

# ANDROID AND KOTLIN MIX QUESTION

## Launch Modes in Android

There are four launch modes for the activity.

1. Standard
2. SingleTop
3. SingleTask
4. SingleInstance

### Standard (Default)

- ☐ This is the default launch mode for activities.
- ☐ Each time an activity is launched, a new instance of the activity is created on top of the activity stack.
- ☐ If the activity already exists in the stack, still, a new instance will be created on top of it.

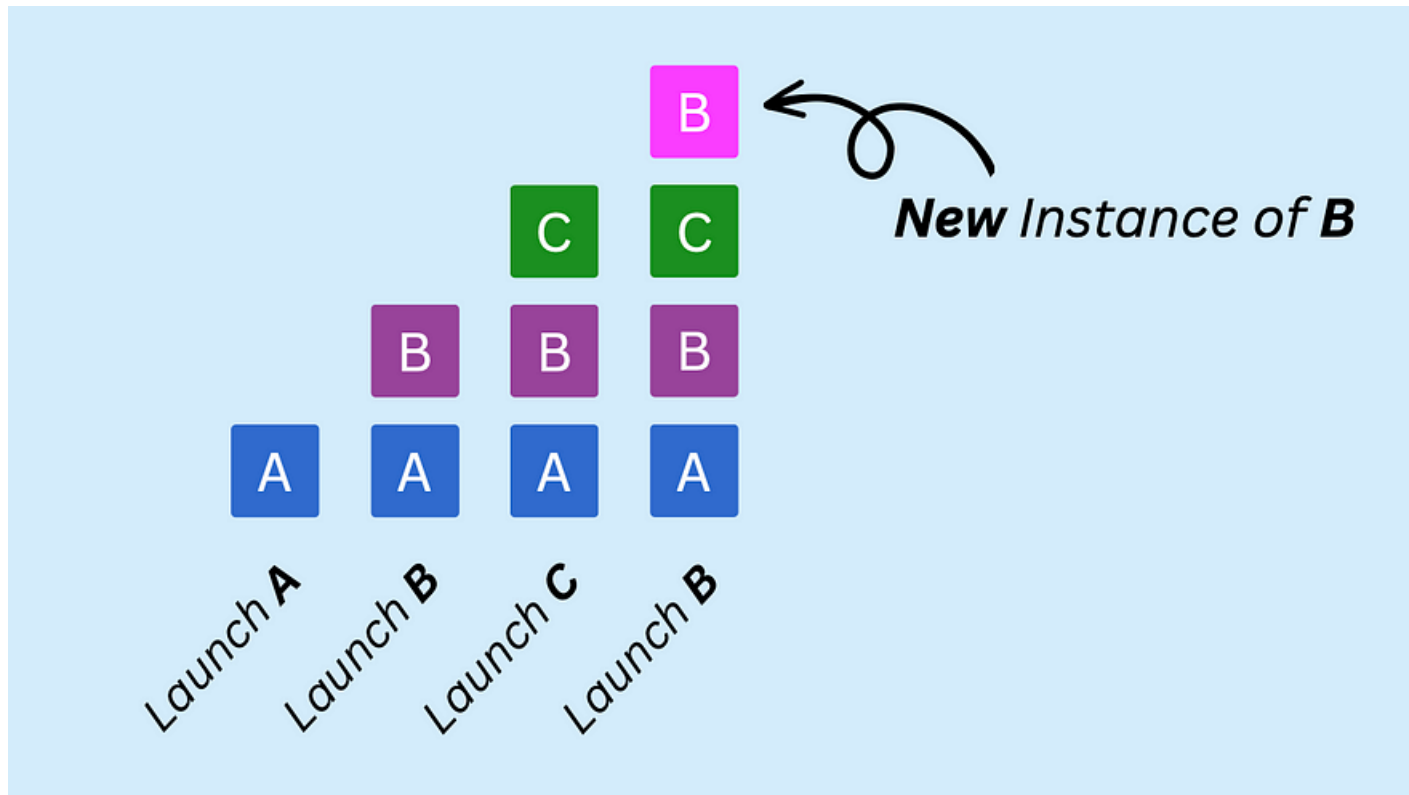


Image: Standard

### SingleTop

☐ With this launch mode, if the activity is already at the top of the stack, a new instance will not be created. Instead, the **onNewIntent()** method of the existing instance will be called with the new intent.

☐ If the activity is not at the top of the stack, a new instance will be created on top of the stack.

