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

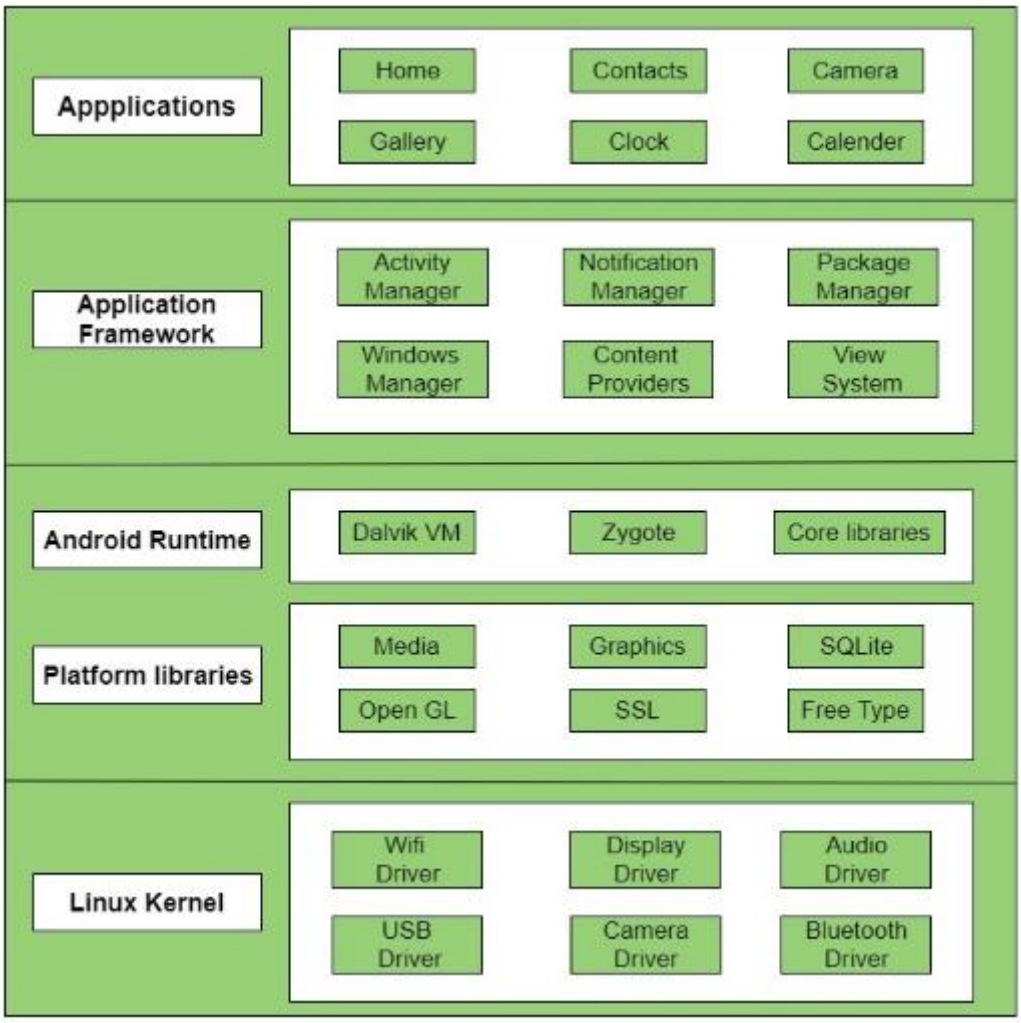


Figure 1.1

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

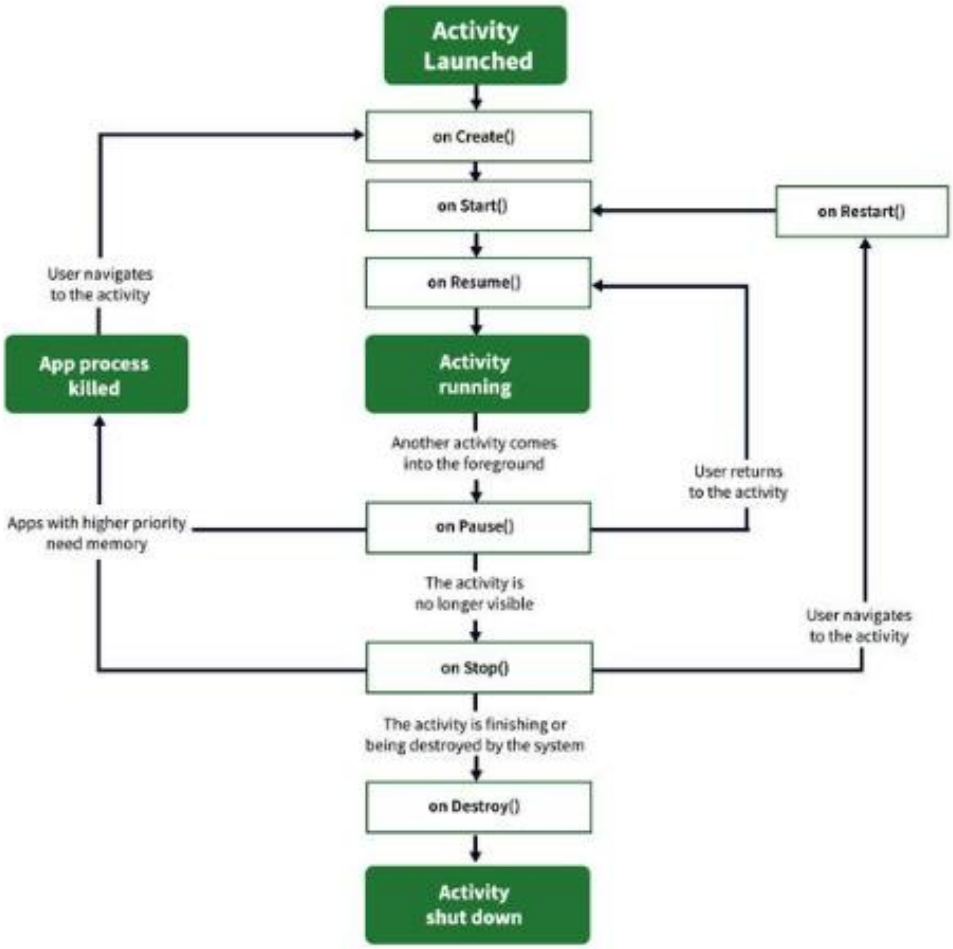
- Introduction
- Activity Life Cycle
- Use cases
- Save state of an activity
- Restore data on configuration change
- Cross questions

➢ 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)

