**Serializable** is a standard Java interface. You simply mark a class Serializable by implementing the interface, and Java will automatically serialize it in certain situations.

**Parcelable** is an Android specific interface where you implement the serialization yourself. It was created to be far more efficient and faster than Serializable, and to get around some problems with the default Java serialization scheme.

**Summarized :**

* Parcelable is faster than serializable interface
* Parcelable need more implemetation time of compared to serializable interface
* serializable interface is easier to implement
* Parcelable array can be pass via Intent in android
* serializable interface create a lot of temporary objects and cause quite a bit of garbage collection

**Q7. What is the difference between Service and IntentService ?**

**Service :**This is the base class for all services. When you extend this class, it’s important that you create a new thread in which to do all the service’s work, because the service uses your application’s main thread, by default, which could slow the performance of any activity your application is running.

**IntentService :** Service is a base class for IntentService Services that handle asynchronous requests (expressed as Intents) on demand. Clients send requests through startService(Intent) calls; the service is started as needed, handles each Intent in turn using a worker thread, and stops itself when it runs out of work.

**Q16. What is An android manifest file?**

Every application must have an AndroidManifest.xml file (with precisely that name) in its root directory. The manifest file provides essential information about your app to the Android system, which the system must have before it can run any of the app’s code.

**Q8. What is a Sticky Intent?**

Sticky Intent is a type of Intent which allows communication between a function and a service. This intent will stick with android system for future broadcast receiver requests. Sticky Broadcast() performs **sendBroadcast(Intent)** with an Intent. Sticky Broadcast is deprecated in Android 5+.

**Q9. What is Pending Intent in Android?**

Pending Intent is another type of intent which allows to trigger at some time in future, even when your application’s process is killed. This intent can be used by other application (e.g., Notification Manager, Alarm Manager or other 3rd party applications), which allows it to execute that intent with the same permissions as of our application.

## **compileSdkVersion:**

compileSdkVersion defines **which Android SDK version will be used by gradle to compile your app.**For example, if you try and use API 31 features but set compileSdkVersion to 30, you will get a compilation error. If you set compileSdkVersion to 31 you can still run the app on an API 30 device as long as your app’s execution paths do not attempt to invoke any APIs specific to API 31. You should set this to the highest API level that Google has made available. Using the highest available level ensures that all software libraries your project is using, compile successfully.

## **targetSdkVersion:**

The targetSdkVersion doesn't have anything to do with how your application is compiled or what APIs you can use. The targetSdkVersion indicates that you have tested your app on (presumably up to and including) the version you specify. This is like a certification or sign-off you are giving the Android OS as a hint to how it should handle your application in terms of OS features. Depending on the functionality of your app, it can target API versions lower than the current. For instance, you can target API 25 even if the current version is 33.

**1.** **What is the sealed class ?**

**Sealed** classes give us the flexibility of having different types of subclasses and also containing the state.

The important point to be noted here is the subclasses that are extending the Sealed classes should be either nested classes of the Sealed class or should be declared in the same file as that of the Sealed class.

**2.** **What is suspend function in Kotlin Coroutines?**

**Suspend function** is the building block of the Coroutines in Kotlin. 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.**

3. **What is the difference between Launch and Async in Kotlin Coroutines?**

The difference is that the launch{} does not return anything and the async{} returns an instance of Deferred<T>,

which has an await() function that returns the result of the coroutine

·like we have future in Java in which we do future.get() to the get the result.

In other words:

**launch**: fire and forget

**async**: perform a task and return a result

**4.** **What are lambdas expressions?**

**Lambdas** expressions are anonymous functions that can be treated as values i.e. we can pass the lambdas expressions as arguments to a function return them, or do any other thing we could do with a normal object

val add : (Int, Int) -> Int = { a, b -> a + b }

val result = add(9, 10)

**5.** **What are Higher-Order functions in Kotlin?**

· A higher-order function is a function that takes functions as parameters or returns a function. For example, A function can take functions as parameters.

**6.** **What are extension functions in Kotlin**

· Extension functions are like extensive properties attached to any class in Kotlin. By using extension functions,

· you can add some methods or functionalities to an existing class even without inheriting the class.

· For example: Let’s say, we have views where we need to play with the visibility of the views. So, we can create an extension function for views.

**7.** **What is an infix function in Kotlin?**

· 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.

**8.** **What is an inline function in Kotlin?**

· Inline function instruct 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.

**9.What is the difference between lateinit and lazy in Kotlin?**

· **lazy** can only be used for **val** properties, whereas **lateinit** can only be applied to var because it can’t be compiled to a final field, thus no immutability can be guaranteed.

· If you want your property to be initialized from outside in a way probably unknown beforehand, use **lateinit**.

**10.** **What is the use of @JvmStatic, @JvmOverloads, and @JvmFiled in Kotlin?**

· **@JvmStatic**: This annotation is used to tell the compiler that the method is a static method and can be used in Java code.

· **@JvmOverloads**: To use the default values passed as an argument in Kotlin code from the Java code, we need to use the @JvmOverloads annotation.

