Android Questions For Interview:

Core Android

Base

- **Tell all the Android application components.** - Learn from here
- **What is the project structure of an Android Application?** - Learn from here
- **What is Context? How is it used?** - Learn from here
- **What is AndroidManifest.xml?** - Learn from here
- **What is Application class?**
  - The Application class in Android is the base class within an Android app that contains all other components such as activities and services. The Application class, or any subclass of the Application class, is instantiated before any other class when the process for your application/package is created.

Activity and Fragment

- **What is Activity and its lifecycle?** - Learn from here
- **What is the difference between onCreate() and onStart()** - Learn from here
- **When only onDestroy is called for an activity without onPause() and onStop()?** - Learn from here
- **Why do we need to call setContentView() in onCreate() of Activity class?** - Learn from here
- **What is onSavedInstanceState() and onRestoreInstanceState() in activity?**
  - onSavedInstanceState() - This method is used to store data before pausing the activity.
  - onRestoreInstanceState() - This method is used to recover the saved state of an activity when the activity is recreated after destruction. So, the onRestoreInstanceState() receive the bundle that contains the instance state information.
- **What is Fragment and its lifecycle.** - Learn from here
- **What are "launch modes"?** - Learn from here
- **What is the difference between a Fragment and an Activity? Explain the relationship between the two.** - Learn from here
- **When should you use a Fragment rather than an Activity?**
  - When you have some UI components to be used across various activities
  - When multiple view can be displayed side by side just like viewPager
- **What is the difference between FragmentPagerAdapter vs FragmentStatePagerAdapter?**
  - FragmentPagerAdapter: Each fragment visited by the user will be stored in the memory but the view will be destroyed. When the page is revisited, then the view will be created not the instance of the fragment.
  - FragmentStatePagerAdapter: Here, the fragment instance will be destroyed when it is not visible to the user, except the saved state of the fragment.
- **What is the difference between adding/replacing fragment in backstack?** - Learn from here
- **Why is it recommended to use only the default constructor to create a Fragment?** - Learn from here

- **How would you communicate between two Fragments?** - Learn from here
- **What is retained Fragment?**
  - By default, Fragments are destroyed and recreated along with their parent Activity's when a configuration change occurs. Calling setRetainInstance(true) allows us to bypass this destroy-and-recreate cycle, signaling the system to retain the current instance of the fragment when the activity is recreated.
- **What is the purpose of addToBackStack() while commiting fragment transaction?**
  - By calling addToBackStack(), the replace transaction is saved to the back stack so the user can reverse the transaction and bring back the previous fragment by pressing the Back button. For more Learn from here

Views and ViewGroups

- **What is View in Android?** - Learn from here
- **Difference between View.GONE and View.INVISIBLE?** - Learn from here
- **Can you a create custom view? How?** - Learn from here
- **What are ViewGroups and how they are different from the Views?**
  - View: View objects are the basic building blocks of User Interface(UI) elements in Android. View is a simple rectangle box which responds to the user's actions. Examples are EditText, Button, CheckBox etc. View refers to the android.view.View class, which is the base class of all UI classes.
  - ViewGroup: ViewGroup is the invisible container. It holds View and ViewGroup. For example, LinearLayout is the ViewGroup that contains Button(View), and other Layouts also. ViewGroup is the base class for Layouts.
- **What is a Canvas?** - Learn from here
- **What is a SurfaceView?** - Learn from here
- **Relative Layout vs Linear Layout.** - Learn from here
- **Tell about Constraint Layout** - Learn from here
- **Do you know what is the view tree? How can you optimize its depth?** - Learn from here
- **How does the Touch Control and Events work in Android?** - Learn from here and here

Displaying Lists of Content

- **What is the difference between ListView and RecyclerView?** - Learn from here
- **How does RecyclerView work internally?** - Learn from here and here
- **What is the ViewHolder pattern? Why should we use it?** - Learn from here
- **RecyclerView Optimization - Scrolling Performance Improvement** - Learn from here
- **What is SnapHelper?** - Learn from here

Dialogs and Toasts

- **What is Dialog in Android?** - Learn from here
- **What is Toast in Android?** - Learn from here
- **What the difference between Dialog and Dialog Fragment?** - Learn from here

Intents and Broadcasting

- **What is Intent?** - Learn from here
- **What is an Implicit Intent?** - Learn from here
- **What is an Explicit Intent?** - Learn from here
- **What is a BroadcastReceiver?** - Learn from here
- **What is a LocalBroadcastManager?** - Learn from here
- **What is the function of an IntentFilter?** - Learn from here
- **What is a Sticky Intent?**
  - Sticky Intents allows communication between a function and a service. sendStickyBroadcast() performs a sendBroadcast(Intent) known as sticky, i.e. the Intent you are sending stays around after the broadcast is complete, so that others can quickly retrieve that data through the return value of registerReceiver(BroadcastReceiver, IntentFilter). For example, if you take an intent for ACTION_BATTERY_CHANGED to get battery change events: When you call registerReceiver() for that action — even with a null BroadcastReceiver — you get the Intent that was last Broadcast for that action. Hence, you can use this to find the state of the battery without necessarily registering for all future state changes in the battery.
- **Describe how broadcasts and intents work to be able to pass messages around your app?** - Learn from here
- **What is a PendingIntent?**
  - If you want someone to perform any Intent operation at future point of time on behalf of you, then we will use Pending Intent.
- **What are the different types of Broadcasts?** - Learn from here

Services

- **What is Service?** - Learn from here
- **Service vs IntentService.** - Learn from here
- **What is a JobScheduler?** - Learn from here

Inter-process Communication

- **How can two distinct Android apps interact?** - Learn from here
- **Is it possible to run an Android app in multiple processes? How?** - Learn from here
- **What is AIDL? Enumerate the steps in creating a bounded service through AIDL.** - Learn from here
- **What can you use for background processing in Android?** - Learn from here
- **What is a ContentProvider and what is it typically used for?** - Learn from here and here