10. example of receiving component

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

11. Example of sending components

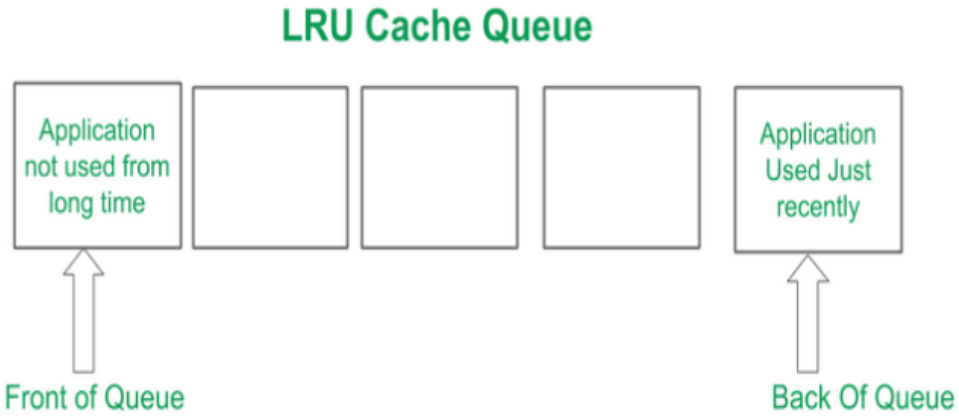
```
fun startActivity(){
    val bundle : Bundle = Bundle();
    bundle.putString("NAME","Ravivarma")
    val intent : Intent = Intent( packageContext: this@FirstActivity,SecondActivity::class.java)
    intent.putExtras(bundle);
    startActivity(intent)
}
```

