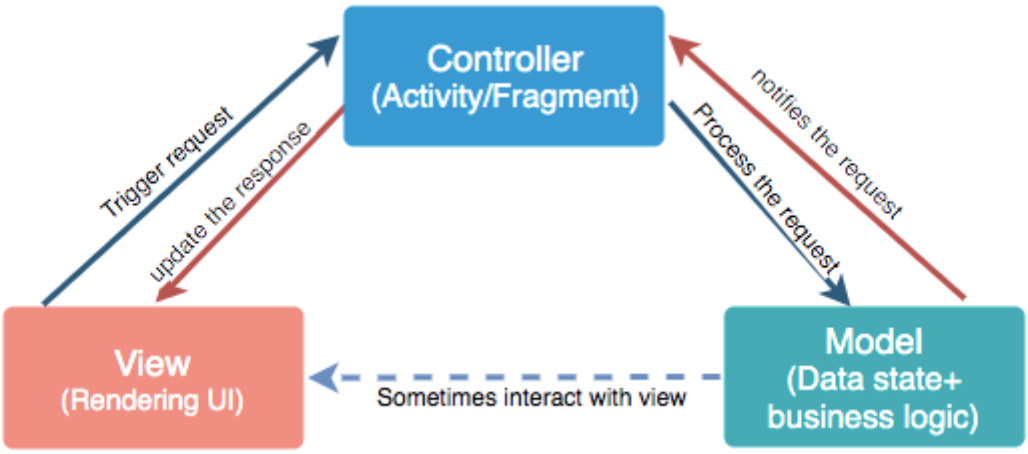
- **Introduction**
- **MVC**
- **MVP**
- **MVVM**

Introduction

1. Architectural patter helps to organize the software in proper way and easy to maintain.
2. Testing is made easy and code maintenance is handled in proper way
3. There are mainly 3 architectural patterns
   a. MVC
   b. MVP
   c. MVVM

MVC

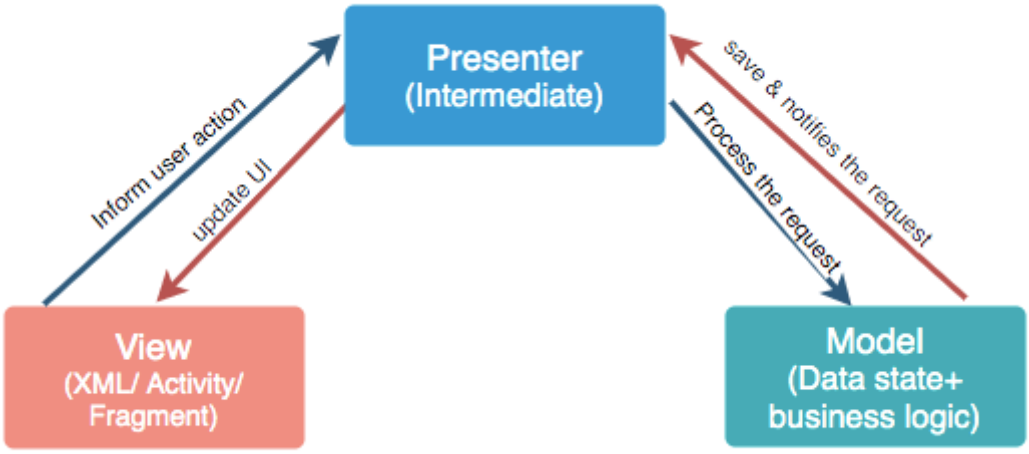
1. MVC is model view controller pattern
2. Model is used to store the data this layer is used to communicate with Database and network for the data
3. View is UI which displays the data held by model layer
4. Controller contains core logic
5. Increase the testability model and controller can be tested as they don't extend any android class
6. View can be tested through the UI test
7. Code is dependent on each other



MVP

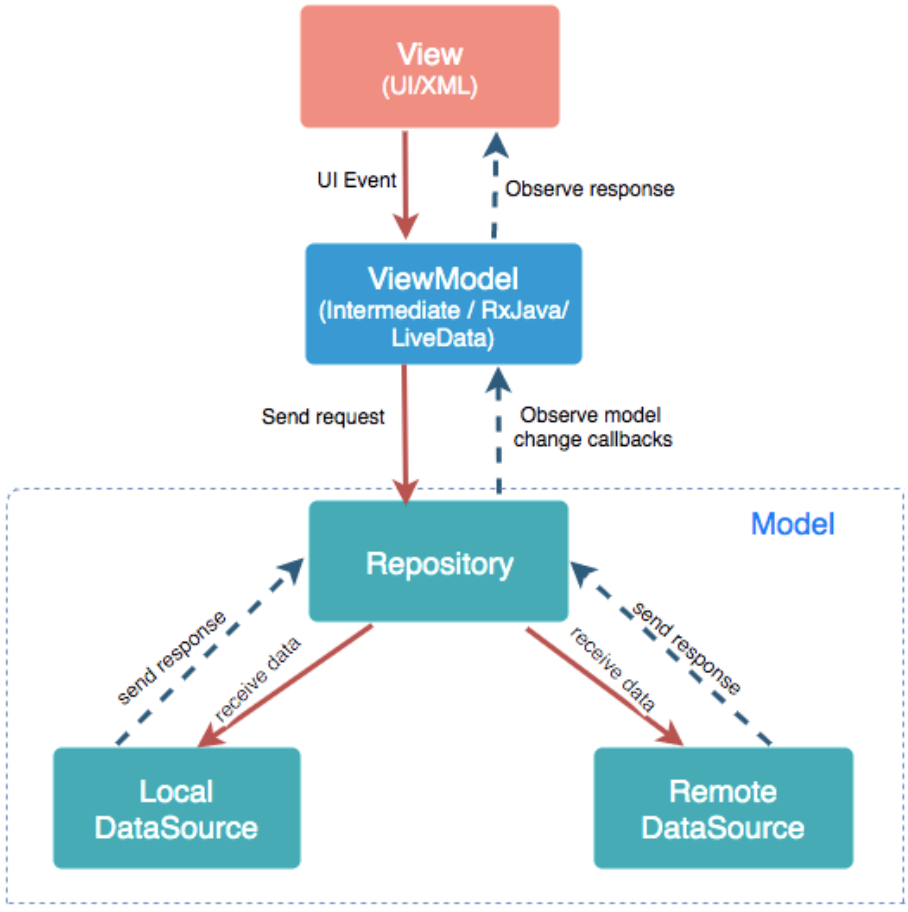
1. MVP is the model view presenter pattern
2. Here Presenter acts as the intermediate between model and view
3. Model Is used to store the data and can be communicate with data sources like DB or network resources etc
4. View is used to display the data which is held by model layer
5. In MVP there is 1 to 1 relation between presenter and view