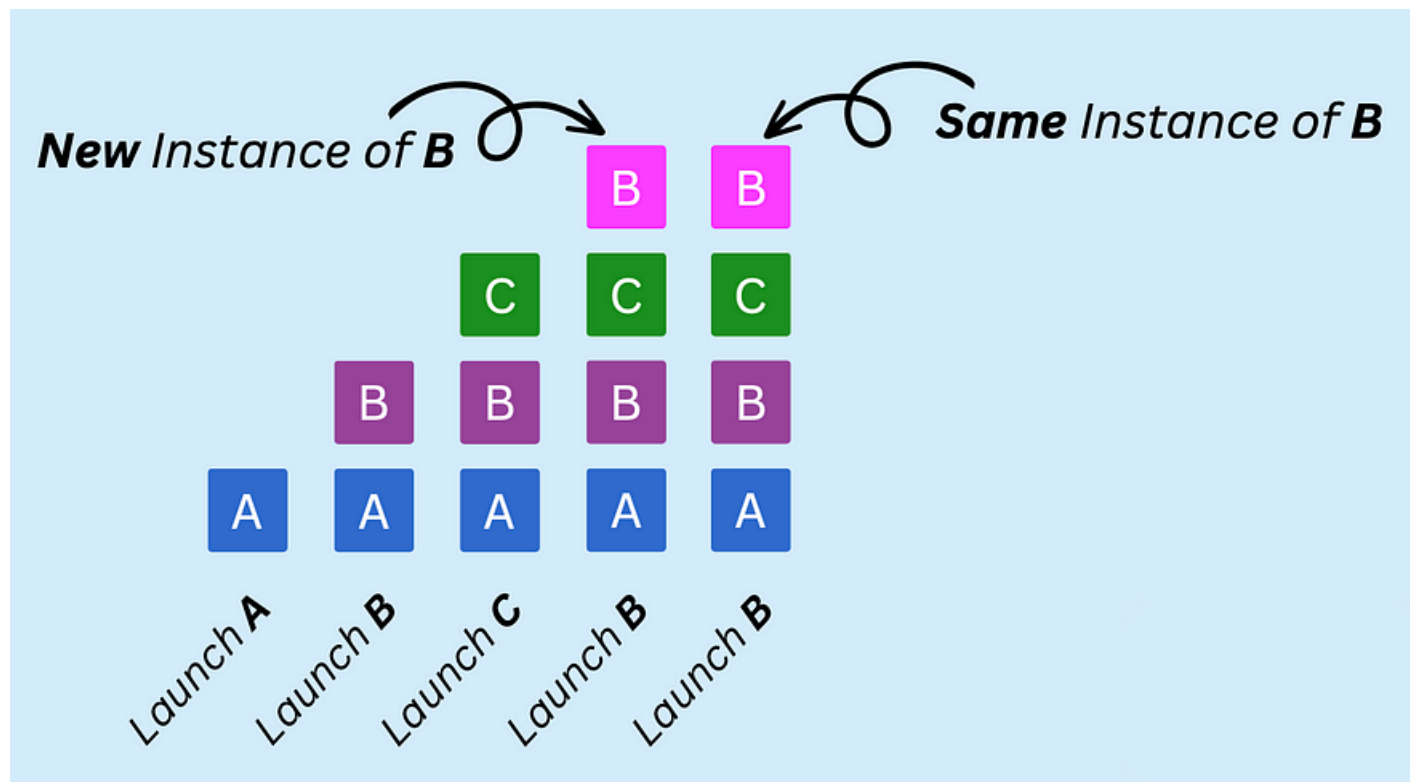


Image: SingleTop

### SingleTask

- ☐ In this launch mode, If there are no existing instances of the activity in the task, a new instance is created.
- ☐ If there is an existing instance of the activity at the top of the task, the system will pass the intent data to the `onNewIntent()` method of that existing instance.
- ☐ If there is an existing instance of the activity in the task but not at the top, the task is brought to the foreground, and all the activities above the singleTask activity in the stack are cleared or destroyed.