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'

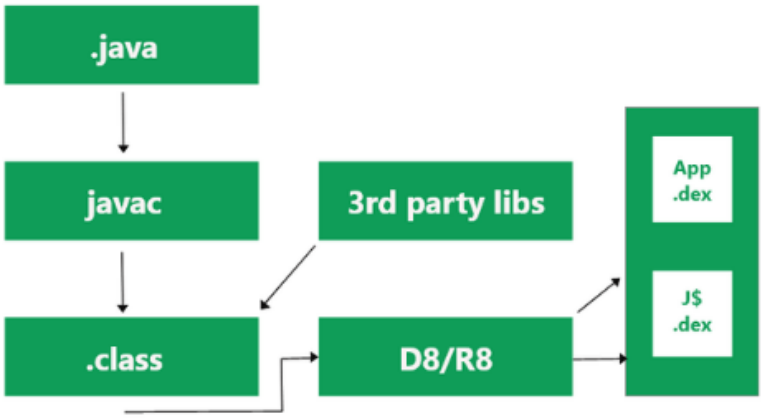
2. Add **corelibraryDesugaringEnabled true** in compileOption block

```
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

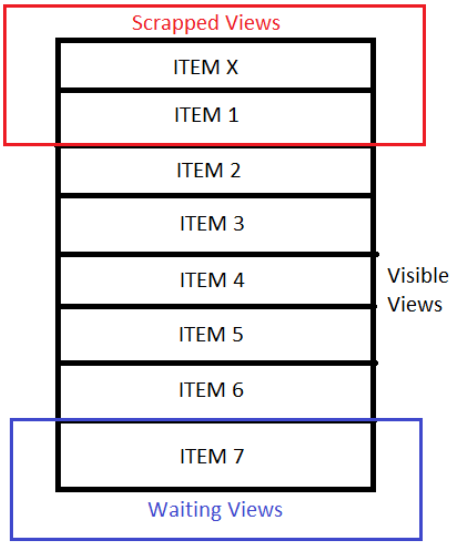
- **Introduction**
- **Working of recycler view**
- **Implementing recycler view**
- **Optimizing recycler view**

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

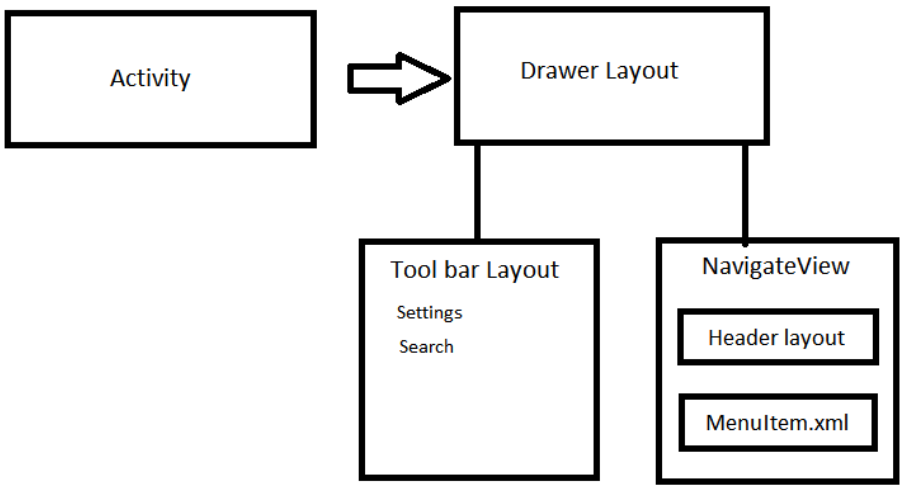
- **Introduction**