6. In MVP lot of interface implementations are involved
7. View and presenter are tightly coupled



➢ MVVM

1. MVVM is the Model view View-model pattern
2. View model make use of live data
3. Live data is the component which emits the data to observers
4. it does not care about who is observing the data
5. in MVVM there is 1 to many relation between view and view – model

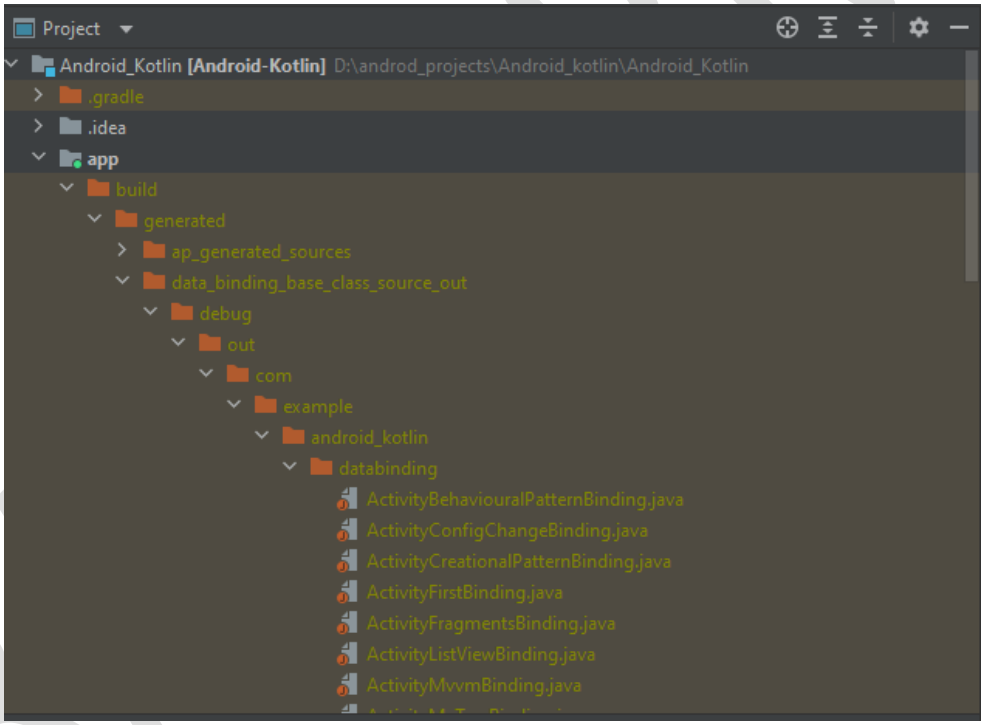


View Binding

- **Introduction**
- **Implementation**

➢ Introduction

1. View Binding is Jet Pack component used to deal with creating objects of views and introduced in the gradle 3.6 and android 4
2. Which replaces the findViewById() also Works faster than findViewById()
3. By using View Binding boilerplate code can be reduced
4. View Binding must follows the naming conventions for example
   a. **activity_mail.xml** is generated as **ActivityMainBinding** which contains all the properties and instances of views
   b. View Ids i**d:"button_submit"** are created as **buttonSubmit** instance
5. All the View binding class are generated in Project folder structure



➢ Implementation

1. Enable View Binding in the gradle file using **buildFeatures{}** block and **viewBinding = true**
2. Create an xml file for the activity which contains views
3. Create an Activity class and instantiate binding object using auto generated binding class
4. Using this binding object we can access properties and method of the view

```
41
42      /** Gradle file
43       * */
44      buildFeatures {
45          viewBinding true
46      }
47
```

```
8   /** View Binding Examole activity
9    * */
10  class ViewBindingExampleActivity : AppCompatActivity() {
11      private lateinit var mBinding: ActivityViewBindingExampleBinding
12      override fun onCreate(bundle: Bundle?){
13          super.onCreate(bundle)
14          mBinding = ActivityViewBindingExampleBinding.inflate(layoutInflater)
15          setContentView(mBinding.root)
16
17          mBinding.submitBtn.setOnClickListener {  it: View!
18              val enteredText = mBinding.enterTextET.text
19              Toast.makeText(applicationContext,enteredText,Toast.LENGTH_SHORT).show()
20          }
21      }
22  }
```

```
6
7       <EditText
8           android:id="@+id/enterTextET"
9           android:layout_width="200dp"
10          android:layout_height="wrap_content"
11          android:layout_marginTop="100dp"
12          app:layout_constraintStart_toStartOf="parent"
13          app:layout_constraintEnd_toEndOf="parent"
14          app:layout_constraintTop_toTopOf="parent"
15          />
16
17      <Button
18          android:id="@+id/submitBtn"
19          android:layout_width="wrap_content"
20          android:layout_height="wrap_content"
21          android:text="@string/submit"
22          app:layout_constraintStart_toStartOf="parent"
23          app:layout_constraintEnd_toEndOf="parent"
24          app:layout_constraintTop_toBottomOf="@+id/enterTextET"
25          />
26
```

Data Binding

- **Introduction**
- **Expressions and operators**
- **Implementation**
- **NOTE**

