**Java and Android Basics**

* **Name some oops concept in JAVA?**
* { Abstraction
* Encapsulation
* Polymorphism
* Inheritance
* Association
* Aggregation
* Composition}

1. **Can we inherit multiple classes to a single class? :- No , We can use interfaces.**
2. **What is the diamond problem?** { This leads to the ambiguity as compiler doesn’t know which superclass method to execute. Because of the diamond shaped class diagram, it’s referred as Diamond Problem in java. Diamond problem in java is the main reason java doesn’t support multiple inheritance in classes.}
3. **Used ENUM, what is it? should we use them in android? if yes or no.. why, why not?**
4. **Can we declare a class as static? (ans -> no, only inner classes can be static)**
5. **String vs StringBuilde vs StringBuffer?**

Answer:

1. String is immutable whereas StringBuffer and StringBuider are mutable classes.
2. StringBuff+er is thread safe and synchronized whereas StringBuilder is not, thats why StringBuilder is more faster than StringBuffer
3. String concat + operator internally uses StringBuffer or StringBuilder class.
4. For String manipulations in non-multi threaded environment, we should use StringBuilder else use StringBuffer class.
5. **What is Context in Android?**
6. **What is Application?** {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, orany subclass of the Application class, is instantiated before any other class when the process for your application/package is created.}

**Activity Lifecycle**

1. **Name main lifecycle methods of an Android activities**
2. **What happen in activity and in fragment if you rotate screen.**
3. **Scenario in which only onDestroy is called for an activity without onPause() and onStop()?** {If finish() is called in the OnCreate method of an activity, the system will invoke onDestroy() method directly.}
4. **Why would you do the setContentView() in onCreate() of Activity class?** {As onCreate() of an Activity is called only once, this is the point where most initialisation should go. It is inefficient to set the content in onResume() or onStart() (which are called multiple times) as the setContentView() is a heavy operation.}
5. **Which method is called only once in a fragment life cycle**? {onAttached}
6. **Launch modes in Android? {**standard, singleTop, singleTask,SingleInstance}
7. **how and where can we define them?** {Manifest and intent flags}

**Service**

1. **Have you worked on background services? why and when should we basically use it?**
2. **Can we give service same priority as a foreground Activity, if yes how?**
3. **difference in Service and Intent service ?** { Service is the base class for Android services that can be extended to create any service. A class that directly extends Service runs on the main thread so it will block the UI (if there is one) and should therefore either be used only for short tasks or should make use of other threads for longer tasks.}

IntentService is a subclass of Service that handles 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. Writing an IntentService can be quite simple; just extend the IntentService class and override the onHandleIntent(Intent intent) method where you can manage all incoming requests.}

**Broadcast Receiver**

1. **what is BroadcastReceiver?**
2. **examples where we can use it?**
   1. **what is PendingIntent?**

**Intent**

1. **what is intent? do we have different forms of it?** {Explicit - internal activities & Implicit - no specified component, just information}
2. **what is the most common use of intent, can we launch every component of android app using intent**? {Content resolver can’t be launched}
3. **can we send an Object inside Intent, if yes how?**
4. **what is intent filter?**

**Content Provider**

1. **what is it? where have u used it?**
2. **How would you access data in a ContentProvider?**

**OTHERS**

**Multithreading, SQLite,**

**what if i started an asynctask and rotate the device screen while it's doInBG method is still running.**

**can i call network related task in onCreate method directly?**

**wokred with sensors? related classes?**

**Build variants**

singleton patterns

what data-structures have you mainly used in your applications?

Memory management, how can we find memory leakS? what should we consider to prevent memory leaks.

Network connectivity, communication with REST services volley, Retrofit (JSON, XML parsing)

**why you used retrofit? whats new in it over Volley? or vice versa**

**Custom view/layouts**

**FTP**

**kotlin?**

**internationalization/localization? How to achieve multilingual in android?**

**Ever worked on chat application? any idea about XMPP**

**Application of custom fonts ?**

**Hashmap, Arraylist, --> sparseArray**

**Android Architecture Components**

Android architecture components are part of [Android Jetpack](https://developer.android.com/jetpack). They are a collection of libraries that help you design robust, testable, and maintainable apps. Start with classes for managing your UI component lifecycle and handling data persistence.