- **Implementation**

➢ 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

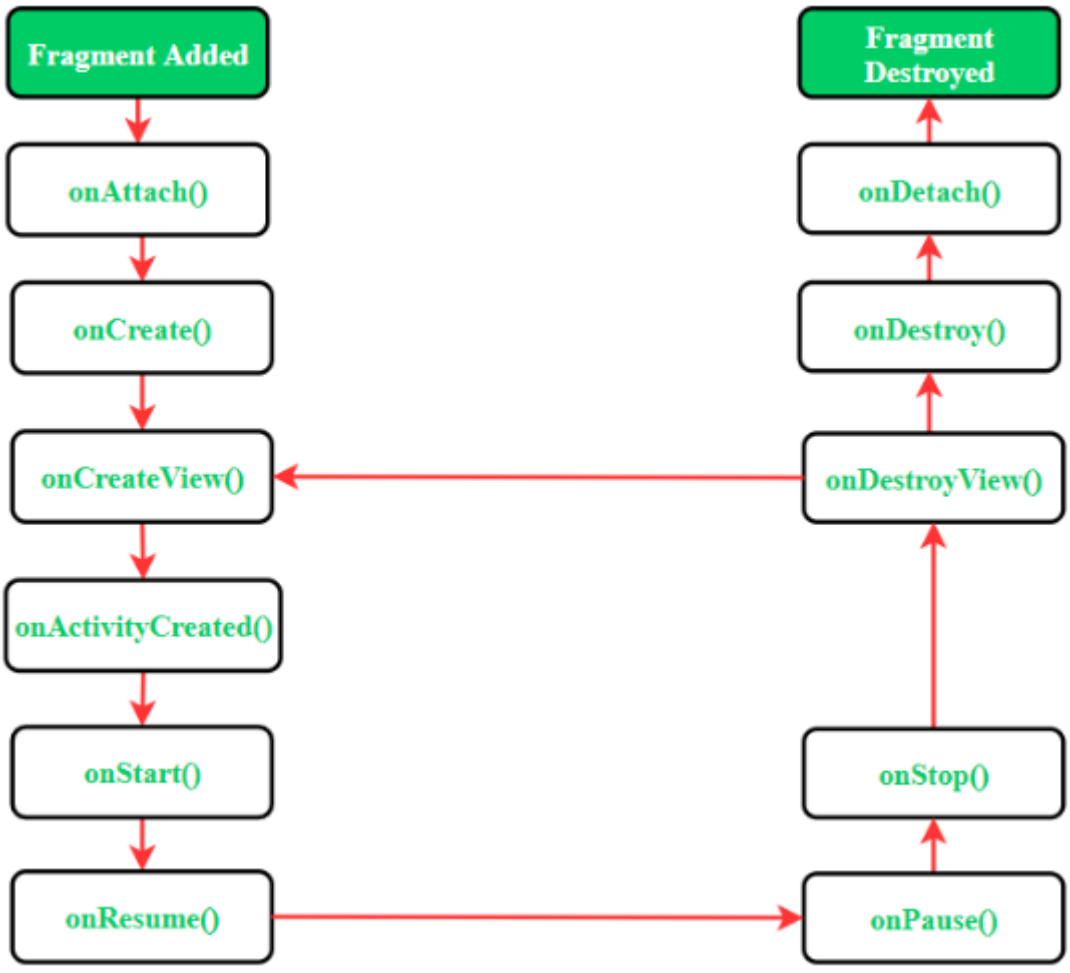
- **Introduction**
- **LifeCycle of Fragments**
- **Implementation**
- **Transaction**
- **Difference between add() and replace()**
- **Difference between activity and fragment**
- **Cross questions**

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

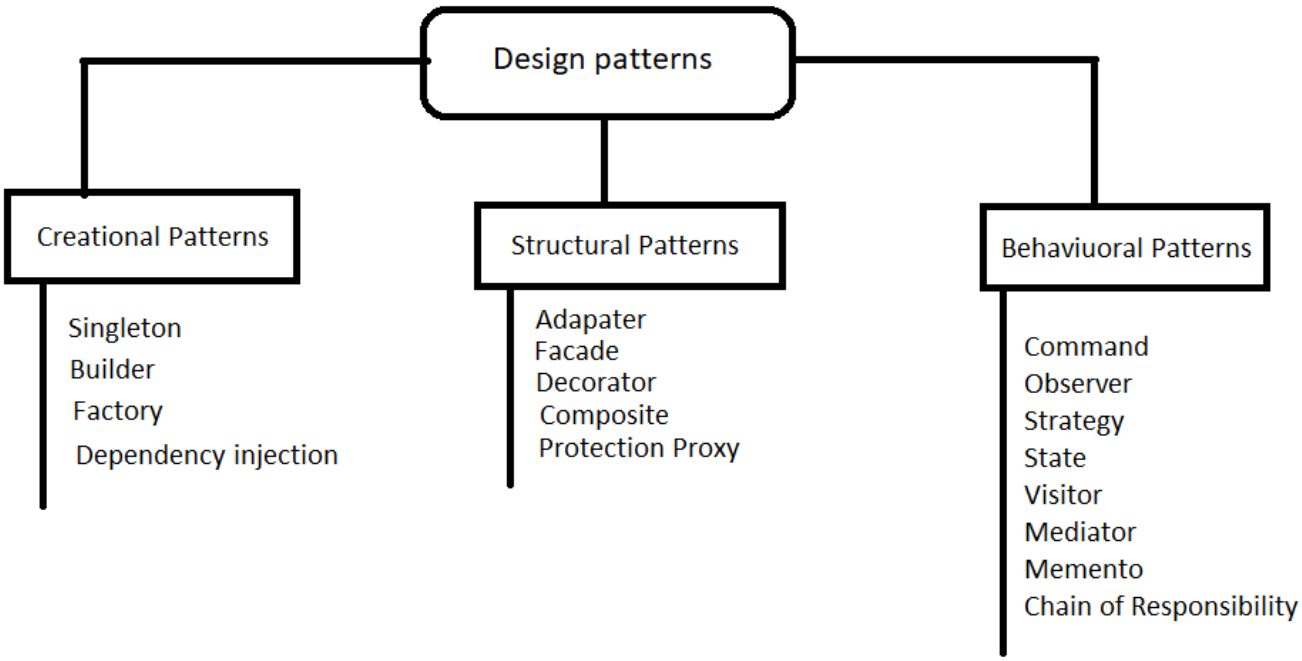
1. If we start fragment from activity buttons of the activity will overlapped on