***Technical Concepts Handbook***

***- Siddhant Patni***

***Mobile: +91-7588811796***

***Email: siddhant4patni@gmail.com***

Contents

[***Java*** 25](#_Toc195650033)

[**Java Fundamentals** 25](#_Toc195650034)

[ **Explain how to install java? Explain directories in it?** 25](#_Toc195650035)

[ **How to configure java in windows?** 26](#_Toc195650036)

[ **Explain Java Program execution?** 26](#_Toc195650037)

[ **Explain the byte stream in java?** 28](#_Toc195650038)

[ **What is Classloader? Different types of it?** 29](#_Toc195650039)

[ **Why is java more secure?** 30](#_Toc195650040)

[ **Explain public static void main (String[] args) method?** 31](#_Toc195650041)

[ **What is the marker interface?** 32](#_Toc195650042)

[ **What is serialization and deserialization?** 32](#_Toc195650043)

[ **What is the use of the transient keyword in serialization?** 34](#_Toc195650044)

[ **How can I make below Employee class as Serializable class?** 35](#_Toc195650045)

[ **In above example, what if we have empId as static then what will happen at the time of serialization?** 36](#_Toc195650046)

[ **What is the final class?** 37](#_Toc195650047)

[ **Where can we use the final keyword?** 38](#_Toc195650048)

[ **What will happen if I create the final class?** 39](#_Toc195650049)

[ **What is String?** 40](#_Toc195650050)

[ **Why String is immutable in java?** 41](#_Toc195650051)

[ **What is StringBuffer?** 42](#_Toc195650052)

[ **What is StringBuilder?** 43](#_Toc195650053)

[ **Difference between String, StringBuffer and StringBuilder?** 44](#_Toc195650054)

[ **Difference between comparable and comparator?** 45](#_Toc195650055)

[ **What is static in java?** 46](#_Toc195650056)

[ **Explain Generics? What is the use of Generics?** 47](#_Toc195650057)

[ **What is Singleton in java?** 48](#_Toc195650058)

[ **Consider we have Strings as String s1 = “Welcome”, String s2 = “Welcome” and String s3 = “WelcomeOne”. What does s1==s2 represent?** 49](#_Toc195650059)

[ **Explain String constant pool?** 50](#_Toc195650060)

[ **What is the difference between equals() and hashCode() method?** 50](#_Toc195650061)

[ **Does Java support Pointers?** 51](#_Toc195650062)

[ **Why doesn't java support pointers?** 52](#_Toc195650063)

[ **Explain which language you prefer in terms of security among C, C++, Java and Python?** 53](#_Toc195650064)

[ **Is Java 100% object oriented?** 54](#_Toc195650065)

[ **How can we make java 100% object oriented?** 55](#_Toc195650066)

[ **Can we create objects of static class?** 56](#_Toc195650067)

[ **Which compiler is used by Java?** 58](#_Toc195650068)

[ **Where are the hashCode() and equals() methods defined in java?** 58](#_Toc195650069)

[ **How many design patterns are present in java?** 58](#_Toc195650070)

[ **Can we override the static method?** 58](#_Toc195650071)

[ **Explain access modifiers in java?** 58](#_Toc195650072)

[ **What is volatile in java? Where can we use it?** 58](#_Toc195650073)

[ **What is an idempotent method in java?** 58](#_Toc195650074)

[ **What is final, static and non-static (instance) in java?** 58](#_Toc195650075)

[ **How will I use two threads using a singleton design pattern?** 58](#_Toc195650076)

[ **How to implement thread safety in java using singleton?** 58](#_Toc195650077)

[ **Write an immutable class in Java?** 58](#_Toc195650078)

[ **Can we override the protected method?** 59](#_Toc195650079)

[ **What is marker interface in java? What is the purpose of the marker interface?** 59](#_Toc195650080)

[ **How can we ignore the finally block?** 59](#_Toc195650081)

[ **What is synchronous and asynchronous call?** 59](#_Toc195650082)

[ **Can we write the main method as private?** 59](#_Toc195650083)

[ **What is the use of static keywords in java?** 59](#_Toc195650084)

[ **What is inner class? Have you used inner class in your project?** 59](#_Toc195650085)

[ **What is the purpose of inner class?** 59](#_Toc195650086)

[ **What is Enum? Write a syntax for enum?** 59](#_Toc195650087)

[ **Difference between static and non-static in java?** 59](#_Toc195650088)

[ **How can we break the singleton pattern?** 59](#_Toc195650089)

[ **Difference between default and protected access modifier?** 59](#_Toc195650090)

[ **Why is string immutable?** 60](#_Toc195650091)

[ **What is an immutable class?** 60](#_Toc195650092)

[ **What is atomic integer in java?** 60](#_Toc195650093)

[ **How can you make a singleton class as thread safe?** 60](#_Toc195650094)

[ **Difference between compareTo and compare method?** 60](#_Toc195650095)

[ **What is the return type of compare method?** 60](#_Toc195650096)

[ **What is contract in equals and hashcode method?** 60](#_Toc195650097)

[ **Difference between Marker interface and Functional interface?** 60](#_Toc195650098)

[ **Difference between Abstract class and Interface? Explain with scenarios which one you will it be used with conditions?** 60](#_Toc195650099)

[ **What is the use of static and instance variables in java?** 60](#_Toc195650100)

[ **How will you access JNDI in java code?** 60](#_Toc195650101)

[ **Why is the main method in java static?** 60](#_Toc195650102)

[ **What is the use of the static keyword?** 60](#_Toc195650103)

[ **How many ways can we create beans in java?** 60](#_Toc195650104)

[ **Explain different types of calling beans?** 60](#_Toc195650105)

[ **What are the default methods?** 60](#_Toc195650106)

[ **Can I add one more new default method in my child class? Does it give any problems with my existing classes?** 60](#_Toc195650107)

[ **Difference between Comparable and Comparator interface?** 60](#_Toc195650108)

[ **In Java, which class is the base class of all the classes?** 60](#_Toc195650109)

[ **Why is the equals method required?** 60](#_Toc195650110)

[ **Is it possible to have the same hashcode for multiple objects?** 61](#_Toc195650111)

[ **Enlist a few predefined interfaces from Java 8?** 61](#_Toc195650112)

[ **If I have a 200 MB excel file so can java read that file?** 61](#_Toc195650113)

[ **What is immutability and mutability in java?** 61](#_Toc195650114)

[ **Custom implementation of Singleton beans?** 61](#_Toc195650115)

[ **What is the default scope of beans?** 61](#_Toc195650116)

[ **Difference between static and final?** 61](#_Toc195650117)

[ **Explain about String class in java?** 61](#_Toc195650118)

[ **What is a special feature of String?** 61](#_Toc195650119)

[ **Create an immutable class in java?** 61](#_Toc195650120)

[ **How to access Singleton class from another class?** 61](#_Toc195650121)

[ **Where will the Singleton class be stored in memory?** 61](#_Toc195650122)

[ **Difference between bucket level and segment level locking?** 61](#_Toc195650123)

[ **How is the static method bound with an object?** 61](#_Toc195650124)

[ **What is the use of the intern() method in String class?** 61](#_Toc195650125)

[**Java Memory Management** 62](#_Toc195650126)

[ **Explain JDK, JRE?** 62](#_Toc195650127)

[ **Explain JVM?** 62](#_Toc195650128)

[ **What are the different loaders in JVM?** 62](#_Toc195650129)

[ **What is JVM profiling?** 62](#_Toc195650130)

[ **Explain Java memory management?** 63](#_Toc195650131)

[ **What is Stack / Heap memory?** 63](#_Toc195650132)

[ **Explain memory management below? Int and Integer?** 63](#_Toc195650133)

[ **Where local variables are stored in memory?** 63](#_Toc195650134)

[ **Where objects will be stored in memory?** 63](#_Toc195650135)

[ **Explain Heap memory in java?** 63](#_Toc195650136)

[ **Difference between Stack and Heap?** 63](#_Toc195650137)

[ **What is Garbage Collection in java?** 63](#_Toc195650138)

[ **What will happen if I assign null to some object then will that be garbage collected?** 64](#_Toc195650139)

[ **When does the finalize method get called?** 64](#_Toc195650140)

[**OOPs** 67](#_Toc195650141)

[ **Explain OOPs concept?** 67](#_Toc195650142)

[ **What is an object?** 67](#_Toc195650143)

[ **Explain method in Object class? Explain 9 methods in Object class?** 67](#_Toc195650144)

[ **How many ways can we create objects of class in java?** 67](#_Toc195650145)

[ **Explain Abstraction?** 67](#_Toc195650146)

[ **What is Encapsulation? Explain with a real time example** 67](#_Toc195650147)

[ **How do we achieve encapsulation in Java?** 68](#_Toc195650148)

[ **What is abstract in java?** 69](#_Toc195650149)

[ **Explain Composition in Oops?** 70](#_Toc195650150)

[ **What is composition and aggregation?** 70](#_Toc195650151)

[ **Parent and child class methods and how to access those?** 70](#_Toc195650152)

[ **Difference between interface and abstract class?** 70](#_Toc195650153)

[ **Where will you use interface and abstract class with real time examples?** 70](#_Toc195650154)

[ **What is encapsulation? How are you currently using it?** 70](#_Toc195650155)

[ **Explain Compile time and Run time polymorphism?** 70](#_Toc195650156)

[ **What is method overriding?** 70](#_Toc195650157)

[ **How to achieve method overriding?** 70](#_Toc195650158)

[ **What is method overloading?** 70](#_Toc195650159)

[ **What is Polymorphism? Explain with an example?** 70](#_Toc195650160)

[ **What is Inheritance?** 70](#_Toc195650161)

[ **What is the interface in java?** 70](#_Toc195650162)

[ **Can we have a static method inside the interface?** 71](#_Toc195650163)

[ **Can we override the static method?** 71](#_Toc195650164)

[ **We have Class A and Class B which contains m1() method in both the class, and class B extends parent class A. Asked which method will get called on which object?** 71](#_Toc195650165)

[ **Explain SOLID Principles?** 71](#_Toc195650166)

[**Collection Framework** 74](#_Toc195650167)

[ **Difference between List and Array?** 74](#_Toc195650168)

[ **Difference between List and Set?** 74](#_Toc195650169)

[ **Difference between ArrayList and LinkedList?** 74](#_Toc195650170)

[ **Difference between Array and Collections?** 74](#_Toc195650171)

[ **Difference between HashMap and LinkedHashMap?** 74](#_Toc195650172)

[ **Difference between HashMap and HashTable?** 75](#_Toc195650173)

[ **Difference between Fail-Fast and Fail-Safe iterator?** 75](#_Toc195650174)

[ **Which is more efficient among the ArrayList and LinkedList?** 75](#_Toc195650175)

[ **Difference between Stream and Collections?** 75](#_Toc195650176)

[ **Write a syntax for integer ArrayList?** 75](#_Toc195650177)

[ **How can Set not store duplicate values? Explain internal working?** 75](#_Toc195650178)

[ **Which collection you will use in order to remove duplicate elements and preserve the insertion order?** 75](#_Toc195650179)

[ **Which Collection will you avoid duplicating and store data in a sorted manner?** 75](#_Toc195650180)

[ **In which scenarios Linked list comes into fixture in HashMap?** 75](#_Toc195650181)

[ **Which among ArrayList and LinkedList will be more efficient for random access of data?** 75](#_Toc195650182)

[ **Difference between Array, ArrayList and LinkedList? When you will use ArrayList and LinkedList?** 75](#_Toc195650183)

[ **How to sort the ArrayList?** 75](#_Toc195650184)

[ **How to sort the ArrayList?** 75](#_Toc195650185)

[ **What are the practical use cases of LinkedList?** 75](#_Toc195650186)

[ **Compare ArrayList objects from the employee and return the highest employee age?** 75](#_Toc195650187)

[ **How will you find the 3rd last element from the linked list in a single iteration?** 75](#_Toc195650188)

[ **How does ArrayList internally work?** 75](#_Toc195650189)

[ **What is the default size of ArrayList?** 75](#_Toc195650190)

[ **How to increase the size of ArrayList? By how much?** 75](#_Toc195650191)

[ **Difference between Map and Set?** 76](#_Toc195650192)

[ **How HashMap calls hashCode and equals method internally?** 76](#_Toc195650193)

[ **Explain internal working of ArrayList?** 76](#_Toc195650194)

[ **How does ArrayList increase its size dynamically?** 76](#_Toc195650195)

[ **How does ArrayList grow?** 76](#_Toc195650196)

[ **Explain internal working of ensureCapacity method?** 76](#_Toc195650197)

[ **How do you write a custom ArrayList which will take the input capacity?** 76](#_Toc195650198)

[ **Which data structure is used by ArrayList?** 76](#_Toc195650199)

[ **Difference between ArrayList and Vector?** 76](#_Toc195650200)

[ **Best time complexity of Arraylist and Linkedlist? Which is better?** 76](#_Toc195650201)

[ **What is HashMap?** 76](#_Toc195650202)

[ **Explain internal working of HashMap?** 79](#_Toc195650203)

[ **What is hash collision in HashMap?** 80](#_Toc195650204)

[ **What are the types of HashMap?** 81](#_Toc195650205)

[ **What happens in a HashMap when multiple keys have the same hashCode()??** 82](#_Toc195650206)

[ **Can HashMap store different keys that have the same hashCode?** 82](#_Toc195650207)

[ **What if equals() returns true for two keys with same hashCode()?** 83](#_Toc195650208)

[ **Can we store null value in HashMap?** 84](#_Toc195650209)

[ **Can we store infinite data in a HashMap?** 84](#_Toc195650210)

[ **Can we access null value in HashMap?** 85](#_Toc195650211)

[ **How hashcode calculations happened on the Employee object?** 86](#_Toc195650212)

[ **Can we have duplicate keys in HashMap?** 86](#_Toc195650213)

[ **How does HashMap come to know that two objects are the same or different?** 86](#_Toc195650214)

[ **What is the use of a ConcurrentHashMap?** 86](#_Toc195650215)

[ **How does ConcurrentHashMap works?** 86](#_Toc195650216)

[ **Difference between HashMap and ConcurrentHashMap?** 86](#_Toc195650217)

[ **Difference between HashMap and HashSet** 86](#_Toc195650218)

[ **What is the default size of ArrayList?** 86](#_Toc195650219)

[ **How to increase the size of ArrayList?** 86](#_Toc195650220)

[ **Can we add infinite elements to ArrayList?** 86](#_Toc195650221)

[ **Can we store infinite elements in a list?** 86](#_Toc195650222)

[ **How can we add data while iterating ArrayList?** 86](#_Toc195650223)

[ **What is the difference between List and Map?** 86](#_Toc195650224)

[ **Explain sorted collections in java?** 86](#_Toc195650225)

[ **Which data structure is used by HashMap?** 86](#_Toc195650226)

[ **How can Set can’t store duplicate data?** 86](#_Toc195650227)

[ **Explain HashSet?** 86](#_Toc195650228)

[ **Which data structure used by HashSet?** 89](#_Toc195650229)

[ **If we have added the same key in HashMap with the same value and different key with the same hash with the same value, then how will it be stored? At what position data will store and how it will store? What will happen when we use ConcurrentHashMap?** 89](#_Toc195650230)

[ **What is the default capacity of HashMap?** 89](#_Toc195650231)

[ **Where will you use ConcurrentHashMap?** 89](#_Toc195650232)

[ **Explain HashMap vs ConcurrentHashMap?** 89](#_Toc195650233)

[ **How is HashMap faster as compared to ConcurrentHashMap?** 89](#_Toc195650234)

[ **Suppose I have multiple Employee objects with different names, so how does HashMap differentiate between different objects?** 89](#_Toc195650235)

[ **What is a Blocking Queue?** 89](#_Toc195650236)

[ **What is Enumeration in java?** 90](#_Toc195650237)

[**Exception Handling** 92](#_Toc195650238)

[ **What is Exception Handling?** 92](#_Toc195650239)

[ **Explain the hierarchy of Exception?** 92](#_Toc195650240)

[ **How to create Custom Exceptions?** 92](#_Toc195650241)

[ **How to handle IOException?** 92](#_Toc195650242)

[ **Difference between Checked Exception and Unchecked Exception?** 92](#_Toc195650243)

[ **How to handle unchecked exceptions?** 92](#_Toc195650244)

[ **What will happen if I use Exception and custom exceptions together? Priority?** 92](#_Toc195650245)

[ **Have you created a custom exception? What is the purpose of that?** 92](#_Toc195650246)

[ **What is the use of finally block in exception handling?** 92](#_Toc195650247)

[ **What is a try with resources?** 92](#_Toc195650248)

[ **Have you come across the OutOfMemory exception? How will you handle it? Explain with standalone code?** 92](#_Toc195650249)

[ **If you have try, catch and finally block and if I have written another exception inside catch block so how control will flow?** 92](#_Toc195650250)

[ **Have you ever got outOfMemory exceptions? Can you tell me any scenarios where you have faced this exception?** 92](#_Toc195650251)

[**Java 8 Features** 93](#_Toc195650252)

[ **Explain Java8 features?** 93](#_Toc195650253)

[ **What is Functional Interface?** 94](#_Toc195650254)

[ **Why we need Functional Interface?** 94](#_Toc195650255)

[ **Difference between normal interface and Functional Interface?** 94](#_Toc195650256)

[ **What is the default method in functional interface?** 94](#_Toc195650257)

[ **What is Lambda Expression?** 94](#_Toc195650258)

[ **What is the prerequisite for Lambda Expression?** 94](#_Toc195650259)

[ **What is the map function in stream?** 94](#_Toc195650260)

[ **What is the filter in the stream?** 94](#_Toc195650261)

[ **What is Stream API?** 95](#_Toc195650262)

[ **What is the use of @FunctionalInterface annotation? Will it be checked manually or by the compiler?** 95](#_Toc195650263)

[ **Difference between Map and FlatMap in stream?** 95](#_Toc195650264)

[ **Difference between Map and Filter in java 8?** 95](#_Toc195650265)

[ **What is the intermediate and terminal operator in Java stream API?** 95](#_Toc195650266)

[ **Advantages of Stream over for loop?** 95](#_Toc195650267)

[ **What is the lambda expression in java 8?** 95](#_Toc195650268)

[ **How to use lambda expressions with one example?** 95](#_Toc195650269)

[ **What is Stream and Parallel stream?** 96](#_Toc195650270)

[ **Explain Optional class in java 8?** 96](#_Toc195650271)

[ **What is the purpose of Optional class?** 96](#_Toc195650272)

[ **Explain Java 8 Functional interface?** 96](#_Toc195650273)

[ **Where can you use FlatMap in a stream?** 96](#_Toc195650274)

[ **What is the lambda expression?** 96](#_Toc195650275)

[ **What is the use map method in java 8 stream?** 96](#_Toc195650276)

[ **How to sort the list of objects in java using Stream API?** 96](#_Toc195650277)

[ **What is the inbuilt functional interface in Java 8?** 96](#_Toc195650278)

[ **What is Predicate and Consumer in java 8?** 96](#_Toc195650279)

[ **Difference between Predicate and Function?** 97](#_Toc195650280)

[ **Explain Consumer and Supplier?** 97](#_Toc195650281)

[ **What is the use of Map?** 98](#_Toc195650282)

[ **What is Optional class? What are the advantages of using Optional class?** 98](#_Toc195650283)

[ **What are all functional interfaces present in java?** 98](#_Toc195650284)

[ **Optional case returning null and String, how to handle it? What are the advantages of using it?** 98](#_Toc195650285)

[ **Give an example of Functional Interface in java?** 98](#_Toc195650286)

[ **Suppose you are using lambda expressions and other than lambda expressions then are we getting any performance benefits?** 99](#_Toc195650287)

[ **Can you elaborate on the Stream API, purpose and use of it?** 99](#_Toc195650288)

[ **Can we have more than one abstract method in a functional interface?** 99](#_Toc195650289)

[ **Have you worked on NullPointerException? How to handle NullPointerException in java 8?** 99](#_Toc195650290)

[ **Difference between Parallel Stream and Serial Stream?** 99](#_Toc195650291)

[**Java 11 Features** 101](#_Toc195650292)

[ **Explain Java 11 features?** 101](#_Toc195650293)

[**Java 17 Features** 103](#_Toc195650294)

[ **Explain Java 17 features?** 103](#_Toc195650295)

[**Java 21 Features** 105](#_Toc195650296)

[ **Explain Java 21 features?** 105](#_Toc195650297)

[ **What is the added version after Java 17 to Java 21?** 106](#_Toc195650298)

[**Multithreading** 107](#_Toc195650299)

[ **What is multithreading in java?** 107](#_Toc195650300)

[ **Difference between Process and Thread?** 107](#_Toc195650301)

[ **Explain the Thread Life Cycle?** 107](#_Toc195650302)

[ **What is a deadlock? Explain with an example?** 107](#_Toc195650303)

[ **What is a Synchronized Thread?** 107](#_Toc195650304)

[ **Difference between synchronized and volatile keyword?** 107](#_Toc195650305)

[ **What happens if a thread calls notify() instead of notifyAll()?** 107](#_Toc195650306)

[ **How to create threads using java 1.7 and java 1.8?** 107](#_Toc195650307)

[ **Difference between wait and sleep?** 107](#_Toc195650308)

[ **What is Thread.sleep() method, and what does it do internally?** 107](#_Toc195650309)

[ **Difference between Thread and Runnable in Java?** 107](#_Toc195650310)

[ **What is significance of Executors in Java?** 107](#_Toc195650311)

[ **Explain the Producer Consumer problem and how would you implement it using wait(), notify() or BlockingQueue?** 107](#_Toc195650312)

[ **Difference between Callable and Runnable. When should you use each?** 107](#_Toc195650313)

[ **What is ForkJoinPool in Java and how does it differ from normal ExecutorService?** 107](#_Toc195650314)

[ **Explain Deadlock and how do you avoid it in Java Multithreading?** 107](#_Toc195650315)

[ **What are the ways of blocking the threads in java?** 107](#_Toc195650316)

[ **What is thread synchronization in java?** 107](#_Toc195650317)

[ **What are the methods of thread?** 107](#_Toc195650318)

[ **What is thread pool? What are the types of thread pool?** 108](#_Toc195650319)