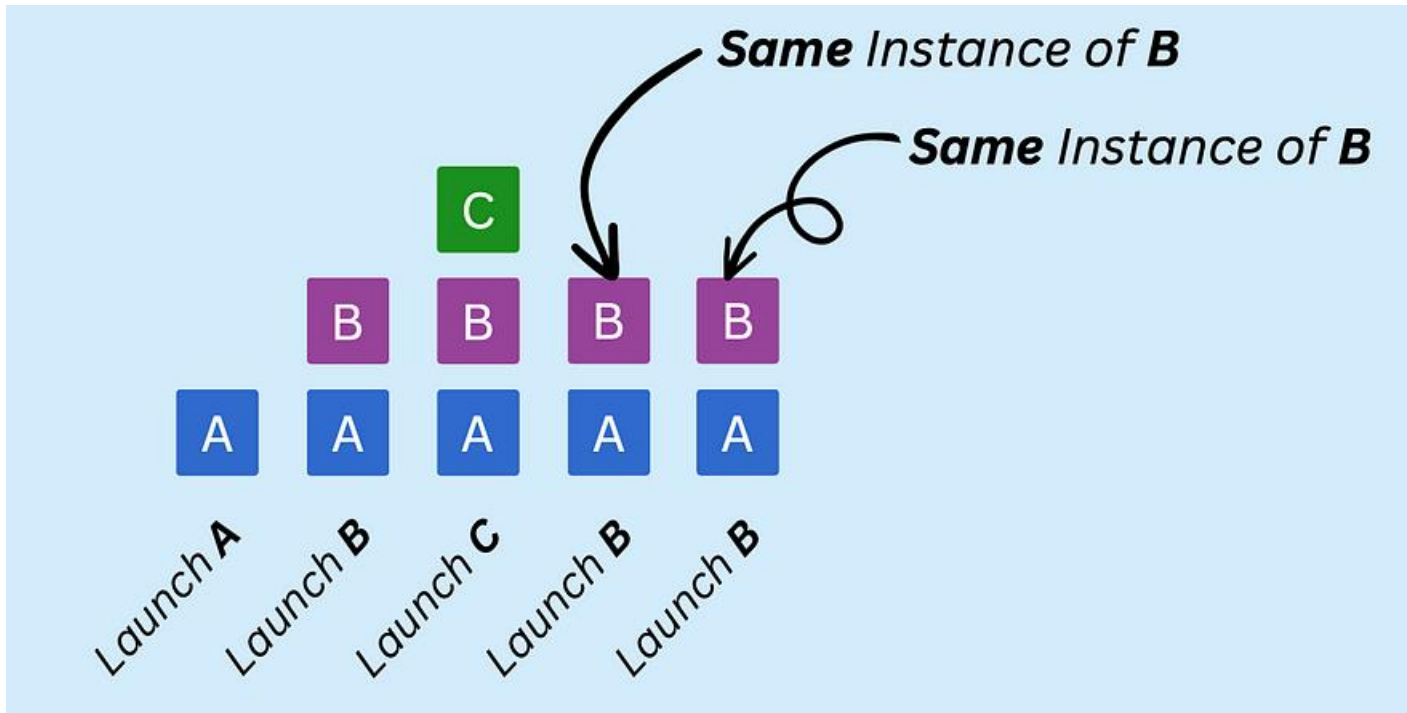


Image: SingleTask

### SingleInstance

- ☐ In SingleInstance launch mode, it creates a new task and ensures that no other activities (even from different applications) are placed in the same task.
- ☐ The SingleInstance activity is always placed in a new task and is isolated from the rest of the application and other tasks.