fragment container view, activity buttons get higher priority than other view.
2. To avoid overlapping of button on fragment container view wrap buttons in any of the view group ex constraint layouts

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

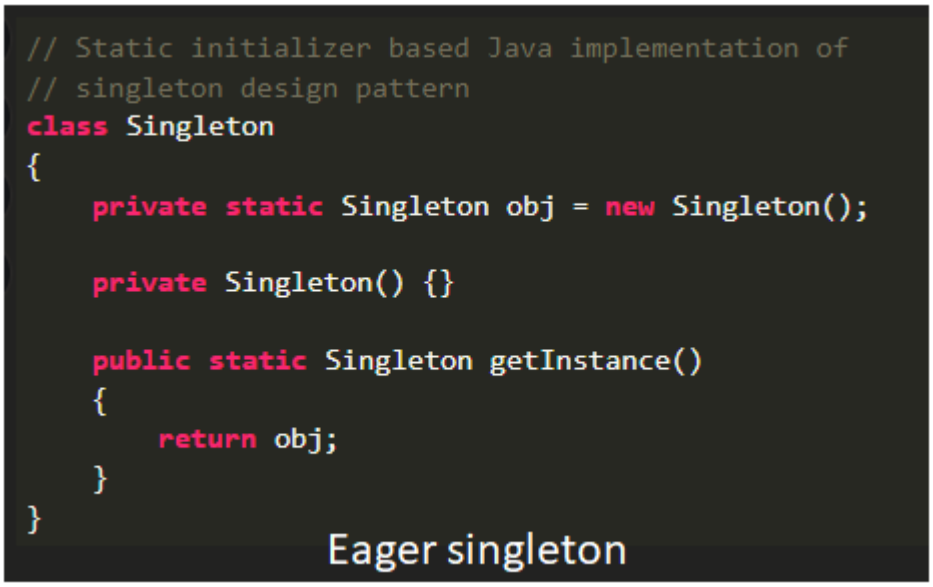
4. Classic singleton are not thread safe if we start two threads at same time different objects may get created for singleton
5. Using CountDownLatch we can demonstrate classic singleton are not safe