[ **In Thread Pool Executor which data structure is used to store the request data?** 108](#_Toc195650320)

[ **When we setup a Thread Pool Executor so what will be the ideal count, how many thread we should keep in it?** 109](#_Toc195650321)

[ **What are thread pool methods?** 110](#_Toc195650322)

[ **How to create threads in java?** 110](#_Toc195650323)

[ **How to start a thread in java?** 110](#_Toc195650324)

[ **How to create threads in two different ways?** 110](#_Toc195650325)

[ **What is the necessity of thread synchronization?** 110](#_Toc195650326)

[ **How to create a Thread pool in java?** 110](#_Toc195650327)

[ **Which method you will override in order to create a thread?** 110](#_Toc195650328)

[ **What are all methods present in Thread class?** 110](#_Toc195650329)

[ **How do notify methods work in multithreading?** 110](#_Toc195650330)

[ **What is synchronized in multithreading?** 110](#_Toc195650331)

[ **Thread safety with Singleton class?** 110](#_Toc195650332)

[ **Explain Thread executors?** 110](#_Toc195650333)

[ **How to create three threads so that it will execute simultaneously?** 110](#_Toc195650334)

[ **What can we replace instead of this keyword in the statement? synchronized(this) {?** 110](#_Toc195650335)

[ **What is runAsync vs supplyAsync?** 110](#_Toc195650336)

[ **What is ReentrantLock in java?** 111](#_Toc195650337)

[ **Difference between lock and synchronized?** 113](#_Toc195650338)

[ **What is Callable interface?** 113](#_Toc195650339)

[ **What is Runnable interface?** 113](#_Toc195650340)

[ **Difference between Future and CompletableFuture?** 114](#_Toc195650341)

[**Scenario based Questions** 117](#_Toc195650342)

[ **Scenario based question - 1?** 117](#_Toc195650343)

[ **Scenario based question - 2?** 117](#_Toc195650344)

[ **Scenario based question - 3?** 117](#_Toc195650345)

[ **Scenario based question - 4?** 117](#_Toc195650346)

[ **Scenario based question - 5?** 118](#_Toc195650347)

[**JDBC** 119](#_Toc195650348)

[ **What is JDBC?** 119](#_Toc195650349)

[***Java Programs*** 120](#_Toc195650350)

[ **Program – 1** 120](#_Toc195650351)

[ **Program – 2** 120](#_Toc195650352)

[ **Program – 3** 120](#_Toc195650353)

[ **Program – 4** 120](#_Toc195650354)

[ **Program – 5** 120](#_Toc195650355)

[ **Program – 6** 120](#_Toc195650356)

[ **Program – 7** 121](#_Toc195650357)

[ **Program – 8** 121](#_Toc195650358)

[ **Program – 9** 122](#_Toc195650359)

[ **Program – 10** 122](#_Toc195650360)

[ **Program – 11** 122](#_Toc195650361)

[ **Program – 12** 122](#_Toc195650362)

[ **Program – 13** 122](#_Toc195650363)

[ **Program – 14** 122](#_Toc195650364)

[ **Program – 15** 122](#_Toc195650365)

[ **Program – 16** 122](#_Toc195650366)

[ **Program – 17** 122](#_Toc195650367)

[ **Program – 18** 122](#_Toc195650368)

[ **Program – 19** 122](#_Toc195650369)

[ **Program – 20** 122](#_Toc195650370)

[ **Program – 21** 123](#_Toc195650371)

[ **Program – 22** 123](#_Toc195650372)

[ **Program – 23** 123](#_Toc195650373)

[ **Program – 24** 123](#_Toc195650374)

[ **Program – 25** 123](#_Toc195650375)

[ **Program – 26** 123](#_Toc195650376)

[ **Program – 27** 123](#_Toc195650377)

[ **Program – 28** 123](#_Toc195650378)

[ **Program – 29** 123](#_Toc195650379)

[ **Program – 30** 123](#_Toc195650380)

[ **Program – 31** 124](#_Toc195650381)

[ **Program – 32** 124](#_Toc195650382)

[ **Program – 33** 124](#_Toc195650383)

[ **Program – 34** 125](#_Toc195650384)

[ **Program – 35** 125](#_Toc195650385)

[ **Program – 36** 125](#_Toc195650386)

[ **Program – 37** 125](#_Toc195650387)

[ **Program – 38** 125](#_Toc195650388)

[ **Program – 39** 125](#_Toc195650389)

[ **Program – 40** 125](#_Toc195650390)

[ **Program – 41** 125](#_Toc195650391)

[ **Program – 42** 125](#_Toc195650392)

[ **Program – 43** 125](#_Toc195650393)

[ **Program – 44** 126](#_Toc195650394)

[ **Program – 45** 126](#_Toc195650395)

[ **Program – 46** 126](#_Toc195650396)

[ **Program – 47** 126](#_Toc195650397)

[ **Program – 48** 126](#_Toc195650398)

[ **Program – 49** 126](#_Toc195650399)

[ **Program – 50** 126](#_Toc195650400)

[ **Program – 51** 127](#_Toc195650401)

[ **Program – 52** 127](#_Toc195650402)

[ **Program – 53** 127](#_Toc195650403)

[ **Program – 54** 127](#_Toc195650404)

[ **Program – 55** 127](#_Toc195650405)

[ **Program – 56** 127](#_Toc195650406)

[ **Program – 57** 128](#_Toc195650407)

[ **Program – 58** 128](#_Toc195650408)

[ **Program – 59** 128](#_Toc195650409)

[ **Program – 60** 128](#_Toc195650410)

[ **Program – 61** 128](#_Toc195650411)

[ **Program – 62** 128](#_Toc195650412)

[ **Program – 63** 128](#_Toc195650413)

[ **Program – 64** 129](#_Toc195650414)

[ **Program – 65** 129](#_Toc195650415)

[ **Program – 66** 130](#_Toc195650416)

[ **Program – 67** 130](#_Toc195650417)

[ **Program – 68** 130](#_Toc195650418)

[ **Program – 69** 131](#_Toc195650419)

[ **Program – 70** 131](#_Toc195650420)

[ **Program – 71** 131](#_Toc195650421)

[ **Program – 72** 131](#_Toc195650422)

[ **Program – 73** 131](#_Toc195650423)

[ **Program – 74** 131](#_Toc195650424)

[ **Program – 75** 131](#_Toc195650425)

[ **Program – 76 – Person Management System, Write model classes** 131](#_Toc195650426)

[ **Program – 77 -** Flatten a List of Lists using Java 8 Streams 133](#_Toc195650427)

[ **Program – 78 - How to find the second largest element from the list?** 133](#_Toc195650428)

[Advance Java / J2EE 135](#_Toc195650429)

[ **What is servlet?** 135](#_Toc195650430)

[ **Explain servlet lifecycle?** 135](#_Toc195650431)

[ **What is a dispatcher servlet?** 135](#_Toc195650432)

[ **What is JSP?** 135](#_Toc195650433)

[ **What is the difference between servlet and JSP?** 135](#_Toc195650434)

[ **What is the JSP lifecycle?** 135](#_Toc195650435)

[ **What is Spring MVC? Explain architecture of spring MVC?** 135](#_Toc195650436)

[ **What is the singleton design pattern?** 135](#_Toc195650437)

[ **What is the REST API?** 135](#_Toc195650438)

[ **Difference between API and REST?** 135](#_Toc195650439)

[ **Difference between SOAP and REST?** 135](#_Toc195650440)

[ **Which among SOAP and REST is easy?** 135](#_Toc195650441)

[ **Explain REST architecture?** 135](#_Toc195650442)

[ **What do you know about DTD and XSD?** 135](#_Toc195650443)

[ **Explain SOAP Web Service?** 135](#_Toc195650444)

[ **What is Path in XML?** 135](#_Toc195650445)

[ **Write a path for a given xml file?** 135](#_Toc195650446)

[ **What is the use of XSLT?** 135](#_Toc195650447)

[ **How does SOAP work?** 135](#_Toc195650448)

[ **What are the action types of http?** 136](#_Toc195650449)

[ **Explain how you will create a SOAP web service?** 136](#_Toc195650450)

[Data Structure 137](#_Toc195650451)

[ **Find the middle element from the singly linked list without finding the length?** 137](#_Toc195650452)

[ **Complexity to find length of singly linked list?** 137](#_Toc195650453)

[ **Do you know how LinkedList works? Can you create a data structure for LinkedList?** 137](#_Toc195650454)

[ **How will you traverse LinkedList? Your List will contains multiple nodes so how will you traverse over it? Can you write a pseudo code? You have your head with you so how will you move to the next node? How will the current node be null?** 137](#_Toc195650455)

[ **Explain Merge Sort??** 137](#_Toc195650456)

[Design Patterns 138](#_Toc195650457)

[ **What is the singleton design pattern? Explain it?** 138](#_Toc195650458)

[ **Explain Factory design pattern?** 138](#_Toc195650459)

[ **What is the Observer design pattern?** 138](#_Toc195650460)

[ **Except for the creational design pattern, can you explain other design patterns?** 138](#_Toc195650461)

[ **Explain Saga design pattern?** 138](#_Toc195650462)

[Spring 139](#_Toc195650463)

[ **Explain Spring MVC architecture?** 139](#_Toc195650464)

[ **Explain Spring IOC?** 139](#_Toc195650465)

[ **Explain Spring Dependency Injection?** 139](#_Toc195650466)

[ **What are the spring bean scopes?** 139](#_Toc195650467)

[ **Explain any scope of spring bean?** 139](#_Toc195650468)

[ **How to add scope in bean configuration?** 139](#_Toc195650469)

[ **Difference between session and prototype scope?** 139](#_Toc195650470)

[ **What is the Spring batch? Explain where you have used the spring batch in your project?** 139](#_Toc195650471)

[ **How spring batch works?** 139](#_Toc195650472)

[ **How to implement the spring batch?** 139](#_Toc195650473)

[ **What is the difference between SOAP and REST? Which is more efficient?** 139](#_Toc195650474)

[ **How REST API works?** 139](#_Toc195650475)

[ **Difference between Spring and Spring Boot?** 139](#_Toc195650476)

[ **Difference between application context and beanfactory in spring?** 139](#_Toc195650477)

[ **Explain Spring MVC architecture?** 139](#_Toc195650478)

[ **Difference between Spring and Spring Boot?** 139](#_Toc195650479)

[ **What is the use of @Transactional annotation in spring?** 139](#_Toc195650480)

[ **What are different stereotypes annotations in spring?** 139](#_Toc195650481)

[ **What is the front controller?** 139](#_Toc195650482)

[ **Difference between applicationcontext and beanfactory?** 140](#_Toc195650483)

[ **Explain spring bean life cycle?** 140](#_Toc195650484)

[ **If I have prototype bean and singleton bean, then how can we call prototype beans inside singleton bean?** 140](#_Toc195650485)

[ **What are containers present in the spring?** 140](#_Toc195650486)

[ **How transactions work in spring?** 140](#_Toc195650487)

[ **Do you know the web.xml file? What is the significance of a web.xml file?** 140](#_Toc195650488)

[ **What is the dependency injection?** 140](#_Toc195650489)

[ **What are the different ways of injection?** 140](#_Toc195650490)

[ **Who will take care of objects in Spring?** 140](#_Toc195650491)

[ **What is ApplicationContext and its types in Spring?** 140](#_Toc195650492)

[ **Difference between Filter and Interceptor in Spring MVC?** 140](#_Toc195650493)

[ **Which one has access to spring beans, interceptors or filters?** 140](#_Toc195650494)

[ **Why do interceptors have access to Spring beans?** 140](#_Toc195650495)

[ **Difference between Spring data JPA, Hibernate, JDBC, My batis?** 140](#_Toc195650496)

[ **Why do we use Spring data JPA?** 140](#_Toc195650497)

[Spring Security 141](#_Toc195650498)

[ **What is Spring Security?** 141](#_Toc195650499)

[ **How does Spring Security work internally?** 141](#_Toc195650500)

[ **Difference between Authentication and Authorization?** 141](#_Toc195650501)

[ **What is the role of UserDetailsService in Spring Security?** 142](#_Toc195650502)

[ **How do you implement Role-Based Authorization?** 142](#_Toc195650503)

[ **What is CSRF and how does Spring Security protect against it?** 143](#_Toc195650504)

[ **What is the use of SecurityContextHolder?** 143](#_Toc195650505)

[ **How is password encryption handled in Spring Security?** 143](#_Toc195650506)

[ **What is the difference between hasRole() and hasAuthority()?** 143](#_Toc195650507)

[ **How do you secure REST APIs using Spring Security?** 143](#_Toc195650508)

[ **How to implement JWT in Spring Security?** 143](#_Toc195650509)

[ **What is the difference between stateless and stateful authentication?** 144](#_Toc195650510)

[ **What is OncePerRequestFilter?** 144](#_Toc195650511)

[ **How to exclude certain endpoints (like /login, /register) from Spring Security?** 144](#_Toc195650512)

[ **What is OAuth2 and how is it supported in Spring Security?** 144](#_Toc195650513)

[Hibernate 145](#_Toc195650514)

[ How data is persisted in database by using hibernate**?** 145](#_Toc195650515)

[ **How to retrieve select query using Hibernate?** 147](#_Toc195650516)

[ **How to retrieve data through hibernate?** 147](#_Toc195650517)

[ **Hibernate caching?** 147](#_Toc195650518)

[ **Explain L1 and L2 cache in Hibernate?** 147](#_Toc195650519)

[ **Explain the hibernate sessions?** 147](#_Toc195650520)

[ **Difference between JDBC and Hibernate?** 147](#_Toc195650521)

[ **What is the difference between get and load method in hibernate?** 147](#_Toc195650522)

[ **Explain bean scopes?** 147](#_Toc195650523)

[ **Can we have beans without bean id?** 147](#_Toc195650524)

[ **Difference between get and load method?** 147](#_Toc195650525)

[ **Explain Performance tuning and indexing in hibernate?** 147](#_Toc195650526)

[ **Explain different classes and annotations in hibernate?** 148](#_Toc195650527)

[ **How to call stored procedures using hibernate?** 148](#_Toc195650528)

[ **What is Many to many associations in hibernate?** 148](#_Toc195650529)

[ **What is JPA?** 148](#_Toc195650530)

[ **What is the many to many associations in hibernate?** 148](#_Toc195650531)

[ **What is a Session factory in hibernate?** 148](#_Toc195650532)

[ **Explain Many to many associations in hibernate?** 148](#_Toc195650533)

[ **If I want to retrieve a few entries from an employee table, how to retrieve employee details through JPA?** 148](#_Toc195650534)

[ **Explain a built-in function in JPA?** 148](#_Toc195650535)

[ **What is lazy loading in hibernate?** 148](#_Toc195650536)

[ **What is circular dependency?** 148](#_Toc195650537)

[ **What is the second level cache in hibernate?** 148](#_Toc195650538)

[ **Can we use Hibernate without JPA?** 148](#_Toc195650539)

[ **What are the advantages of ORM?** 148](#_Toc195650540)

[Spring Boot 149](#_Toc195650541)

[ **Difference between Spring and Spring Boot?** 149](#_Toc195650542)

[ **What are the advantages of SprigBoot?** 149](#_Toc195650543)

[ **What is a yaml file in SpringBoot?** 149](#_Toc195650544)

[ **What is what is @transient in spring boot?** 149](#_Toc195650545)

[ **What are the types of propagation?** 149](#_Toc195650546)

[ **What @Qualifier in spring? What is the use of @Qualifier?** 149](#_Toc195650547)

[ **What are the advantages of Spring boot?** 149](#_Toc195650548)

[ **How to create multiple modules using spring boot?** 149](#_Toc195650549)

[ **What are all methods present in the controller?** 149](#_Toc195650550)

[ **Explain annotations in Spring boot?** 149](#_Toc195650551)

[ **What is the query parameter?** 149](#_Toc195650552)

[ **How to change the server in Spring boot?** 149](#_Toc195650553)

[ **Difference between @Controller and @RestController?** 149](#_Toc195650554)

[ **What is the use of @Responsebody annotation?** 149](#_Toc195650555)

[ **Explain @RestController annotation in Spring boot?** 149](#_Toc195650556)

[ **Which two annotations are used in built by @RestController?** 149](#_Toc195650557)

[ **How to access application.properties properties in java code?** 149](#_Toc195650558)

[ **Explain Spring boot application? How does it work?** 149](#_Toc195650559)

[ **What is the use of @Service annotation in spring?** 149](#_Toc195650560)

[ **What is the use of @Component annotation in spring?** 149](#_Toc195650561)

[ **What is the use of @Repository annotation in spring?** 150](#_Toc195650562)

[ **What is the difference between @Service and @Controller annotation?** 150](#_Toc195650563)

[ **What is @ComponentScan?** 150](#_Toc195650564)

[ **What is the use of @Autowired annotation?** 150](#_Toc195650565)

[ **Write a REST API to accept user name as input, return the response with message as "Hello, user" and status as "success" and Endpoint - /api/message?** 150](#_Toc195650566)

[ **How to handle Exceptions in Spring boot?** 150](#_Toc195650567)

[ **How to provide security to spring boot applications?** 150](#_Toc195650568)

[ **What is @ControllerAdvice in Spring boot?** 150](#_Toc195650569)

[ **How to retrieve query parameters in Spring boot?** 150](#_Toc195650570)

[ **How is the data persisted in DB with the REST API?** 150](#_Toc195650571)

[ **Explain GET and POST methods?** 150](#_Toc195650572)

[ **Write a SQL or JPQL to retrieve the data from DB?** 150](#_Toc195650573)

[ **How will you add Employee object data in the DB using POST call?** 150](#_Toc195650574)

[ **What is the size of Spring in your project? Who will decide that?** 150](#_Toc195650575)

[ **Explain about spring security?** 150](#_Toc195650576)

[ **Explain the flow of REST api?** 150](#_Toc195650577)

[ **What will happen when we replace @Service instead of @Repository?** 150](#_Toc195650578)

[ **Explain uses of SpringBoot application?** 150](#_Toc195650579)

[ **How is https used in SpringBoot applications?** 150](#_Toc195650580)

[ **Difference between RequestMapping and GetMapping?** 150](#_Toc195650581)

[ **If you have to create a Spring boot application? What steps will you follow?** 151](#_Toc195650582)

[ **What is a starter kit in spring boot application?** 151](#_Toc195650583)

[ **What are the starters in Spring boot application?** 151](#_Toc195650584)

[ **What are Actuators in spring boot applications?** 151](#_Toc195650585)

[ **What are custom exceptions? How do you define your own custom exception?** 151](#_Toc195650586)

[ **Have you worked on Unit testing? Which framework have you used in your project?** 151](#_Toc195650587)

[ **What is the difference between lazy loading and eager loading?** 151](#_Toc195650588)

[ **Difference between put and patch?** 151](#_Toc195650589)

[ **Can we use @RequestMapping at class level and method level both?** 151](#_Toc195650590)

[ **What will happen if I use @RequestMapping at method level?** 151](#_Toc195650591)

[ **How to map Service class to RestController?** 151](#_Toc195650592)

[ **What is Spring security? Have you used Spring security in your project?** 151](#_Toc195650593)

[ **What are Profiles in Spring boot?** 151](#_Toc195650594)

[ **How will you do CRUD operations using spring boot applications?** 151](#_Toc195650595)

[ **How to change from JAR to WAR in Spring boot?** 151](#_Toc195650596)

[ **Where will you give database details in the spring boot application?** 151](#_Toc195650597)

[ **How will you give database details for the individual environment?** 151](#_Toc195650598)

[ **What is a JPA repository?** 151](#_Toc195650599)

[ **What is the use of @SpringBootApplication annotation in Spring boot?** 151](#_Toc195650600)

[ **How will the Spring boot application know whether its JAR or WAR type?** 151](#_Toc195650601)

[ **How are you providing security to your services?** 151](#_Toc195650602)

[ **As an end user how can I consume your services?** 152](#_Toc195650603)

[ **How to validate a token?** 152](#_Toc195650604)

[ **Explain a few security algorithms?** 152](#_Toc195650605)

[ **What is native queries in Spring boot?** 152](#_Toc195650606)

[ **Where do you use native queries?** 152](#_Toc195650607)

[ **Suppose I am having different microservices, if I want to invoke different microservice so how can we interact?** 152](#_Toc195650608)

[ **What kind of information are you passing to the GET method?** 152](#_Toc195650609)

[ **Suppose I want to start a spring boot application, so what are all the steps?** 152](#_Toc195650610)

[ **How to configure a custom server in a Spring boot application?** 152](#_Toc195650611)

[ **How to configure multiple databases in a Spring boot application?** 152](#_Toc195650612)

[ **How does your application know I am using a particular database?** 152](#_Toc195650613)

[ **What is the use of @Qalifier annotation?** 152](#_Toc195650614)

[ **How are REST calls intercepted in our project? How is the call happening?** 152](#_Toc195650615)

[ **When I hit a URL how data will be invoked with your application?** 152](#_Toc195650616)

[ **How is the DTO getting invoked with REST endpoints?** 152](#_Toc195650617)

[ **I have a URL and I hit the URL, so which is the first class which is invoked in our product?** 152](#_Toc195650618)

[ **If you have to expose a new endpoint, what steps will you follow?** 152](#_Toc195650619)

[ **What is the difference between @Bean and @Configuration?** 152](#_Toc195650620)

[ **What is the difference between RequestParam and path variable? Give me one example with one URL?** 152](#_Toc195650621)

[ **How to change the port of the embedded tomcat server in Spring boot? Write a syntax for that?** 153](#_Toc195650622)

[ **What are Profiles in Spring boot applications? Give one example?** 153](#_Toc195650623)

[ **Which annotations are we using in order to use the Exception?** 153](#_Toc195650624)

[ **What is the default scope for spring beans?** 153](#_Toc195650625)

[ **Which database have you used? How do you integrate databases with JPA?** 153](#_Toc195650626)

[ **How to communicate between two spring boot applications using asynchronous calls?** 153](#_Toc195650627)

[ **Have you worked messaging services?** 153](#_Toc195650628)

[ **How will you handle errors / exceptions in your project?** 153](#_Toc195650629)

[ **What are Actuators?** 153](#_Toc195650630)