Introduction

1. Data binding allows us to bind UI components with Data sources directly, thus we can simpler and easier to maintain the code
2. Data binding library provides imports, variables and includes
3. **<layout>** tag is the root element of the layout
4. **<data>** tag is used to declare the variables and variables can be declared using <variable> tag
5. **<variable>** tag contains two attributes name and type
6. Here type can be the view model class
7. Data binding supports **two-way** data binding meaning binding supports the ability to receive data changes to a property and listen to user updates to that property at the same time.
8. In Data tag we can do the import of particular class using <Import> tag <import type="android.view.View"/>
9. Using **default** key-word we can set default values to the views @{dataBindingExampleVM.textViewLiveData, default = My_Default_value}

Expressions and operators

1. All the binding expressions should be defined inside @{} block
2. Mathematic **(+ - * / %)**, Logical **(&& ||),** Comparison **(== > < >= <=)**
3. Binary **(& | ^)**, Unary **(+ - ! ~)**, Shift **(>> >>> <<)**
4. String concat +, Ternary ?:, Grouping (), Array access []
5. **Null coalescing operator (??)** works same as ternary operator
   a. android:text="@{user.displayName ?? user.lastName}"
   b. android:text="@{user.displayName != null ? user.displayName : user.lastName}".

Implementation

1. Data binding supports android version 4.0(API level 14) and higher
2. Enable data binding in **buildFeatures()** block using **dataBinding true**
3. Create Activity and the layout XML which will have <layout> as root element
4. Create View Model this view model will be declared in layout data tag
5. Create VM object using ViewModelProvider and data binding object using DataBindingUtil
6. Set life cycle owner to the data binding and View model object to the property defined in data tag

```xml
<layout xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android">
    <data>
        <variable
            name="dataBindingExampleViewModel"
            type="com.example.android_kotlin.dataBinding.DataBindingExampleViewModel" />
        <import type="android.view.View"/>
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">
```

```kotlin
class DataBindingExampleViewModel : ViewModel() {
    companion object{
        val U_NAME = "RAVI"
        val PASSWORD = "RAVI"
    }
    val textViewLiveData = MutableLiveData<String>()
    val textViewVisibility = MutableLiveData<Boolean>()
    val loginBtnClk = MutableLiveData<Boolean>()

    val errorTVLiveData = MutableLiveData<String>()
    val errorTVVisibility = MutableLiveData<Boolean>()

    val dummy = MutableLiveData<String>()

    public fun login(view : View){
        loginBtnClk.setValue(true)
    }
}
```

```kotlin
lateinit var dataBinding: ActivityDataBindingExampleBinding
lateinit var vm : DataBindingExampleViewModel

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    vm = ViewModelProvider( owner: this)[DataBindingExampleViewModel::class.java]
    dataBinding = DataBindingUtil.setContentView( activity: this, R.layout.activity_data_binding_example)
    /** must set Life cycle owner or
     * else UI never update*/
    dataBinding.lifecycleOwner = this
    dataBinding.dataBindingExampleViewModel = vm
```

➢ NOTE

1. @{dataBindingExampleViewModel::login} used for data binding on button clicks Login is the method name its signature must match on click listener's method signature **ex: public fun login(view : View)**
2. Must set life cycle owner to the data binding object or else UI will wont update **dataBinding.lifecycleOwner = this**

Broadcast Receiver

- **Introduction**
- **Receiving Broadcast**
- **Sending Broadcast**
- **Broadcast receiver with permissions**
- **Security consideration**

➢ Introduction

1. Broadcast are used to send messages from one application to other applications
2. Broadcast Receivers follow publish and subscribe pattern
3. Messages can be sent and receive in the form of intents

➢ Receiving Broadcast

1. There are two way of receiving broadcast one is context register and another is manifest register broadcast
2. Manifest registered are declared in <receiver> tag in manifest file
3. System Create Broadcast component object and this object valid until returns from onReceive()
4. Context registered broad cast are declared in particular component and valid as long as context is valid

➢ Sending Broadcast

1. sendOrderdBroadCast is method used to send messages in order, to only one receiver messages will be sent at one time
2. sendBroadCast is method used to send messages to all the registered receiver at the same time
3. LocalBoradCastManager.sendBroadCast() is used to send messages within application.

➢ Broadcast receiver with permissions

1. Send broadcast methods takes two arguments
   a. Intent
   b. Permission
2. if sender is sending the broadcast with permission receiver must request the permission while registration
3. **<uses permission name= “ ”>** tag is used to declare the permission for context registered broadcast

➢ Security consideration

1. To avoid communication with outside of application use Local Broadcast Receiver
2. Many broadcast may cause problem to system to handle
3. Do not broad cast sensitive information in the intents
4. Malicious broadcast can be avoided by using permissions
5. Do not do long running task in onReceive

```
/** register local broadcast and global bradcast
 * must pass the string arguments in intent filter
 * if not passed broad cast will not be sent*/
val localIntentFilter = IntentFilter(LOCAL_RECEIVER)
localBroadcastManager = LocalBroadcastManager.getInstance( context: this)
localBroadcastManager.registerReceiver(localReceiver, localIntentFilter)

//global broadcast
val globalIntentFilter = IntentFilter(GLOBAL_RECEIVER)
registerReceiver(globalReceiver,globalIntentFilter)
```

```
/**send local and global broadcast
 * string argument in the intent must be same as registered*/
activityVm.sendLocalBroadCastBtn.observe( owner: this, Observer { it: Boolean!
    COUNT++
    val intent = Intent(LOCAL_RECEIVER)
    intent.putExtra(KEY, COUNT)
    localBroadcastManager.sendBroadcast(intent)
})

//send global broadcast
activityVm.sendGlobalBroadCastBtn.observe( owner: this, Observer { it: Boolean!
    COUNT++
    val intent = Intent(GLOBAL_RECEIVER)
    intent.putExtra(KEY, COUNT)
    sendBroadcast(intent)
})
```

```
/** Defining of local braodcast */
val localReceiver : BroadcastReceiver = object  : BroadcastReceiver(){
    val broadCastReceivedLiveData = MutableLiveData<String>()
    override fun onReceive(p0: Context?, intent: Intent?) {
        Toast.makeText(applicationContext, text: "local broad cast", Toast.LENGTH_SHORT).show()
        val count = intent?.extras?.get(MyBroadCastActivity.KEY)
        activityVm.boardCastMsgTv.value = "$count"
    }
}
```

```
<!--Broad Cast-->
<activity android:name=".broadCastReceiver.MyBroadCastActivity"/>
<receiver android:name=".broadCastReceiver.MtBroadCatReceiver"/>
```