Long-running Operations

- **How to run parallel tasks in Java or Android, and get callback when all complete?** - Learn from here
- **Why should you avoid to run non-ui code on the main thread?** - Learn from here
- **What is ANR? How can the ANR be prevented?** - Learn from here
- **What is an AsyncTask(Deprecated in API level 30) ?** - Learn from here
- **What are the problems in AsyncTask?** - Learn from here
- **When would you use java thread instead of an AsyncTask?** - Learn from here
- **What is a Loader? (Deprecated)** - Learn from here

- **What is the relationship between the life cycle of an `AsyncTask` and an `Activity`? What problems can this result in? How can these problems be avoided?**
  - An AsyncTask is not tied to the life cycle of the Activity that contains it. So, for example, if you start an AsyncTask inside an Activity and the user rotates the device, the Activity will be destroyed (and a new Activity instance will be created) but the AsyncTask will not die but instead goes on living until it completes.
  - Then, when the AsyncTask does complete, rather than updating the UI of the new Activity, it updates the former instance of the Activity (i.e., the one in which it was created but that is not displayed anymore!). This can lead to an Exception (of the type java.lang.IllegalArgumentException: View not attached to window manager if you use, for instance, findViewById to retrieve a view inside the Activity).
  - There's also the potential for this to result in a memory leak since the AsyncTask maintains a reference to the Activity, which prevents the Activity from being garbage collected as long as the AsyncTask remains alive.
  - For these reasons, using AsyncTasks for long-running background tasks is generally a bad idea . Rather, for long-running background tasks, a different mechanism (such as a service) should be employed.
  - Note: AsyncTasks by default run on a single thread using a serial executor, meaning it has only 1 thread and each task runs one after the other.
- **Explain `Looper`, `Handler` and `HandlerThread`.** - Learn from here and from video
- **How does the threading work in Android?** - Learn from here
- **Android Memory Leak and Garbage Collection** - Learn from here

Working With Multimedia Content

- **How do you handle bitmaps in Android as it takes too much memory?** - Learn from here and here
- **What is the difference between a regular `Bitmap` and a nine-patch image?**
  - In general, a Nine-patch image allows resizing that can be used as background or other image size requirements for the target device. The Nine-patch refers to the way you can resize the image: 4 corners that are unscaled, 4 edges that are scaled in 1 axis, and the middle one that can be scaled into both axes.
- **Tell about the `Bitmap` pool.** - Learn from here
- **How to play sounds in Android?** - Learn from here
- **How image compression is preformed?** - Learn from here

Data Saving

- **How to persist data in an Android app?** - Learn from here
- **What is ORM? How does it work?** - Learn from here
- **How would you preserve `Activity` state during a screen rotation?** - Learn from here
- **What are different ways to store data in your Android app?** - Learn from here
- **Explain Scoped Storage in Android.** - Learn from here
- **How to encrypt data in Android?** - Learn from here
- **What is commit() and apply() in SharedPreferences?**
  - commit() returns a boolean value of success or failure immediately by writing data synchronously.

    - apply() is asynchronous and it won't return any boolean response. If you have an apply() outstanding and you are performing commit(), then the commit() will be blocked until the apply() is not completed.

Look and Feel

- **What is a Spannable?** - Learn from here
- **What is a SpannableString?**
    - A SpannableString has immutable text, but its span information is mutable. Use a SpannableString when your text doesn't need to be changed but the styling does. Spans are ranges over the text that include styling information like color, heighliting, italics, links, etc
- **What are the best practices for using text in Android?** - Learn from here
- **How to implement Dark mode in any application?** - Learn from here
- **How to generate dynamic colors based in image?** - Learn from here
- **Explain about Density Independence Pixel** - Learn from here

Memory Optimizations

- **What is the onTrimMemory() method?** - Learn from here
- **How does the OutOfMemory happens?** - Learn from here
- **How do you find memory leaks in Android applications?** - Learn from here and here

Battery Life Optimizations

- **How to reduce battery usage in an android application?** - Learn from here
- **What is Doze? What about App Standby?** - Learn from here
- **What is overdraw?** - Learn from here

Supporting Different Screen Sizes

- **How do you support different types of resolutions?** - Learn from here

Permissions

- **What are the different protection levels in permission?** - Learn from here

Native Programming

- **What is the NDK and why is it useful?** - Learn from here and here and here
- **What is renderscript?** - Learn from here

Android System Internal

- **What is the Dalvik Virtual Machine?** - Learn from here
- **What is the difference JVM, DVM and ART?** - Learn from here
- **What are the differences between Dalvik and ART?** - Learn from here
- **What is DEX?** - Learn from here
- **Can you manually call the Garbage collector?** - Learn from here

Android Jetpack

- **What is Android Jetpack and why to use this?** - Learn from here
- **What are Android Architecture Components?** - Learn from here
- **What is LiveData in Android?** - Learn from here
- **How LiveData is different from ObservableField?** - Learn from here

- **What is the difference between setValue and postValue in LiveData?** - Learn from here
- **How to share ViewModel between Fragments in Android?** - Learn from here
- **Explain Work Manager in Android.** - Learn from here
- **Use-cases of WorkManager in Android.** - Learn from here
- **How ViewModel work internally?** - Learn from here

Others

- **Why Bundle class is used for data passing and why cannot we use simple Map data structure?** - Learn from here
- **How do you troubleshoot a crashing application?** - Learn from here
- **Explain Android notification system?** - Learn from here
- **What is the difference between Serializable and Parcelable? Which is the best approach in Android?** - Learn from here
- **What is AAPT?** - Learn from here
- **What is the best way to update the screen periodically?** - Learn from here
- **FlatBuffers vs JSON.** - Learn from here
- **`HashMap`, `ArrayMap` and `SparseArray`** - Learn from here
- **What are Annotations?** - Learn from here, Link, and from video
- **How to create custom Annotation?** - Learn from here and here
- **How to handle multi-touch in android?** - Learn from here
- **How to implement XML namespaces?** - Learn from here
- **What is the support library? Why was it introduced?** - Learn from here
- **What is Android Data Binding?** - Learn from here
- **How to check if Software keyboard is visible or not?** - Learn from here
- **How to take screenshot in Android programmatically?** - Learn from here