* Manage your app's lifecycle with ease. New [lifecycle-aware components](https://developer.android.com/topic/libraries/architecture/lifecycle) help you manage your activity and fragment lifecycles. Survive configuration changes, avoid memory leaks and easily load data into your UI.
* Use [LiveData](https://developer.android.com/topic/libraries/architecture/livedata) to build data objects that notify views when the underlying database changes.
* [ViewModel](https://developer.android.com/topic/libraries/architecture/viewmodel) Stores UI-related data that isn't destroyed on app rotations.
* [Room](https://developer.android.com/topic/libraries/architecture/room) is an a SQLite object mapping library. Use it to Avoid boilerplate code and easily convert SQLite table data to Java objects. Room provides compile time checks of SQLite statements and can return RxJava, Flowable and LiveData observables.

Notification ->

Resource not found exception

Network on main thread exception

Memory leakage

Wrong Class cast exception

How to restrict a user to install my application in the case if camera is not there in mobile,

How old application is working on new OS even when it does not have a code to ask permissions at runtime?

Build, deployment, code review,

Prasun – { Uses-feature… 0.5, dangerous permission- 0.3, code review -0.5, take url input from user, send a file to this server using FTP and the application should not crash.}

Neelesh – { }

### 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.

* Array
  + An Array consists of a group of elements of the same data type. It is stored contiguously in memory and by using its' index, you can find the underlying data. Arrays can be one dimensional and multi-dimensional. One dimensional array is the simplest data structure, and also most commonly used. It is worth noting that in Java language multi-dimensional arrays are implemented as arrays of arrays. For example, int[10][5] is actually one array with its' cells pointing to ten 5-element arrays.

| **Algorithm** | **Average** | **Worst Case** |
| --- | --- | --- |
| Space | Θ(n) | O(n) |
| Search | Θ(n) | O(n) |
| Insert | Θ(n) | O(n) |
| Delete | Θ(n) | O(n) |

* LinkedList
  + A LinkedList, just like a tree and unlike an array, consists of a group of nodes which together represent a sequence. Each node contains data and a pointer. The data in a node can be anything, but the pointer is a reference to the next item in the LinkedList. A LinkedList contains both a head and a tail. The "Head" is the first item in the LinkedList, while the "Tail" is the last item. It is not a circular data structure, therefore the tail does not have its' pointer pointing at the Head - the pointer is just null. The run time complexity for each of the base methods are as follows:

| **Algorithm** | **Average** | **Worst Case** |
| --- | --- | --- |
| Space | Θ(n) | O(n) |
| Search | Θ(n) | O(n) |
| Insert | Θ(1) | O(1) |
| Delete | Θ(1) | O(1) |

* DoublyLinkedList
  + A DoublyLinkedList is based on a LinkedList, but there is two pointers in each node, "previous" pointer holds reference to the previous node and "next" pointer holds reference to the next node. It also has a Head node, head node's next pointer references the first node in this DoublyLinkedList. The last node's "next" reference points to null, but if last node's next pointer points to the first node, such DoublyLinkedList is called "Circular DoublyLinkedList". This data structure is very convenient if you need to be able to traverse stored elements in both directions.

| **Algorithm** | **Average** | **Worst Case** |
| --- | --- | --- |
| Space | Θ(n) | O(n) |
| Search | Θ(n) | O(n) |
| Insert | Θ(1) | O(1) |
| Delete | Θ(1) | O(1) |

* Stack
  + A Stack is a basic data structure with a "Last-in-First-out" (LIFO) semantics. This means that the last item that was added to the stack is the first item that comes out of the stack. A Stack is like a stack of books in that in order to get to the first book that was added in the stack (the bottom book), all of the books that were added after need to be removed first. Adding to a Stack is called "Push", removing from a stack is called "Pop", and getting the last item inserted into the stack without removing it is called "Top". The most common way to implement a stack is by using a LinkedList, but there is also StackArray (implemented with an array) which does not replace null entries, and there is also a Vector implementation that does replace null entries. [Wikipedia](https://en.wikibooks.org/wiki/Data_Structures/Stacks_and_Queues#Performance_Analysis)