[ **How are you keeping track of logs in your project?** 153](#_Toc195650631)

[ **If you have millions of data then how will you manage it?** 153](#_Toc195650632)

[ **What is the difference between REST API and Http?** 153](#_Toc195650633)

[ **Explain @Autowired usage?** 153](#_Toc195650634)

[ **How does the REST API work?** 153](#_Toc195650635)

[ **How can we handle a yaml file when it is exposed?** 153](#_Toc195650636)

[ **How to externalize variables in spring boot?** 153](#_Toc195650637)

[ **What is the Spring reactive framework?** 153](#_Toc195650638)

[ **Why should we use Spring reactive?** 153](#_Toc195650639)

[ **What is the use of block and subscribe?** 153](#_Toc195650640)

[ **What is a novel based server?** 153](#_Toc195650641)

[ **How will the findById or findByName method in the repository work with name? When we go with camelcase then where is it implemented internally?** 154](#_Toc195650642)

[ **How does the JPA Repository know there is a class?** 154](#_Toc195650643)

[ **How does the Spring Boot application work internally?** 154](#_Toc195650644)

[ **How will you authenticate API in Spring Boot?** 154](#_Toc195650645)

[ **In the Spring REST API what is the White label error accessing from the browser? What is the status code you will get?** 154](#_Toc195650646)

[Microservices 155](#_Toc195650647)

[ **Monolith vs Microservices differences and when to choose which?** 155](#_Toc195650648)

[ **How to design a microservice from scratch?** 156](#_Toc195650649)

[ **API Gateway pattern and its advantages?** 157](#_Toc195650650)

[ **Inter-service communication: REST vs Messaging?** 159](#_Toc195650651)

[ **Circuit Breaker pattern and its implementation using Resilience4j?** 160](#_Toc195650652)

[ **Load balancing in microservices using Spring Cloud Load Balancer?** 162](#_Toc195650653)

[ **How Spring Cloud Config helps in centralized configuration management?** 163](#_Toc195650654)

[ **Service discovery using Eureka or Consul?** 165](#_Toc195650655)

[ **Feign Client vs WebClient: Which one to use and why?** 165](#_Toc195650656)

[ **Event-driven architecture and Kafka integration?** 165](#_Toc195650657)

[ **Database per service vs Shared Database: Pros and cons?** 165](#_Toc195650658)

[ **Saga Pattern for distributed transactions in microservices?** 165](#_Toc195650659)

[ **JWT-based authentication and OAuth2 in microservices?** 165](#_Toc195650660)

[ **How to handle security in an API Gateway?** 165](#_Toc195650661)

[ **Observability: Logging, tracing, and monitoring best practices?** 165](#_Toc195650662)

[ **Role of Prometheus and Grafana in microservices monitoring?** 165](#_Toc195650663)

[ **Kubernetes deployment strategies for microservices?** 166](#_Toc195650664)

[ **Blue-Green and Canary deployments in microservices?** 166](#_Toc195650665)

[ **When to use WebFlux for reactive microservices?** 166](#_Toc195650666)

[ **CQRS and Event Sourcing: When and why to use them?** 166](#_Toc195650667)

[ **Difference between Monolithic and Microservice architecture?** 166](#_Toc195650668)

[ **Explain Microservices architecture?** 166](#_Toc195650669)

[ **Where are you deploying your microservice?** 166](#_Toc195650670)

[ **What are different Aggregation patterns in Microservices?** 166](#_Toc195650671)

[ **What are challenges in Microservice architecture?** 166](#_Toc195650672)

[ **How are you handling load in your microservices?** 166](#_Toc195650673)

[ **How will you handle resilience and fault tolerance in microservice?** 166](#_Toc195650674)

[ **Can you explain Event driven microservice and how you have used it in your project?** 166](#_Toc195650675)

[SQL 167](#_Toc195650676)

[ **Difference between Primary Key and Unique key?** 167](#_Toc195650677)

[ **What is ordered by and range in the database?** 167](#_Toc195650678)

[ **Write a SQL query to get the second highest salary of an employee from the employee table?** 167](#_Toc195650679)

[ **Write a SQL query to print employee id, employee\_name, depatment\_name Employee - employee\_id, employee\_name, department\_id** 167](#_Toc195650680)

[**Department - department\_id, department\_name?** 167](#_Toc195650681)

[ **SQL query to find department name and number of employees in that department?** 167](#_Toc195650682)

[ **What is a collision in the oracle database?** 167](#_Toc195650683)

[ **If you create multiple folders one inside another and store one file into the last folder then how can you implement using database tables in oracle? How many tables are required?** 167](#_Toc195650684)

[ **What are DDL and DML commands?** 167](#_Toc195650685)

[ **Write a query to update data in the table?** 167](#_Toc195650686)

[ **What is the use of joins in the database?** 167](#_Toc195650687)

[ **Explain ACID properties in the database?** 167](#_Toc195650688)

[ **Explain Normalization?** 167](#_Toc195650689)

[ **What are the types of Normalization?** 167](#_Toc195650690)

[ **What is Denormalization?** 168](#_Toc195650691)

[ **What is the use of join?** 168](#_Toc195650692)

[ **Write a SQL query to return the 7th highest salary of an employee?** 168](#_Toc195650693)

[ **What is Union in SQL?** 168](#_Toc195650694)

[ **Can we perform union operations on one table?** 168](#_Toc195650695)

[ **Difference between where and let?** 168](#_Toc195650696)

[ **What is Semaphore?** 168](#_Toc195650697)

[ **What is View and Index in SQL?** 168](#_Toc195650698)

[ **What is Trigger in SQL? What is the use of Trigger in SQL?** 168](#_Toc195650699)

[ **What is inner join in SQL?** 168](#_Toc195650700)

[ **Write a SQL query to get emp\_sal > 5000 and company\_name = "Kotak"** 168](#_Toc195650701)

[ **Write a JPA query to retrieve the above result?** 168](#_Toc195650702)

[ **What is an index in SQL?** 168](#_Toc195650703)

[ **Explain different types of Index?** 168](#_Toc195650704)

[ **Explain different types of joins?** 168](#_Toc195650705)

[ **Difference between Outer join and Full outer join?** 169](#_Toc195650706)

[ **Write a SQL query to find the name of a city which has more than one customer in it?** 169](#_Toc195650707)

[ **Write SQL query to display 10th highest salary of employee?** 169](#_Toc195650708)

[ **What is the difference between DDL and DML commands?** 169](#_Toc195650709)

[ **Difference between Inner join and Outer join?** 169](#_Toc195650710)

[ **Difference between left outer join and right outer join?** 169](#_Toc195650711)

[ **What are Triggers?** 169](#_Toc195650712)

[ **SQL query to get the youngest employee from the employee table?** 169](#_Toc195650713)

[ **SQL query to get list of employees from Finance department?** 169](#_Toc195650714)

[ **What is Indexing in SQL?** 169](#_Toc195650715)

[ **What are the disadvantages of Index?** 169](#_Toc195650716)

[ **What is the difference between primary key and unique key?** 169](#_Toc195650717)

[ **What is the difference between implicit cursor and explicit cursor?** 169](#_Toc195650718)

[ **What is a correlated subquery?** 169](#_Toc195650719)

[ **What is the left outer join?** 170](#_Toc195650720)

[ **What is the use of stored procedure?** 170](#_Toc195650721)

[ **Explain the database schema for Students and Course table?** 170](#_Toc195650722)

[ **What is UNION ALL?** 170](#_Toc195650723)

[ **SQL query to get second highest salary?** 170](#_Toc195650724)

[ **Write a SQL query to display a list of students which have a Mathematics subject?** 170](#_Toc195650725)

[ **What is Normalization?** 170](#_Toc195650726)

[ **Can you explain different forms of Normalization with one example?** 170](#_Toc195650727)

[ **What kind of fetching strategy are you using in hibernate?** 170](#_Toc195650728)

[ **What is OneToMany mapping?** 170](#_Toc195650729)

[ **Write a SQL query to get the 3rd highest salary of an employee?** 170](#_Toc195650730)

[ **What is dense and rank in SQL?** 170](#_Toc195650731)

[ **What is the difference between SQL and NoSQL?** 170](#_Toc195650732)

[ **Difference between Stored Procedure and Function?** 170](#_Toc195650733)

[ **What is Transaction in SQL?** 170](#_Toc195650734)

[ **Write a SQL query:** 170](#_Toc195650735)

[ **SQL query that categorizes employees based on their salary into different percentage ranges.** 171](#_Toc195650736)

[ **Write a SQL query to find the department name who does not have an employee? (Employee and Department table given)?** 171](#_Toc195650737)

[ **What is the difference between Primary Key and Unique Key?** 171](#_Toc195650738)

[ **What is the Composite key?** 171](#_Toc195650739)

[ **What is Self join?** 171](#_Toc195650740)

[ **We have Table A (2 columns and 4 rows) and Table B (2 columns and 4 rows) and I want to print all 8 rows? Write a query for it? (Use Union All)?** 171](#_Toc195650741)

[ **What is the difference between SQL Union and Union All?** 171](#_Toc195650742)

[ **Write a SQL query and print the employee name which does not have any department?** 172](#_Toc195650743)

[ **Write a query - Employee table and add one extra column in output as category? Let's say if salary is in 1 to 4000 then Category 1, 4000 - 7000 then category 2, 7000 - 10000 then category 3?** 172](#_Toc195650744)

[Kafka 173](#_Toc195650745)

[ **What is Apache Kafka?** 173](#_Toc195650746)

[ **Why does Kafka compare to RabbitMQ?** 173](#_Toc195650747)

[ **How to scale up Kafka?** 173](#_Toc195650748)

[ **What are the default partitions in kafka?** 173](#_Toc195650749)

[ **Difference between topic and partition in Kafka?** 173](#_Toc195650750)

[ **Is there any ordering of messages in partitions in kafka?** 173](#_Toc195650751)

[ **How does partition assignment happen? What method is used?** 173](#_Toc195650752)

[ **What is offset and partition in Kafka?** 173](#_Toc195650753)

[GitHub 174](#_Toc195650754)

[ **What is Version control i.e. Git?** 174](#_Toc195650755)

[ **How to push and commit code in git using commands?** 174](#_Toc195650756)

[ **Difference between rebase and merge?** 174](#_Toc195650757)

[ **Which versioning tool are you using? Explain a few Git commands?** 174](#_Toc195650758)

[ **What is the Git command to create a new branch from an existing branch?** 174](#_Toc195650759)

[ **If we have more consumer groups than partitions and how will it manage?** 174](#_Toc195650760)

[Docker 175](#_Toc195650761)

[ **Explain use cases of Docker?** 175](#_Toc195650762)

[ **You have 2 containers 1 and 2, can they communicate with each other?** 175](#_Toc195650763)

[ **If both are with different hosts then whether they will be able to communicate with each other or not?** 175](#_Toc195650764)

[ **What is the host in this case?** 175](#_Toc195650765)

[ **Imagine you have one volume and you want to share it with multiple containers? How will you do it?** 175](#_Toc195650766)

[ **Where does an object get stored when created inside a docker container?** 175](#_Toc195650767)

[ **What is the process of Docker?** 175](#_Toc195650768)

[ **How is Docker helpful?** 175](#_Toc195650769)

[ **What is the use of Docker?** 175](#_Toc195650770)

[Kubernetes 176](#_Toc195650771)

[ **What are Kubernetes Secrets??** 176](#_Toc195650772)

[AWS 177](#_Toc195650773)

[ **Explain AWS services?** 177](#_Toc195650774)

[ **What is cloud computing?** 178](#_Toc195650775)

[ **Explain services provided by cloud computing?** 178](#_Toc195650776)

[ **How to deploy service to AWS?** 178](#_Toc195650777)

[ **Which services have you used?** 178](#_Toc195650778)

[ **What are different services in EKS?** 178](#_Toc195650779)

[ **What is the benefit of Serverless services? Is EKS serverless?** 178](#_Toc195650780)

[Azure 179](#_Toc195650781)

[ **question?** 179](#_Toc195650782)

[Angular 180](#_Toc195650783)

[ **What are the components in the Angular?** 180](#_Toc195650784)

[ **How to pass data from components in Angular?** 180](#_Toc195650785)

[ **What does the component.ts file contain in angular?** 180](#_Toc195650786)

[ **Explain two way data binding in angular?** 180](#_Toc195650787)

[ **How to make backend calls in angular?** 180](#_Toc195650788)

[ **What is authguard in angular?** 180](#_Toc195650789)

[ **What is interpolation in angular?** 180](#_Toc195650790)

[ **How to create components in angular?** 180](#_Toc195650791)

[ **Explain angular directives?** 180](#_Toc195650792)

[ **What are observables in angular?** 180](#_Toc195650793)

[ **How to implement dependency injection in angular?** 180](#_Toc195650794)

[ **What are different components of Angular?** 180](#_Toc195650795)

[ **How to handle caching in Angular?** 180](#_Toc195650796)

[React 181](#_Toc195650797)

[ **Features of ReactJS?** 181](#_Toc195650798)

[ **How does Virtual DOM work?** 181](#_Toc195650799)

[ **Features of Virtual DOM?** 181](#_Toc195650800)

[ **What is reducing modifications in Virtual DOM?** 181](#_Toc195650801)

[ **What do you call the backend API from ReactJS?** 181](#_Toc195650802)

[ **What are other Hooks in React?** 181](#_Toc195650803)

[ **What is the purpose of Hook?** 181](#_Toc195650804)

[ **What is the dependency array in useEffect?** 181](#_Toc195650805)

[ **What is the use and significance of dependency arrays?** 181](#_Toc195650806)

[ **How to make a call to an external API from ReactJS?** 181](#_Toc195650807)

[ **What is an alternative to Axios?** 181](#_Toc195650808)

[Others 182](#_Toc195650809)

[ **Explain Jira?** 182](#_Toc195650810)

[ **What is the output of below program?** 182](#_Toc195650811)

[ **What is https protocol?** 182](#_Toc195650812)

[ **How does https internally work?** 182](#_Toc195650813)

[ **What is a certificate?** 182](#_Toc195650814)

[ **Difference between http and https?** 182](#_Toc195650815)

[ **What is a Sprint retrospective?** 182](#_Toc195650816)

[ **Do you know anything about CI/CD?** 182](#_Toc195650817)

[ **How will Jenkins' job work? Explain the process?** 182](#_Toc195650818)

[ **What is Node JS?** 182](#_Toc195650819)

[ **How is your application getting deployed? Which platform are you using in your project?** 182](#_Toc195650820)

[ **What do you know about Asynchronous communication?** 182](#_Toc195650821)

[ **What are the measure objects i.e key things in order to make Asynchronous calls?** 182](#_Toc195650822)

[ **If I have two different technologies then how can we achieve asynchronous communication?** 182](#_Toc195650823)

[ **Do you have exposure in AI and ML?** 182](#_Toc195650824)

[ **Examples of Fund Transfer are Synchronous or Asynchronous?** 182](#_Toc195650825)

[ **Is WhatsApp Synchronous or Asynchronous?** 183](#_Toc195650826)

[ **Difference between 2 Tier and 3 Tier Architecture?** 183](#_Toc195650827)

[ **Have you heard about the log4j vulnerability?** 183](#_Toc195650828)

[ **What is https protocol?** 183](#_Toc195650829)

[ **Difference between http and https?** 183](#_Toc195650830)

[ **Suppose in server one job at 9 AM and copy json file and paste it in temp folder another job 6 PM read it and load in db and no one should change the file. How to prevent file from any Change?** 183](#_Toc195650831)

[ **If we increase timeout then how can we resolve it? Deployment related issue and how will you handle it?** 183](#_Toc195650832)

[Linux 184](#_Toc195650833)

[ **Explain Linux commands?** 184](#_Toc195650834)

[ **Command - Read a file and print first two lines from file Ans. head -2 filename?** 184](#_Toc195650835)

[ **Search a word from a file?** 184](#_Toc195650836)

[ **Why do you want to change your current organization?** 184](#_Toc195650837)

[Puzzles 185](#_Toc195650838)

[ **Puzzle – 1** 185](#_Toc195650839)

[ **Puzzle – 2** 185](#_Toc195650840)

[ **Puzzle – 3** 185](#_Toc195650841)

# ***Java***

## **Java Fundamentals**

### **Explain how to install java? Explain directories in it?**

**Step 1: Download Java**

Go to the official Oracle website or use OpenJDK: Oracle JDK / OpenJDK

**Step 2: Choose Version**

Choose the appropriate JDK version (e.g., Java 17 or Java 21 – latest LTS versions).

Select the installer for your operating system (Windows, macOS, Linux).

**Step 3: Install Windows:**

Run the .exe installer and follow the wizard.

It installs Java to a default path like: C:\Program Files\Java\jdk-21

macOS/Linux:

Use .tar.gz or package managers like Homebrew, apt, yum.

sudo apt install openjdk-21-jdk # For Ubuntu

**Step 4: Set Environment Variables (Windows)**

Go to System Properties > Environment Variables.

Add to Path: C:\Program Files\Java\jdk-21\bin

Set JAVA\_HOME: JAVA\_HOME = C:\Program Files\Java\jdk-21

**Step 5: Verify Installation**

Open a terminal/command prompt and run:

java -version

javac -version



### **How to configure java in windows?**

✅ **Step 1: Download Java JDK**

Go to the official Oracle website or OpenJDK site:

* [Oracle JDK](https://www.oracle.com/java/technologies/javase-downloads.html)
* [OpenJDK (Adoptium)](https://adoptium.net/)

Download the installer for Windows x64.

**✅ Step 2: Install Java**

Run the installer (.msi or .exe)

Choose installation path (e.g., C:\Program Files\Java\jdk-21)

Complete the installation

✅ **Step 3: Set Environment Variables**

**🌐 1. Set JAVA\_HOME**

Open System Properties  
Right click on This PC → Properties → Advanced system settings → Environment Variables

Under System Variables, click New:

Variable Name: JAVA\_HOME

Variable Value: C:\Program Files\Java\jdk-21

🌐 **2. Add to Path variable**

In the same "Environment Variables" window:

Find Path under System Variables

Click Edit → New and add:

%JAVA\_HOME%\bin

✅ **Step 4: Verify Java Installation**

Open Command Prompt and run:

java -version

Example output:

java version "21.0.1" 2023-10-17 LTS

Java(TM) SE Runtime Environment ...

Also try:

javac -version

To verify the Java compiler (javac) is set correctly.

### **Explain Java Program execution?**

* **1. Writing the Source Code**
  + You create a .java file containing your Java code (e.g., HelloWorld.java)
  + Example:

public class HelloWorld {

public static void main(String[] args) {

System.out.println("Hello, World!");

}

}

**2. Compilation (javac)**

* + The Java compiler (javac) converts your source code into **bytecode** (platform-independent intermediate code)
  + Creates .class files (e.g., HelloWorld.class)
  + Bytecode is not machine code - it's instructions for the **Java Virtual Machine (JVM)**

**3. Class Loading**

When you run java HelloWorld:

**a) Bootstrap ClassLoader**

* Loads core Java classes (from rt.jar and other core libraries)
* Written in native code (not Java)

**b) Extension ClassLoader**

* Loads classes from Java extension directories

**c) Application ClassLoader**

* Loads your application classes (from the classpath)

**4. Bytecode Verification**

* JVM verifies the bytecode to ensure:
  + No illegal memory access
  + Proper stack manipulation
  + Correct method calls
  + No violation of access restrictions
* This provides security by preventing malicious code

**5. Just-In-Time (JIT) Compilation**

* The JVM's JIT compiler converts frequently executed bytecode into **native machine code**
* Happens at runtime
* Optimizes performance by:
  + Inlining methods
  + Removing dead code
  + Optimizing loops

**6. Execution**

* The JVM executes the program:
  + Creates the main thread
  + Allocates memory for objects in the Heap
  + Manages method calls using the Stack
  + Handles garbage collection automatically

**7. Runtime Memory Areas**

The JVM manages these memory areas during execution:

a) Method Area

* Stores class structures, method code, and static variables

b) Heap

* Stores all objects and their instance variables
* Garbage collection works here

c) JVM Stacks

* Each thread has its own stack
* Stores frames for each method call (local variables, operands, return values)

d) PC Registers

* Tracks the execution position for each thread

e) Native Method Stacks

* For native code (non-Java code)

**8. Garbage Collection**

* Automatically reclaims memory from objects no longer in use
* Runs in the background
* Different algorithms available (Serial, Parallel, G1, ZGC, etc.)



### **Explain the byte stream in java?**

* The Byte Stream in Java is used to perform input and output of 8-bit bytes. It is mainly used to read and write binary data like images, videos, audio files, PDF, etc.

🧱 **Byte Stream Classes (Hierarchy)**

All byte stream classes are derived from these two abstract classes:

| **Stream Type** | **Abstract Class** | **Direction** |
| --- | --- | --- |
| Input Stream | InputStream | Reading |
| Output Stream | OutputStream | Writing |

**📥 Common Byte InputStream Classes**

| **Class** | **Description** |
| --- | --- |
| FileInputStream | Reads data from a file |
| BufferedInputStream | Reads data efficiently using a buffer |
| ByteArrayInputStream | Reads data from a byte array |
| ObjectInputStream | Reads Java objects from a stream (used in serialization) |

**📤 Common Byte OutputStream Classes**

| **Class** | **Description** |
| --- | --- |
| FileOutputStream | Writes data to a file |
| BufferedOutputStream | Writes data efficiently using a buffer |
| ByteArrayOutputStream | Writes to a byte array |
| ObjectOutputStream | Writes Java objects to a stream |

✅ **Simple Example: Reading a file using Byte Stream**

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

public class ByteStreamExample {

public static void main(String[] args) {

try (

FileInputStream in = new FileInputStream("input.txt");

FileOutputStream out = new FileOutputStream("output.txt");

) {

int byteData;

while ((byteData = in.read()) != -1) {

out.write(byteData);

}

System.out.println("File copied successfully using byte stream.");

} catch (IOException e) {

e.printStackTrace();

}

}

}

🎯 **When to Use Byte Stream?**

* When working with non-textual data (images, videos, etc.)
* When you need to process data byte by byte
* When dealing with platform-independent binary formats

📌 **Byte Stream vs Character Stream**

| **Feature** | **Byte Stream (InputStream/OutputStream)** | **Character Stream (Reader/Writer)** |
| --- | --- | --- |
| Data Type | 8-bit bytes (binary) | 16-bit characters (text) |
| Use Case | Images, videos, PDFs | Text files, character data |
| Base Classes | InputStream, OutputStream | Reader, Writer |

### **What is Classloader? Different types of it?**

* The **ClassLoader** is a part of the **Java Runtime Environment (JRE)** responsible for **loading .class files** into memory **at runtime**.