Android Libraries

- **Explain OkHttp Interceptor** - Learn from here
- **OkHttp - HTTP Caching - How caching work in Android** - Learn from here
- **Tell me something about RxJava.** - Learn from here
- **How will you handle error in RxJava?** - Learn from here
- **FlatMap Vs Map Operator** - Learn from here
- **When to use `Create` operator and when to use `fromCallable` operator of RxJava?** - Learn from here
- **When to use `defer` operator of RxJava?** - Learn from here
- **How are Timer, Delay, and Interval operators used in RxJava?** - Learn from here
- **How to make two network calls in parallel using RxJava?** - Learn from here
- **Tell the difference between Concat and Merge.** - Learn from here
- **Explain Subject in RxJava?** - Learn from here
- **What are the types of Observables in RxJava?** - Learn from here
- **How to implement EventBus with RxJava?** - Learn from here
- **How to implement search feature using RxJava in your application?** - Learn from here
- **How The Android Image Loading Library Glide and Fresco Works?** - Learn from here
- **Difference between Schedulers.io() and Schedulers.computation() in RxJava.**

- **Why do we use the Dependency Injection Framework like Dagger in Android?** - Learn from here
- **How does the Dagger work?** - Learn from here and here
- **What is Component in Dagger?** - Learn from here
- **What is Module in Dagger?** - Learn from here
- **How does the custom scope work in Dagger?**
- **When to call dispose and clear on CompositeDisposable in RxJava?** - Learn from here
- **What is Multipart Request in Networking?** - Learn from here
- **What is Flow in Kotlin?** - Learn from here

Android Architecture

- **Describe the architecture of your last app.**
- **Describe MVP.** - Learn from here
- **Describe MVVM.** - Learn from here and here
- **MVC vs MVP vs MVVM architecture.** - Learn from here
- **What is presenter?** - Learn from here
- **What is model?** - Learn from here
- **Describe MVC.** - Learn from here
- **Describe MVI** - Learn from here
- **Describe the repository pattern** - Learn from here
- **What is controller?** - Learn from here
- **Tell something about clean code** - Learn from here

Android Design Problem

- **Design Uber App.** - Learn from here
- **Design Facebook App.**
- **Design Facebook Near-By Friends App.**
- **Design WhatsApp.**
- **Design SnapChat.**
- **Design problems based on location based app.**
- **How to build offline-first app? Explain the architecture.**
- **Design LRU Cache.**
- **Design File Downloader** - Learn from here
- **Design an Image Loading Library** - Learn from here
- **HTTP Request vs HTTP Long-Polling vs WebSockets** - Learn from here

Android Unit Testing

- **What is Espresso?** - Learn from here
- **What is Robolectric?** - Learn from here
- **What are the disadvantages of using Robolectric?** - Learn from here
- **What is UI-Automator?** - Learn from here
- **Explain unit test.** - Learn from here
- **Explain instrumented test.** - Learn from here
- **Have you done unit testing or automatic testing?**
- **Why Mockito is used?** - Learn from here
- **Describe JUnit test.** - Learn from here
- **Describe code coverage.** - Learn from here

Android Tools And Technologies

- **What is ADB?** - Learn from here
- **What is DDMS and what can you do with it?** - Learn from here
- **What is the StrictMode?** - Learn from here
- **What is Lint? What is it used for?** - Learn from here
- **Git.** - Learn from here
- **Android Development Useful Tools.** - Learn from here
- **Firebase.** - Learn from here
- **How to measure method execution time in Android?** - Learn from here
- **Can you access your database of SQLite Database for debugging?** - Learn from here
- **What are things that we need to take care while using Proguard?** - Learn from here
- **What is Multidex in Android?** - Learn from here
- **How to use Android Studio Memory Profiler?** - Learn from here
- **How to use Firebase realtime database in your app?** - Learn from here
- **What is Gradle?** - Learn from here
- **APK Size Reduction.** - Learn from here and here
- **How can you speed up the Gradle build?** - Learn from here
- **About gradle build system.** - Learn from here
- **About multiple apk for android application.** - Learn from here
- **What is proguard used for?** - Learn from here
- **What is obfuscation? What is it used for? What about minification?** - Learn from here
- **How to change some parameters in an app without app update?** - Learn from here

Java and Kotlin

OOP

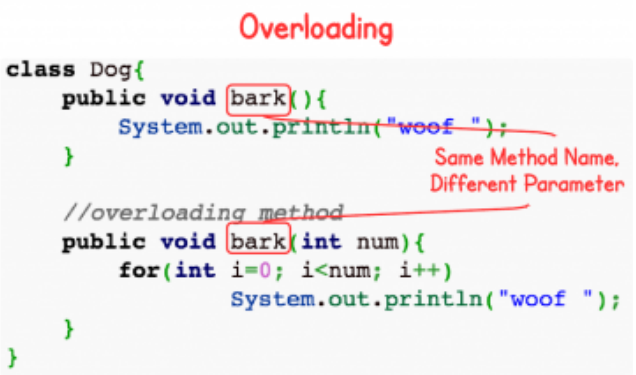
- **Explain OOP Concepts.**
  - Object-Oriented Programming is a methodology of designing a program using classes, objects, inheritance, polymorphism, abstraction, and encapsulation.
- **What is the difference between a constructor and a method?**
  - The name of the constructor is same as that of the class name, whereas the name of the method can be anything.
  - There is no return type of a constructor.
  - When you make an object of a class, then the constructor of that class will be called automatically. But for methods, we need to call it explicitely.
  - Constructors can't be inherited but you can call the constructor of the parent class by calling `super()`.
  - Constructor and a method they both run a block of code but the difference is in calling them.
  - We can call method directly using their name.
  - Constructor Syntax -
  - ```
    public class SomeClassName{
        SomeClassName(parameter_list){
    ```
    ```

```
    ...
  }
  ...
}
```

- Note: In the above syntax, the name of the constructor is the same as that of class and it has no return type.
- Method Syntax