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Average** | **Worst Case** | **Image representation** |
| Space | Θ(n) | O(n) |  |
| Search | Θ(n) | O(n) |
| Insert (Push) | Θ(1) | O(1) |
| Delete (Pop) | Θ(1) | O(1) |
| Top | Θ(1) | O(1) |

* Queue
* PriorityQueue
* Binary Tree [Wikipedia](https://en.wikipedia.org/wiki/Binary_tree?oldformat=true)
* Binary Search Tree
* Hash Table or Hash Map
* Sorting Algorithms [Wikipedia](https://en.wikipedia.org/wiki/Sorting_algorithm?oldformat=true)
  + 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](https://stackoverflow.com/questions/487258/what-is-a-plain-english-explanation-of-big-o-notation)) 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](http://bigocheatsheet.com/)
  + Bubble sort [Wikipedia](https://en.wikipedia.org/wiki/Bubble_sort?oldformat=true)
    - 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](https://www.wikiwand.com/en/Selection_sort)
    - 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](https://en.wikipedia.org/wiki/Insertion_sort?oldformat=true)
    - 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](https://en.wikipedia.org/wiki/Merge_sort?oldformat=true)
    - 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(n log(n)) | Θ(n log(n)) | O(n log(n)) | O(n) |

* + Quicksort [Wikipedia](https://en.wikipedia.org/wiki/Quicksort?oldformat=true)
    - 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(1) |

* + 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".
* Dynamic Programming
* Greedy Algorithm
* String Manipulation
* Pathfinding algorithms [Wikipedia](https://en.wikipedia.org/wiki/Greedy_algorithm?oldformat=true)
  + Dijkstra algorithm
  + A\* algorithm
  + Breadth First Search
  + Depth First Search

### Core Java

#### OOP

* Explain OOP Concepts.
  + Object-Oriented Programming is a methodology of designing a program using classes, objects, [inheritance](https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)),[polymorphism](https://en.wikipedia.org/wiki/Polymorphism_(computer_science)), [abstraction](https://en.wikipedia.org/wiki/Abstraction_(software_engineering)), and [encapsulation](https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)).
* Differences between abstract classes and interfaces? [GitHub](https://arjun-sna.github.io/java/2017/02/02/abstractvsinterface/)
  + 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?
* Do you agree we use composition over inheritance? [Composition vs Inheritance](https://www.journaldev.com/12086/composition-vs-inheritance)
* Difference between method overloading and overriding.

[](https://github.com/codeshef/android-interview-questions/blob/master/assets/overloading-vs-overriding.png)

* + 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](https://stackoverflow.com/questions/19017258/static-vs-dynamic-binding-in-java).
  + 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):
    1. 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.
    2. 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.
    3. 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.
    4. 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 subinterface.
* 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 [Blog](http://codeinventions.blogspot.in/2014/07/can-interface-extend-multiple.html)
* What are the design patterns? [GitHub](https://github.com/iluwatar/java-design-patterns)
  + Creational patterns
    1. Builder [Wikipedia](https://en.wikipedia.org/wiki/Builder_pattern?oldformat=true)
    2. Factory [Wikipedia](https://en.wikipedia.org/wiki/Factory_method_pattern?oldformat=true)
    3. Singleton [Wikipedia](https://en.wikipedia.org/wiki/Singleton_pattern) A singleton is a class that can only be instantiated once. This singleton pattern restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects.
    4. Monostate [Wikipedia](http://wiki.c2.com/?MonostatePattern)
    5. Fluent Interface Pattern [Wikipedia](https://martinfowler.com/bliki/FluentInterface.html)
  + Structural patterns
    1. Adapter [Wikipedia](https://en.wikipedia.org/wiki/Adapter_pattern?oldformat=true)
    2. Decorator [Wikipedia](https://en.wikipedia.org/wiki/Decorator_pattern?oldformat=true)
    3. Facade [Wikipedia](https://en.wikipedia.org/wiki/Facade_pattern?oldformat=true)
  + Behavioural patterns
    1. Chain of responsibility [Wikipedia](https://en.wikipedia.org/wiki/Chain-of-responsibility_pattern?oldformat=true)
    2. Iterator [Wikipedia](https://en.wikipedia.org/wiki/Iterator_pattern?oldformat=true)
    3. Strategy [Wikipedia](https://en.wikipedia.org/wiki/Strategy_pattern?oldformat=true)