```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**

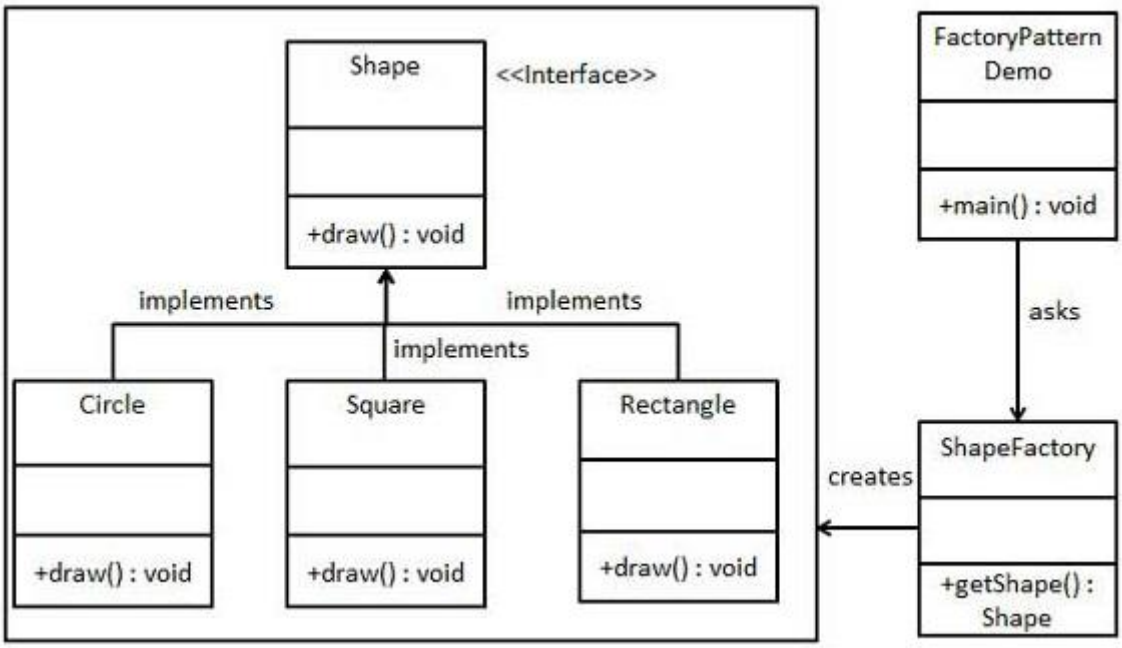
```
public class JavaCarBuilder {
    private int carModel;
    private String brand;
    private String color;

    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

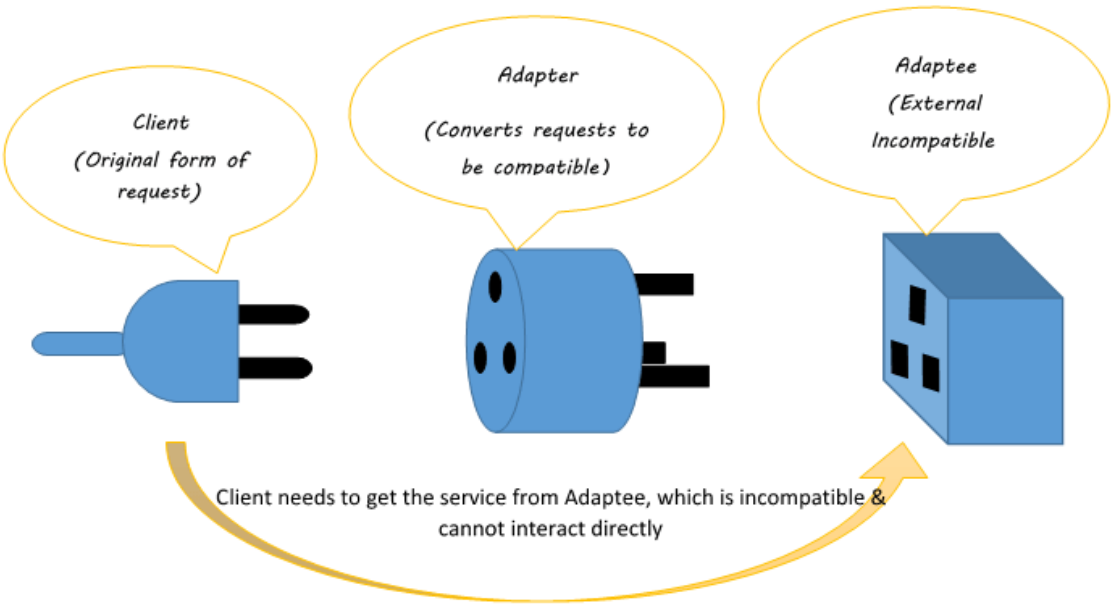


Figure 1-Adapter Pattern Concept

4. **Example**

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

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

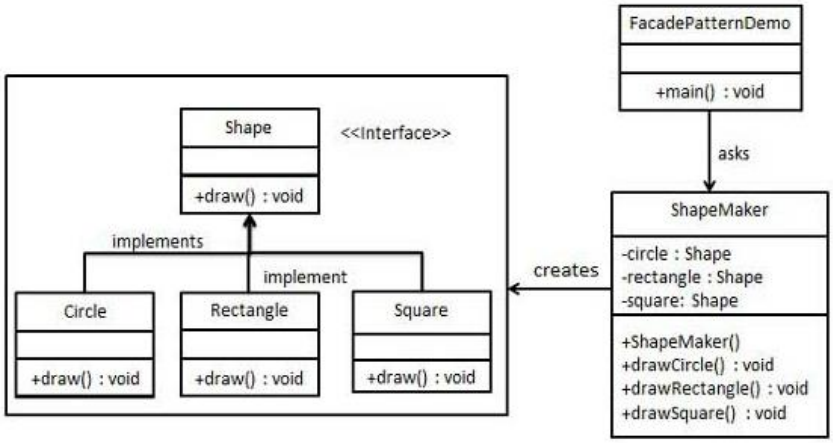
```kotlin
/** Adaptee interface */
interface UkPlug {
    fun provideElectricity() : String
}