```
public class SomeClassName{
  public void someMethodName(parameter_list){
    ...
  }
  // call method
  someMethodName(parameter_list)
}
```

- **Differences between abstract classes and interfaces?**
  - An abstract class, is a class that contains both concrete and abstract methods (methods without implementations). An abstract method must be implemented by the abstract class sub-classes. Abstract classes cannot be instantiated and need to be extended to be used.
  - An interface is like a blueprint/contract of a class (or it may be thought of as a class with methods, but without their implementation). It contains empty methods that represent, what all of its subclasses should have in common. The subclasses provide the implementation for each of these methods. Interfaces are implemented.
- **What is the difference between iterator and enumeration in java?**
  - In Enumeration we don't have remove() method and we can only read and traverse through a collection.
  - Iterators can be applied to any collection. In Iterator, we can read and remove items from a collection.
- **Do you agree we use composition over inheritance?** Learn from here
- **Difference between method overloading and overriding.**



  - Overloading happens at compile-time while Overriding happens at runtime: The binding of overloaded method call to its definition happens at compile-

time however binding of overridden method call to its definition happens at runtime. More info on static vs. dynamic binding: StackOverflow.

- Static methods can be overloaded which means a class can have more than one static method of same name. Static methods cannot be overridden, even if you declare a same static method in child class it has nothing to do with the same method of parent class as overridden static methods are chosen by the reference class and not by the class of the object.
  So, for example:

```
public class Animal {
    public static void testClassMethod() {
        System.out.println("The static method in Animal");
    }

    public void testInstanceMethod() {
        System.out.println("The instance method in Animal");
    }
}

public class Cat extends Animal {
    public static void testClassMethod() {
        System.out.println("The static method in Cat");
    }

    public void testInstanceMethod() {
        System.out.println("The instance method in Cat");
    }

    public static void main(String[] args) {
        Cat myCat = new Cat();
        myCat.testClassMethod();
        Animal myAnimal = myCat;
        myAnimal.testClassMethod();
        myAnimal.testInstanceMethod();
    }
}
```

  Will output:

```
The static method in Cat     // testClassMethod() is called from "Cat" reference

The static method in Animal // testClassMethod() is called from "Animal" reference,
                            // ignoring actual object inside it (Cat)

The instance method in Cat  // testInstanceMethod() is called from "Animal" reference,
                             // but from "Cat" object underneath
```

  The most basic difference is that overloading is being done in the same class while for overriding base and child classes are required. Overriding is all about giving a specific implementation to the inherited method of parent class.
  Static binding is being used for overloaded methods and dynamic binding is

being used for overridden/overriding methods. Performance: Overloading gives better performance compared to overriding. The reason is that the binding of overridden methods is being done at runtime.
Private and final methods can be overloaded but they cannot be overridden. It means a class can have more than one private/final methods of same name but a child class cannot override the private/final methods of their base class.
Return type of method does not matter in case of method overloading, it can be same or different. However in case of method overriding the overriding method can have more specific return type (meaning if, for example, base method returns an instance of Number class, all overriding methods can return any class that is extended from Number, but not a class that is higher in the hierarchy, like, for example, Object is in this particular case).
Argument list should be different while doing method overloading. Argument list should be same in method Overriding. It is also a good practice to annotate overridden methods with `@Override` to make the compiler be able to notify you if child is, indeed, overriding parent's class method during compile-time.

- **What are the access modifiers you know? What does each one do?**
    - There are four access modifiers in Java language (from strictest to the most lenient):
        - `private` *variables*, *methods*, *constructors* or *inner classes* are only visible to its' containing class and its' methods. This modifier is most commonly used, for example, to allow variable access only through getters and setters or to hide underlying implementation of classes that should not be used by user and therefore maintain encapsulation. Singleton constructor is also marked `private` to avoid unwanted instantiation from outside.
        - `Default` (no keyword is used) this modifier can be applied to *classes*, *variables*, *constructors* and *methods* and allows access from classes and methods inside the same package.
        - `protected` can be used on *variables*, *methods* and *constructors* therefore allowing access only to subclasses and classes that are inside the same package as protected members' class.
        - `public` modifier is widely-used on *classes*, *variables*, *constructors* and *methods* to grant access from any class and method anywhere. It should not be used everywhere as it implies that data marked with `public` is not sensitive and can not be used to harm the program.
- **Can an Interface implement another Interface?**
    - Yes, an interface can implement another interface (and more than one), but it needs to use `extends`, rather than `implements` keyword. And while you can not remove methods from parent interface, you can add new ones freely to your sub-interface.
- **What is Polymorphism? What about Inheritance?**
    - Polymorphism in Java has two types: Compile time polymorphism (static binding) and Runtime polymorphism (dynamic binding). Method overloading is an example of static polymorphism, while method overriding is an example of dynamic polymorphism.
      An important example of polymorphism is how a parent class refers to a

child class object. In fact, any object that satisfies more than one IS-A relationship is polymorphic in nature.
For instance, let's consider a class `Animal` and let `Cat` be a subclass of `Animal`. So, any cat IS animal. Here, Cat satisfies the IS-A relationship for its own type as well as its super class Animal.

- Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order.
  The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).
  Inheritance uses the keyword `extends` to inherit the properties of a class.
  Following is the syntax of extends keyword.