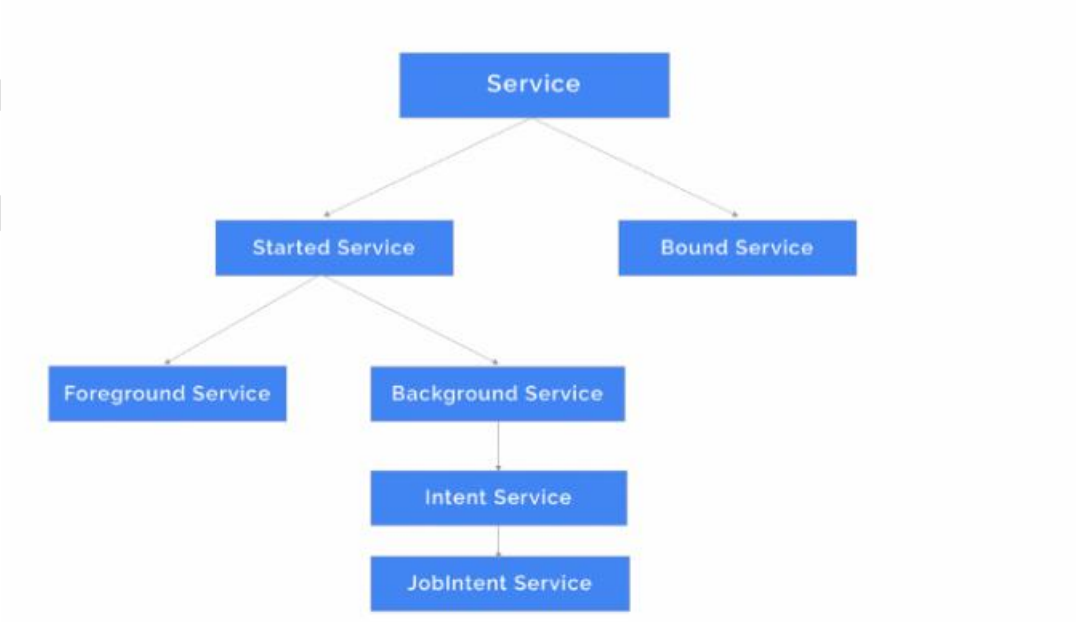
```
/** defination of global broadcast */
class MtBroadCatReceiver : BroadcastReceiver() {
    val broadCastReceivedLiveData = MutableLiveData<String>()
    override fun onReceive(p0: Context?, intent: Intent?) {
        Toast.makeText(p0, text: "global broad cast", Toast.LENGTH_SHORT).show()
        val count = intent?.extras?.get(MyBroadCastActivity.KEY)
        broadCastReceivedLiveData.value = "$count "
    }
}
```

Services

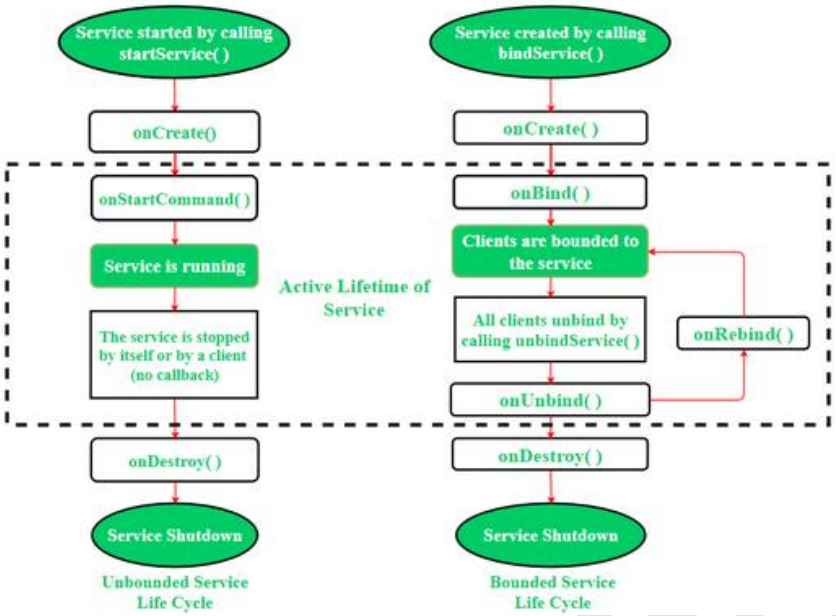
- **Introduction**
- **Life cycle of services**
- **Started services**
- **Bound services**
- **Foreground services**

Introduction

1. Service is the application component for doing long running background task
2. It runs in the main thread of the application process
3. There are 3 types of services
   a. Foreground service
   b. Background Service
   c. Bound Service
4. Foreground services are noticeable to the user, user can be aware of the notification that service is running
5. Background service are not aware to the user and there are some restriction on background services from API level 26
6. Bound services allow components interaction with other process or same process
7. **onStartCommand()** this method is called once startService() method is called indication service is started
8. **onBind()** is invoked by calling bindService() method, indicates service is bound to the particular components
9. **onCreate()** method is called only once in the life time of service
10. **onDestroy()** method is used to indicate service is stopped, this method can be invoked by calling stopService() or stopSelf() methods