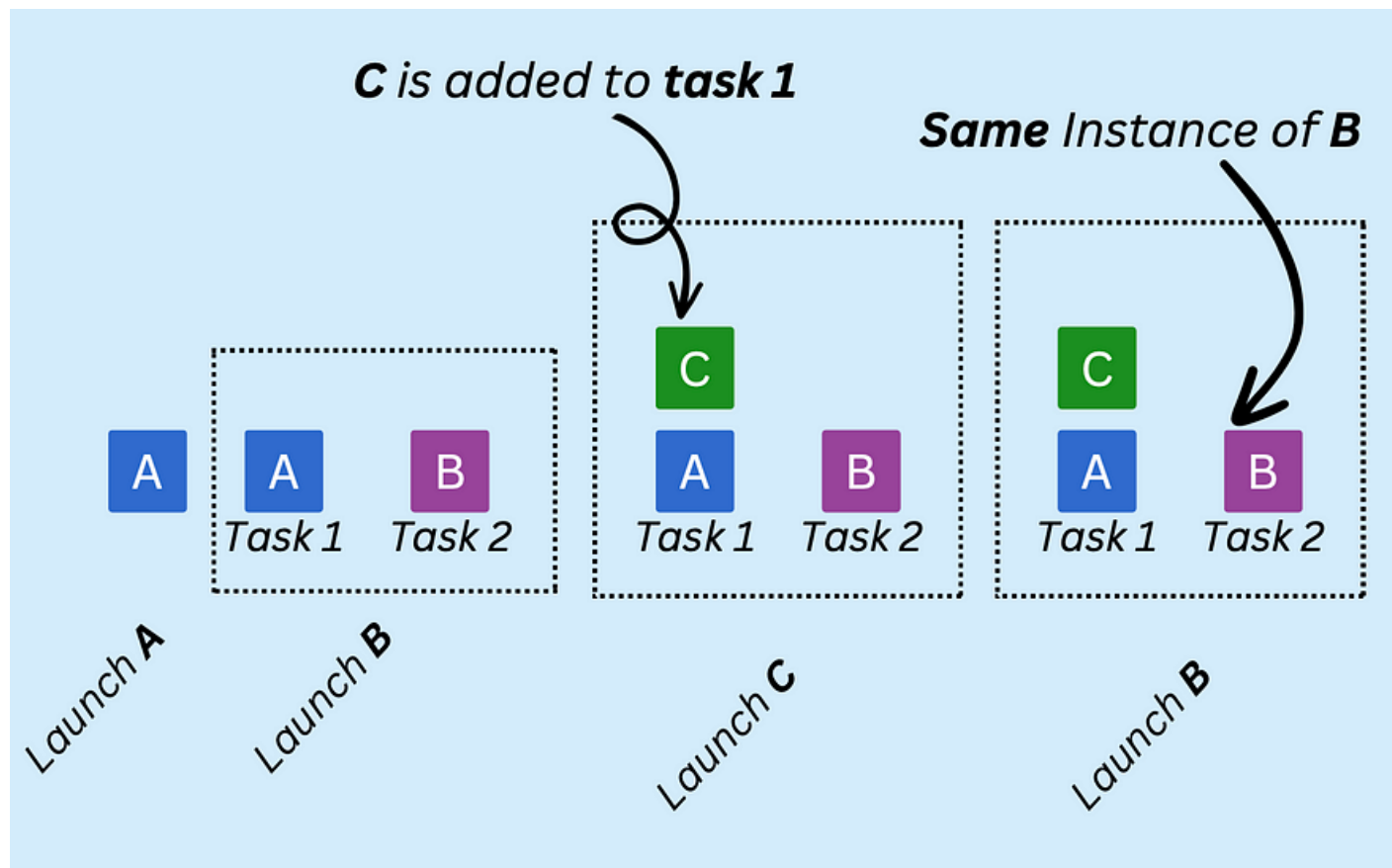
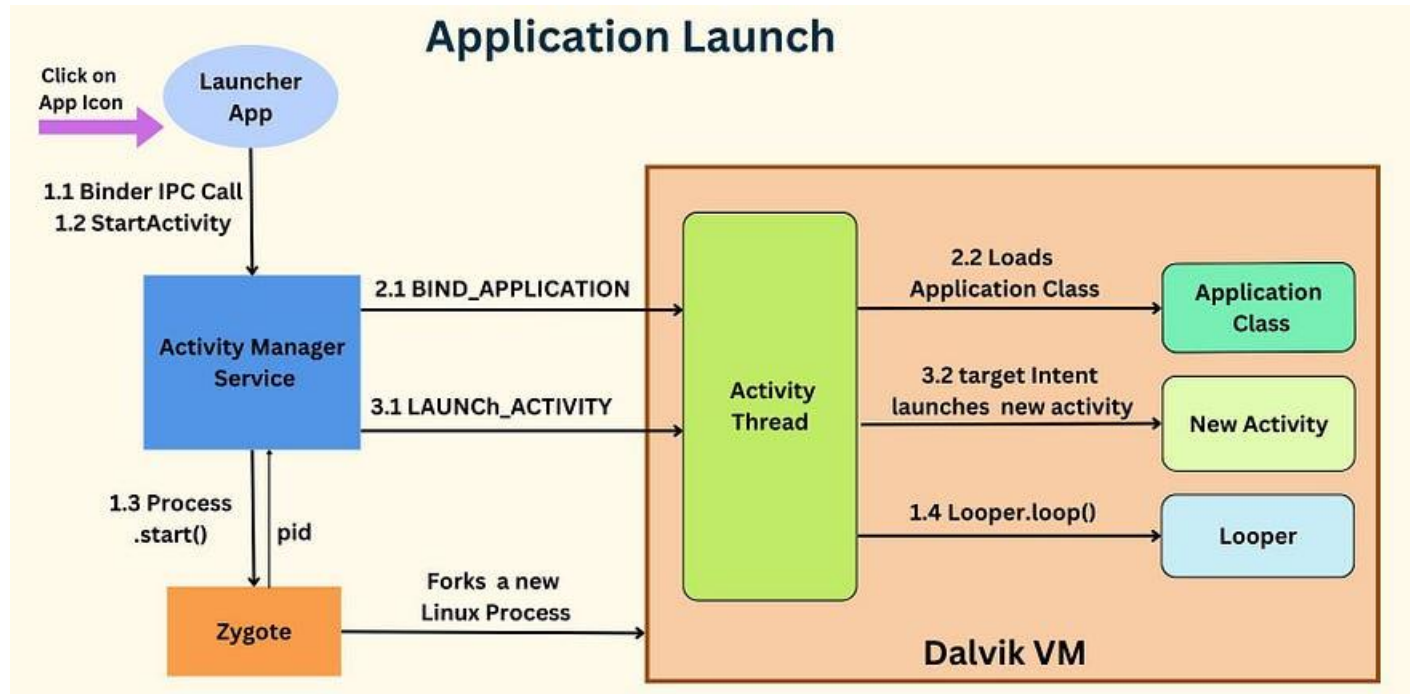