```
class Super {
   .....
   .....
}
class Sub extends Super {
   .....
   .....
}
```

- **Multiple inheritance in Classes and Interfaces in java** Learn from here
- **What are the design patterns?** Learn from here
  - Creational patterns
    - Builder Wikipedia
    - Factory Wikipedia
    - Singleton Wikipedia
    - Monostate Wikipedia
    - Fluent Interface Pattern Wikipedia
  - Structural patterns
    - Adapter Wikipedia
    - Decorator Wikipedia
    - Facade Wikipedia
  - Behavioural patterns
    - Chain of responsibility Wikipedia
    - Iterator Wikipedia
    - Strategy Wikipedia

Collections and Generics

- **Arrays Vs ArrayLists** - Learn from here and here
- **HashSet Vs TreeSet** - Learn from here
- **HashMap Vs Set** - Learn from here
- **Stack Vs Queue** - Learn from here
- **Explain Generics in Java?**
  - Generics were included in Java language to provide stronger type checks, by allowing the programmer to define, which classes can be used with other classes
    In a nutshell, generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods. Much like the more familiar formal parameters used in method declarations, type

parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types. (Official Java Documentation)

    - This means that, for example, you can define:
      `List<Integer> list = new ArrayList<>();`

      And let the compiler take care of noticing, if you put some object, of type other than `Integer` into this list and warn you.
    - It should be noted that standard class hierarchy *does not* apply to generic types. It means that `Integer` in `List<Integer>` is not inherited from `<Number>` - it is actually inherited directly from `<Object>`. You can still put some constraints on what classes can be passed as a parameter into a generic by using wildcards like `<?>`, `<? extends MyCustomClass>` or `<? super Number>`.
    - While generics are very useful, late inclusion into Java language has put some restraints on their implementation - backward compatibility required them to remain just "syntactic sugar" - they are erased (type erasure) during compile-time and replaced with `object` class.
- **What is Java PriorityQueue?** - In Priority Queue, each element is having some priority and all the elements are present in a queue. The operations are performed based on the priority.

Objects and Primitives

- **How is `String` class implemented? Why was it made immutable?**
    - There is no primitive variant of `String` class in Java language - all strings are just wrappers around underlying array of characters, which is declared `final`. This means that, once a `String` object is instantiated, it cannot be changed through normal tools of the language (Reflection still can mess things up horribly, because in Java no object is truly immutable). This is why `String` variables in classes are the first candidates to be used, when you want to override `hashCode()` and `equals()` of your class - you can be sure, that all their required contracts will be satisfied.
      Note: The String class is immutable, so that once it is created a String object cannot be changed. The String class has a number of methods, some of which will be discussed below, that appear to modify strings. Since strings are immutable, what these methods really do is create and return a new string that contains the result of the operation. (Official Java Documentation)

      This class is also unique in a sense, that, when you create an instance like this:
      `String helloWorld = "Hello, World!";`

      `"Hello, World!"` is called a *literal* and compiler creates a `String` object with its' value. So
      `String capital = "Hello, World!".toUpperCase();`

      is a valid statement, that, firstly, will create an object with literal value "Hello, World!" and then will create and return another object with value "HELLO, WORLD!"

    - String was made immutable to prevent malicious manipulation of data, when, for example, user login or other sensitive data is being send to a server.
- **What does it means to say that a String is immutable?**
    - It means that once created, String object's char[] (its' containing value) is declared final and, therefore, it can not be changed during runtime.
- **What is String.intern()? When and why should it be used?**
    - String.intern() is used to mange memory in Java code. It is used when we have duplicates value in different strings. When you call the String.intern(), then if in the String pool that string is present then the equals() method will return true and it will return that string only.
- **Can you list 8 primitive types in java?**
    - byte
    - short
    - int
    - long
    - float
    - double
    - char
    - String
    - boolean
- **What is the difference between an Integer and int?**
    - int is a primitive data type (with boolean, byte, char, short, long, float and double), while Integer (with Boolean, Byte, Character, Short,Long, Float and Double) is a wrapper class that encapsulates primitive data type, while providing useful methods to perform different tasks with it.
- **What is Autoboxing and Unboxing?**
    - Autoboxing and Unboxing is the process of automatic wrapping (putting in a box) and unwrapping (getting the value out) of primitive data types, that have "wrapper" classes. So int and Integer can (almost always) be used interchangeably in Java language, meaning a method void giveMeInt(int i) { ... } can take int as well as Integer as a parameter.
- **Typecast in Java**
    - In Java, you can use casts to polymorph one class into another, compatible one. For example: `long i = 10l;`
    - `int j = (int) i;`
    - `long k = j;`

      Here we see, that, while narrowing (long i -> int j) requires an explicit cast to make sure the programmer realizes, that there may be some data or precision loss, widening (int j -> long k) does not require an explicit cast, because there can be no data loss (long can take larger numbers than int allows).
- **Do objects get passed by reference or value in Java? Elaborate on that.**
    - In Java all primitives and objects are passed by value, meaning that their copy will be manipulated in the receiving method. But there is a caveat - when you pass an object reference into a method, a *copy of this reference* is made, so it still points to the same object. This means, that any changes that you make to the insides of this object are retained, when the method exits. `public class Pointer {`
    -

```
    int innerField;

    public Pointer(int a) {
        this.innerField = a;
    }
}

public class ValueAndReference {

    public static void main(String[] args) {

        Pointer a = new Pointer(0);
        int b = 1;

        print("Before:");
        print("b = " + b);
        print("a.innerField = " + a.innerField);
        exampleMethod(a, b);
        print("After:");
        print("b = " + b);
        print("a.innerField = " + a.innerField);
    }

    static void exampleMethod(Pointer a, int b) {
        a.innerField = 2;
        b = 10;
    }

    static void print(String text) {
        System.out.println(text);
    }
}
```

Will output: Before:

```
    b = 1

    a.innerField = 0

    After:

    b = 1       // a new local int variable was created and operated on, so "b" didn't change