➢ Life cycle of services



➢ Started services

1. Started services are started by other component and call will be invoked onStartCommand()
2. **startService()** is the method which is used to start the service and service runs in the main thread of the application
3. Best practice is to create worker thread to do the long running task in the service so main thread will not be blocked
4. Service can be started in three ways
5. **START_NOT_STICKY** once system kills service, service will not be recreated unless pending intent to delivered
6. **START_STICKY** If system kills service, service will be recreated without last intent for example media player
7. **START_REDELIVER_INTENT** Service will recreate with last intent for example downloading file
8. **stopService() and stopSelf()** is the method which is used to stop the service.
9. If onStartCommand() is handling multiple request it should not be stopped other - wise other request will also stops
10. To overcome this concurrent stopping problem we should use **stopSelf(int)** which takes request id as argument, only that request will be stopped

➢ Bound services

1. Bound service are used to serve the particular components, if no components are attached then service will be killed
2. **bindService()** is used to bind the service to the component
3. **IBinder** is used for the communication with the component and service
4. **unBindService()** is used to detach service with component

➢ Foreground services

1. Foreground services are aware to the user, from API level 28 we must need **FOREGROUND_SERVICE permission**
2. StartForeGround (ID, notification) method is used to start the foreground service which take two argument one is ID, the notification
3. stopForeGround() is used to stop the foreground service

➢ Started services

```
/** Started Service */
class MyStartedService : Service() {
    private val list = ArrayList<MediaPlayer>()
    override fun onCreate() { super.onCreate() }

    override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
        super.onStartCommand(intent, flags, startId)

        val mediaPlayer = MediaPlayer.create( context: this, Settings.System.DEFAULT_RINGTONE_URI)
        list.add(mediaPlayer)
        mediaPlayer.isLooping = true
        mediaPlayer.start()

        return START_STICKY
    }

    override fun onDestroy() {
        for (player in list){ player.stop() }
        super.onDestroy()
    }

    override fun onBind(p0: Intent?): IBinder? {
        return null
    }
}
```

```
val intent = Intent(context, MyStartedService::class.java)
fragmentVm.startServiceBtn.observe(viewLifecycleOwner, Observer { it: Boolean!
    requireActivity().startService(intent)
})

fragmentVm.stopServiceBtm.observe(viewLifecycleOwner, Observer { it: Boolean!
    requireActivity().stopService(intent)
})
```

➢ Bound services

```
/** bind service will be call only once on client is bound to service, bindService()*/
override fun onBind(p0: Intent?): IBinder? { return iBinder }

override fun onRebind(intent: Intent?) { super.onRebind(intent) }

override fun onUnbind(intent: Intent?): Boolean {
    super.onUnbind(intent)
    return true
}

override fun onDestroy() {
    super.onDestroy()
    chronometer.stop()
}
```

```
var myBoundService : MyBoundService? = null
private val serviceConnection = object : ServiceConnection{
    /** Service connection provides object of Service through iBinder*/
    override fun onServiceConnected(componentName: ComponentName?, binder: IBinder?) {
        val myBinder = binder as MyBoundService.MyBinder
        myBoundService = myBinder.getService()
    }

    /** this method will be called if service stops unexpectedly by system*/
    override fun onServiceDisconnected(p0: ComponentName?) { myBoundService = null }

    override fun onBindingDied(name: ComponentName?) { super.onBindingDied(name) }

    override fun onNullBinding(name: ComponentName?) { super.onNullBinding(name) }
}
```

```
fragmentVm.startServiceBtn.observe(viewLifecycleOwner, Observer { it: Boolean!
    val intent = Intent(context, MyBoundService::class.java)
    context?.bindService(intent, serviceConnection, Context.BIND_AUTO_CREATE)
})

fragmentVm.stopServiceBtm.observe(viewLifecycleOwner, Observer { it: Boolean!
    context?.unbindService(serviceConnection)
    myBoundService = null
})

fragmentVm.showTimeBtn.observe(viewLifecycleOwner, Observer { it: Boolean!
    if(myBoundService != null){
        fragmentVm.setTimeTv.value = myBoundService?.getTimeStamp()
    }else{
        fragmentVm.setTimeTv.value = "Service is disconnected"
    }
})
```

➢ Foreground services

```
val intent = Intent(context, MyForeGroundService::class.java)
fragmentVm.startServiceBtn.observe(viewLifecycleOwner, Observer { it: Boolean!
    requireContext().startService(intent)
})

fragmentVm.stopServiceBtm.observe(viewLifecycleOwner, Observer { it: Boolean!
    requireContext().stopService(intent)
})
```

```kotlin
class MyForeGroundService : Service() {
    companion object{
        val REQUEST_CODE = 100
        val CHANNEL_ID = "MyForeGroundServiceChannel"
    }
    fun showNotification(){
        val notificationIntent = Intent( packageContext: this, ServiceExampleActivity::class.java)
        val pendingIntent = PendingIntent.getActivity( context: this, REQUEST_CODE, notificationIntent,
            PendingIntent.FLAG_IMMUTABLE)

        val notificationChannel = NotificationChannel(CHANNEL_ID, name: "channel",
            NotificationManager.IMPORTANCE_DEFAULT)
        val notificationManager = getSystemService(NotificationManager::class.java)
        notificationManager.createNotificationChannel(notificationChannel)

        val notification = NotificationCompat.Builder( context: this,CHANNEL_ID)
            .setContentTitle("MY_FOREGROUND")
            .setContentText("App running in background")
            .setSmallIcon(R.drawable.ic_baseline_info_24)
            .setContentIntent(pendingIntent)
            .build()

        startForeground( id: 1, notification)
    }

    override fun onCreate() { super.onCreate() }

    override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
        super.onStartCommand(intent, flags, startId)
        showNotification()
        return START_STICKY
    }

    override fun onDestroy() { super.onDestroy() }

    override fun onBind(p0: Intent?): IBinder? { return null }
}
```

Intent Services

- **Introduction**
- **Service Vs Intent Service**
- **Implementation**

➢ Introduction

1. Intent service is the sub class of Service whatever the task that can be done by service can also be done by the Intent service
2. Intent service is deprecated in the API level 30 that is android 11 due to back ground restriction, user job intent service instead of intent service
3. Intent Service uses single worker thread to do the task instead of main thread