Image: SingleInstance

**What happens when you click on the app icon behind the scenes?**



When you click on the app icon, there are three distinct phases that occur during the Process launch.

### 1 Process Creation:

- The **ActivityManagerService** initiates the process creation by using the `startProcessLocked()` method.
- This method sends the necessary arguments to the **Zygote process** through a socket connection.
- The Zygote process then forks itself and calls **ZygoteInit.main()** which then instantiates **ActivityThread** object and returns a process id of a newly created process.
- **ActivityThread** calls **Looper.loop()** and runs infinitely.

### 2 Binding Application:

- The next step is to attach the newly created process to a specific application.
- This is done by calling the **bindApplication()** method on the **ActivityThread** object.
- The method sends a **BIND\_APPLICATION** message to the message queue.
- The **Handler** object retrieves this message and triggers the message-specific action, which is handled by the **handleBindApplication()** method.
- Inside this method, the **makeApplication()** method is invoked to load the application-specific classes into memory.

### 3 Launching Activity / Starting Service / Invoking Intent Receiver:

- After the previous steps, the system now has the process responsible for the application, with the application classes loaded into its private memory.
- The sequence of steps to launch an activity is the same for both a newly created process and an existing process.

- The actual process of launching an activity begins in the ***realStartActivity()*** method, which calls ***scheduleLaunchActivity()*** on the application thread object.
- This method sends a ***LAUNCH\_ACTIVITY*** message to the message queue.
- The ***handleLaunchActivity()*** method handles this message and initiates the necessary actions to launch the activity.

These three phases collectively describe the process that occurs behind the scenes when you click on an app icon.

## JvmStatic, JvmOverloads, JvmField Annotations in Kotlin



**@JvmStatic:**

```
object DataBase {
    fun runSql(query: String) {}
}
```

### In Java

```
DataBase.INSTANCE.runSql("SELECT * FROM Table");
```



```
DataBase.runSql("SELECT * FROM Table");
```



In object or companion object, if we use **@JvmStatic** annotation on **methods or properties**. It will create additional **static methods** and **static setter and getter methods** in that object or the class containing the companion object.

```
In Kotlin:
object DataBase {
    @JvmStatic
    fun runSql(query: String) {}
}
```

```
In Java:
DataBase.runSql("SELECT * FROM Table");
```

### @JvmOverloads:

```
data class Event(val name: String, val screenName: String = "")
```

### In Kotlin

```
Event("PhotoClick", "Profile")
```



```
Event("ButtonClick")
```



### In Java

```
new Event("PhotoClick", "Profile");
```



```
new Event("ButtonClick");
```



Kotlin function or constructor with default parameters, they will be visible in Java only as a full signature, with all parameters present.

When we will use **@JvmOverloads**, compiler will generate **overloads** for this function that **substitute default parameter values**



```
In Kotlin:
data class Event @JvmOverloads constructor (
    val name: String, val screenName: String = "" )
```

```
In Java:
new Event("ButtonClick")
```

### @JvmField:

```
data class Event( val name: String, val screenName: String = "" )
```

#### In Kotlin

```
val event = Event("PhotoClick", "Profile")
println(event.name) ✓
```

#### In Java

```
Event event = new Event("PhotoClick", "Profile");
System.out.println(event.getName()); ✓
```

```
System.out.println(event.name); ✗
```

When accessing Kotlin properties from Java code, we need to use setter and getter methods.