#### Collections and Generics

* Arrays vs ArrayLists.
* HashSet vs TreeSet.
* HashMap vs Set
* Stack vs Queue
* 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](https://docs.oracle.com/javase/tutorial/java/generics/why.html))

* + 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](https://docs.oracle.com/javase/tutorial/extra/generics/wildcards.html) 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](https://docs.oracle.com/javase/tutorial/java/generics/erasure.html)) during compile-time and replaced with object class.
* What is Java PriorityQueue?

#### Objects and Primitives

#### String

* 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](https://docs.oracle.com/javase/tutorial/java/data/strings.html))

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?
* Can you list 8 primitive types in java?
* 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](https://docs.oracle.com/javase/tutorial/java/data/numberclasses.html) 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?
  + Initialization is the process of the memory allocation, when a new variable is created. Variables should be explicitly given a value, otherwise they may contain a random value that remained from the previous variable that was using the same memory space. To avoid this problem, Java language assigns default (right after initialization) values to some data types:
    - boolean defaults to false;
    - byte defaults to 0;
    - short defaults to 0;
    - int defaults to 0;
    - long defaults to 0L;
    - char defaults to \u0000;
    - float defaults to 0.0f;
    - double defaults to 0.0d;
    - object defaults to null.
  + Instantiation is the process of explicitly assigning definitive value to a declared variable:
  + int j; // Initialized variable (int defaults to 0 right after)

j = 10; // Instantiated variable

* 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? [Jenkov](http://tutorials.jenkov.com/java-concurrency/java-memory-model.html)
* What is memory leak and how does Java handle it?
* What are strong, soft, weak and phantom references in Java?

#### Concurrency

* What does the keyword synchronized mean? [Link](https://stackoverflow.com/a/1085745/2621950)
* What is a ThreadPoolExecutor? [MindOrks](https://blog.mindorks.com/threadpoolexecutor-in-android-8e9d22330ee3)
* What is volatile modifier? [Jenkov](http://tutorials.jenkov.com/java-concurrency/volatile.html)
* The clasess 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? [Link](http://tutorials.jenkov.com/java-exception-handling/basic-try-catch-finally.html)
* What is the difference between a Checked Exception and an Un-Checked Exception?

#### 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 Serializeable 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](https://stackoverflow.com/questions/4076946/parcelable-where-when-is-describecontents-used/4914799#4914799). 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? [JavaTPoint](https://www.javatpoint.com/transient-keyword)
* What are anonymous classes?[OracleDoc](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html)
* What is the difference between using == and .equals on an object?[GeeksForGeeks](http://www.geeksforgeeks.org/difference-equals-method-java/)
* What is the hashCode() and equals() used for?
* Why would you not call abstract method in constructor?
* Can you list 8 primitive types in java?
* 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);
  + }
  + 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 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 downcasted), 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? [Jenkov](http://tutorials.jenkov.com/java-reflection/index.html)
* What is Dependency Injection? Can you name few libraries? Have you used any?
  + Dependency injection is a very powerful technique, where you relay the task of providing object with its' dependencies on instances of other objects (OOP Composition, [Wikipedia](https://en.wikipedia.org/wiki/Object_composition?oldformat=true)) to a separate class. This allows for fewer constructors, setters, factories and builders as all those functions are taken care of by the DI framework that you use. Also, and it may seem as a minor advantage, but if you use DI framework you need not worry about going through the project and changing all of (example names) YourCustomInterface customInterfaceObject = new YourCustomClass(); to a new implementaion, as long as your new class (in place of YourCustomClass) still implements CustomInterface - you can just tweak the DI factory class to produce new class and voila - this new class will be automatically instantiated throughout your code. This allows for better maintenence and control over the program. Another example of DI usage is unit-testing - it allows to conveniently inject all needed dependencies and keep the amount of written code at a lower level.
  + One of the most popular libraries for DI for Android is Dagger 2. [MindOrks](https://blog.mindorks.com/a-complete-guide-to-learn-dagger-2-b4c7a570d99c)
* How is a StringBuilder implemented to avoid the immutable string allocation problem?
* Difference between StringBuffer and StringBuilder? [Link](http://www.journaldev.com/137/stringbuffer-vs-stringbuilder)
* What’s the difference between an Enumeration and an Iterator? [Link](http://javaconceptoftheday.com/differences-between-enumeration-vs-iterator-in-java/)
* What is the difference between fail-fast and fail-safe iterators in Java?
* What is Java NIO? [Link](http://tutorials.jenkov.com/java-nio/index.html)