➢ Service Vs Intent Service

	Service	Intent Service
1	In service Intents will be executed concurrently	In intent service intents will be automatically queued up
2	runs in the main thread of the process	Single worker thread will be created for executing the tasks
3	All the tasks are executed in onstartCommand method	In intent service all the tasks are executed in the onHandleIntent method
4	To stop the service stopService() or stopSelf() methods will be used	Intent Service automatically stops once all the intents are handled
5	Service can be used in small background task	Intent service can be used for long running task

➢ Implementation

1. Create a class which extends IntentService and must override method OnHHandleIntent() which is to do background task
2. Declare this intent service class in manifest file
3. Use startService() and stopService() method to start and stop the service from fragment of activity
4. Use broadcast receiver for the communication from activity to service through intent extras
5. Intent service will automatically stop once all tasks are done and onDestroy() call back will be invoked

```kotlin
/** Intent Service*/
class MyIntentService : IntentService( name: "MyIntentService"){
    companion object{
        var COUNT = 0;
        val INTENT_SERVICE = "INTENT_SERVICE"
        val COUNT_KEY = "COUNT_KEY"
        val PROGRESS_KEY = "PROGRESS_KEY"
    }
    override fun onCreate() { super.onCreate() }

    override fun onStart(intent: Intent?, startId: Int) { super.onStart(intent, startId) }

    override fun onHandleIntent(intent: Intent?) {

        val handler = android.os.Handler(mainLooper)
        handler.post(Runnable {
            Toast.makeText(applicationContext, text: "onHandleIntent", Toast.LENGTH_SHORT).show()
        })

        COUNT ++
        var intent = Intent(INTENT_SERVICE)
        intent.putExtra(COUNT_KEY, COUNT)
        intent.putExtra(PROGRESS_KEY, value: "In progress")
        LocalBroadcastManager.getInstance( context: this).sendBroadcast(intent)

        Thread.sleep( millis: 10000)

        intent = Intent(INTENT_SERVICE)
        intent.putExtra(COUNT_KEY, COUNT)
        intent.putExtra(PROGRESS_KEY, value: "Work Done")
        LocalBroadcastManager.getInstance( context: this).sendBroadcast(intent)
    }

    override fun onDestroy() { super.onDestroy() }
}
```

```kotlin
/** Register Broadcast for communication with service */
broadcastManager = LocalBroadcastManager.getInstance(requireContext())
intentFilter = IntentFilter(INTENT_SERVICE)
broadcastManager.registerReceiver(broadCastReceiver, intentFilter)

/** Start or Stop the service on button click*/
val intent = Intent(context, MyIntentService::class.java)
fragmentVm.startServiceClick.observe(viewLifecycleOwner, Observer { it: Boolean!
    requireActivity().startService(intent)
})
fragmentVm.stopServiceClick.observe(viewLifecycleOwner, Observer { it: Boolean!
    requireActivity().stopService(intent)
})
```

```kotlin
/** Create local Broad cast in fragment or activity*/
val broadCastReceiver : BroadcastReceiver = object : BroadcastReceiver(){
    override fun onReceive(context: Context?, intent: Intent?) {
        val count = intent?.extras?.get(COUNT_KEY)
        val progress = intent?.extras?.get(PROGRESS_KEY)
        fragmentVm.serviceUpdateTV.value = "Task $count is $progress"
    }
}
```

Job Intent Services

- **Introduction**
- **Implementation**

Introduction

1. Intent service does not work well in android Orea, in android O a background service will run few minutes, once after app enters background, service will automatically stops and call ondestroy()
2. To over come this problem of intent Service android provided Job Intent service
3. Job intent service internally uses JoScehdulers for API 26 and above and uses Intent service for APi level 25 and below

4. So this is how Job Intent service provides backward compatibility
5. Pre android Oreo devices job intent service require wake lock, so we must mention WAKE_LOCK permission In manifest
6. For Orea devices job intent service require BIND_JOB_SERVICE permission in manifest
7. Job Intent Service is deprecated in the favor of android jet pack Work Manager

Implementation

1. Create a class which extends job intent service class, and declare the service class in manifest file
2. After declaring service, declare BIND_JOB_SERVICE inside service tag in manifest file

```
<service android:name=".services.jobIntentService.MyJobIntentService"
    android:permission="android.permission.BIND_JOB_SERVICE"/>
```

3. Override method onHandleWork(Intent) which takes intent as parameters and responsible for doing back ground task
4. expose on static method which is responsible for enqueue work using method enqueueWork() which take 4 arguments context, service claaa name, job_id, intent
5. This static method will be called from activity or fragment

```kotlin
/** Job Intent Service Example */
class MyJobIntentService : JobIntentService() {
    companion object{
        val JOB_ID = 1
        var WORK_COUNT = 0;
        val JOB_INTENT_SERVICE = "JOB_INTENT_SERVICE"
        val PROGRESS_KEY = "PROGRESS_KEY"
        fun enQueueMyWork(context: Context, intent : Intent) {
            enqueueWork(context, MyJobIntentService::class.java, JOB_ID, intent)
        }
    }
    override fun onCreate() { super.onCreate() }

    override fun onHandleWork(intent: Intent) {
        WORK_COUNT++
        val handler = Handler(mainLooper)
        handler.post(Runnable {
            Toast.makeText(applicationContext, text: "onHandleWork", Toast.LENGTH_SHORT).show()
        })

        var intent = Intent(JOB_INTENT_SERVICE)
        intent.putExtra(PROGRESS_KEY, value: "Work ${WORK_COUNT} start")
        LocalBroadcastManager.getInstance( context: this).sendBroadcast(intent)

        for(i in 1 ≤ .. ≤ 10){
            Thread.sleep( millis: 1000)
            val intent = Intent(JOB_INTENT_SERVICE)
            intent.putExtra(PROGRESS_KEY, value: "Work ${WORK_COUNT} Progress ${i*10} % complete")
            LocalBroadcastManager.getInstance( context: this).sendBroadcast(intent)
        }

        intent = Intent(JOB_INTENT_SERVICE)
        intent.putExtra(PROGRESS_KEY, value: "Work ${WORK_COUNT} done")
        LocalBroadcastManager.getInstance( context: this).sendBroadcast(intent)

        Thread.sleep( millis: 2000)
    }
    override fun onDestroy() { super.onDestroy() }
}
```

```kotlin
fragmentVm.enqueueWorkBtn.observe(viewLifecycleOwner, Observer { it: Boolean!
    val intent = Intent(context, MyJobIntentService::class.java)
    MyJobIntentService.enQueueMyWork(requireContext(),intent)
})
localBroadcastManager = LocalBroadcastManager.getInstance(requireContext())
intentFilter = IntentFilter(JOB_INTENT_SERVICE)
localBroadcastManager.registerReceiver(receiver,intentFilter)
```

```kotlin
val receiver : BroadcastReceiver = object : BroadcastReceiver(){
    override fun onReceive(context: Context?, intent: Intent?) {
        val progress = intent?.extras?.get(PROGRESS_KEY) as String
        fragmentVm.progressTV.value = progress
    }
}
```

```kotlin
override fun onDestroyView() {
    super.onDestroyView()
    localBroadcastManager.unregisterReceiver(receiver)
}
```