Java doesn't load all classes at once — it loads them **on demand**, and the ClassLoader is what performs that task.

**🔁 Class Loading Process (High-Level)**

1. **Load** – The class file is located and loaded into memory.
2. **Link** – It verifies, prepares, and optionally resolves the class.
3. **Initialize** – Static blocks and static variables are initialized.

**🧱 Types of ClassLoaders in Java**

Java has a **hierarchical delegation model**, and these are the key built-in classloaders:

| **ClassLoader** | **Role** |
| --- | --- |
| 🔹 **Bootstrap ClassLoader** | Loads **core Java classes** from the rt.jar (like java.lang.\*) |
| 🔹 **Extension ClassLoader** | Loads classes from the ext directory ($JAVA\_HOME/lib/ext) |
| 🔹 **System/Application ClassLoader** | Loads classes from the **classpath** (e.g., your app code) |

**➕ Custom ClassLoaders**

You can create your own ClassLoader by extending ClassLoader to load classes dynamically (e.g., plugin systems, frameworks like OSGi).

**🔄 Delegation Hierarchy**

When a class is requested to be loaded, the process follows this **parent-first delegation model**:

App ClassLoader → Ext ClassLoader → Bootstrap ClassLoader

Each loader **delegates to its parent** before trying to load the class itself.

**✅ Example: Get the ClassLoader of a Class**

public class Demo {

public static void main(String[] args) {

ClassLoader cl = Demo.class.getClassLoader();

System.out.println("ClassLoader: " + cl);

}

}

**🔍 Real-World Use Cases of ClassLoader**

* **Hot deployment** in application servers
* **Dynamic loading** of plugins/modules
* **Custom loading** of encrypted/obfuscated classes
* Used heavily in frameworks like **Spring**, **Hibernate**, **Tomcat**, etc.

### **Why is java more secure?**

* Java was **designed with security in mind** from the start. Here's how it achieves that:

**1. No Pointer Arithmetic**

* Java **doesn't allow direct memory access** using pointers like C/C++.
* Prevents buffer overflows, memory corruption, and access to arbitrary memory.

**2. Automatic Memory Management (Garbage Collection)**

* Reduces memory leaks and dangling pointer issues.
* JVM handles allocation and deallocation safely.

**3. Bytecode Verification**

* Before execution, **Java bytecode is verified** by the JVM to ensure it:
  + Obeys access restrictions
  + Doesn't overflow/underflow memory
  + Has correct data types
* Prevents malicious or corrupted bytecode from running.

**4. ClassLoader and SecurityManager**

* Java’s **ClassLoader** architecture ensures classes are loaded securely and isolates class loading for apps.
* **SecurityManager** (though deprecated in Java 17+) allows you to define **runtime access control policies**.

**5. Sandboxing**

* Java applets (now obsolete) used to run in a **sandbox**, restricting file access, network, etc.
* Still applies to custom environments like **JVM containers** or plugin systems.

**6. Strong Access Modifiers**

* Keywords like private, protected, final prevent unwanted access or overrides.
* Supports **encapsulation**, keeping internal logic safe from external interference.

**7. Exception Handling**

* Java's exception system helps handle runtime errors gracefully, reducing crashes or undefined behavior.

**8. Built-in Cryptography & SSL APIs**

* Java provides APIs for **secure communication**, **data encryption**, and **message integrity** (like javax.crypto, java.security, etc.).

**9. Runtime Monitoring with JVM Tools**

* Java has profiling, monitoring, and diagnostic tools (e.g., **JFR**, **JConsole**, **VisualVM**) to catch issues early.

**10. Regular Security Patches & Community Support**

* Oracle and the OpenJDK community release **frequent security updates**.
* Vast community support ensures vulnerabilities are spotted and patched quickly.

**🛡️ Summary Table**

| **Security Feature** | **Java's Advantage** |
| --- | --- |
| Memory Safety | No pointers, GC prevents memory attacks |
| Access Control | Strong access modifiers and class isolation |
| Code Verification | Bytecode is checked before execution |
| Runtime Security | SecurityManager, ClassLoader protection |
| Communication Security | SSL, TLS, Encryption libraries available |

### **Explain public static void main (String[] args) method?**

**🔍 Breakdown of each keyword:**

| **Part** | **Meaning** |
| --- | --- |
| public | Access modifier – means this method can be called from anywhere. |
| static | No need to create an object to call this method — JVM calls it directly. |
| void | Return type – it doesn't return any value to the JVM. |
| main | Method name – JVM looks for this specific method as the entry point. |
| String[] args | Command-line arguments – passed to your program when it starts. |

**🧠 Why each part is important?**

**✅ public**

* Must be **public** so that **JVM can access it** from outside the class.

**✅ static**

* JVM doesn't create an object of your class to call main().
* Declared static so it can be called **without creating an instance**.

**✅ void**

* main() doesn’t return anything. It's just the entry point.

**✅ main**

* JVM always looks for this method name to start the program.

**✅ String[] args**

* An array of String values passed during command-line execution.
* **Example:**

java MyApp Hello World

Then args[0] = "Hello", args[1] = "World"

**✅ Complete Example:**

public class HelloWorld {

public static void main(String[] args) {

System.out.println("Hello, Java!");

}

}

**🧪 Fun fact:**

You can actually write it like:

static public void main(String[] args) { }

...because the order of static and public doesn't matter in this context (but the conventional order is public static).

### **What is the marker interface?**

* A **marker interface** is an interface that has **no methods or fields**.  
  Its sole purpose is to **mark** a class with some metadata so that the **JVM or framework** can identify and treat the object differently.

**🔍 Example:**

public interface Serializable {

// No methods inside — it's a marker interface

}

If a class implements Serializable, it **indicates** that objects of that class can be **serialized** (converted into a byte stream).

**✅ Common Marker Interfaces in Java**

| **Interface** | **Purpose** |
| --- | --- |
| Serializable | Marks objects that can be serialized |
| Cloneable | Allows object cloning using Object.clone() |
| Remote | Used in RMI to indicate remote method calling |
| ThreadSafe (custom) | Can be used to tag thread-safe components |

**🧠 How Does It Work?**

Although a marker interface has **no methods**, Java libraries and JVM use **instanceof checks or reflection** to apply behavior.

if (object instanceof Serializable) {

// safe to serialize

}

**🛠️ Custom Marker Interface Example**

public interface Auditable {} // marker interface

public class Transaction implements Auditable {

// business logic

}

A framework might look for Auditable-tagged classes and log changes automatically.

**🆚 Marker Interface vs Annotations**

| **Aspect** | **Marker Interface** | **Annotation** |
| --- | --- | --- |
| Introduced in | Java 1.0 | Java 5 |
| Inheritance | Can use instanceof | Cannot use instanceof |
| Flexibility | Limited | More flexible (can include data) |
| Preferred now? | ❌ Old style | ✅ Recommended (modern approach) |

**✅ Summary**

* Marker interfaces = **empty interfaces** used to **mark classes** for special treatment
* Still used (e.g., Serializable), but **annotations are more powerful and flexible** today

### **What is serialization and deserialization?**

* **📦 Serialization**

**Definition:** Serialization is the process of **converting a Java object into a byte stream**, so it can be stored in a file, sent over a network, or saved in memory.

**Why?** To persist (save) the state of an object or transfer it.

**Java Syntax:**

ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("data.ser"));

out.writeObject(object); // object must be Serializable

out.close();

**📥 Deserialization**

**Definition:**  
Deserialization is the reverse process — **reconstructing a Java object from a byte stream**.

**Java Syntax:**

ObjectInputStream in = new ObjectInputStream(new FileInputStream("data.ser"));

MyObject obj = (MyObject) in.readObject();

in.close();

**✅ Requirements:**

* The class **must implement** java.io.Serializable interface.
* Example:

import java.io.Serializable;

public class Person implements Serializable {

private String name;

private int age;

// constructors, getters, setters

}

**🛠️ Real-World Use Cases:**

* Saving user session data to a file
* Sending objects between systems (e.g., via sockets)
* Caching objects
* Storing objects in databases as BLOBs

**🚫 Things to Watch Out:**

* **transient** keyword skips fields during serialization.

transient String password;

* Version conflicts can happen — use serialVersionUID for version control:

private static final long serialVersionUID = 1L;

**🧪 Simple Example:**

// Serializable class

import java.io.\*;

class Student implements Serializable {

private static final long serialVersionUID = 1L;

String name;

int age;

public Student(String name, int age) {

this.name = name;

this.age = age;

}

}

// Main class

public class SerializationExample {

public static void main(String[] args) throws Exception {

Student s1 = new Student("John", 22);

// Serialize

ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("student.ser"));

out.writeObject(s1);

out.close();

// Deserialize

ObjectInputStream in = new ObjectInputStream(new FileInputStream("student.ser"));

Student s2 = (Student) in.readObject();

in.close();

  System.out.println("Deserialized Student: " + s2.name + ", " + s2.age);

}

}

### **What is the use of the transient keyword in serialization?**

* The **transient** keyword in Java is used to mark a field of a class as "non-serializable." When an object is serialized (i.e., converted into a byte stream for storage or transmission), the transient keyword ensures that the marked field is not included in the serialization process. This is useful when you have fields that should not be persisted, such as sensitive data (e.g., passwords), or fields that can be recalculated and don't need to be saved.

Example:

import java.io.\*;

class Employee implements Serializable {

private String name;

private transient int salary; // This field will not be serialized

public Employee(String name, int salary) {

this.name = name;

this.salary = salary;

}

public String getName() {

return name;

}

public int getSalary() {

return salary;

}

}

 public class TestSerialization {

public static void main(String[] args) throws Exception {

Employee emp = new Employee("John", 50000);

// Serialize the object

FileOutputStream fileOut = new FileOutputStream("employee.ser");

ObjectOutputStream out = new ObjectOutputStream(fileOut);

out.writeObject(emp);

out.close();

fileOut.close();

// Deserialize the object

FileInputStream fileIn = new FileInputStream("employee.ser");

ObjectInputStream in = new ObjectInputStream(fileIn);

Employee deserializedEmp = (Employee) in.readObject();

in.close();

fileIn.close();

// Output the deserialized object's fields

System.out.println("Name: " + deserializedEmp.getName());

System.out.println("Salary: " + deserializedEmp.getSalary()); // Will be 0 since it's transient

}

}

In this example, the **salary** field is marked as **transient**, so when the object is serialized and then deserialized, the **salary** field will not retain its value and will default to **0** in the deserialized object.

### **How can I make below Employee class as Serializable class?**

@Data class Employee {

String name;

String empld;

Date dob;

Address address;

}

@Data

class Address{

String linel;

String line2;

}

import java.io.Serializable;

import java.util.Date;

@Data

class Employee implements Serializable {

private static final long serialVersionUID = 1L; // Optional, but good practice to define

private String name;

private String empld;

private Date dob;

private Address address;

}

@Data

class Address implements Serializable {

private static final long serialVersionUID = 1L; // Optional, but good practice to define

private String line1;

private String line2;

}

**Key Points:**

1. **Implements Serializable:** Both Employee and Address classes implement Serializable, making instances of these classes eligible for serialization.
2. **serialVersionUID:** It's recommended to include a serialVersionUID for version control. If a class definition changes after objects have been serialized, the serialVersionUID is used to verify that the deserialization process is compatible with the serialized object.
3. **Field Naming Correction:** Fixed the typo in the field names (linel to line1 and adress to address).

Now both classes are serializable and can be serialized into a byte stream and deserialized back into objects.

### **In above example, what if we have empId as static then what will happen at the time of serialization?**

* If empId is declared as static in the Employee class, it will **not be serialized** during the serialization process. This is because static fields are associated with the class itself, not with instances of the class, and serialization in Java only deals with instance data (non-static fields).

Here's what will happen when empId is static:

**Example:**

import java.io.Serializable;

import java.util.Date;

@Data

class Employee implements Serializable {

private static final long serialVersionUID = 1L;

private String name;

private String empld;

private Date dob;

private Address address;

// Static field (will not be serialized)

private static String empId;

public Employee(String name, String empld, Date dob, Address address, String empId) {

this.name = name;

this.empld = empld;

this.dob = dob;

this.address = address;

Employee.empId = empId;

}

}

@Data

class Address implements Serializable {

private static final long serialVersionUID = 1L;

private String line1;

private String line2;

}

**What happens during serialization:**

* The instance fields (name, empld, dob, address) of the Employee class will be serialized.
* The static field (empId) will **not** be serialized because it is tied to the class, not the individual object instance.
* After deserialization, the value of the empId field will be **lost** for that particular instance because it was not serialized. The empId will not be preserved across serialization and deserialization unless it is explicitly handled (e.g., through custom serialization or manually restoring the value after deserialization).

**Example of behavior:**

1. **Before Serialization:**
   * empId = "12345" (static)
   * empld = "E123" (instance field)
2. **After Serialization and Deserialization:**
   * empld will retain its value (e.g., "E123").
   * empId will be reset to its default value (null for String since it's static, or it may stay as it was set before the serialization if shared among instances).

To ensure that empId is correctly handled across serialization and deserialization, you might need to manually manage it. For instance, you could mark the field transient and handle its restoration manually, or you could implement custom readObject and writeObject methods to control how static fields are serialized.

### **What is the final class?**

* **🧱 What is a final class?**

A **final class** is a class that **cannot be extended (inherited)**.

public final class MyClass {

// class code

}

* Once a class is marked as final, **no other class** can subclass it.

**✅ Why use a final class?**

| **Reason** | **Benefit** |
| --- | --- |
| 🔐 Prevent Inheritance | Avoid accidental or malicious subclassing |
| 🛡️ Security | Makes class behavior predictable and secure |
| 🚀 Performance Optimization | JVM can make certain optimizations as it knows the class won't change |
| 📦 Encapsulation | Helps in creating immutable or well-contained utility classes |

**🔐 Example:**

public final class SecurityManager {

public void checkPermission() {

System.out.println("Permission granted.");

}

}

// ❌ This will cause a compile-time error

public class MySecurityManager extends SecurityManager {

// Cannot inherit from final class

}

**📌 Real-life Example: java.lang.String**

public final class String {

// String class is final in Java

}

* Why? Because String is **immutable and heavily used**. Preventing subclassing avoids unpredictable behavior.

**⚠️ Key Points**

* You **can create objects** of a final class.
* You **cannot extend (subclass)** a final class.
* A final class **can have final or non-final methods**.
* A class can be final, a method can be final, and a variable can also be final — but they have **different meanings**.

**Related Concepts**

| **Keyword** | **Meaning** |
| --- | --- |
| final variable | Value cannot be changed after initialization |
| final method | Method cannot be overridden |
| final class | Class cannot be extended |

### **Where can we use the final keyword?**

* The final keyword in Java is used to **restrict modification** — it can be applied to **variables, methods, and classes**.

Let’s break it down:

**✅ 1. final Variable**

* **Prevents reassignment** — once initialized, its value **cannot be changed**.

**🔹 Example:**

final int x = 10;

// x = 20; // ❌ Compilation error

**🔸 Use Case:**

* Constants (static final)
* Ensuring values remain unchanged

**✅ 2. final Method**

* Prevents a method from being **overridden** by subclasses.

**🔹 Example:**

class Parent {

final void show() {

System.out.println("Parent show");

}

}

class Child extends Parent {

// void show() { } // ❌ Error: Cannot override final method

}

**🔸 Use Case:**

* Security or utility methods that shouldn't be altered

**✅ 3. final Class**

* Prevents a class from being **subclassed** (i.e., inherited).

**🔹 Example:**

final class Animal {

void speak() {

System.out.println("Roar");

}

}

// class Dog extends Animal {} // ❌ Error: Cannot inherit from final class

**🔸 Use Case:**

* Immutable or security-sensitive classes (like java.lang.String)

**✅ 4. final with Reference Variables**

* You **can’t reassign** the reference, but the object it points to **can still be modified**.

final List<String> names = new ArrayList<>();

names.add("Alice"); // ✅ Allowed

// names = new ArrayList<>(); // ❌ Not allowed

**✅ 5. final with Method Parameters**

* Makes the parameter **read-only** within the method.

void print(final int num) {

// num = 5; // ❌ Error

System.out.println(num);

}

**✅ 6. final and static together**

Used to define **constants**.

public static final double PI = 3.14159;

**📝 Summary Table**

| **Context** | **Effect of final** |
| --- | --- |
| Variable | Value can't be changed after initialization |
| Method | Cannot be overridden |
| Class | Cannot be extended (no subclass) |
| Reference | Reference can't be reassigned (but object can change) |
| Parameter | Value cannot be modified inside method |
| static final | Creates constants |

### **What will happen if I create the final class?**

* A final class in Java is a class that **cannot be extended or inherited**.

**✅ What happens when you create a final class?**

**1. You can use it as-is**

You **can create objects**, call its methods, and use it like any regular class.

final class Vehicle {

void run() {

System.out.println("Vehicle is running");

}

}

public class Main {

public static void main(String[] args) {

Vehicle v = new Vehicle();

v.run(); // ✅ Works fine

}

}

**2. You cannot extend it**

If any other class tries to inherit it, the compiler throws an error.

class Car extends Vehicle {

// ❌ Compilation Error: Cannot inherit from final class

}

**💡 Why create a final class?**

| **Reason** | **Description** |
| --- | --- |
| 🔐 **Security** | Prevents classes from being modified via inheritance |
| 🧱 **Immutability** | Helps build immutable classes (like String) |
| 📦 **Framework Control** | Used in APIs to avoid unexpected subclassing |
| 🔄 **Stable Behavior** | Prevents method overriding and logic changes |

**✅ Real-world example**

The String class in Java is final:

public final class String {

// internal implementation

}

Why? Because String is:

* Immutable
* Used heavily in JVM internals
* Needs guaranteed behavior

**🚫 What you CANNOT do with a final class:**

* ❌ Subclass it
* ❌ Override its methods in a child class (since no child class can exist)

But you **can**:

* ✅ Instantiate it
* ✅ Use its methods
* ✅ Pass it around like any other class

### **What is String?**

* **📌 What is String in Java?**
* In Java, String is a **class** in the **java.lang** package.
* It is used to represent **a sequence of characters** (text data).
* **Immutable**: Once a String object is created, its value **cannot be changed**.

String name = "Alice";

Here, name is a String object containing the text "Alice".

**🔍 String is a Class (Not a Primitive)**

* String is **not a primitive type** (like int, char, boolean)
* It's a **reference type**, but behaves like a basic data type due to special handling by the JVM.

**🔐 String is Immutable**

String s1 = "Hello";

s1.concat(" World");

System.out.println(s1); // Output: Hello (not "Hello World")

* A new String is created internally, but s1 still points to the original "Hello".
* For modifications, use StringBuilder or StringBuffer.

**🧠 How Strings are Stored**

**String Pool (String Interning):**

String a = "Java";

String b = "Java";

System.out.println(a == b); // true (points to same object in string pool)

* Strings created with **literals** are stored in the **String constant pool**.
* Helps in **memory optimization**.

**🔧 Ways to Create Strings**

// 1. Using literals (preferred)

String str1 = "Hello";

// 2. Using constructor (avoids pool)

String str2 = new String("Hello");

**✅ Common String Methods:**

| **Method** | **Description** |
| --- | --- |
| length() | Returns the length |
| charAt(index) | Returns character at index |
| equals() | Checks value equality |
| == | Checks reference equality |
| toLowerCase() | Converts to lowercase |
| toUpperCase() | Converts to uppercase |
| substring(start) | Extracts a part of the string |
| contains(str) | Checks if substring is present |
| split(regex) | Splits string into array |
| replace(a, b) | Replaces characters or substrings |
| trim() | Removes leading/trailing spaces |

**🔄 Mutable Alternatives:**

| **Class** | **Thread Safe** | **Use Case** |
| --- | --- | --- |
| StringBuilder | ❌ No | Faster in single-threaded apps |
| StringBuffer | ✅ Yes | For multi-threaded environments |

**🧪 Example:**

public class Example {

public static void main(String[] args) {

String greet = "Hello";

greet += " World";

System.out.println(greet); // Hello World

}

}

### **Why String is immutable in java?**

* **🔐 Why is String immutable in Java?**

Because once a String object is created, **its value cannot be changed**. Any operation that seems to "modify" a string actually creates a **new object**.

**🧠 But WHY did Java make Strings immutable?**

**✅ 1. Security**

* Strings are widely used in **file paths, network connections, class loading, etc.**
* If a String were mutable, someone could change:

// Original path

String path = "/home/user/app";

// Imagine someone changes it to a malicious path!

🔐 Immutability prevents tampering with sensitive data like usernames, passwords, and URLs.

**✅ 2. String Pooling**

* Java uses a **String Pool** to save memory.
* Since strings don't change, they can be **safely shared** among many references.

String s1 = "Java";

String s2 = "Java";

System.out.println(s1 == s2); // true → Same object in memory

**✅ 3. Thread Safety**

* Immutable objects are **automatically thread-safe**.
* No need for synchronization since their state can’t change.

Multiple threads can safely use the same String object without causing issues.

**✅ 4. HashCode Consistency**

* String is often used as a key in **HashMap, HashSet, etc.**
* Since the value never changes, the **hashCode stays constant**, ensuring map integrity.

Map<String, String> map = new HashMap<>();

map.put("key", "value");

// If "key" changed, we couldn't retrieve "value" reliably!

**✅ 5. Performance**

* JVM optimizes the use of strings due to immutability.
* Reuse from the pool and predictable behavior helps the JVM optimize better.

**🧪 Example of how it's immutable:**

String s = "Java";

s.concat(" World");

System.out.println(s); // Java → Original string unchanged

s.concat(" World") creates a new string "Java World" but doesn't change s.