### Core Android

#### Base

* Tell all the Android application components. [Android Official](https://developer.android.com/guide/components/fundamentals.html#Components)
* What is the structure of an Android Application?
* What is Context? How is it used? [MindOrks](https://blog.mindorks.com/understanding-context-in-android-application-330913e32514)
* What is AndroidManifest.xml? [Android Official](https://developer.android.com/guide/topics/manifest/manifest-intro)
* What is Application class?

#### Activity

* What is Activity?
* Explain Activity and Fragment lifecycle. (Complete diagram [GitHub](https://github.com/xxv/android-lifecycle), simplified diagram for [Activity](https://developer.android.com/guide/components/activities/activity-lifecycle.html#alc), [Fragment](https://developer.android.com/guide/components/fragments.html#Lifecycle))
* What are "launch modes"? [MindOrks](https://blog.mindorks.com/android-activity-launchmode-explained-cbc6cf996802)

#### Fragments

* What is Fragment?
* What is the difference between a Fragment and an Activity? Explain the relationship between the two.
* Why is it recommended to use only the default constructor to create a Fragment? [StackOverflow](https://stackoverflow.com/a/16042750/2809326)
* How would you communicate between two Fragments? [Android Official](https://developer.android.com/training/basics/fragments/communicating.html)
* What is retained Fragment? [AndroidDesignPatterns](https://www.androiddesignpatterns.com/2013/04/retaining-objects-across-config-changes.html)

#### Views and ViewGroups

* What is View in Android?
* Difference between View.GONE and View.INVISIBLE? [StackOverflow](https://stackoverflow.com/questions/11556607/android-difference-between-invisible-and-gone)
* Can you create custom views? How? Yes!Excellent documentation in [Google Developers Training advance course](https://google-developer-training.github.io/android-developer-advanced-course-concepts/unit-5-advanced-graphics-and-views/lesson-10-custom-views/10-1-c-custom-views/10-1-c-custom-views.html), custom views chapter
* What are ViewGroups and how they are different from the Views?
* What is a canvas?
* What is a SurfaceView?
* Relative Layout vs Linear Layout.
* Tell about Constraint Layout [MindOrks](https://blog.mindorks.com/using-constraint-layout-in-android-531e68019cd)
* Do you know what is the view tree? How can you optimize its depth?

#### Displaying Lists of Content

* What is the difference between ListView and RecyclerView?
* What is the ViewHolder pattern? Why should we use it?
* What is SnapHelper? [MindOrks](https://blog.mindorks.com/using-snaphelper-in-recyclerview-fc616b6833e8)

#### Dialogs and Toasts

* What is Dialog in Android?
* What is Toast in Android?
* What the difference between Dialog and Dialog Fragment?

#### Intents and Broadcasting

* What is Intent? [StackOverflow](https://stackoverflow.com/questions/6578051/what-is-an-intent-in-android)
* What is an Implicit Intent?
* What is an Explicit Intent?
* What is a BroadcastReceiver? [StackOverflow](https://stackoverflow.com/questions/5296987/what-is-broadcastreceiver-and-when-we-use-it)
* What is a LocalBroadcastManager? [Developer Android](https://developer.android.com/reference/android/support/v4/content/LocalBroadcastManager.html)
* What is the function of an IntentFilter?
* What is a Sticky Intent? [AndroidInterview](http://www.androidinterview.com/what-is-a-sticky-intent/)
* Describe how broadcasts and intents work to be able to pass messages around your app?
* What is a PendingIntent?
* What are the different types of Broadcasts?