Job Schedulers

- **Introduction**
- **Job Info**
- **Job Service**
- **Implementation**

➢ Introduction

1. Job schedulers are used to schedule the task based on the contract
2. Contract is how our job need to be executed and in what condition jobs need to be executed
3. Job scheduler service is provided by android system this service is responsible for executing and scheduling the task
4. Job Info and Job service are the two classes helps in creating jobs
5. Job schedulers runs in main thread of the application

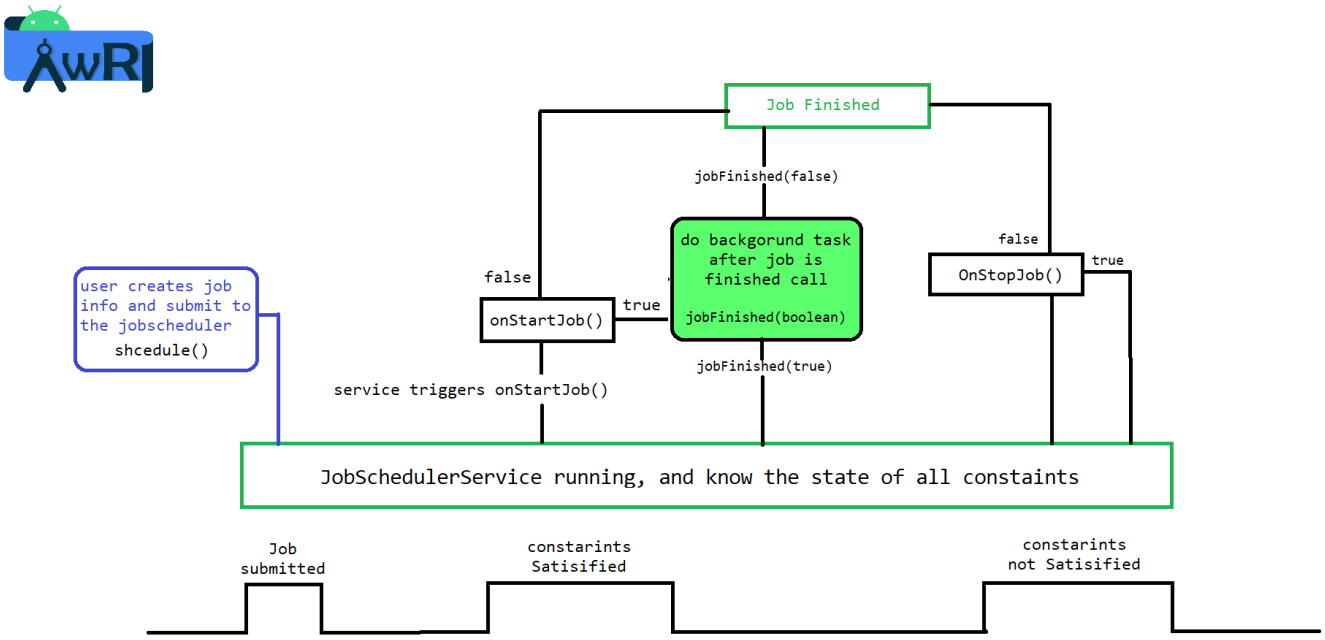
➢ Job Info

1. Job Info class is abstraction which hides task is to executed by the system
2. By using job info we can create contract which defines condition in which and what condition task is to be executed
3. This job info will be given to the job schedulers for the execution and we can post multiple work in one job info
4. **JobInfo.Builder()** is used to build the JobInfo and defines the contact
5. **Builder()** takes two arguments JobID and the ComponentName
6. Before android Q we must have to mention at least one contract
7. After android Q job schedulers can accept job info without any constraints

➢ Job Service

1. Job service is the class which will be extended by other class, this is the actualt job to be executed
2. This job will be given to the job scheduler service via Job Info with constraints
3. Job service is the component that must be declared in the manifest with permission **BIND_JOB_SERVICE**
4. We can enqueue one or more task inside the job service using JobInfo
5. Job service exposed with 3 methods
   a. onStartJob()
   b. onStopJob()
   c. JobFinished()

6. **onStartJob()** is the method is used to execute the Task once all the constraints are met.
7. onStartJob() returns Boolean, false indicates that job is done and indicates system to release all the resources like wake locks
8. if true is returned this indicates job is not yet finished and running in background, developer must indicate the system that job is finished using **jobFinished()** then system can release all the resources
9. **onStopJob()** is method get called when constraints are no longer satisfied, once this method is returned system releases all the resources
10. **onStopJob()** returns Boolean, false indicates job is no need to scheduled again
11. if false is returned by the onStopJob() then system has to reschedule the task again