    a.innerField = 2 // Pointer a got its' innerField variable changed
                     //  from 0 to 2, because method was operating on
                     //  the same reference to an instance
```

- **What is the difference between instantiation and initialization of an object?** - Learn from here
- **What the difference between local, instance and class variables?**

    - Local variables exist only in methods that created them, they are stored separately in their respected Thread Stack (for more information, see question about Java Memory Model) and cannot have their reference passed outside of the method scope. That also means that they cannot be assigned any access modifier or made `static` - because they only exist during enclosing method's execution and those modifiers just do not make sense, since no other outside method can get them anyway.
    - Instance variables are the ones, that are declared in classes and their value can be different from one instance of the class to another, but they always require that class' instance to exist.
    - Class variables are those, that are marked with `static` keyword in their class' body. They can only have one value across all instances of that class (changing it in one place will change it in their class and, therefore, in all instances) and can even be retrieved without that class' instance (if their access modifier allows it).

Java Memory Model and Garbage Collector

- **What is garbage collector? How does it work?**
    - All objects are allocated on the heap area managed by the JVM. As long as an object is being referenced, the JVM considers it alive. Once an object is no longer referenced and therefore is not reachable by the application code, the garbage collector removes it and reclaims the unused memory.
- **What is Java Memory Model? What contracts does it guarantee? How are its' Heap and Stack organized?** - Learn from here
- **What is memory leak and how does Java handle it?** - Learn from here
- **What are strong, soft, weak and phantom references in Java?** - Learn from here

Concurrency

- **What does the keyword `synchronized` mean?** Learn from here
- **What is a `ThreadPoolExecutor`?** Learn from here
- **What is `volatile` modifier?** Learn from here
- **The classes in the atomic package expose a common set of methods: `get`, `set`,, `lazyset`, `compareAndSet`, and `weakCompareAndSet`. Please describe them.**

Exceptions

- **How does the `try{}`, `catch{}`, `finally` works?** - Learn from here
- **What is the difference between a `Checked Exception` and an `Un-Checked Exception`?** - Learn from here

Others

- **What is serialization? How do you implement it?**
    - Serialization is the process of converting an object into a stream of bytes in order to store an object into memory, so that it can be recreated at a later time, while still keeping the object's original state and data. In Android you may use either the Serializable, Externalizable (implements Serializable) or Parcelable interfaces.
    - While Serializable is the easiest to implement, Externalizable may be used if you need to insert custom logic into the process of serialization (although it

is almost never used nowadays as it is considered a relic from early versions of Java). But it is highly recommended to use Parcelable in Android instead, as Parcelable was created exclusively for Android and it performs about 10x faster than Serializable, because Serializable uses reflection, which is a slow process and tends to create a lot of temporary objects and it may cause garbage collection to occur more often.

- To use Serializable all you have to do is implement the interface:

```
/**
*  Implementing the Serializable interface is all that is required
*/
public class User implements Serializable {

    private String name;
    private String email;

        public User() {
        }

        public String getName() {
            return name;
        }

        public void setName(final String name) {
            this.name = name;
        }

        public String getEmail() {
            return email;
        }

        public void setEmail(final String email) {
            this.email = email;
        }
    }
```

- Parcelable requires a bit more work:

```
    public class User implements Parcelable {

        private String name;
        private String email;

        /**
         * Interface that must be implemented and provided as a public
CREATOR field
         * that generates instances of your Parcelable class from a Parcel.
         */
        public static final Creator<User> CREATOR = new Creator<User>() {

            /**
             * Creates a new USer object from the Parcel. This is the reason why
             * the constructor that takes a Parcel is needed.
             */
```

```
        @Override
        public User createFromParcel(Parcel in) {
            return new User(in);
        }

        /**
         * Create a new array of the Parcelable class.
         * @return an array of the Parcelable class,
         * with every entry initialized to null.
         */
        @Override
        public User[] newArray(int size) {
            return new User[size];
        }
    };

    public User() {
    }

    /**
     * Parcel overloaded constructor required for
     * Parcelable implementation used in the CREATOR
     */
    private User(Parcel in) {
        name = in.readString();
        email = in.readString();
    }

    public String getName() {
        return name;
    }

    public void setName(final String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(final String email) {
        this.email = email;
    }

    @Override
    public int describeContents() {
        return 0;
    }

    /**
     * This is where the parcel is performed.
     */
```

```
    @Override
    public void writeToParcel(final Parcel parcel, final int i) {
        parcel.writeString(name);
        parcel.writeString(email);
    }
}
```

Note: For a full explanation of the **describeContents()** method see StackOverflow. In Android Studio, you can have all of the parcelable code auto generated for you, but like with everything else, it is always a good thing to try and understand everything that is happening.

- **What is transient modifier?** Learn from here
- **What are anonymous classes?** Learn from here
- **What is the difference between using == and .equals on an object?** - Learn from here
- **What is the hashCode() and equals() used for?** - Learn from here
- **Why would you not call abstract method in constructor?** - Learn from here
- **When would you make an object value final?**
- **What are these final, finally and finalize keywords?**
  - final is a keyword in the java language. It is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.

```
class FinalExample {
    public static void main(String[] args) {
        final int x=100;
        x=200;//Compile Time Error because x is final
    }
}
```

  - finally is a code block and is used to place important code, it will be executed whether exception is handled or not.

```
class FinallyExample {
    public static void main(String[] args) {
        try {
            int x=300;
        }catch(Exception e) {
            System.out.println(e.getMessage());          }
        finally {
            System.out.println("finally block is executed");
        }
    }
}
```

  - Finalize is a method used to perform clean up processing just before object is garbage collected.

```
class FinalizeExample {
    public void finalize() {
        System.out.println("finalize called");
    }

    public static void main(String[] args) {
        FinalizeExample f1=new FinalizeExample();
        FinalizeExample f2=new FinalizeExample();
```