#### Services

* What is Serivce?
* Service vs IntentService. [StackOverflow](https://stackoverflow.com/a/15772151/5153275)
* What is a JobScheduler? [Vogella](http://www.vogella.com/tutorials/AndroidTaskScheduling/article.html)

#### Inter-process Communication

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

#### Long-running Operations

* How would you perform a long-running operation in an application?
* Why should you avoid to run non-ui code on the main thread?
* What is ANR? How can the ANR be prevented?
* What is an AsyncTask?
* What are the problems in asynctask?
* When would you use java thread instead of an asynctask?
* What is a Loader?
* 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?
* Explain Looper, Handler and HandlerThread. [MindOrks](https://blog.mindorks.com/android-core-looper-handler-and-handlerthread-bd54d69fe91a) and [MindOrks Video](https://www.youtube.com/watch?v=rfLMwbOKLRk&list=PL6nth5sRD25hVezlyqlBO9dafKMc5fAU2)

#### Working With Multimedia Content

* How do you handle bitmaps in Android as it takes too much memory?
* What is the difference between a regular Bitmap and a nine-patch image?
* Tell about the Bitmap pool. [MindOrks](https://blog.mindorks.com/how-to-use-bitmap-pool-in-android-56c71a55533c)
* How to play sounds in Android? [Vogella](http://www.vogella.com/tutorials/AndroidMedia/article.html)

#### Data Saving

* How to persist data in an Android app?
* What is ORM? How does it work?
* How would you preserve Activity state during a screen rotation? [StackOverflow](https://stackoverflow.com/questions/3915952/how-to-save-state-during-orientation-change-in-android-if-the-state-is-made-of-m)
* What are different ways to store data in your Android app?

#### Look and Feel

* What is a Spannable?

#### Memory Optimizations

* What is the onTrimMemory() method?
* How does the OutOfMemory happens?
* How do you find memory leaks in Android applications? [MindOrks](https://mindorks.com/blog/detecting-and-fixing-memory-leaks-in-android)

#### Battery Life Optimizations

* How to reduce battery usage in an android application? [MindOrks](https://blog.mindorks.com/battery-optimization-for-android-apps-f4ef6170ff70)
* What is Doze? What about App Standby?
* What is overdraw? [Developer Android](https://developer.android.com/topic/performance/rendering/overdraw.html)

#### Supporting Different Screen Sizes

* How did you support different types of resolutions?

#### Permissions

* What are the different protection levels in permission?

#### Native Programming

* What is the NDK and why is it useful?
* What is renderscript? [MindOrks](https://blog.mindorks.com/comparing-android-ndk-and-renderscript-1a718c01f6fe)

#### Android System Internal

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

#### Debugging and Programming Tools

* What is ADB?
* What is DDMS and what can you do with it?
* What is the StrictMode? [MindOrks](https://blog.mindorks.com/use-strictmode-to-find-things-you-did-by-accident-in-android-development-4cf0e7c8d997)
* What is Lint? What is it used for?

#### Others

* Why Bundle class is used for data passing and why cannot we use simple Map data structure
* How do you troubleshoot a crashing application?
* Explain Android notification system?
* What is the difference between Serializable and Parcelable? Which is the best approach in Android?
* Have you developed widgets? Describe. [MindOrks](https://blog.mindorks.com/android-widgets-ad3d166458d3)
* What is AAPT?
* What is the best way to update the screen periodically?
* FlatBuffers vs JSON. [MindOrks](https://blog.mindorks.com/why-consider-flatbuffer-over-json-2e4aa8d4ed07)
* HashMap, ArrayMap and SparseArray [MindOrks](https://blog.mindorks.com/android-app-optimization-using-arraymap-and-sparsearray-f2b4e2e3dc47)
* What are Annotations? [Mindorks](https://blog.mindorks.com/creating-custom-annotations-in-android-a855c5b43ed9), [Link](https://blog.mindorks.com/improve-your-android-coding-through-annotations-26b3273c137a)
* How to handle multi-touch in android [GitHub](https://arjun-sna.github.io/android/2016/07/20/multi-touch-android/)
* How to implement XML namespaces?
* What is the support library? Why was it introduced?[MartianCraft](http://martiancraft.com/blog/2015/06/android-support-library/)
* What is Android Data Binding? [Developer Android](https://developer.android.com/topic/libraries/data-binding/index.html)
* What are Android Architecture Components? [Developer Android](https://developer.android.com/topic/libraries/architecture/index.html)
* How to implement search using RxJava operators? [MindOrks](https://blog.mindorks.com/implement-search-using-rxjava-operators-c8882b64fe1d)