To use a **property** as a **field** in **Java code**, we can annotate it with **@JvmField**. This way, the compiler will not generate setter and getter methods for it.

```
In Kotlin:
data class Event (@JvmField val name: String, val screenName: String = "")
```

```
In Java:
Event event = new Event ( "Photoclick", "Profile"):
System.out.println(event.name):
```

## Types of Function In Kotlin



### Extension Functions:

*Extension functions* are like extensive properties attached to any class in Kotlin. Extension functions are used to add methods or functionalities to an existing class even without inheriting the class.

**For example:** Suppose, we have views where we need to play with the visibility of the views.

### Higher-Order Function:

A **high order function (Higher level function)** is a function which can accept a function as a parameter or can return a function is called **Higher-Order function**. Instead of Integer, String, or Array as a parameter to function, we will pass anonymous function or lambdas. Frequently, lambdas are passed as parameter in Kotlin functions for the convenience.

### Scope Functions:

*By definition, Scoped functions are functions that execute a block of code within the context of an object.*

Well, what does this mean? These functions provide a way to give temporary scope to the object under consideration where specific operations can be applied to the object within the block of code, thereby, resulting in a clean and concise code. Not clear still right? Yeah. In Software Development, things are only better

understood by implementing rather than reading. So, let's go ahead and understand these scoped functions with some examples.

## Application of using scope functions

Scope functions make code more **clear**, **readable**, and **concise** which are Kotlin language's main features.

## Types of scope functions

There are five types of scope functions:

1. **let** : working with nullable objects to avoid NullPointerException.
2. **run** : operate on a nullable object, executing lambda expressions.
3. **with** : operating on non-null objects.
4. **apply** : changing object configuration.
5. **also** : adding additional operations.

## Infix function:

An infix function is used to call the function without using any bracket or parenthesis. You need to use the infix keyword to use the infix function.

## Inline function:

An **inline function** is declared with a keyword **inline**. The use of inline function enhances the performance of *higher order function*. The inline function tells the compiler to copy parameters and functions to the call site.

The inline function instructs compiler to insert complete body of the function wherever that function got used in the code. To use an inline function, all you need to do is just add an inline keyword at the beginning of the function declaration.

- Declaration of local classes
- Declaration of inner nested classes

- Function expressions
- Declarations of local function
- Default value for optional parameters

### Suspend function:

The ***suspend function*** is the building block of the Coroutines in Kotlin. The suspend function is a function that could be started, paused, and resume. To use a suspend function, we need to use the suspend keyword in our normal function definition.

### Recursion Function:

A *recursion function* is a function which calls itself continuously. This technique is called recursion.

```
fun functionName()
{
    functionName() //calling same function
}
```

### Lambda Function:

Lambda is a function which has no name. Lambda is defined with a curly braces **{ }** which takes *variable as a parameter* (if any) and body of function. The *body of the function* is written after variable (if any) followed by - > operator.

### Syntax of lambda:

```
{ variable -> body_of_function }
```

### Recursive Function:

A function that calls itself is known as a recursive function. And, this technique is known as ***recursion***.

A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

### How does recursion work in programming?

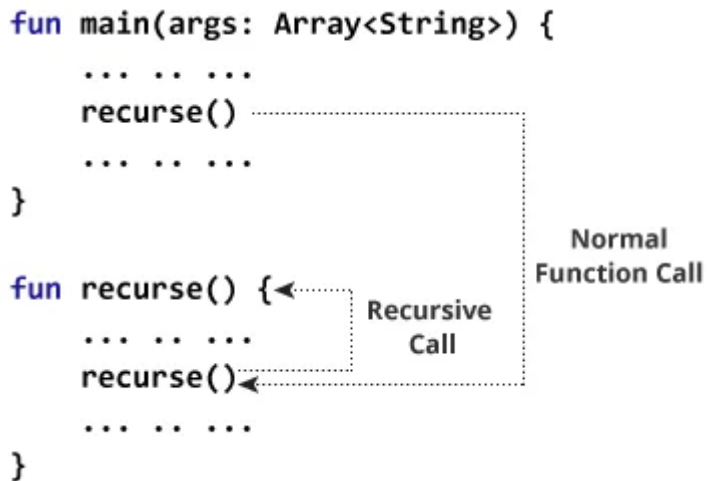
```

fun main(args: Array<String>) {
    ... ..
    recurse()
    ... ..
}

fun recurse() {
    ... ..
    recurse()
    ... ..
}