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

```kotlin
/** 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()
    }
}
```

```kotlin
/** 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
}
```

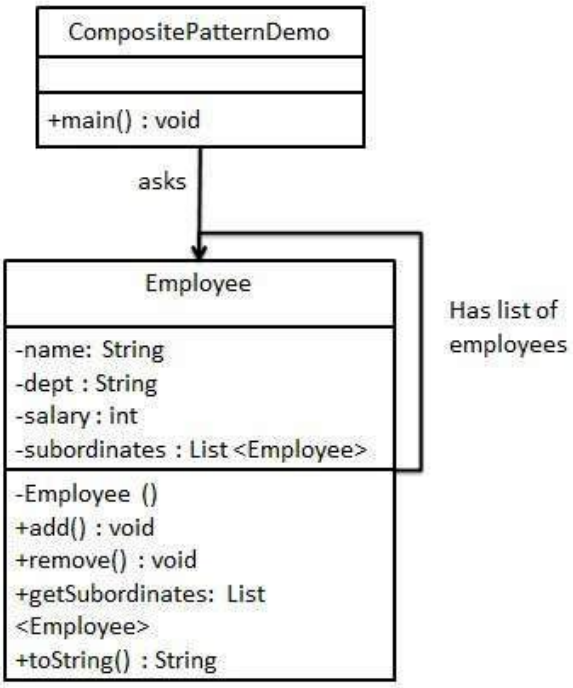
```
class Circle : IShape {
    override fun draw(): String {
        return "Circle"
    }
}
```

```
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]

}
```