**🧱 Internally**

The String class has a private final char array:

public final class String {

private final char value[];

}

* final → reference to the array can't change
* private → can't access directly
* Immutable at its core

**✅ Summary**

| **Reason** | **Explanation** |
| --- | --- |
| Security | Prevents sensitive data manipulation |
| String Pooling | Enables memory-efficient reuse |
| Thread Safety | Can be shared between threads without risk |
| HashCode Stable | Reliable use in HashMap, HashSet, etc. |
| JVM Optimization | Better performance due to predictability |

### **What is StringBuffer?**

* **🔤 StringBuffer in Java?**
* StringBuffer is a **mutable** class used to create and manipulate **strings of characters**.
* Unlike String, which is **immutable**, StringBuffer objects **can be changed** without creating new objects.
* It is **thread-safe**, meaning **methods are synchronized** so multiple threads can use it safely.

**🔧 Declaration & Example:**

StringBuffer sb = new StringBuffer("Hello");

sb.append(" World");

System.out.println(sb); // Output: Hello World

* append() modifies the original object itself, no new object is created.

**🧪 Key Features of StringBuffer**

| **Feature** | **Description** |
| --- | --- |
| ✅ Mutable | Can modify content without creating new objects |
| ✅ Thread-safe | Methods are synchronized |
| 📏 Dynamic length | Grows as needed (like a resizable array) |
| ⚡ Slower than StringBuilder | Due to synchronization overhead |

**🛠️ Common Methods**

StringBuffer sb = new StringBuffer("Java");

sb.append(" Rocks"); // Add text

sb.insert(4, " Programming"); // Insert at index

sb.replace(0, 4, "Python"); // Replace part of string

sb.delete(0, 6); // Delete part

sb.reverse(); // Reverse string

System.out.println(sb);

**💡 When to use StringBuffer?**

Use StringBuffer when:

* You need a **mutable string**
* You are working in a **multi-threaded environment**
* You don’t want to use manual synchronization

**✅ Summary**

* StringBuffer = **mutable + thread-safe** version of string manipulation.
* Prefer StringBuilder in single-threaded contexts for better performance.
* Use String when data is constant and doesn't change.

### **What is StringBuilder?**

* **🔤 StringBuilder in Java?**
* StringBuilder is a **mutable** sequence of characters, like StringBuffer, but it is **not synchronized**, meaning it is **not thread-safe**.
* It is designed for **single-threaded applications** where performance is important and you need to frequently modify strings.
* **Faster** than StringBuffer because it doesn't incur the overhead of synchronization.

**🔧 StringBuilder Example:**

StringBuilder sb = new StringBuilder("Hello");

sb.append(" World");

System.out.println(sb); // Output: Hello World

* append() modifies the content of the original object, and no new object is created.

**🧪 Key Features of StringBuilder**

| **Feature** | **Description** |
| --- | --- |
| ✅ Mutable | Can modify content without creating new objects |
| ⚡ Performance | **Faster** than StringBuffer (no synchronization) |
| 🛠️ Thread-safety | **Not synchronized**, hence not thread-safe |
| 📏 Dynamic length | Resizable — automatically grows as needed |

**🛠️ Common Methods in StringBuilder**

StringBuilder sb = new StringBuilder("Java");

// Append new text

sb.append(" Rocks");

// Insert text at a specified index

sb.insert(4, " Programming");

// Replace text in the specified range

sb.replace(0, 4, "Python");

// Delete text from a specified range

sb.delete(0, 6);

// Reverse the string

sb.reverse();

System.out.println(sb); // Output: Python Rocks

* StringBuilder is ideal for frequent modifications, such as **concatenation**, **insertion**, and **reversals**.

**✅ When to use StringBuilder?**

* Use **StringBuilder** when:
  + You need a **mutable string**.
  + The **performance** of string operations is a concern (since StringBuilder is faster).
  + You're working in a **single-threaded** environment.
* Avoid StringBuilder in multi-threaded contexts where multiple threads may need to access the same object, as it is **not thread-safe**.

**📈 Performance Comparison: String, StringBuffer, and StringBuilder**

* **String**: Immutable, creates a new object on every change — good for constant data.
* **StringBuffer**: Thread-safe but **slower** due to synchronization.
* **StringBuilder**: **Faster** than StringBuffer but not thread-safe — ideal for single-threaded use cases.

**✅ Summary**

* StringBuilder = **mutable + thread-unsafe** string manipulation.
* **Best choice for performance** in **single-threaded** environments.
* Choose StringBuilder when you need to frequently **modify strings** and do not need thread safety.

### **Difference between String, StringBuffer and StringBuilder?**

| **Feature** | **String** | **StringBuffer** | **StringBuilder** |
| --- | --- | --- | --- |
| Mutable | ❌ No | ✅ Yes | ✅ Yes |
| Thread Safe | ✅ Yes | ✅ Yes | ❌ No |
| Performance | 🐢 Slow | 🐌 Moderate | 🏎️ Fastest |
| Synchronized | ❌ No | ✅ Yes | ❌ No |
| Use Case | Constant Data | Multi-threaded | Single-threaded |
| Memory Efficient? | ✅ If constant | ❌ More overhead | ✅ More efficient |

**Example Comparison:**

// String (Immutable)

String str = "Java";

str.concat(" World");

System.out.println(str); // Java

// StringBuffer (Mutable + Thread-safe)

StringBuffer sb = new StringBuffer("Java");

sb.append(" World");

System.out.println(sb); // Java World

// StringBuilder (Mutable + Not thread-safe)

StringBuilder sb2 = new StringBuilder("Java");

sb2.append(" World");

System.out.println(sb2); // Java World

### **Difference between comparable and comparator?**

| **Feature** | **Comparable** | **Comparator** |
| --- | --- | --- |
| **Package** | java.lang | java.util |
| **Purpose** | Defines **natural ordering** of objects | Defines **custom ordering** of objects |
| **Interface method** | int compareTo(T o) | int compare(T o1, T o2) |
| **Implemented by** | The class whose objects are being compared | Separate class (or anonymous class / lambda) |
| **Affects original class?** | Yes, class must implement Comparable | No, keeps original class unchanged |
| **Used in** | Collections.sort(list) (no comparator) | Collections.sort(list, comparator) |
| **Java 8+ usage** | Can use lambda indirectly through compareTo() | Often used with lambda expressions |

**Example:**

**Using Comparable:**

class Student implements Comparable<Student> {

int id;

String name;

public int compareTo(Student s) {

return this.id - s.id; // Natural order by id

}

}

**Using Comparator:**

class NameComparator implements Comparator<Student> {

public int compare(Student s1, Student s2) {

return s1.name.compareTo(s2.name); // Custom order by name

}

}

Java 8+ Lambda Comparator:

Comparator<Student> nameComparator = (s1, s2) -> s1.name.compareTo(s2.name);

Collections.sort(studentList, nameComparator);

### **What is static in java?**

* **Static** is a modifier that can be applied to variables, methods, blocks, and nested classes.

**Static Variable (Class Variable):**

* Belongs to the class rather than any object
* Shared by all instances of the class
* Initialized when class is loaded

class Counter {

static int count = 0; // static variable

Counter() {

count++;

}

}

**Static Method:**

* Belongs to the class rather than instances
* Can be called without creating an object
* Can only access static members directly

class MathUtils {

static int add(int a, int b) { // static method

return a + b;

}

}

// Usage: MathUtils.add(5, 3);

**Static Block:**

* Used for static initialization of a class
* Executed when the class is loaded

class MyClass {

static {

System.out.println("Static block executed");

}

}

**Example Combining Both:**

abstract class Database {

static final String DEFAULT\_URL = "jdbc:default"; // static constant

abstract void connect(); // abstract method

static void printDefaultUrl() { // static method

System.out.println(DEFAULT\_URL);

}

}



**Note:**

Static members are resolved at compile-time (early binding), while abstract methods enable runtime polymorphism (late binding).

### **Explain Generics? What is the use of Generics?**

* **Generics** in Java enable you to write **type-safe and reusable code**. They allow you to define classes, interfaces, and methods with **type parameters**, so you can work with any object type while maintaining compile-time type checking.

🧠 **Why Use Generics?**

| **Feature** | **Benefit** |
| --- | --- |
| **Type Safety** | Catches type errors at compile time. No ClassCastException at runtime. |
| **Code Reusability** | Write a single class/method for any data type. |
| **Readability & Maintainability** | Avoids unnecessary casting and makes the code easier to read. |
| **Performance** | Eliminates need for boxing/unboxing (with collections). |

📌 **Example Without Generics (Old Java Style):**

List names = new ArrayList();

names.add("Alice");

String name = (String) names.get(0); // Manual cast

📌 **Example With Generics:**

List<String> names = new ArrayList<>();

names.add("Alice");

String name = names.get(0); // No cast needed

💡 **Generic Class Example:**

class Box<T> {

private T item;

public void set(T item) {

this.item = item;

}

public T get() {

return item;

}

}

**Usage:**

Box<String> stringBox = new Box<>();

stringBox.set("Hello");

System.out.println(stringBox.get());

🔄 **Generic Method Example:**

public <T> void printArray(T[] array) {

for (T element : array) {

System.out.println(element);

}

}

🧱 **Bounded Generics:**

class Calculator<T extends Number> {

public double square(T num) {

return num.doubleValue() \* num.doubleValue();

}

}

🎯 **Common Use in Collections:**

Map<String, List<Integer>> studentMarks = new HashMap<>();

### **What is Singleton in java?**

* A **Singleton** is a **design pattern** that ensures **only one instance** of a class exists throughout the application and provides a **global access point** to that instance.

It’s commonly used for things like:

* Configuration classes
* Logger classes
* Database connections
* Caching mechanisms

**✅ Key Characteristics:**

1. **Private constructor** – so no other class can instantiate it.
2. **Static instance** – holds the single instance.
3. **Public static method** – provides access to the instance.

**🔧 Basic Singleton Implementation (Lazy Initialization):**

public class MySingleton {

private static MySingleton instance;

private MySingleton() {

// private constructor

}

public static MySingleton getInstance() {

if (instance == null) {

instance = new MySingleton();

}

return instance;

}

}

**🔐 Thread-Safe Singleton (Synchronized Method):**

public class MySingleton {

private static MySingleton instance;

private MySingleton() {}

public static synchronized MySingleton getInstance() {

if (instance == null) {

instance = new MySingleton();

}

return instance;

}

}

**⚡ Best Practice: Bill Pugh Singleton (Thread-safe & Lazy without synchronization overhead):**

public class MySingleton {

private MySingleton() {}

private static class Holder {

private static final MySingleton INSTANCE = new MySingleton();

}

public static MySingleton getInstance() {

return Holder.INSTANCE;

}

}

**🚀 Singleton with Enum (Recommended for simplicity & thread safety):**

public enum MySingleton {

INSTANCE;

public void doSomething() {

System.out.println("Singleton doing work...");

}

}

**Usage:**

MySingleton.INSTANCE.doSomething();

### **Consider we have Strings as String s1 = “Welcome”, String s2 = “Welcome” and String s3 = “WelcomeOne”. What does s1==s2 represent?**

* s1 == s2 in Java compares references, not content.

🔍 **Given:**

String s1 = "Welcome";

String s2 = "Welcome";

String s3 = "WelcomeOne";

✅ s1 == s2 → true ✅

Why? Because:

"Welcome" is a String literal.

String literals are stored in the String Pool (a part of the heap).

Java optimizes memory by reusing string literals from the pool.

So both s1 and s2 point to the same memory location in the pool.

🧠 What about s1.equals(s2)?

This checks content.

Also returns true, because "Welcome".equals("Welcome").

❌ == is false if:

String s1 = new String("Welcome");

String s2 = "Welcome";

System.out.println(s1 == s2); // false (different memory locations)

System.out.println(s1.equals(s2)); // true (same content)

🎯 **Summary:**

| **Comparison** | **Meaning** | **Result** |
| --- | --- | --- |
| s1 == s2 | Reference equality | ✅ true (same pool object) |
| s1.equals(s2) | Content equality | ✅ true (same content) |

### **Explain String constant pool?**

* The **String Constant Pool** (aka **String Intern Pool**) is a special memory area inside the **heap** where **Java stores all string literals** to optimize memory usage.

**✅ How it works:**

When you do:

String s1 = "Java";

String s2 = "Java";

"Java" is stored only once in the String pool.

Both s1 and s2 will point to the same memory reference.

So:

s1 == s2 // true (same object)

s1.equals(s2) // true (same content)

**📦 Where is the pool stored?**

**In heap memory**, but **separately managed**.

Pre-Java 7: Stored in the **PermGen** space.

Java 7+: Moved to the **heap space** (more flexible).

**🚫 When a new String is created with new:**

String s1 = new String("Java");

String s2 = "Java";

s1 is a new object in the heap (outside the pool).

s2 is from the pool.

So s1 == s2 → ❌ false, but s1.equals(s2) → ✅ true

**🧵 How to manually put strings into the pool?**

**Use intern():**

String s1 = new String("Hello").intern();

String s2 = "Hello";

System.out.println(s1 == s2); // true

**✅ Benefits of String Pool:**

* **Memory efficiency**: Avoids duplicate string literals.
* **Performance boost**: Comparisons with == are faster than .equals().

**⚠️ Important Notes:**

* Only string literals are automatically pooled.
* String pool is managed by the **JVM** — no need for manual cleanup.

### **What is the difference between equals() and hashCode() method?**

| **Feature** | **equals()** | **hashCode()** |
| --- | --- | --- |
| Purpose | Compares contents of two objects | Returns an integer hash code for the object |
| Defined in | java.lang.Object | java.lang.Object |
| Return type | boolean | int |
| Usage | Used to check logical equality | Used in hash-based collections like HashMap, HashSet, Hashtable |
| Must override together? | ✅ Yes – if you override one, override the other | ✅ Yes – to maintain contract |

🧠 **Example:**

class Person {

String name;

int age;

// Override equals()

@Override

public boolean equals(Object obj) {

if (this == obj) return true;

if (obj == null || getClass() != obj.getClass()) return false;

Person p = (Person) obj;

return age == p.age && name.equals(p.name);

}

// Override hashCode()

@Override

public int hashCode() {

return Objects.hash(name, age); // Java 7+ utility

}

}

🚨 **Contract between equals() and hashCode():**

* If two objects are equal (equals() returns true), then their hashCode() must be the same.
* If hashCode() is different, equals() must return false.
* If hashCode() is the same, equals() may still return false (hash collision).

✅ **Used in Collections:**

Map<Person, String> map = new HashMap<>();

* hashCode() is used to locate the bucket
* equals() is used to find the correct key in the bucket

💥 **If You Only Override equals() but Not hashCode():**

Hash-based collections won’t work properly – e.g., duplicate keys may be stored.

### **Does Java support Pointers?**

* **No**, Java does not support pointers like C or C++ does.

**Why?**

Java was designed with the goal of:

* Safety
* Simplicity
* Platform-independence

Allowing raw memory access through pointers (like in C/C++) can:

* Lead to security vulnerabilities
* Cause memory corruption
* Make garbage collection impossible

✅ **What Java does instead:**

| Feature | Java's Alternative |
| --- | --- |
| Pointers | References (object handles, no direct memory access) |
| Manual memory management | Automatic Garbage Collection |
| Pointer arithmetic | ❌ Not allowed at all |

**Example:**

String s = "Hello";

Here, s holds a reference to a String object in memory (not a pointer you can manipulate).

You can’t get its address or do pointer math like s++.

🧠 **So What's the Difference?**

| **Concept** | **Pointers (C/C++)** | **References (Java)** |
| --- | --- | --- |
| Memory Access | Direct | Indirect (no control) |
| Arithmetic | Allowed | ❌ Not allowed |
| Null Safety | Risk of Segfault | NullPointerException (still safer) |
| Use Cases | System-level programming | Application-level programming |

🔐 **JVM Handles the Memory:**

Java manages memory behind the scenes using the JVM and Garbage Collector, freeing developers from manual memory management.

### **Why doesn't java support pointers?**

1**. 🔐 Security**

* Pointers allow direct memory access.
* This can lead to:
  + Buffer overflows
  + Memory leaks
  + Unauthorized memory access
* By hiding pointers, Java prevents low-level memory manipulation and increases application security.

2. 🧼 **Simplicity**

* Java was designed to be simple and easy to use.
* Pointers can be hard to understand and error-prone, especially for beginners (e.g., pointer arithmetic, null pointer dereferencing).
* Removing pointers removes a major source of bugs.

3. **🧹 Garbage Collection**

* Java uses automatic garbage collection.
* If raw pointers were allowed, it would confuse or break the garbage collector, which tracks objects using internal references.
* Pointers would let developers manipulate memory outside the GC’s knowledge, making safe memory management impossible.

4. 🧠 **Avoids Manual Memory Management**

* In languages like C/C++, you need to malloc/free memory manually.
* Java automates memory allocation/deallocation, making code less error-prone and more maintainable.

✅ **What Java Uses Instead?**

* References — like pointers, but:
  + No direct access to memory addresses
  + No pointer arithmetic
  + Managed and tracked by the JVM

### **Explain which language you prefer in terms of security among C, C++, Java and Python?**

* Each of these languages has its own strengths and weaknesses when it comes to **security**, and the level of security often depends on how the developer writes the code. However, I can break down the **security aspects** for each language to help you understand which one is inherently more secure.

**🏆 1. Java:**

**Security Strengths:**

**Memory Management**: Java uses **automatic garbage collection**, which helps avoid memory leaks and dangling pointers.

**No Pointers**: Java doesn’t support pointers, which prevents **buffer overflows**, **dangling pointer issues**, and **memory corruption**.

**Exception Handling**: Java has a built-in **robust exception handling system** that helps catch and manage runtime errors, making it safer.

**Bytecode**: Java runs on the **JVM** and uses **bytecode**, which makes it platform-independent and reduces the risk of platform-specific vulnerabilities.

**Security Manager**: The **Java Security Manager** restricts what operations code can perform (e.g., file access, network access) to prevent unauthorized actions.

**Strong Typing**: Java is **strongly typed**, so **type-related errors** are caught at compile-time.

**Security Weaknesses:**

**Java Libraries**: The security of a Java application can be compromised if third-party libraries are vulnerable (e.g., outdated versions of libraries).

**Deserialization Issues**: Older or improperly implemented code may be vulnerable to **Java deserialization attacks**, where malicious data is processed by the application.

**🛡️ 2. Python:**

**Security Strengths:**

**High-Level Language**: Python abstracts away low-level details like memory management, which reduces the risk of memory-related vulnerabilities like buffer overflows.

**Garbage Collection**: Like Java, Python has **automatic memory management** through garbage collection.

**Readability**: Python's **readable syntax** helps developers write secure code by reducing the likelihood of mistakes.

**Dynamic Typing**: Python is dynamically typed, which can reduce errors in the development process when used carefully.

**Security Weaknesses:**

**Dynamic Typing**: The lack of compile-time type checking means that **type-related bugs** (such as passing incorrect arguments to functions) are only caught at runtime.

**Security Libraries**: Python has security issues related to **unsafe libraries** (e.g., using insecure functions for handling user input or file handling).

**Interpreted Language**: Because Python is interpreted, malicious code can be more easily injected and executed without needing to go through a compilation phase.

**⚠️ 3. C/C++:**

**Security Strengths:**

**Control Over System Resources**: C and C++ give developers more control over system resources like memory and CPU, allowing for optimization.

**Used in High-Security Systems**: These languages are often used in **security-critical applications**, like operating systems, firewalls, and cryptography libraries, where developers must be mindful of security.

**Security Weaknesses:**

**Memory Management**: C and C++ require **manual memory management** (malloc, free), which is prone to **memory leaks**, **buffer overflows**, and **dangling pointers**.

**Pointer Arithmetic**: Pointers and pointer arithmetic make **buffer overflow** attacks more likely, which can lead to arbitrary code execution or access violations.

**No Built-in Protection**: C/C++ provides minimal **runtime checks** for errors like out-of-bounds access or null pointer dereferencing, which can result in undefined behavior and security vulnerabilities.

**Complexity**: Low-level operations and manual memory management can lead to human error, introducing bugs that compromise security.

**⚔️ 4. C++ (Compared to C):**

C++ inherits most of the security risks of C, but it adds some additional features like **object-oriented programming**, **RAII (Resource Acquisition Is Initialization)**, and **exceptions** for better error handling.

**C++'s extra features** can potentially help reduce some security risks, but C++ still requires **manual memory management** and **pointer arithmetic**, making it still prone to security vulnerabilities like C.

**🔑 Which Language is More Secure?**

**Java** is the most secure of the four languages, primarily because:

It doesn’t have **pointers**, avoiding many low-level vulnerabilities (e.g., **buffer overflows**, **memory corruption**).

It provides **automatic memory management** and **runtime security checks**.

It has the **Java Security Manager** and runs in a **sandboxed environment** which limits the potential for security breaches.

**📊 Security Ranking (Most to Least Secure):**

**Java** — Safest due to lack of pointers, garbage collection, and a robust security model.

**Python** — Generally safe with its high-level abstraction and garbage collection, but dynamic typing can cause runtime issues.

**C++** — Adds some security improvements over C (like exceptions and RAII), but still inherits the risk of **manual memory management** and **pointer manipulation**.

**C** — Least secure due to **manual memory management**, **pointer arithmetic**, and lack of built-in error checking, which makes it prone to **security vulnerabilities**.

### **Is Java 100% object oriented?**

* No, Java is not 100% object-oriented. However, it is primarily object-oriented, with a few exceptions.

**Why Java is Considered Object-Oriented:**

Java is designed with Object-Oriented Programming (OOP) principles at its core. Key features of OOP in Java include:

Encapsulation: Wrapping data and methods together into a class.

Inheritance: One class can inherit the properties and behaviors of another.

Polymorphism: Objects can take on many forms through method overriding or interfaces.

Abstraction: Hiding complex implementation details and showing only the necessary features of objects.

**Why Java is Not 100% Object-Oriented:**

**Primitive Data Types:**

* Java has primitive data types like int, char, float, double, boolean, etc.
* These primitives are not objects, meaning they don’t have methods or properties, unlike objects in Java.

**Example:**

int x = 10; // Primitive type

Integer y = new Integer(10); // Object type (Wrapper class)

* While Java provides wrapper classes like Integer, Character, etc., to "wrap" primitive types into objects, the primitives themselves are not object-oriented.