```

Here, the `recurse()` function is called from the body of `recurse()` function itself. Here's how this program works:

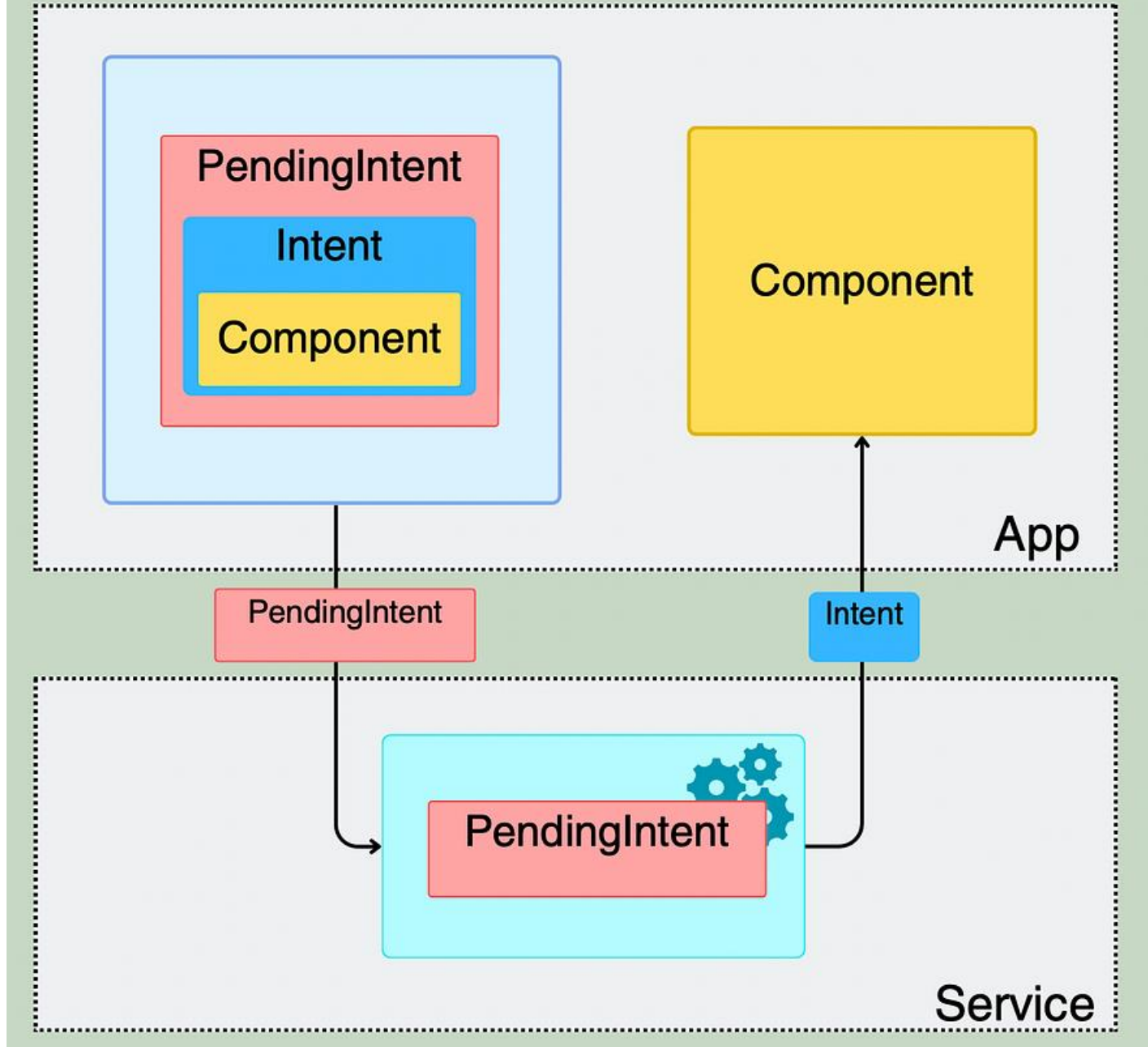


Here, the recursive call continues forever causing infinite recursion.

To avoid infinite recursion, if...else (or similar approach) can be used where one branch makes the recursive call and other doesn't.