· **@JvmField**: To access the fields of a Kotlin class from Java code without using any getters and setters, we need to use the @JvmField in the Kotlin code.

# What are the differences between Lateinit and Lazy?

1**-** The modifier**“lateinit” is restricted to mutable(var) variable properties,** whereas the modifier “**lazy” is exclusively used with read-only(val) properties.**

2- A property marked with “lateinit” **can be assigned a value multiple times as needed during runtime**, whereas a property initialized with “lazy” can **only be assigned a value once upon its first use.**

3- It is **not possible to declare a primitive data type as a “lateinit” property,** whereas a **“lazy” property can be of either primitive or non-primitive data types.**(Such as Int)

4- While it is**not possible to ensure thread safety for a “lateinit” property**, for a **“lazy” property we have the option to choose from synchronization options such as SYNCHRONIZED, PUBLICATION, or NONE. (**That’s why we use lazy when using the Singleton design pattern.**)**

5- Unlike a**“lateinit” property, which cannot be declared as nullable**, a “lazy” property can be defined **as either nullable or non-nullable.**

6- While a**“lateinit” property cannot have a customized getter**, a **“lazy” property contains a block of code that runs the first time the property is called.**

7- Attempting to access a “lateinit” property **before it has been initialized results in a distinct exception that specifies the uninitialized property.** On the other hand, a **“lazy” property cannot be accessed before its initialization.** It is important to**note that a “lazy” property can be null, yet it will still be initialized the first time the property is accessed.**

**Send data from one fragment to other**

|  |
| --- |
| 1#  setFragmentResult( |
|  | "requestKey", |
|  | bundleOf("bundleKey" to yourResult) |
|  | ) |

2#

|  |
| --- |
| setFragmentResultListener("requestKey") { requestKey, bundle -> |
|  |  |
|  | val result = bundle.getString("bundleKey") |
|  | setFragmentResult( |
|  | "requestKeyChild", |
|  | bundleOf("bundleKeyChild" to yourResult) |
|  | ) |
|  |  |
|  | } |

**1.let:** *This function is used to perform operations on a nullable object or execute a block of code only if the object is not null. The result of the operation is returned. The syntax for using let is as follows:*

object?.let {  
 // perform operations on the object  
}

fun main() {  
 val message = "Hello, world!"  
 val result = message.let {  
 val reversedMessage = it.reversed()  
 reversedMessage.toUpperCase()  
 }  
 println(result) // output: "!DLROW ,OLLEH"  
}

**2. also**: *This function is used to perform some additional operation on an object without changing its state, and then return the object itself. The syntax for using also is as follows:*

object.also {  
 // perform additional operation on the object  
}

fun main() {  
 val message = "Hello, world!"  
 message.also {  
 println("The original message is: $it")  
 }.run {  
 toUpperCase()  
 }.also {  
 println("The uppercase message is: $it")  
 }  
}

**3. apply**: *This function is used to configure the properties of an object and then return the object itself. The syntax for using apply is as follows:*

object.apply {  
 // configure the properties of the object  
}

data class Person(var name: String, var age: Int)  
  
fun main() {  
 val person = Person("John", 25)  
 person.apply {  
 name = "Jane"  
 age = 30  
 }  
 println(person) // output: "Person(name=Jane, age=30)"  
}

**4. run:** *This function is used to perform a block of code on an object, and return the result of the block. The syntax for using run is as follows:*

object.run {  
 // perform the block of code on the object  
}

fun main() {  
 val message = "Hello, world!"  
 val upperCaseMessage = message.run {  
 toUpperCase()  
 }  
 println(upperCaseMessage) // output: "HELLO, WORLD!"  
}

**5. with:** *This function is used to perform a block of code on an object, without the need to call the object again within the block. The syntax for using with is as follows:*

with(object) {  
 // perform the block of code on the object  
}

data class Person(val name: String, var age: Int)  
  
fun main() {  
 val person = Person("John", 25)  
 val greeting = with(person) {  
 "Hello, $name! You are $age years old."  
 }  
 println(greeting) // output: "Hello, John! You are 25 years old."  
}

**LayoutManager**

A LayoutManager is responsible for measuring and positioning item views within a RecyclerView as well as determining the policy for when to recycle item views that are no longer visible to the user.

**1. What is Work Manager?**

Work Manager is an Android Jetpack library that provides an API for scheduling background tasks. It allows you to perform tasks that need to run in the background, even if the app is closed or the device restarts. Work Manager intelligently chooses the appropriate execution strategy based on factors like device conditions and power constraints.

Example:

val workRequest = PeriodicWorkRequest.Builder(MyWorker::class.java, 1, TimeUnit.DAYS).build()  
WorkManager.getInstance(context).enqueue(workRequest)

**2. Differentiate VAL vs VAR vs Const val:**

- ‘val’ is used to declare a read-only (immutable) variable, meaning its value cannot be changed once assigned.

- ‘var’ is used to declare a mutable variable, allowing its value to be reassigned.

- ‘const val’ is used to declare a compile-time constant value, which cannot be changed and is evaluated at compile-time.