### Architecture

* Describe the architecture of your last app.
* Describe MVP. [MindOrks](https://mindorks.com/course/android-mvp-introduction)
* What is presenter?
* What is model?
* Describe MVC.
* Describe MVI
* Describe the repository pattern
* What is controller?
* Describe MVVM. [GitHub](https://github.com/MindorksOpenSource/android-mvvm-architecture)
* Tell something about clean code [MindOrks](https://blog.mindorks.com/every-programmer-should-read-this-book-6755dedec78d)

### Design Problem

* Design Uber App.
* Design Facebook App.
* Design Facebook Near-By Friends App.
* Design WhatsApp.
* Design SnapChat.
* Design problems based on location based app.

### Tools And Technologies

* Git. [MindOrks Youtube](https://www.youtube.com/watch?v=D4h8Dbrjt4M&list=PL6nth5sRD25itbyNVUULAebzL-VLrLfkK)
* RxJava. [MindOrks](https://blog.mindorks.com/a-complete-guide-to-learn-rxjava-b55c0cea3631)
* Dagger 2. [MindOrks](https://blog.mindorks.com/a-complete-guide-to-learn-dagger-2-b4c7a570d99c)
* Android Development Useful Tools. [MindOrks](https://blog.mindorks.com/android-development-useful-tools-fd73283e82e3)
* Firebase. [Firebase.google.com](https://firebase.google.com/)

### Android Test Driven Development

* What is Espresso? [Developer Android](https://developer.android.com/training/testing/ui-testing/espresso-testing.html)
* What is Robolectric? [Robolectric](http://robolectric.org/)
* What are the disadvantages of using Roboelectric?
* What is UI-Automator? [Developer Android](https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html)
* Explain unit test.
* Explain instrumented test.
* Have you done unit testing or automatic testing?
* Why Mockito is used? [Official site](http://site.mockito.org/)
* Describe JUnit test.

### Others

* Describe how REST APIs work. What is REST?
* Describe other forms of web API architecutre. [GraphQL] (<https://medium.com/mindorks/what-is-graphql-and-using-it-on-android-ab8e493abdd7>) [SOAP] (<https://www.w3.org/TR/ws-arch/>)
* Describe SQLite.
* Describe database.
* Project Management tool - trello, basecamp, kanban, jira, asana.
* About build System - gradle, maven, ant, buck.
* About multiple apk for android application. [MindOrks](https://mindorks.com/blog/how-to-create-multiple-apk-files-for-android-application)
* Reverse Engineering an APK.
* What is proguard used for?
* What is obfuscation? What is it used for? What about minification?
* How do you build your apps for release?
* How do you control the application version update to specific number of users?
* Can we identify users who have uninstalled our application?
* Implement Search Using RxJava Operators. [MindOrks](https://blog.mindorks.com/implement-search-using-rxjava-operators-c8882b64fe1d)
* APK Size Reduction. [MindOrks](https://blog.mindorks.com/how-to-reduce-apk-size-in-android-2f3713d2d662)
* Android Development Best Practices. [MindOrks](https://blog.mindorks.com/android-development-best-practices-83c94b027fd3)
* Android Code Style And Guidelines. [MindOrks](https://blog.mindorks.com/android-code-style-and-guidelines-d5f80453d5c7)
* Have you tried Kotlin? [MindOrks](https://blog.mindorks.com/why-you-must-try-kotlin-for-android-development-e14d00c8084b)
* What are the metrics that you should measure continuously while android application development? [MindOrks](https://blog.mindorks.com/android-app-performance-metrics-a1176334186e)