```
        f1=null;
        f2=null;
        System.gc();
    }
}
```

- **What is the difference between "throw" and "throws" keyword in Java?**
  - throws is just used to indicated which exception is to be thrown. But the throw keyword is used to throw some exception from any static block or any method.
- **What does the static word mean in Java?**
  - In case of static variable it means that this variable (its' value or the object it references) spans across all instances of enclosing class (changing it in one instance affects all others), while in case of static methods it means that these methods can be invoked without an instance of their enclosing class. It is useful, for example, when you create util classes that need not be instantiated every time you want to use them.
- **Can a static method be overridden in Java?**
  - While child class can override a static method with another static method with the same signature (return type can be down-casted), it is not truly overridden - it becomes "hidden", but both methods can still be accessed under right circumstances (see question about overloading/overriding above).
- **When is a static block run?**
  - Code inside static block is executed only once: the first time you make an object of that class or the first time you access a static member of that class (even if you never make an object of that class).
- **What is reflection?**
  - You can inspect classes, interfaces, fields, and method at runtime with the help of reflection and the best part is that you need not know the names of these classes, methods, etc.
- **What is Dependency Injection?** Learn from here
- **How is a StringBuilder implemented to avoid the immutable string allocation problem?** - Learn from here
- **Difference between StringBuffer and StringBuilder?** - Learn from here
- **What is the difference between fail-fast and fail-safe iterators in Java?**
  - Fail-safe iterator will not throw any exception even if the collection is modified while iteration over it. But in Fail-safe iterator, it throws a ConcurrentModificationException when you try to modify the collection while using it.
- **What is Java NIO?** - Learn from here
- **Monitor and Synchronization** - Learn from here
- **Tell some advantages of Kotlin.** - Learn from here
- **What is the difference between val and var?** - Learn from here
- **What is the difference between const and val?** - Learn from here
- **How to ensure null safety in Kotlin?** - Learn from here
- **When to use lateint keyword used in Kotlin?** - Learn from here
- **How to check if a lateinit variable has been initialized?** - Learn from here
- **How to do lazy initialization of variables in Kotlin?** - Learn from here and here
- **What are companion objects in Kotlin?** - Learn from here

- **What are the visibility modifiers in Kotlin?** - Learn from here
- **What is the equivalent of Java static methods in Kotlin?** - Learn from here
- **What is a data class in Kotlin?** - Learn from here
- **How to create a Singleton class in Kotlin?** - Learn from here
- **What is the difference between open and public in Kotlin?** - Learn from here
- **Explain the use-case of let, run, with, also, apply in Kotlin.** - Learn from here and here
- **Difference between List and Array types in Kotlin** - Learn from here
- **What are Labels in Kotlin?** - Learn from here
- **What is an Init block in Kotlin?** - Learn from here
- **Explain pair and triple in Kotlin.** - Learn from here
- **How to choose between apply and with?** - Learn from here
- **How to choose between switch with when?** - Learn from here
- **What are Coroutines in Kotlin?** - Learn from here
- **What is Coroutine Scope?** - Learn from here
- **What is Coroutine Context?** - Learn from here
- **Launch vs Async in Kotlin Coroutines** - Learn from here
- **What is inline function in Kotlin?** - Learn from here
- **When to use Kotlin sealed classes?** - Learn from here
- **Explain function literals with receiver in Kotlin?** - Learn from here
- **Tell about Kotlin DSL.** - Learn from here
- **What are higher-order functions in Kotlin?** - Learn from here
- **What are Lambdas in Kotlin** - Learn from here
- **Tell about the Collections in Kotlin** - Learn from here

Data Structures And Algorithms
The level of questions asked on the topic of Data Structures And Algorithms totally depends on the company for which you are applying.

Whiteboard Interview Series - Data Structures and Algorithms on Youtube - Check here

Tech Interview Preparation Kit - Check here

Android Developer should know these Data Structures for Next Interview - Check here

- **Complexity Analysis** - Learn from here
  - What is Input, Output, Correctness, Efficiency of Algorithms?
  - What is Input Size and Running Time of Algorithms?
  - Explain the Worst, Best, and Average case analysis of Algorithms.
  - What is Big-O Notation with respect to Time and Space Complexity?
- **Iteration and Two Pointer Approach** - Learn from here
  - Explain Initialization, Maintenance, and Termination used in iteration.
  - Explain the use-case of Two Pointer approach
- **Recursion and Divide & Conquer Approach** - Learn from here
  - Explain Recursion with the help of an example and also draw the recursion call stack for the same.
  - How will you analyse the recursive solution of some problem?
  - Is there any difference between Recursion and Iteration?

    - Explain the Divide and Conquer technique with the help of a real-world example.
- **Arrays and Linked List** - Learn from here
    - What do you mean by Linear Data Structures?
    - Explain the basic operations that can be performed on Arrays? Also, tell about Amortized analysis of array.
    - What is a Linked List? Explain with an example by performing some operations on Linked List.
    - What are the types of Linked List?
    - Can you tell the difference between an Array and a Linked List?
- **Stack and Queue** - Learn from here
    - What is a Stack? Explain various operations that can be performed on a Stack.
    - Can you implement Stack using an Array or using a Linked List? How?
    - What is a Queue? Explain various operations that can be performed on a Queue.
    - Can you implement Queue using an Array or using a Linked List? How?
    - Is there any difference between a Stack and a Queue?
- **Sorting Algorithms** - Wikipedia
    - Using the most efficient sorting algorithm (and correct data structures that implement it) is vital for any program, because data manipulation can be one of the most significant bottlenecks in case of performance and the main purpose of spending time, determining the best algorithm for the job, is to drastically improve said performance. The efficiency of an algorithm is measured in its' "Big O" (StackOverflow) score. Really good algorithms perform important actions in O(n log n) or even O(log n) time and some of them can even perform certain actions in O(1) time (HashTable insertion, for example). But there is always a trade-off - if some algorithm is really good at adding a new element to a data structure, it is, most certainly, much worse at data access than some other algorithm. If you are proficient with math, you may notice that "Big O" notation has many similarities with "limits", and you would be right - it measures best, worst and average performances of an algorithm in question, by looking at its' function limit. It should be noted that, when we are speaking about O(1) - constant time - we are not saying that this algorithm performs an action in one operation, rather that it can perform this action with the same number of operations (roughly), regrardless of the amount of elements it has to take into account. Thankfully, a lot of "Big O" scores have been already calculated, so you don't have to guess, which algorithm or data structure will perform better in your project. "Big O" cheat sheet
    - Bubble sort Wikipedia