**Static Methods and Variables:**

* Java allows static methods and static variables that belong to the class rather than an instance of the class.
* Static methods are not tied to object instances and do not operate on object data.

**Example:**

public class MyClass {

static int count = 0;

static void incrementCount() {

count++;

}

}

Since static members are not associated with an object, they do not adhere strictly to OOP principles.

**Constructors:**

* Java’s constructor mechanism doesn’t completely follow OOP, as constructors are not considered methods but are used to initialize objects.
* They are used in a special way to create objects, but they aren't true methods of the object.

**Summary:**

* Java is mostly object-oriented but not 100% object-oriented.
* Java’s primitive types and static methods break the pure OOP paradigm.
* Despite these exceptions, Java is still widely considered an object-oriented language due to its class-based structure and focus on objects.

**Key Takeaway:**

Java is 99% object-oriented, but its inclusion of primitive types and static members means it’s not purely object-oriented like some other languages (e.g., Smalltalk, which has no primitives).

### **How can we make java 100% object oriented?**

* To make Java **100% object-oriented**, you would need to address the aspects that break from pure object-oriented principles. Here’s what you'd need to change:

**1. Eliminate Primitive Data Types**

In a purely object-oriented language, there are no primitive types like int, float, char, etc. Every type, including basic data types, must be an object.

**Solution:**

In pure object-oriented languages like **Smalltalk**, everything is an object, including numbers and characters.

In Java, this can be achieved by using **wrapper classes** for primitives:

int becomes Integer

char becomes Character

double becomes Double

This would involve **boxing** and **unboxing** for converting between primitives and objects, which can be inefficient.

**Example:**

Integer x = new Integer(10); // Instead of using primitive int

Character c = new Character('A'); // Instead of char

However, Java uses **autoboxing** and **unboxing** to automatically convert primitives to wrapper classes and vice versa, but to fully embrace OOP, everything would be manually boxed as objects.

**2. Remove Static Methods and Variables**

In Java, **static** members (variables and methods) are tied to the **class**, not the instance, which breaks the OOP principle of everything being an object.

**Solution:**

You would need to remove static methods and variables altogether.

Instead of using static methods, you would need to call methods on **objects** (instances of classes).

**Example:**

Instead of this:

class MyClass {

static int count = 0;

static void incrementCount() {

count++;

}

}

You would need to remove the static keyword:

class MyClass {

int count = 0;

void incrementCount() {

count++;

}

}

In this case, every **operation** would need an **instance** of MyClass.

**3. Treat Constructors as Objects**

In Java, constructors are special methods used to initialize objects, but they are not technically methods.

**Solution:**

To make Java 100% object-oriented, you could treat **constructors as objects**. However, this isn’t possible in the current version of Java, as constructors are a part of the language syntax.

You’d need to shift to a language where even the construction of objects is treated as an operation on an object (like in **Smalltalk**), where everything is an object.

**4. Remove the null Keyword**

The null keyword represents the **absence of an object** in Java, which means Java’s design recognizes the concept of "nothing," breaking the idea that **everything is an object**.

**Solution:**

Remove the concept of null, meaning that every object must be instantiated before it can be used. This is a theoretical idea that could make Java more object-oriented, but it would also increase complexity and have practical implications for memory management.

**Summary of Changes to Make Java 100% Object-Oriented:**

**Eliminate Primitive Data Types**: Use only objects, no primitives (int, float, etc.), replacing them with their wrapper classes.

**Remove Static Members**: All methods and variables would have to be instance-level members, with no static context.

**Treat Constructors as Objects**: In a truly object-oriented world, even object creation (constructors) would be handled in an object-like manner.

**Remove null**: Every variable must always refer to a valid object, and null would be eliminated.

**Realistic Consideration:**

While making Java fully object-oriented would be an interesting exercise in OOP purity, it would introduce significant **performance overheads** due to:

**Autoboxing** and **unboxing** for primitives

The need for managing more objects (even simple integers or booleans)

Complexity in the language's design, potentially making it less efficient.

### **Can we create objects of static class?**

* In Java, a **static class** typically refers to a **nested static class** (i.e., a class declared as static inside another class). The **static** keyword can be used with a nested class, but it has certain implications on object creation and behavior.

**Understanding Static Classes:**

**Static Nested Class**:

A **nested static class** is a class that is **declared static** inside another class.

It **does not** have a reference to the outer class’s instance, meaning it can exist independently of the outer class’s instance.

You can **create an object** of a static nested class without needing an instance of the outer class.

**Static Inner Class**:

A **static inner class** can be instantiated without needing the outer class’s instance, because it is not tied to an instance of the enclosing class.

**Can We Create Objects of a Static Nested Class?**

Yes, **you can create objects of a static class** (nested static class) in Java. Since a static nested class is **independent of the outer class instance**, it can be instantiated directly using the **class name**.

**Example of Static Nested Class:**

class OuterClass {

static class StaticNestedClass {

void display() {

System.out.println("Inside static nested class");

}

}

public static void main(String[] args) {

// Create an object of the static nested class without an instance of OuterClass

OuterClass.StaticNestedClass nestedObj = new OuterClass.StaticNestedClass();

nestedObj.display(); // Outputs: Inside static nested class

}

}

**In this example:**

StaticNestedClass is a **static nested class** inside OuterClass.

You can create an instance of StaticNestedClass **without** creating an instance of OuterClass.

**Static vs. Non-Static Inner Class:**

**Static Nested Class**:

Can be instantiated without an instance of the outer class.

Can access only **static members** of the outer class.

**Non-Static Inner Class**:

Requires an instance of the outer class to create an object of the inner class.

Can access both **instance and static members** of the outer class.

**Example of Non-Static Inner Class:**

class OuterClass {

class InnerClass {

void display() {

System.out.println("Inside non-static inner class");

}

}

public static void main(String[] args) {

// You need an instance of OuterClass to create an instance of the inner class

OuterClass outer = new OuterClass();

OuterClass.InnerClass innerObj = outer.new InnerClass();

innerObj.display(); // Outputs: Inside non-static inner class

}

}

**In this example:**

InnerClass is a **non-static inner class**.

To create an instance of InnerClass, you need to first create an instance of the outer class OuterClass.

**Summary:**

**Yes**, you can create objects of a static nested class without creating an instance of the outer class.

A **static nested class** is independent of the outer class’s instance, whereas a **non-static inner class** requires an instance of the outer class.

### **Which compiler is used by Java?**

* answer

### **Where are the hashCode() and equals() methods defined in java?**

* answer

### **How many design patterns are present in java?**

* answer

### **Can we override the static method?**

* answer

### **Explain access modifiers in java?**

* answer

### **What is volatile in java? Where can we use it?**

* answer

### **What is an idempotent method in java?**

* answer

### **What is final, static and non-static (instance) in java?**

* answer

### **How will I use two threads using a singleton design pattern?**

* answer

### **How to implement thread safety in java using singleton?**

* answer

### **Write an immutable class in Java?**

import java.io.Serializable;

import java.util.Date;

public final class Employee implements Serializable {

private static final long serialVersionUID = 1L;

private final String name;

private final String empId;

private final double sal;

private final Date dob;

// Constructor to initialize the Employee object

public Employee(String name, String empId, double sal, Date dob) {

this.name = name;

this.empId = empId;

this.sal = sal;

this.dob = new Date(dob.getTime()); // Defensive copy to protect mutable Date

}

// Getter methods for the fields (no setters to ensure immutability)

public String getName() {

return name;

}

public String getEmpId() {

return empId;

}

public double getSal() {

return sal;

}

public Date getDob() {

return new Date(dob.getTime()); // Defensive copy to avoid external modification

}

}

**Key Points:**

**Final Class:** The Employee class is final to prevent subclassing, ensuring that its immutability is not compromised.

**Final Fields:** All instance fields (name, empId, sal, dob) are final, ensuring that they can only be assigned once, during object construction.

**No Setters:** There are no setter methods, ensuring that the fields cannot be modified after the object is created.

Defensive Copying for Date: Since Date is mutable, a defensive copy is made in the constructor (new Date(dob.getTime())) and when accessing it via the getter (new Date(dob.getTime())), to prevent the internal dob field from being changed externally.

This makes the Employee class immutable, and its state cannot be changed once the object is created.

### **Can we override the protected method?**

* answer

### **What is marker interface in java? What is the purpose of the marker interface?**

* answer

### **How can we ignore the finally block?**

* answer

### **What is synchronous and asynchronous call?**

* answer

### **Can we write the main method as private?**

* answer

### **What is the use of static keywords in java?**

* answer

### **What is inner class? Have you used inner class in your project?**

* answer

### **What is the purpose of inner class?**

* answer

### **What is Enum? Write a syntax for enum?**

* answer

### **Difference between static and non-static in java?**

* answer

### **How can we break the singleton pattern?**

* answer

### **Difference between default and protected access modifier?**

* answer

### **Why is string immutable?**

* answer

### **What is an immutable class?**

* answer

### **What is atomic integer in java?**

* answer

### **How can you make a singleton class as thread safe?**

* answer

### **Difference between compareTo and compare method?**

* answer

### **What is the return type of compare method?**

* answer

### **What is contract in equals and hashcode method?**

* answer

### **Difference between Marker interface and Functional interface?**

* answer

### **Difference between Abstract class and Interface? Explain with scenarios which one you will it be used with conditions?**

* answer

### **What is the use of static and instance variables in java?**

* answer

### **How will you access JNDI in java code?**

* answer

### **Why is the main method in java static?**

* answer

### **What is the use of the static keyword?**

* answer

### **How many ways can we create beans in java?**

* answer

### **Explain different types of calling beans?**

* answer

### **What are the default methods?**

* answer

### **Can I add one more new default method in my child class? Does it give any problems with my existing classes?**

* answer

### **Difference between Comparable and Comparator interface?**

* answer

### **In Java, which class is the base class of all the classes?**

* answer

### **Why is the equals method required?**

* answer

### **Is it possible to have the same hashcode for multiple objects?**

* answer

### **Enlist a few predefined interfaces from Java 8?**

* answer

### **If I have a 200 MB excel file so can java read that file?**

* answer

### **What is immutability and mutability in java?**

* answer

### **Custom implementation of Singleton beans?**

* answer

### **What is the default scope of beans?**

* answer

### **Difference between static and final?**

* answer

### **Explain about String class in java?**

* answer

### **What is a special feature of String?**

* answer

### **Create an immutable class in java?**

* answer

### **How to access Singleton class from another class?**

* answer

### **Where will the Singleton class be stored in memory?**

* answer

### **Difference between bucket level and segment level locking?**

* answer

### **How is the static method bound with an object?**

* answer

### **What is the use of the intern() method in String class?**

* answer

## **Java Memory Management**

### **Explain JDK, JRE?**

* answer

### **Explain JVM?**

* Answer

### **What are the different loaders in JVM?**

* Answer

### **What is JVM profiling?**

* **JVM profiling** is the process of **monitoring and analyzing the performance** of a Java application running on the **Java Virtual Machine (JVM)** to identify bottlenecks, memory leaks, thread issues, CPU usage, and more.

It helps developers understand **how the application behaves at runtime**, which parts of the code consume the most resources, and how to optimize it.

**✅ Key Aspects of JVM Profiling:**

**Memory Usage**:

Monitor **heap** and **non-heap** memory usage.

Detect **memory leaks** or **excessive garbage collection (GC)**.

Analyze **object creation** and memory retention.

**CPU Usage**:

Identify **methods or classes** consuming the most CPU time.

Analyze **hotspots** in the code.

**Garbage Collection (GC)**:

Track how often GC occurs and how much time it takes.

Tune JVM GC settings based on profiling.

**Thread Activity**:

Monitor thread states (running, waiting, blocked).

Detect **deadlocks**, **thread contention**, and **race conditions**.

**Class Loading**:

Analyze which classes are loaded/unloaded and their memory footprint.

**🛠 Common JVM Profiling Tools:**

| **Tool** | **Description** |
| --- | --- |
| **JVisualVM** | Bundled with JDK; visual profiler for memory, CPU, GC, threads. |
| **JConsole** | Lightweight monitoring tool; good for basic profiling. |
| **YourKit** | Commercial, advanced features for memory, threads, GC analysis. |
| **JProfiler** | Commercial, very powerful for CPU/memory/thread profiling. |
| **Flight Recorder & Mission Control** | Built into Oracle JDK; low-overhead continuous profiling. |
| **Async Profiler** | Open-source, low-overhead async CPU and memory profiler. |
| **Perf (Linux)** + **Java Flame Graphs** | Advanced profiling using Linux performance tools. |

**🧠 Example Use Case:**

* You notice your Java app is **slowing down** over time.
* You use **JVisualVM** to attach to the running JVM.
* It shows:
* **Heap memory** constantly growing → Possible **memory leak**.
* Certain method taking 80% of **CPU time** → Optimize that logic.
* **Thread deadlock** between two threads → Fix synchronization issue.

**🧪 How to Start JVM Profiling (Example with JVisualVM):**

* 1. Start your Java application with:

java -jar myapp.jar

* 1. Open jvisualvm (comes with JDK).
  2. Attach to the running process.
  3. View tabs for **CPU**, **Memory**, **Threads**, **GC**, etc.
  4. Use **heap dump** and **profiling snapshots** for deep analysis.

**✅ Benefits:**

* Optimizes performance.
* Improves resource utilization.
* Detects memory leaks early.
* Helps fix concurrency issues.

### **Explain Java memory management?**

* answer

### **What is Stack / Heap memory?**

* Answer

### **Explain memory management below? Int and Integer?**

* answer

### **Where local variables are stored in memory?**

* answer

### **Where objects will be stored in memory?**

* answer

### **Explain Heap memory in java?**

* Answer

### **Difference between Stack and Heap?**

* Answer

### **What is Garbage Collection in java?**

* **Garbage Collection** in Java is the **automatic process** of identifying and **reclaiming memory** used by objects that are **no longer reachable** or needed by the application.

It is performed by the **JVM (Java Virtual Machine)** to **free up heap memory** and prevent memory leaks, so developers don't have to manually manage memory (unlike in C/C++).

**✅ Why is GC Important?**

* Prevents **OutOfMemoryError**.
* Frees memory from **unused objects**.
* Improves **application stability**.
* Reduces chances of **memory leaks**.

**🧠 How GC Works (High-Level):**

* Java objects are stored in **Heap Memory**.
* When no references point to an object, it becomes **eligible for GC**.
* The JVM periodically runs the **Garbage Collector**, which:
  + Finds **unreachable objects**.
  + Reclaims their memory.
  + Compacts memory (optional, depending on collector).

**📦 JVM Memory Areas (relevant to GC):**

| **Area** | **Description** |
| --- | --- |
| **Young Generation** | New objects created (faster GC – minor GC) |
| **Old Generation** | Long-lived objects (slower GC – major GC) |
| **Metaspace** | Class metadata (not garbage collected as objects) |

**🧪 Example:**

public class GCDemo {

public static void main(String[] args) {

String str = new String("Hello");

str = null; // Now eligible for GC

System.gc(); // Hint to JVM to run GC (not guaranteed)

}

}

**🧰 Common Garbage Collectors in Java:**

| **GC Algorithm** | **Description** |
| --- | --- |
| **Serial GC** | Simple, single-threaded. Good for small apps. |
| **Parallel GC** | Uses multiple threads for minor/major GC. Good for throughput. |
| **CMS (Concurrent Mark-Sweep)** | Minimizes pause times (deprecated). |
| **G1 GC (Garbage First)** | Good balance of performance and low pause times. |
| **ZGC** / **Shenandoah** | Ultra-low pause times (Java 11+ / 12+). |

**🛠 GC Tuning (Optional for advanced users):**

You can tune the JVM GC behavior using flags:

java -Xms512m -Xmx2g -XX:+UseG1GC MyApp

**🔍 How to Know if GC Is Happening?**

Use logs:

-verbose:gc

-XX:+PrintGCDetails

Use tools like:

* Jvisualvm
* Jconsole
* GCViewer
* Java Flight Recorder

**✅ Summary:**

* GC is Java's way of **automatic memory management**.
* It helps **keep memory clean** by removing unused objects.
* Developers **don’t control** GC directly, but they can influence it with tuning and understanding object lifecycles.

### **What will happen if I assign null to some object then will that be garbage collected?**

* answer

### **When does the finalize method get called?**

* The finalize() method is called **by the Garbage Collector (GC)** **just before** an object is **destroyed (i.e., garbage collected)** to give it a chance to clean up resources.

**✅ Key Points:**

| **Aspect** | **Description** |
| --- | --- |
| **When** | Automatically called **before the GC reclaims the object's memory** |
| **Defined in** | java.lang.Object class |
| **Signature** | protected void finalize() throws Throwable |
| **Purpose** | Used to perform **cleanup** like closing files, network sockets, etc. |
| **Manual call** | You can call it explicitly, but it's **not recommended** (obj.finalize() is just a method call, not GC) |
| **Only once?** | Yes — GC calls it **at most once** per object |
| **Deprecated?** | ✅ **Yes**, deprecated in **Java 9**, and **removed in Java 18+** because it's **unreliable and unpredictable** |

**⚠️ Why Not to Use finalize()?**

No guarantee **when or even if** it will be called.

Can lead to **resource leaks** if relied on.

**Better alternatives**: use **try-with-resources** or explicitly close resources via close().

**🔧 Example:**

class MyObject {

@Override

protected void finalize() throws Throwable {

System.out.println("Finalize called!");

}

}

public class Main {

public static void main(String[] args) {

MyObject obj = new MyObject();

obj = null;

System.gc(); // Suggests GC, not guaranteed

System.out.println("Main ends.");

}

}

Output may or may not show "Finalize called!" — because GC behavior is **non-deterministic**.

## **OOPs**

### **Explain OOPs concept?**

* Object-Oriented Programming (OOPs) is a programming paradigm based on the concept of "objects", which can contain data and code. Java is a fully object-oriented language (except for primitive types) and supports the following core OOPs concepts:
* **Abstraction**
  + Hides internal implementation and shows only functionality.
  + Achieved using **abstract classes** and **interfaces**.
* **Encapsulation**
  + Bundling of data (variables) and methods into a single unit (class).
  + Achieved using **access modifiers** (private, public, protected).
* **Inheritance**
  + Allows a class (child) to inherit fields and methods from another class (parent).
  + Promotes code reusability and method overriding.
* **Polymorphism**
  + One entity behaves differently based on context.
  + **Compile-time (method overloading)** and **Run-time (method overriding)** are the two types.

### **What is an object?**

* answer

### **Explain method in Object class? Explain 9 methods in Object class?**

* Answer

### **How many ways can we create objects of class in java?**

* Answer

### **Explain Abstraction?**

* answer

### **What is Encapsulation? Explain with a real time example**

* Encapsulation is one of the fundamental OOPs principles in Java. It refers to wrapping data (variables) and the code (methods) that operates on the data into a single unit — typically a class. It helps protect the internal state of an object from unwanted external modifications.
* Encapsulation is achieved using:
  + Private fields to restrict direct access.
  + Public getters and setters to allow controlled access.
* Example: Bank Account

public class BankAccount {

private double balance; // private field — can't be accessed directly

public BankAccount(double initialBalance) {

this.balance = initialBalance;

}

// public method to get balance

public double getBalance() {

return balance;

}

// public method to deposit money (with validation)

public void deposit(double amount) {

if (amount > 0) {

balance += amount;

}

}

// public method to withdraw money (with validation)

public void withdraw(double amount) {

if (amount > 0 && balance >= amount) {

balance -= amount;

}

}

}

**Explanation:**

* The balance variable is **encapsulated** — it's private.
* Access is provided **only through public methods** (getBalance, deposit, withdraw), ensuring that:
  + The balance can't be set to a negative value directly.
  + Deposits and withdrawals go through validation.

This prevents incorrect or unauthorized changes to the data, which is the essence of **encapsulation** in real-world applications.

### **How do we achieve encapsulation in Java?**

* Encapsulation is one of the four fundamental OOP concepts in Java. It refers to bundling data (variables) and methods that operate on that data into a single unit (class) while restricting direct access to some of the object's components.

**Key Ways to Achieve Encapsulation:**

**1. Using Private Access Modifier**

Make class fields private to prevent direct access from outside the class.

public class Person {

private String name; // private field

private int age;

}

**2. Providing Public Getter and Setter Methods**

Create public methods to access and modify private fields.

public class Person {

private String name;

private int age;

// Getter for name

public String getName() {

return name;

}

// Setter for name

public void setName(String name) {

this.name = name;

}

// Getter for age

public int getAge() {

return age;

}

// Setter for age with validation

public void setAge(int age) {

if(age > 0) { // validation logic

this.age = age;

}

}

}

**3. Implementing Constructors with Validation**

Initialize fields through constructors with validation.

public Person(String name, int age) {

this.name = name;

if(age > 0) {

this.age = age;

}

}

**Benefits of Encapsulation:**

* **Data Hiding**: Internal representation is hidden from outside
* **Increased Flexibility**: Can change internal implementation without affecting other code
* **Reusability**: Encapsulated code is easier to reuse
* **Control**: Can add validation logic in setters
* **Maintainability**: Easier to maintain and modify code

**Example Usage:**

public class Main {

public static void main(String[] args) {

Person person = new Person("Alice", 30);

// Access through getters

System.out.println(person.getName()); // Alice

System.out.println(person.getAge()); // 30

// Modify through setters

person.setAge(31);

person.setName("Alice Smith");

// Invalid age won't be set

person.setAge(-5); // age remains 31

}

}

### **What is abstract in java?**

* **Abstract** is a non-access modifier that can be applied to classes and methods.