➢ Implementation

1. Create a class which extends JobService class and override onStartJob() and onStopJob()
2. Create a component object and create Job Info using component
3. Schedule the job by obtaining JOB_SCHEDULE_SERVICE from getSystemService

```kotlin
/** JOB SCHEDULERS */
class MyJobSchedulers : JobService() {
    companion object{
        var COUNT = 0
        var WORK_DONE_COUNT = 0
        val JOB_ID = 123
        val JOB_SCHEDULER = "JOB_SCHEDULER"
        val PROGRESS_KEY = "PROGRESS_KEY"
    }
    override fun onStartJob(jobParam: JobParameters?): Boolean {
        COUNT ++
        var intent = Intent(JOB_SCHEDULER)
        intent.putExtra(PROGRESS_KEY, value: "job started $COUNT")
        LocalBroadcastManager.getInstance( context: this).sendBroadcast(intent)

        Thread(Runnable {
            for (i in 1 ≤ .. ≤ 10){
                Thread.sleep( millis: 1000)
                intent = Intent(JOB_SCHEDULER)
                intent.putExtra(PROGRESS_KEY, value: "job progress ${i*10}% complete")
                LocalBroadcastManager.getInstance( context: this).sendBroadcast(intent)
            }

            Thread.sleep( millis: 1000)
            Handler(mainLooper).post(Runnable {
                WORK_DONE_COUNT ++
                intent = Intent(JOB_SCHEDULER)
                intent.putExtra(PROGRESS_KEY, value: "job done ${WORK_DONE_COUNT}")
                LocalBroadcastManager.getInstance( context: this).sendBroadcast(intent)
                jobFinished(jobParam, wantsReschedule: true)
            })
        }).start()
        return true
    }
    override fun onStopJob(p0: JobParameters?): Boolean { return true }
}
```

```kotlin
/** In Activity onCreate() */
val intentFilter = IntentFilter(JOB_SCHEDULER)
LocalBroadcastManager.getInstance( context: this).registerReceiver(receiver,intentFilter)
val jobScheduler = getSystemService(JOB_SCHEDULER_SERVICE) as JobScheduler

activityVm.startJobBtn.observe( owner: this, Observer { it: Boolean!
    val componentName = ComponentName(applicationContext, MyJobSchedulers::class.java)
    val jobInfo = JobInfo.Builder(JOB_ID,componentName).build()
    val result = jobScheduler.schedule(jobInfo)

    if(result == JobScheduler.RESULT_SUCCESS){
        Toast.makeText(applicationContext, text: "Job scheduled", Toast.LENGTH_SHORT).show()
    }else{
        Toast.makeText(applicationContext, text: "Job not scheduled", Toast.LENGTH_SHORT).show()
    }
})

activityVm.cancelJobBtn.observe( owner: this, Observer { jobScheduler.cancel(JOB_ID) })
```

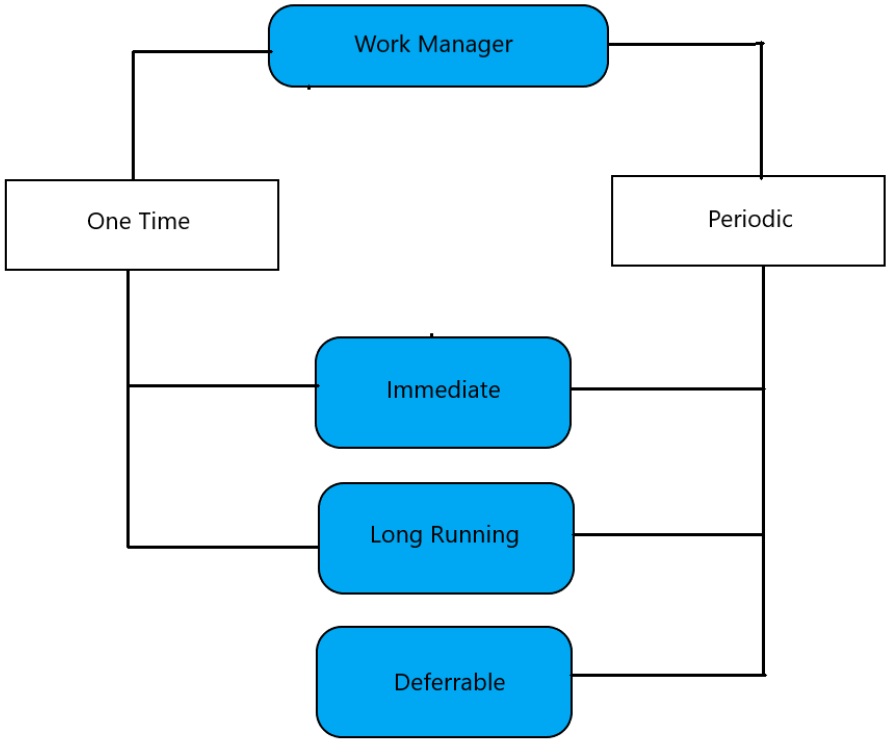
```kotlin
/** Local broad cast in Activity class*/
private val receiver : BroadcastReceiver = object : BroadcastReceiver(){
    override fun onReceive(p0: Context?, p1: Intent?) {
        val progress = p1?.extras?.get(PROGRESS_KEY) as String
        activityVm.updateProgressTv.value = progress
    }
}
```

Job Schedulers

- **Introduction**
- **Features**
- **Defining work**
- **Defining work request**
- **Work state**
- **Managing work**
- **Observing work progress**
- **Chaining work together**

Introduction

1. Work manager is recommended solution for persistent work that means work is persistent even though app restart and reboots.
2. Work manager supports two types of work request
   a. One time work request
   b. Periodic work request
3. Work manager has 3 types of Persistent work
   a. Immediate
   b. Long running
   c. Deferrable
4. Immediate task must begin immediately and complete soon
5. Long running task might run longer period of time
6. Deferrable tasks are scheduled task may start later and can run periodically



- **Features**
- **Defining work**
- **Defining work request**
- **Work state**
- **Managing work**
- **Observing work progress**
- **Chaining work together**