## Pending Intent in Android

# Pending Intent in Android



## Pending Intent:

A **PendingIntent** is a token that you give to a foreign application (e.g. NotificationManager, AlarmManager, or other 3rd party applications), **which allows the foreign application to use your application's permissions to execute a predefined piece of code.**

```

val intent = Intent(applicationContext, MainActivity::class.java)
    .apply { ... }
val pendingIntent = PendingIntent.getActivity(
    applicationContext,
    NOTIFICATION_REQUEST_CODE,
    intent,
    PendingIntent.FLAG_IMMUTABLE
)
val notification = NotificationCompat.Builder(
    ...
).apply {
    setContentIntent(pendingIntent)
    ...
}.build()

notificationManager.notify(
    ...
notification
)

```

### Pending Intent Flags:

**FLAG\_IMMUTABLE** indicates that a PendingIntent is immutable, and the creator of the PendingIntent can only update it via the FLAG\_UPDATE\_CURRENT.

**FLAG\_MUTABLE** indicates that a PendingIntent is mutable.

**FLAG\_UPDATE\_CURRENT**, if the PendingIntent already exists, updates the extra data; otherwise, it creates a new one.

**FLAG\_NO\_CREATE** indicates that if the PendingIntent does not already exist, then return null instead of creating it.

## Kotlin: Inline, noinline, crossinline functions

# Inline functions

Inline

crossinline

noinline

## **Inline:**

When you mark a function as inline, the compiler will attempt to inline the function's code wherever it is called.




```
fun house() {  
    println("room")  
  
    kitchen()  
  
    println("washroom")  
}
```

```
inline fun kitchen() {  
    println("kitchen")  
}
```

//Compiled Version

```
public void house() {  
    System.out.println("room")  
    System.out.println("kitchen")  
  
    System.out.println("washroom")  
}
```

Inlined within  
house() method



### Inline with higher-order function:

Inlining the function that contains lambda expressions will also inline the lambda itself, and prevents the creation of additional obj for the lambda instances.

```

fun house() {
    println("room")

    kitchen {
        println("cooking")
    }

    println("washroom")
}

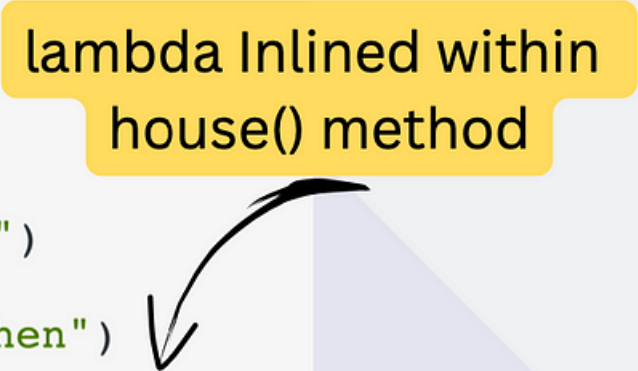
inline fun kitchen(cook: () -> Unit) {
    println("kitchen")
    cook()
    println("serving")
}

//Compiled Version
public void house(){
    System.out.println("room")

    System.out.println("kitchen")
    System.out.println("cooking")
    System.out.println("serving")
    System.out.println("washroom")
}

```

lambda Inlined within  
house() method



#### Inline with Return:

If we **return** from the lambda, it will treat return as local return and compiler will discard all the code after it.

```
fun house() {
    println("room")

    kitchen {
        println("cooking")
        return
    }

    println("washroom")
}

inline fun kitchen(cook: () -> Unit) {
    println("kitchen")
    cook()
    println("serving")
}

//Compiled Version
public void house(){
    System.out.println("room")

    System.out.println("kitchen")

    System.out.println("cooking")
}
```

All code after lambda  
is discarded



### **crossinline:**

If we want to restrict that no one can return from the lambda, we can mark the function type parameter as ***crossinline***.

```
fun house() {  
    println("room")
```

```
    kitchen {  
        println("cooking")  
        return  
    }  
    println("washroom")  
}
```

Compile Time Error:  
'return' is not allowed here

```
inline fun kitchen(crossinline cook: () -> Unit) {  
    println("kitchen")  
    cook()  
    println("serving")  
}
```

### noinline:

We mark function type parameter as `noinline`, compiler will not optimize the code for it, also we can not use `return` in it.

```

fun house() {
    println("room")

    kitchen {
        println("cooking")
    }

    println("washroom")
}

inline fun kitchen(noinline cook: () -> Unit) {
    println("kitchen")

    cook()
}

//Compiled Version
public void house(){
    System.out.println("room");

    Function0 fun0 = new Function0(){
        @Override
        public void invoke() {
            System.out.print("cooking");
        }
    }

    System.out.println("kitchen");

    fun0.invoke();

    System.out.println("washroom");
}

```

This lambda is not optimized and creating object of it