**Abstract Class:** Abstract is a non-access modifier that can be applied to classes and methods.

* Cannot be instantiated (cannot create objects)
* Can contain both abstract and concrete methods
* Used as a base class for inheritance

abstract class Animal {

// Abstract method (no implementation)

abstract void makeSound();

// Concrete method

void eat() {

System.out.println("Animal is eating");

}

}

**Abstract Method:**

* Has no body (no implementation)
* Must be overridden by the first concrete subclass
* Can only exist in abstract classes

abstract class Shape {

abstract double calculateArea(); // abstract method

}

**Key Points:**

* If a class has even one abstract method, the class must be declared abstract
* Abstract classes can have constructors (called when subclass is instantiated)
* Used to define common interface for subclasses

### **Explain Composition in Oops?**

* Answer

### **What is composition and aggregation?**

* answer

### **Parent and child class methods and how to access those?**

* answer

### **Difference between interface and abstract class?**

* answer

### **Where will you use interface and abstract class with real time examples?**

* answer

### **What is encapsulation? How are you currently using it?**

* answer

### **Explain Compile time and Run time polymorphism?**

* answer

### **What is method overriding?**

* Answer

### **How to achieve method overriding?**

* Answer

### **What is method overloading?**

* answer

### **What is Polymorphism? Explain with an example?**

* answer

### **What is Inheritance?**

* Answer

### **What is the interface in java?**

* Answer

### **Can we have a static method inside the interface?**

* Answer

### **Can we override the static method?**

* answer

### **We have Class A and Class B which contains m1() method in both the class, and class B extends parent class A. Asked which method will get called on which object?**

* answer

### **Explain SOLID Principles?**

The **SOLID** principles are the five key design principles in object-oriented programming (OOP) that help create **clean, maintainable, and scalable software**. Here’s a breakdown with **simple explanations** and **Java examples**:

🔵 **S – Single Responsibility Principle (SRP)**

**A class should have only one reason to change.**

**❌ Bad:**

class Report {

public String getReportData() { return "data"; }

public void saveToFile(String data) { /\* writes to file \*/ }

}

**✅ Good:**

class Report {

public String getReportData() { return "data"; }

}

class ReportSaver {

public void saveToFile(String data) { /\* writes to file \*/ }

}

**✅ Why?** Now Report handles only data, and ReportSaver handles saving — **separation of concerns**.

🟠 **O – Open/Closed Principle (OCP)**

**Software entities should be open for extension, but closed for modification.**

**❌ Bad:**

class ShapePrinter {

public void printShape(String shape) {

if (shape.equals("Circle")) { /\* draw circle \*/ }

else if (shape.equals("Square")) { /\* draw square \*/ }

}

}

**✅ Good:**

interface Shape {

void draw();

}

class Circle implements Shape {

public void draw() { System.out.println("Drawing Circle"); }

}

class Square implements Shape {

public void draw() { System.out.println("Drawing Square"); }

}

class ShapePrinter {

public void printShape(Shape shape) {

shape.draw();

}

}

**✅ Why?** You can add new shapes without changing ShapePrinter.

🟡 **L – Liskov Substitution Principle (LSP)**

**Subtypes must be substitutable for their base types.**

**✅ Example:**

class Bird {

public void fly() {

System.out.println("Bird is flying");

}

}

class Sparrow extends Bird {

@Override

public void fly() {

System.out.println("Sparrow is flying");

}

}

**❌ Violation:**

class Ostrich extends Bird {

@Override

public void fly() {

throw new UnsupportedOperationException("Ostrich can't fly");

}

}

**✅ Fix:** Use better hierarchy:

interface Bird {}

interface FlyingBird extends Bird {

void fly();

}

🟢 **I – Interface Segregation Principle (ISP)**

**Clients should not be forced to depend on methods they do not use.**

**❌ Bad:**

interface Worker {

void work();

void eat();

}

class Robot implements Worker {

public void work() {}

public void eat() {} // unnecessary for robot

}

**✅ Good:**

interface Workable { void work(); }

interface Eatable { void eat(); }

class Robot implements Workable {

public void work() {}

}

🔴 **D – Dependency Inversion Principle (DIP)**

**High-level modules should not depend on low-level modules. Both should depend on abstractions.**

**❌ Bad:**

class MySQLDatabase {

public void connect() {}

}

class App {

private MySQLDatabase db = new MySQLDatabase();

}

**✅ Good:**

interface Database {

void connect();

}

class MySQLDatabase implements Database {

public void connect() {}

}

class App {

private Database db;

public App(Database db) {

this.db = db;

}

}

🧠 **Summary:**

| **Principle** | **Description** |
| --- | --- |
| **S** | One class = One job |
| **O** | Add features by extending, not modifying |
| **L** | Derived classes should work like the base |
| **I** | Split large interfaces into smaller ones |
| **D** | Depend on abstractions, not implementations |

## **Collection Framework**

### **Difference between List and Array?**

| **Feature** | **Array** | **List (java.util.List)** |
| --- | --- | --- |
| Type | Data structure | Interface (implemented by ArrayList, LinkedList, etc.) |
| Size | Fixed (set at creation time) | Dynamic (can grow or shrink) |
| Syntax | int[] arr = new int[5]; | List<Integer> list = new ArrayList<>(); |
| Data Type Support | Both primitives and objects | Only objects (e.g., Integer, String) |
| Flexibility | Less flexible | More flexible (resize, insert, delete easily) |
| Memory Allocation | Contiguous memory | Uses internal structures like arrays or nodes |
| Performance | Slightly faster (no overhead) | Slightly slower due to dynamic nature |
| Index Access | Fast (O(1)) | Fast for ArrayList, slower for LinkedList |
| Useful Methods | No built-in methods (except via Arrays class) | Many built-in methods: add(), remove(), contains(), etc. |
| Part of Collections API | No | Yes |

**✅ When to Use What?**

| **Use Case** | **Prefer** |
| --- | --- |
| Fixed-size, performance-critical code | Array |
| Need dynamic resizing, easy insert/remove | List |
| Storing primitives (like int, char) | Array (or use List<Integer>, with boxing) |
| Use Collection API features | List |

**Example:**

📦 **Array:**

int[] numbers = new int[3];

numbers[0] = 10;

numbers[1] = 20;

System.out.println(numbers[1]); // Output: 20

📦 **List:**

List<Integer> numbers = new ArrayList<>();

numbers.add(10);

numbers.add(20);

System.out.println(numbers.get(1)); // Output: 20

### **Difference between List and Set?**

* answer

### **Difference between ArrayList and LinkedList?**

* Answer

### **Difference between Array and Collections?**

* Answer

### **Difference between HashMap and LinkedHashMap?**

* Answer

### **Difference between HashMap and HashTable?**

* answer

### **Difference between Fail-Fast and Fail-Safe iterator?**

* answer

### **Which is more efficient among the ArrayList and LinkedList?**

* Answer

### **Difference between Stream and Collections?**

* Answer

### **Write a syntax for integer ArrayList?**

* answer

### **How can Set not store duplicate values? Explain internal working?**

* Answer

### **Which collection you will use in order to remove duplicate elements and preserve the insertion order?**

* answer

### **Which Collection will you avoid duplicating and store data in a sorted manner?**

* answer

### **In which scenarios Linked list comes into fixture in HashMap?**

* answer

### **Which among ArrayList and LinkedList will be more efficient for random access of data?**

* Answer

### **Difference between Array, ArrayList and LinkedList? When you will use ArrayList and LinkedList?**

* answer

### **How to sort the ArrayList?**

* Answer

### **How to sort the ArrayList?**

* Answer

### **What are the practical use cases of LinkedList?**

* answer

### **Compare ArrayList objects from the employee and return the highest employee age?**

* Answer

### **How will you find the 3rd last element from the linked list in a single iteration?**

* answer

### **How does ArrayList internally work?**

* answer

### **What is the default size of ArrayList?**

* answer

### **How to increase the size of ArrayList? By how much?**

* Answer

### **Difference between Map and Set?**

* answer

### **How HashMap calls hashCode and equals method internally?**

* answer

### **Explain internal working of ArrayList?**

* answer

### **How does ArrayList increase its size dynamically?**

* answer

### **How does ArrayList grow?**

* Answer

### **Explain internal working of ensureCapacity method?**

private void ensureCapacityInternal(int minCapacity)

{

if (elementData == DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA) {

minCapacity = Math.max(DEFAULT\_CAPACITY, minCapacity);

}

ensureExplicitCapacity(minCapacity);

}

### **How do you write a custom ArrayList which will take the input capacity?**

answer

### **Which data structure is used by ArrayList?**

* answer

### **Difference between ArrayList and Vector?**

* Answer

### **Best time complexity of Arraylist and Linkedlist? Which is better?**

* Answer

### **What is HashMap?**

* HashMap<K, V> is a **part of java.util package** that implements the **Map interface** and allows you to store **key-value pairs** and allows efficient retrieval, insertion, and deletion operations.

Map<String, String> map = new HashMap<>();

map.put("Name", "Alice");

map.put("City", "New York");

**Key Characteristics**

1. **Implements Map Interface**: HashMap implements the java.util.Map interface.
2. **Key-Value Pairs**: Stores data as key-value pairs (like a dictionary).
3. **No Duplicate Keys**: Doesn't allow duplicate keys (each key maps to exactly one value).
4. **Permits Null Values**: Allows one null key and multiple null values.
5. **Not Thread-Safe**: Not synchronized by default.
6. **Unordered Collection**: Doesn't guarantee any specific order of elements.
7. **Dynamic Resizing**: Automatically resizes when the number of elements exceeds a threshold.

**Internal Data Structure:**

Internally, a HashMap is an array of Node<K,V>, called a bucket array:

transient Node<K,V>[] table;

**Important Parameters**

1. **Initial Capacity**: The number of buckets when the HashMap is created (default is 16).
2. **Load Factor**: The measure of how full the hash table is allowed to get before its capacity is increased (default is 0.75).
3. **Threshold**: When the size exceeds (capacity \* load factor), the hash table is rehashed.

**Constructors**

1. HashMap(): Default capacity (16) and load factor (0.75)
2. HashMap(int initialCapacity): Specified initial capacity, default load factor
3. HashMap(int initialCapacity, float loadFactor): Both specified
4. HashMap(Map<? extends K, ? extends V> m): Creates from another map

**Each node stores:**

static class Node<K,V> implements Map.Entry<K,V> {

final int hash;

final K key;

V value;

Node<K,V> next; // linked list

}

**Hashing and Index Calculation:**

When you call map.put(key, value):

1. hashCode() is called on the key.
2. A hash value is calculated (with some bit manipulation for better distribution).
3. An index is computed as:

index = (n - 1) & hash // where n is array length

1. Entry is stored in the bucket at that index.

**Collision Handling:**

A collision occurs when two keys hash to the same index.

* Java 7: uses Linked List
* Java 8+: uses Linked List → Tree (Red-Black Tree) if the list size > 8 and bucket size > 64

In case of collision:

* New node is appended to the list/tree.
* During retrieval, equals() is used to find the right key.

**Rehashing (Resizing)**

* When size > capacity \* load factor (default 16 \* 0.75 = 12), HashMap resizes.
* New array is double in size.
* All existing entries are rehashed and moved to new buckets.

**Common Methods:**

**Basic Operations:**

1. put(K key, V value): Associates the specified value with the specified key
2. get(Object key): Returns the value to which the specified key is mapped
3. remove(Object key): Removes the mapping for the specified key
4. containsKey(Object key): Returns true if this map contains a mapping for the key
5. containsValue(Object value): Returns true if this map maps one or more keys to the value
6. size(): Returns the number of key-value mappings
7. isEmpty(): Returns true if this map contains no key-value mappings
8. clear(): Removes all mappings

**Bulk Operations:**

1. putAll(Map<? extends K, ? extends V> m): Copies all mappings from the specified map
2. keySet(): Returns a Set view of the keys
3. values(): Returns a Collection view of the values
4. entrySet(): Returns a Set view of the mappings

**Example:**

Map<Integer, String> map = new HashMap<>();

map.put(1, "A");

map.put(2, "B");

map.put(3, "C");

System.out.println(map.get(2)); // Output: B

**Example Usage:**

import java.util.HashMap;

public class HashMapExample {

public static void main(String[] args) {

// Create a HashMap

HashMap<String, Integer> map = new HashMap<>();

// Add elements

map.put("Apple", 10);

map.put("Banana", 20);

map.put("Orange", 30);

// Access elements

System.out.println("Apple count: " + map.get("Apple")); // 10

// Check if key exists

System.out.println("Contains Banana? " + map.containsKey("Banana")); // true

// Remove element

map.remove("Orange");

System.out.println("After removal: " + map); // {Apple=10, Banana=20}

// Iterate through HashMap

for (String key : map.keySet()) {

System.out.println(key + ": " + map.get(key));

}

// Using entrySet() for iteration

for (Map.Entry<String, Integer> entry : map.entrySet()) {

System.out.println(entry.getKey() + " => " + entry.getValue());

}

}

}

**When to Use HashMap**

1. When you need fast access to elements by key
2. When insertion and retrieval operations are frequent
3. When you don't need to maintain insertion order
4. When thread safety is not a requirement

**Best Practices:**

* Choose **initial capacity** and **load factor** wisely if you expect large volume.
* Use immutable objects as keys to prevent unexpected behavior.
* Use **containsKey()** before using get() to avoid null confusion.
* Override equals() and hashCode() properly for custom key objects.
* Consider ConcurrentHashMap for multi-threaded environments.

**Real-World Use Cases:**

* Caching
* Indexing data for quick lookups
* Counting frequencies (e.g., word count)
* Mapping IDs to entities

Performance Considerations

* **Average Case**:
  + get(): O(1)
  + put(): O(1)
  + remove(): O(1)
  + containsKey(): O(1)
* **Worst Case** (if all keys collide):
  + get(): O(log n) in Java 8+ (due to tree conversion), O(n) before Java 8
  + put(): O(log n) in Java 8+, O(n) before Java 8
  + remove(): O(log n) in Java 8+, O(n) before Java 8

### **Explain internal working of HashMap?**

* A HashMap stores **key-value pairs** using a technique called **hashing**. It offers **O(1)** time complexity (on average) for put() and get() operations.

**🔧 Internal Components**

static class Node<K,V> implements Map.Entry<K,V> {

final int hash;

final K key;

V value;

Node<K,V> next; // for handling collisions

}

Internally, HashMap maintains:

* An **array of Node<K, V>[]** called table
* **Each Node** stores a key, value, hash, and reference to the next node (linked list)

**🔄 How put(key, value) Works**

**Step-by-step:**

1. **Compute Hash:**

int hash = hash(key);

Java enhances the hashCode with a **hash spreading function**:

static final int hash(Object key) {

int h;

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}

1. **Find Bucket Index:**

index = (n - 1) & hash;

n is the length of the array (must be a power of 2 for performance).

1. **Check Bucket:**
   * If **null**, create new node at index
   * If **occupied** (collision):
     + Traverse linked list or tree at that index
     + If key exists (checked via equals()), update value
     + Else, add new node (as next in list or as tree node)
2. **Resize if Needed:** If number of entries exceeds threshold = capacity \* loadFactor, rehash (resize).

**📦 How get(key) Works**

1. Compute hash of the key
2. Get index = (n - 1) & hash
3. Traverse list or tree at that index:
   * Compare hash and use equals() to find the correct key
4. Return value if key found, else return null

**⚠️ Collision Handling**

When two keys hash to the same index:

**Java 7 and before:**

* **Linked List** in each bucket

**Java 8 and after:**

* **Linked List** used **until 8 entries**
* When entries in a bucket > 8 and capacity > 64, it converts list to a **Red-Black Tree** for faster lookup

**🔁 Resizing (Rehashing)**

* When size > capacity \* loadFactor (default loadFactor = 0.75)
* Array is resized to **double the size**
* All entries are **rehashed** and **moved to new positions**

**🧠 Example**

HashMap<String, String> map = new HashMap<>();

map.put("name", "Alice");

map.put("city", "London");

1. "name".hashCode() → hash → index
2. Stored in table[index] as a node
3. If "city" hashes to same index, a **collision** occurs, and it's linked

**🔍 Summary of Internal Logic**

| **Operation** | **Description** |
| --- | --- |
| hashCode() | Generates hash for key |
| index = (n - 1) & hash | Finds correct bucket |
| equals() | Ensures key uniqueness |
| Collision | Resolved via Linked List or Tree |
| Resize | Doubles capacity and rehashes entries |

### **What is hash collision in HashMap?**

* A **hash collision** occurs in a HashMap when **two different keys** generate the **same hash value**, and thus, map to the **same bucket index** in the internal array.

**💡 Why do collisions happen?**

HashMap uses hashCode() to determine the bucket index.

Different keys can return the same hashCode() or map to the same index after applying the hashing formula:

index = (n - 1) & hash

Since the number of possible keys is much greater than the number of buckets (array length), collisions are inevitable.

**📦 Example:**

String key1 = "FB"; // hashCode = 2236

String key2 = "Ea"; // hashCode = 2236

Map<String, String> map = new HashMap<>();

map.put("FB", "Facebook");

map.put("Ea", "Electronic Arts");

Even though key1 and key2 are different, they may end up in the same bucket due to hash collision.

**⚙️ How HashMap handles collisions**

**Java 7 and earlier:**

Collisions are handled using a linked list.

New entry is added to the head or tail of the list at that index.

equals() is used to differentiate between keys.

**Java 8+:**

If the number of entries in a bucket exceeds 8, and capacity > 64:

The linked list is converted into a Red-Black Tree to improve performance (O(log n) instead of O(n)).

**🔍 Visual Representation**

Index 5: [Node(key=FB, value=Facebook)] -> [Node(key=Ea, value=Electronic Arts)]

🧠 **Summary**

| **Term** | **Meaning** |
| --- | --- |
| Hash Collision | Two keys map to the same bucket index |
| Cause | Same hashCode() or same computed index |
| Resolution | Linked List (Java 7), Tree (Java 8+) |
| Problem | Can degrade performance to O(n) if not handled efficiently |

### **What are the types of HashMap?**

* In Java, HashMap is a specific implementation of the Map interface, but there are **several variants** of Map implementations that behave similarly with some additional or modified behavior.

Let’s go through the **types of HashMap and its variations**:

**✅ 1. HashMap**

* **Default implementation**
* **Not thread-safe**
* Allows **1 null key** and **multiple null values**
* No ordering guaranteed

Map<String, String> map = new HashMap<>();

**✅ 2. LinkedHashMap**

* Extends **HashMap**
* Maintains insertion order or access order (if enabled)
* Slightly slower than **HashMap** due to ordering overhead

Map<String, String> map = new LinkedHashMap<>();

Useful when you want predictable iteration order.

**✅ 3. TreeMap**

* Implements **NavigableMap** interface
* **Sorted** by keys (natural order or comparator)
* **No null keys** (throws NullPointerException), but allows null values

Map<String, String> map = new TreeMap<>();

Internally uses a **Red-Black Tree**.

**✅ 4. Hashtable (Legacy)**

* Older synchronized version of HashMap
* **Thread-safe**, but **slower**
* **No null key or null values** allowed

Map<String, String> map = new Hashtable<>();

Rarely used in modern Java (replaced by ConcurrentHashMap).

**✅ 5. ConcurrentHashMap**

* Designed for **concurrent access** in multithreaded environments
* **Thread-safe**, high performance
* **Does not allow null keys or values**
* Uses **segment locking** in older versions; Java 8+ uses **CAS and synchronized blocks**

Map<String, String> map = new ConcurrentHashMap<>();

**🧠 Comparison Summary**

| **Feature** | **HashMap** | **LinkedHashMap** | **TreeMap** | **Hashtable** | **ConcurrentHashMap** |
| --- | --- | --- | --- | --- | --- |
| Thread-safe | ❌ | ❌ | ❌ | ✅ | ✅ |
| Null key allowed | ✅ (1) | ✅ (1) | ❌ | ❌ | ❌ |
| Ordering | ❌ | ✅ (insertion) | ✅ (sorted) | ❌ | ❌ |
| Performance | High | Slightly lower | Moderate | Low | High (for concurrent use) |

**🔧 Use Case Guidelines**

| **Use Case** | **Recommended Type** |
| --- | --- |
| Fast access, no thread safety | HashMap |
| Maintain insertion order | LinkedHashMap |
| Sorted map | TreeMap |
| Thread-safe (legacy) | Hashtable |
| Thread-safe (modern) | ConcurrentHashMap |

### **What happens in a HashMap when multiple keys have the same hashCode()??**

* If multiple keys return the same hashCode(), they will land in the **same bucket** inside the HashMap. This is known as a **hash collision**. To resolve this, Java uses the equals() method to check if the keys are actually the same or not.

If equals() returns true, the value is **updated**.

If equals() returns false, the new key-value pair is **added to the same bucket** (via a linked list or red- black tree).

### **Can HashMap store different keys that have the same hashCode?**

* Yes, as long as their equals() method returns false, Java treats them as distinct keys, even if their hashCode() is the same.

**Example:**

import java.util.HashMap;

import java.util.Objects;

class MyKey {

private String key;

public MyKey(String key) {

this.key = key;

}

// Override hashCode to always return same value

@Override

public int hashCode() {

return 100;

}

// equals compares actual content

@Override

public boolean equals(Object obj) {

if (this == obj) return true;

if (!(obj instanceof MyKey)) return false;

MyKey other = (MyKey) obj;

return Objects.equals(this.key, other.key);

}

@Override

public String toString() {

return key;

}

}

public class HashMapCollisionExample {

public static void main(String[] args) {

HashMap<MyKey, Integer> map = new HashMap<>();

MyKey key1 = new MyKey("A");

MyKey key2 = new MyKey("B");

MyKey key3 = new MyKey("C");

map.put(key1, 12);

map.put(key2, 12);

map.put(key3, 18);

System.out.println("Value for A: " + map.get(key1)); // 12

System.out.println("Value for B: " + map.get(key2)); // 12

System.out.println("Value for C: " + map.get(key3)); // 18

}

}

🧠 **What's Happening Internally:**

hashCode() is overridden to return 100 for all keys → all keys go to same bucket.

But equals() compares key string, so:

"A".equals("B") → false

"B".equals("C") → false

Hence, all keys are treated as distinct entries in the same bucket.

### **What if equals() returns true for two keys with same hashCode()?**

* Then it replaces the old value:

MyKey k1 = new MyKey("A");

MyKey k2 = new MyKey("A"); // same content

map.put(k1, 100);

map.put(k2, 200); // same key (based on equals), replaces value

System.out.println(map.get(k1)); // 200

**🔚 Summary**

| **Aspect** | **Behavior** |
| --- | --- |
| Same hashCode | Goes to same bucket |
| Different equals | Stored as different keys |
| Same equals | Treated as same key, old value replaced |
| Internal DS | Array + Linked List / Tree (Java 8+) to handle collisions |

### **Can we store null value in HashMap?**

* Yes, we can store null values in a HashMap.