**3. What are Launch Modes of Android activity?**

Launch modes define how a new instance of an activity should be launched in the task stack. There are four launch modes:

- **Standard**: Each time an activity is launched, a new instance is created and placed on top of the stack.

- **SingleTop**: If the activity is already at the top of the stack, it won’t be recreated; otherwise, a new instance is created.

- **SingleTask**: Only one instance of the activity can exist in the stack. If an instance already exists, the system brings it to the foreground instead of creating a new one.

- **SingleInstance**: The activity is launched into a new task and a separate task stack. Only one instance of the activity can exist across all tasks.

4. What is difference between Service and IntentService:

- **Service**: A Service is a component that runs in the background without a user interface. It can perform long-running operations, but it runs on the main thread by default, so you need to handle threading manually.

- **IntentService**: IntentService is a subclass of Service that provides a default implementation for handling asynchronous requests using a worker thread. It automatically stops itself when the work is complete.

**5. What is difference between lateinit vs lazy:**

- ‘**lateinit**’ is used for declaring non-nullable properties that will be initialized later before being used. It allows you to delay the initialization of a variable.

- ‘**lazy**’ is a function that takes a lambda and returns an instance of Lazy<T>. It is used for declaring properties that are initialized lazily when they are first accessed.

Example:

lateinit var name: String  
val fullName: String by lazy { firstName + " " + lastName }

**6.Describe difference betweenLiveData vs MutableLiveData:**

- **LiveData** is an observable data holder class that is part of the Android Architecture Components. It is lifecycle-aware, meaning it only updates the observers when they are active.

- **MutableLiveData** is a subclass of LiveData that allows you to modify its value. It is commonly used when you need to update the value of LiveData from within a ViewModel.

Example:

val data: LiveData<String> = MutableLiveData<String>()  
// Setting a new value using MutableLiveData  
(data as MutableLiveData<String>).value = "New Value"

**7. setValue vs postValue:**

- ‘**setValue**’ is a method of MutableLiveData that sets the value of the LiveData synchronously. It should only be called from the main thread.

- ‘**postValue**’ is a method of MutableLiveData that sets the value of the LiveData asynchronously. It can be called from any thread.

**8. What is the difference between compileSdkVersion and targetSdkVersion?**

- ‘**compileSdkVersion**’ is the version of the Android SDK against which the app is compiled. It determines which APIs and features are available during the build process.

- ‘**targetSdkVersion**’ is the version of the Android framework that the app is targeting. It indicates the highest API level that the app has been tested against and guarantees the app’s compatibility with that version.

**9. What is a Repository in MVVM?**

In the MVVM (Model-View-ViewModel) architecture pattern, a repository acts as a mediator between the ViewModel and the data sources. It abstracts the data access, providing a clean API for the ViewModel to retrieve and manipulate data. The repository can fetch data from a remote server, local database, or any other data source.

**10. HashMap vs HashTable:**

- Both HashMap and HashTable are data structures used to store key-value pairs.

- HashMap is not synchronized and allows null keys and values. It is generally preferred over HashTable in non-threaded environments.

- HashTable is synchronized and does not allow null keys or values. It is thread-safe but has a performance overhead.

11. What is SSL Pinning:

SSL Pinning is a security mechanism used in mobile apps to ensure that the app only communicates with a trusted server. It involves associating the server’s SSL certificate or public key with the app in a secure manner. This prevents attackers from intercepting or tampering with the network traffic by presenting fake certificates.

**12. What is Dependency Injection:**

Dependency Injection is a design pattern that enables loose coupling between components in an application. It allows you to provide dependencies to a class from external sources, rather than the class creating or managing the dependencies itself. This promotes modular and testable code.

**13. Benifits of using Android Studio Profiler:**

Android Studio Profiler is a powerful tool that helps developers analyze the performance and behavior of their Android apps. It provides real-time monitoring of CPU

, memory, network, and other resource usage. The profiler can identify performance bottlenecks, memory leaks, and help optimize app performance.

**14. Diffrenciate Final, finally & Finalize:**

- ‘**final**’ is a keyword used in Java/Kotlin to indicate that a variable, method, or class cannot be further modified or extended.

- ‘**finally**’ is a block of code that is executed regardless of whether an exception is thrown or not. It is typically used to release resources or perform cleanup operations.

- ‘**finalize**’ is a method in Java that is called by the garbage collector before an object is garbage collected. It allows the object to perform any necessary cleanup actions before being destroyed.

**15. Activity and Fragment Lifecycle:**

Both Activity and Fragment have a lifecycle consisting of various stages such as onCreate, onStart, onResume, onPause, onStop, and onDestroy. These lifecycle methods allow you to manage the initialization, execution, and termination of activities and fragments. Understanding the lifecycle is crucial for managing state, handling configuration changes, and ensuring smooth user experiences. Please refer my [Article](https://medium.com/@jecky999/mastering-the-android-activity-lifecycle-android-e52e40f6d92) for more details.

# launchWhenCreated() vs launchWhenStarted() vs launchWhenResumed() vs repeatOnLifeCycle()