        - Bubble sort is one of the simplest sorting algorithms. It just compares neighbouring elements and if the one that precedes the other is smaller - it changes their places. So over one iteration over the data list, it is guaranteed that **at least** one element will be in its' correct place (the biggest/smallest one - depending on the direction of sorting). This is not a very efficient algorithm, as highly unordered arrays will require a lot of reordering (upto O(n^2)), but one of the advantages of this algorithm is its' space complexity - only two elements are compared at once and there is no need to allocate more memory, than those two will occupy.

▪ Time Complexity			▪ Space Complexity
▪ Best	▪ Average	▪ Worst	▪ Worst
▪ Ω(n)	▪ Θ(n^2)	▪ O(n^2)	▪ O(1)

▪

- Selection sort Wikipedia
  - Firstly, selection sort assumes that the first element of the array to be sorted is the smallest, but to confirm this, it iterates over all other elements to check, and if it finds one, it gets defined as the smallest one. When the data ends, the element, that is currently found to be the smallest, is put in the beginning of the array. This sorting algorithm is quite straightforward, but still not that efficient on larger data sets, because to assign just one element to its' place, it needs to go over all data.

▪ Time Complexity			▪ Space Complexity
▪ Best	▪ Average	▪ Worst	▪ Worst

Ω(n^2)	Θ(n^2)	O(n^2)	O(1)

- Insertion sort Wikipedia
  - Insertion sort is another example of an algorithm, that is not that difficult to implement, but is also not that efficient. To do its' job, it "grows" sorted portion of data, by "inserting" new encountered elements into already (innerly) sorted part of the array, which consists of previously encountered elements. This means that in best case (data is already sorted) it can confirm that its' job is done in Ω(n) operations, while, if all encountered elements are not in their required order as many as O(n^2) operations may be needed.

Time Complexity			**Space Complexity**
Best	**Average**	**Worst**	**Worst**
Ω(n)	Θ(n^2)	O(n^2)	O(1)

- Merge sort Wikipedia
  - This is a "divide and conquer" algorithm, meaning it recursively "divides" given array in to smaller parts (up to 1 element) and then sorts those parts, combining them with each other. This approach allows merge sort to achieve very high speed, while doubling required space, of course, but today memory space is more available than it was a couple of years ago, so this trade-off is considered acceptable.

▪ Time Complexity			▪ Space Complexity
▪ Best	▪ Average	▪ Worst	▪ Worst
▪ Ω(nlog(n))	▪ Θ(nlog(n))	▪ O(nlog(n))	▪ O(n)

▪

- Quicksort Wikipedia
  - Quicksort is considered, well, quite quick. When implemented correctly, it can be a significant number of times faster than its' main competitors. This algorithm is also of "divide and conquer" family and its' first step is to choose a "pivot" element (choosing it randomly, statistically, minimizes the chance to get the worst performance), then by comparing elements to this pivot, moving it closer and closer to its' final place. During this process, the elements that are bigger are moved to the right side of it and smaller elements to the left. After this is done, quicksort repeats this process for subarrays on each side of placed pivot (does first step recursively), until the array is sorted.

▪ Time Complexity			▪ Space Complexity
▪ Best	▪ Average	▪ Worst	▪ Worst

Ω(n log(n))	Θ(n log(n))	O(n ^ 2)	O(n)

- There are, of course, more sorting algorithms and their modifications. We strongly recommend all readers to familiarize themselves with a couple more, because knowing algorithms is very important quality of a candidate, applying for a job and it shows understanding of what is happening "under the hood".

- **Binary Tree** - Learn from here
  - What are non-linear data structures? Give example.
  - What is a Tree Data Structure? Explain the properties of tree with an example.
  - How is Binary Tree different from a normal Tree?
  - What is inorder, pre-order, post-order, and level-order traversal of a tree? Explain with an example.
  - Can you find the inorder, pre-order, and post-order of a tree using Stack? How?
  - Explain how searching, insertion, and deletion operations are performed on a Tree?
- **Binary Search Tree** - Learn from here
  - What is a Binary Search Tree? Explain its properties also.
  - Explain how searching, insertion, and deletion operations are performed on a Binary Search Tree?
  - How is Binary Search Tree different from Binary Tree?
- **Heap and Priority Queue** - Learn from here
  - What is a Heap data structure and when it is used?
  - Explain the operations that can be performed on a Heap.
  - What is the difference between a min-heap and a max-heap? How to implement these two?
  - What do you mean by Priority Queue? How to implement Priority Queue?
  - What are the real-life applications of Priority Queue?
- **Hash Table** - Learn from here
  - What do you mean by Direct Address Table?
  - Can you perform search, insert, and delete in O(1)? How?
  - Explain Hash Table and its properties.
  - How to remove collision in Hash Table by Chaining and Open Addressing?
  - What are the real-life applications of Hash Table?
- **Dynamic Programming** - Learn from here
  - What is Dynamic Programming and how to find if a problem can be solved using DP or not?
  - What are two approaches of solving a Dynamic Programming problem?
  - Explain Optimization and Combinatorial problems?
- **Greedy Algorithms** - Learn from here
  - What do you mean by Greedy algorithms? How to find if a problem can be solved by Greedy approach or not?

    - Is there any difference between Dynamic Programming and Greedy Algorithms?
- **Backtracking** - Learn from here
    - What is Backtracking?
    - How to find if a problem can be solved with Backtracking or not?
    - What is Exhaustive Searching?
- **Graph** - Learn from here
    - What is Graph and how to represent a Graph?
    - Explain Depth First Search and Breadth First Search.
    - How to represent a Graph?
    - What are the real-life applications of Graph?
    - What do you mean by Topological Sorting?
    - Explain Dijkstra algorithm with an example.
    - What is a Minimum Spanning Tree?

Other Topics

- **Describe how REST APIs work. What is REST?** - Learn from here and here
- **Describe SQLite.** - Learn from here and here
- **Describe database.** - Learn from here
- **How do you control the application version update to specific number of users?**
- **Can we identify users who have uninstalled our application?**
- **Android Development Best Practices.** - Learn from here
- **Android Code Style And Guidelines.** - Learn from here
- **Have you tried Kotlin?** - Learn from here
- **What are the metrics that you should measure continuously while android application development?** - Learn from here
- **What is Chrome Custom Tabs? How to display web content in your app?** - Learn from here
- **How to avoid API keys from check-in into VCS?** - Learn from here
- **How does the Kotlin Multiplatform work?** - Learn from here
- **How to use Memory Heap Dumps data?** - Learn from here
- **How to implement Dark Theme in your app?** - Learn from here
- **Have you tried Jetpack compose?** - Learn from here
- **How to secure the API keys used in an app?** - Learn from here
- **How to convert check Java equivalent code of Kotlin?** - Learn from here
- **Tell something about memory usage in Android.** - Learn from here
- **What is Benchmarking?** - Learn from here
- **Can you create transparent activity in Android?** - Learn from here
- **How to use Android Sensors?** - Learn from here
- **Explain Annotation processing.** - Learn from here
- **How to increase the Notification delivery rate?** Learn from here
- **How does the notification system work?** Learn from here
- **How to show local Notification at an exact time?** Learn from here