Let’s break it down:

✅ **Null Key and Null Value in HashMap**

**🔹 Null Key:**

HashMap allows exactly one null key

When null is used as a key, it is always mapped to the bucket at index 0

Map<String, String> map = new HashMap<>();

map.put(null, "NullKeyValue");

System.out.println(map.get(null)); // Output: NullKeyValue

**🔹 Null Values:**

HashMap allows multiple keys to be mapped to null values

map.put("key1", null);

map.put("key2", null);

System.out.println(map); // Output: {null=NullKeyValue, key1=null, key2=null}

**⚠️ Be Careful:**

Other map types behave differently:

| Map Type | Null Key | Null Value |
| --- | --- | --- |
| HashMap | ✅ 1 key | ✅ many |
| LinkedHashMap | ✅ 1 key | ✅ many |
| TreeMap | ❌ | ✅ (some cases) |
| Hashtable | ❌ | ❌ |
| ConcurrentHashMap | ❌ | ❌ |

🔧 **Why is only one null key allowed?**

Internally, HashMap uses:

if (key == null) {

// handle specially

}

So it maps it to a special bucket (usually index 0) and ensures there's only one such entry.

### **Can we store infinite data in a HashMap?**

* **No**, we cannot store infinite data in a HashMap.

While HashMap theoretically can grow as long as memory permits (thanks to dynamic resizing), there are practical and system-level limitations that prevent it from holding "infinite" data.

**🚧 Limitations That Prevent Infinite Storage**

**1. Heap Memory Limit**

The biggest constraint is available heap memory (RAM).

Every entry (key + value + Node overhead) consumes memory.

Once heap space is exhausted, you'll get an:

java.lang.OutOfMemoryError: Java heap space

2**. Array Size Limit**

Internally, HashMap uses an array to store buckets.

The maximum array size in Java is Integer.MAX\_VALUE (2^31 - 1) entries — about 2.1 billion buckets.

**3. GC Overhead**

A massive number of objects leads to frequent garbage collection, degrading performance severely.

**4. Rehashing Overhead**

When the size exceeds the threshold (capacity \* loadFactor), HashMap resizes and rehashes — an expensive operation.

**🧠 Example**

Map<Integer, String> map = new HashMap<>();

for (int i = 0; ; i++) {

map.put(i, "value" + i);

}

This will eventually crash with: Exception in thread "main" java.lang.OutOfMemoryError: Java heap space

**✅ Best Practice**

Always design HashMap usage with estimated size and memory in mind.

Use initialCapacity wisely to reduce rehashing:

new HashMap<>(expectedSize, loadFactor);

### **Can we access null value in HashMap?**

* ✅ **Yes**, you can access a null value from a HashMap just like any other value.

Let’s break it down clearly 👇

✅ **Example: Accessing a null value in a HashMap**

import java.util.HashMap;

public class Main {

public static void main(String[] args) {

HashMap<String, String> map = new HashMap<>();

map.put("name", null); // Key with null value

map.put(null, "nullKeyValue"); // Null key with value

System.out.println(map.get("name")); // 👉 Output: null

System.out.println(map.get(null)); // 👉 Output: nullKeyValue

}

}

**📌 Key Notes:**

| **Operation** | **Result** |
| --- | --- |
| map.get("name") | Returns null (actual value is null) |
| map.get("unknown") | Returns null (key doesn't exist) |

**⚠️ Important Distinction:**

If map.get(key) returns null, you can't tell directly if:

the key exists and value is null, or

the key doesn't exist at all

To handle that, use:

if (map.containsKey("name")) {

System.out.println("Key exists with value: " + map.get("name"));

} else {

System.out.println("Key does not exist.");

}

🔍 **Summary**

| **Question** | **Answer** |
| --- | --- |
| Can we store null values in HashMap? | ✅ Yes |
| Can we access them? | ✅ Yes |
| What if key doesn't exist? | Returns null (same as if value is null) |
| How to differentiate? | Use containsKey(key) |

### **How hashcode calculations happened on the Employee object?**

* answer

### **Can we have duplicate keys in HashMap?**

* answer

### **How does HashMap come to know that two objects are the same or different?**

* Answer

### **What is the use of a ConcurrentHashMap?**

* answer

### **How does ConcurrentHashMap works?**

* answer

### **Difference between HashMap and ConcurrentHashMap?**

* Answer

### **Difference between HashMap and HashSet**

* Answer

### **What is the default size of ArrayList?**

* answer

### **How to increase the size of ArrayList?**

* answer

### **Can we add infinite elements to ArrayList?**

* answer

### **Can we store infinite elements in a list?**

* answer

### **How can we add data while iterating ArrayList?**

* answer

### **What is the difference between List and Map?**

* answer

### **Explain sorted collections in java?**

* answer

### **Which data structure is used by HashMap?**

* answer

### **How can Set can’t store duplicate data?**

* Answer

### **Explain HashSet?**

* A HashSet in Java is part of the java.util package and implements the Set interface. It is a collection that does **not allow duplicate elements** and does not maintain any **order** of its elements. HashSet is backed by a hash table (actually a HashMap), so it provides **constant time performance** (O(1)) for basic operations like add(), remove(), and contains(), assuming the hash function disperses the elements properly.

**Key Features of HashSet:**

**No Duplicates:** A HashSet does not allow duplicate elements. If you attempt to add a duplicate, it will not be added to the set.

**Unordered:** The elements in a HashSet are not ordered. There is no guarantee of the order in which elements are stored or retrieved. It may seem random because it is based on hash codes.

**Allows null values:** A HashSet allows at most one null element.

**Constant Time Complexity:** Operations like add(), remove(), and contains() have an average time complexity of O(1), making them very efficient.

**Common Methods:**

**add(E e):** Adds the specified element to the set. If the element already exists, it returns false, and the set is unchanged.

**remove(Object o):** Removes the specified element from the set if it exists.

**contains(Object o):** Returns true if the set contains the specified element.

**size():** Returns the number of elements in the set.

**isEmpty():** Returns true if the set is empty.

**clear():** Removes all elements from the set.

**iterator():** Returns an iterator over the elements in the set.

**Example:**

import java.util.HashSet;

public class HashSetExample {

public static void main(String[] args) {

// Create a HashSet

HashSet<String> set = new HashSet<>();

// Add elements to the HashSet

set.add("Apple");

set.add("Banana");

set.add("Cherry");

// Try adding a duplicate element

boolean isAdded = set.add("Apple"); // This will return false

System.out.println("Apple added: " + isAdded);

// Check if an element exists

System.out.println("Contains 'Banana': " + set.contains("Banana"));

// Remove an element

set.remove("Banana");

// Print all elements (order is not guaranteed)

System.out.println("Elements in set: " + set);

// Check the size of the HashSet

System.out.println("Size of set: " + set.size());

// Check if the HashSet is empty

System.out.println("Is set empty? " + set.isEmpty());

}

}

**Output:**

Apple added: false

Contains 'Banana': true

Elements in set: [Cherry, Apple] // The order is not guaranteed

Size of set: 2

Is set empty? false

**Important Points:**

**No Duplicates:** Even though we tried adding "Apple" twice, the second attempt didn't change the set.

**Unordered Collection:** The output doesn't guarantee the order of elements in the set. It can vary each time the program is run.

**Efficient Operations:** Operations like add(), remove(), and contains() are efficient because they are backed by a hash table.

**Internals of HashSet:**

Internally, a HashSet uses a HashMap to store its elements. The elements themselves are used as keys in the map, and all values are just dummy values (commonly Boolean.TRUE).

The hash codes of the elements determine their positions in the hash table, which is why the hashCode() method of the objects stored in a HashSet must be implemented properly to ensure efficient retrieval and storage.

### **Which data structure used by HashSet?**

* HashSet in Java is internally backed by a **HashMap**.

**✅ Key Points:**

* When you add an element to a HashSet, it is actually stored as a **key** in the internal HashMap.
* The **value** associated with each key is a constant dummy object — typically PRESENT = new Object().

**🔍 Internal Structure:**

private transient HashMap<E, Object> map;

private static final Object PRESENT = new Object();

**🔧 When you do:**

HashSet<String> set = new HashSet<>();

set.add("apple");

Under the hood, it works like:

map.put("apple", PRESENT);

**⚙️ Why HashMap?**

* HashMap uses **hashing** to store and retrieve keys efficiently.
* This allows HashSet to provide **average O(1) time complexity** for operations like add(), remove(), and contains().

**💡 Summary:**

| **Feature** | **Backed By** | **Storage Mechanism** |
| --- | --- | --- |
| HashSet | HashMap | Elements as keys, dummy Object as values |

So, HashSet is essentially a wrapper around HashMap that stores only the **keys**, ignoring the values.

### **If we have added the same key in HashMap with the same value and different key with the same hash with the same value, then how will it be stored? At what position data will store and how it will store? What will happen when we use ConcurrentHashMap?**

* answer

### **What is the default capacity of HashMap?**

* Answer

### **Where will you use ConcurrentHashMap?**

* answer

### **Explain HashMap vs ConcurrentHashMap?**

* answer

### **How is HashMap faster as compared to ConcurrentHashMap?**

* answer

### **Suppose I have multiple Employee objects with different names, so how does HashMap differentiate between different objects?**

* answer

### **What is a Blocking Queue?**

* answer

### **What is Enumeration in java?**

* In Java, **Enumeration** is an interface that was introduced in **Java 1.0** and is part of the **java.util** package. It is used to iterate over a collection of elements, but it has been largely replaced by more modern alternatives such as **Iterator** and **for-each loop**.

**Key Characteristics of Enumeration:**

* **Iterator for Legacy Classes**: Enumeration was primarily designed to iterate over **legacy collections** like **Vector**, **Stack**, and **Hashtable**.
* **Methods**: Enumeration has only two main methods:

1. **hasMoreElements()**: Checks if there are more elements in the collection.
2. **nextElement()**: Retrieves the next element in the collection.

**Basic Syntax and Usage of Enumeration:**

import java.util.\*;

public class EnumerationExample {

public static void main(String[] args) {

// Creating a Vector (legacy collection)

Vector<String> vector = new Vector<>();

vector.add("Apple");

vector.add("Banana");

vector.add("Cherry");

// Creating an Enumeration object from the Vector

Enumeration<String> enumeration = vector.elements();

// Using Enumeration to iterate through the Vector

while (enumeration.hasMoreElements()) {

System.out.println(enumeration.nextElement());

}

}

}

**Methods in Enumeration:**

1. **boolean hasMoreElements()**:

* Returns true if there are more elements in the collection.
* Returns false when all elements have been iterated through.

1. **E nextElement()**:

* Returns the **next element** in the collection.
* Throws **NoSuchElementException** if no more elements are available (when hasMoreElements() returns false).

**Limitations of Enumeration:**

* **No remove() Method**: Unlike **Iterator**, **Enumeration** doesn't have a remove() method. This means you can't remove elements during iteration.
* **Outdated**: It was replaced by **Iterator** (introduced in Java 1.2) as part of the **Java Collections Framework**. The **Iterator** interface has additional functionality like element removal, which **Enumeration** lacks.

**Comparison with Iterator:**

* **Enumeration** is older and is mainly used for legacy classes (like Vector, Hashtable).
* **Iterator** is part of the **Java Collections Framework**, and it is more flexible and powerful because it supports element removal and iteration through modern collections (like ArrayList, HashMap, etc.).

**Iterator Example (modern way):**

import java.util.\*;

public class IteratorExample {

public static void main(String[] args) {

// Using ArrayList (modern collection)

List<String> list = new ArrayList<>();

list.add("Apple");

list.add("Banana");

list.add("Cherry");

// Using Iterator to iterate

Iterator<String> iterator = list.iterator();

while (iterator.hasNext()) {

System.out.println(iterator.next());

}

}

}

**When to Use Enumeration:**

* **Legacy Code**: Enumeration is mostly used when working with older collections such as **Vector** and **Hashtable**.
* **New Projects**: It is recommended to use **Iterator** or **for-each loops** in new projects, as they offer more functionality and are better integrated with modern Java collections.

**Summary:**

* **Enumeration** is an interface used to iterate over legacy collections (like Vector and Hashtable).
* It has two main methods: hasMoreElements() and nextElement().
* It lacks the ability to remove elements during iteration (unlike Iterator).
* **Iterator** has replaced **Enumeration** in most use cases due to its extended functionality and better support for modern collections.

## **Exception Handling**

### **What is Exception Handling?**

* Answer

### **Explain the hierarchy of Exception?**

* answer

### **How to create Custom Exceptions?**

* Answer

### **How to handle IOException?**

* answer

### **Difference between Checked Exception and Unchecked Exception?**

* answer

### **How to handle unchecked exceptions?**

* answer

### **What will happen if I use Exception and custom exceptions together? Priority?**

* answer

### **Have you created a custom exception? What is the purpose of that?**

* answer

### **What is the use of finally block in exception handling?**

* answer

### **What is a try with resources?**

* answer

### **Have you come across the OutOfMemory exception? How will you handle it? Explain with standalone code?**

* Answer

### **If you have try, catch and finally block and if I have written another exception inside catch block so how control will flow?**

* Answer

### **Have you ever got outOfMemory exceptions? Can you tell me any scenarios where you have faced this exception?**

* answer

## **Java 8 Features**

### **Explain Java8 features?**

✅ **Java 8 Overview**

Java 8, released in **March 2014**, was a major release that introduced **functional programming**, **streams**, and **default methods**, making Java much more expressive and modern.

🔹 **1. Lambda Expressions (JEP 126)**

Introduces anonymous functions to make code more concise.

List<String> names = List.of("Alice", "Bob");

names.forEach(name -> System.out.println(name));

✅ Reduces boilerplate for iterators, callbacks, and functional logic.

🔹 **2. Functional Interfaces**

An interface with a single abstract method, used as the target for lambda expressions.

@FunctionalInterface

interface MyFunc {

void show();

}

✅ Examples: Runnable, Callable, Predicate, Function, Consumer

🔹 **3. Stream API (JEP 107)**

Provides a high-level abstraction for processing sequences of elements (like collections) in a declarative way.

List<String> result = names.stream()

.filter(n -> n.startsWith("A"))

.collect(Collectors.toList());

✅ Supports map, filter, reduce, collect, etc.

🔹 **4. Default and Static Methods in Interfaces**

Allows interfaces to have method implementations.

interface MyInterface {

default void show() {

System.out.println("Default Method");

}

static void print() {

System.out.println("Static Method");

}

}

✅ Enables interface evolution without breaking existing implementations.

🔹 **5. Method References**

Shorthand for calling existing methods.

names.forEach(System.out::println);

✅ Cleaner than writing lambdas when reusing existing methods.

🔹 **6. Optional Class**

A container to handle null values more gracefully.

Optional<String> name = Optional.ofNullable(getName());

name.ifPresent(System.out::println);

✅ Avoids NullPointerException and encourages functional style handling of absence.

🔹 **7. Date and Time API (java.time)**

A comprehensive replacement for legacy java.util.Date and Calendar.

LocalDate today = LocalDate.now();

LocalDate birthday = LocalDate.of(1990, Month.MARCH, 10);

✅ Immutable, thread-safe, and more readable.

🔹 **8. Nashorn JavaScript Engine**

Allows Java to execute JavaScript code.

ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");

engine.eval("print('Hello JavaScript')");

✅ Deprecated in later versions.

✅ **Summary Table:**

| **Feature** | **Description** |
| --- | --- |
| **Lambda Expressions** | Anonymous functions for concise code |
| **Functional Interfaces** | Target types for lambdas |
| **Stream API** | Declarative collection processing |
| **Default/Static Methods** | Method implementation in interfaces |
| **Method References** | Shorthand for lambda expressions |
| **Optional Class** | Avoids null pointer exceptions |
| **Date and Time API** | Modern date/time classes in java.time package |
| **Nashorn Engine** | JavaScript execution in Java (deprecated in later) |

### **What is Functional Interface?**

* Answer

### **Why we need Functional Interface?**

* Answer

### **Difference between normal interface and Functional Interface?**

* Answer

### **What is the default method in functional interface?**

* Answer

### **What is Lambda Expression?**

* answer

### **What is the prerequisite for Lambda Expression?**

* answer

### **What is the map function in stream?**

* answer

### **What is the filter in the stream?**

* answer

### **What is Stream API?**

* answer

### **What is the use of @FunctionalInterface annotation? Will it be checked manually or by the compiler?**

* answer

### **Difference between Map and FlatMap in stream?**

* Answer

### **Difference between Map and Filter in java 8?**

* Answer

### **What is the intermediate and terminal operator in Java stream API?**

* Answer

**Intermediate Operators:**

These operators are lazy, meaning they do not process the data immediately but instead return a new stream. They are used to transform or filter elements of the stream and can be chained together.

**Examples of intermediate operators include:**

**filter():** Filters elements based on a predicate.

**map():** Transforms each element into another form.

**flatMap():** Flattens a stream of streams into a single stream.

**distinct():** Removes duplicate elements.

**sorted():** Sorts the elements.

**peek():** Allows you to perform an action on each element, primarily used for debugging.

**Terminal Operators:**

These operators trigger the processing of the stream and produce a result or a side-effect. After a terminal operation is invoked, the stream is considered consumed and cannot be used further.

**Examples of terminal operators include:**

**collect():** Collects the elements into a collection like a list or a set.

**forEach():** Performs an action on each element.

**reduce():** Reduces the stream to a single value based on an accumulator function.

**count():** Returns the number of elements in the stream.

**anyMatch(), allMatch(), noneMatch():** Match operations that return a boolean.

**findFirst(), findAny():** Returns the first or any element from the stream.

**max(), min():** Returns the maximum or minimum element based on a comparator.

Intermediate operations are usually chained together, whereas terminal operations are the ones that conclude the stream processing.

### **Advantages of Stream over for loop?**

* answer

### **What is the lambda expression in java 8?**

* answer

### **How to use lambda expressions with one example?**

* answer

### **What is Stream and Parallel stream?**

* answer

### **Explain Optional class in java 8?**

* answer

### **What is the purpose of Optional class?**

* answer

### **Explain Java 8 Functional interface?**

* answer

### **Where can you use FlatMap in a stream?**

* answer

### **What is the lambda expression?**

* answer

### **What is the use map method in java 8 stream?**

* answer

### **How to sort the list of objects in java using Stream API?**

* answer

### **What is the inbuilt functional interface in Java 8?**

* Answer

### **What is Predicate and Consumer in java 8?**

**🔹 Predicate<T> – For Conditional Checks**

**Purpose:** Represents a condition (boolean-valued function) on an object of type T.

**Method:** boolean test(T t)

**Use case:** Filtering, validations, condition checks

**✅ Example:**

Predicate<String> isLongerThan5 = str -> str.length() > 5;

System.out.println(isLongerThan5.test("Hello")); // false

System.out.println(isLongerThan5.test("Welcome")); // true

**🔹 Consumer<T> – For Performing an Action**

**Purpose:** Performs an operation on an object of type T (no return value).

**Method:** void accept(T t)

**Use case:** Logging, printing, updating, applying changes

✅ **Example:**

Consumer<String> printUpper = str -> System.out.println(str.toUpperCase());

printUpper.accept("hello"); // Output: HELLO

✅ **Real-World Use in Streams:**

List<String> names = Arrays.asList("John", "Sam", "Jennifer", "Joe");

// Using Predicate to filter names longer than 3 characters

names.stream()

.filter(name -> name.length() > 3) // Predicate

.forEach(name -> System.out.println(name)); // Consumer

**🔁 Summary:**

| **Interface** | **Method** | **Return Type** | **Purpose** |
| --- | --- | --- | --- |
| Predicate<T> | test(T t) | boolean | Evaluate a condition |
| Consumer<T> | accept(T t) | void | Perform an action |

### **Difference between Predicate and Function?**

| **Feature** | **Predicate<T>** | **Function<T, R>** |
| --- | --- | --- |
| Purpose | Represents a condition (true/false check) | Represents a transformation or mapping |
| Method | boolean test(T t) | R apply(T t) |
| Return Type | boolean | Any type R (generic return type) |
| Use Case | Filtering, validations | Converting or transforming data |
| Used In | .filter() in streams | .map() in streams |

**✅ Predicate Example:**

Predicate<String> isLong = str -> str.length() > 5;

System.out.println(isLong.test("Hello")); // false

System.out.println(isLong.test("Welcome")); // true

**✅ Function Example:**

Function<String, Integer> getLength = str -> str.length();

System.out.println(getLength.apply("Hello")); // 5

System.out.println(getLength.apply("Welcome")); // 7

🧠 **Real Use in Streams:**

List<String> names = Arrays.asList("Tom", "Jonathan", "Amy");

// Using Predicate to filter names

names.stream()

.filter(name -> name.length() > 3) // Predicate<String>

.forEach(System.out::println);

// Using Function to transform names to lengths

List<Integer> lengths = names.stream()

.map(name -> name.length()) // Function<String, Integer>

.collect(Collectors.toList());

System.out.println(lengths); // [3, 8, 3]

**🔁 Summary Table:**

| **Concept** | **Predicate<T>** | **Function<T, R>** |
| --- | --- | --- |
| Input | T | T |
| Output | boolean (true/false) | R (any type) |
| Functional Method | test(T t) | apply(T t) |
| Stream Use | filter() | map() |

### **Explain Consumer and Supplier?**

🔹 **Consumer<T> – Takes input, returns nothing**

**Purpose:** Represents an operation that accepts a value and performs an action, but returns nothing.

**Method:** void accept(T t)

**Use Case:** Logging, printing, modifying objects, etc.

**✅ Example:**

Consumer<String> greet = name -> System.out.println("Hello, " + name);

greet.accept("John"); // Output: Hello, John

🔹 **Supplier<T> – Gives output, takes nothing**

**Purpose:** Represents a function that supplies a value without any input.

**Method:** T get()

**Use Case:** Providing default values, random numbers, object generation, etc.

**✅ Example:**

Supplier<String> supplyName = () -> "Anonymous";

System.out.println(supplyName.get()); // Output: Anonymous

🔁 **Comparison Table**

| **Feature** | **Consumer<T>** | **Supplier<T>** |
| --- | --- | --- |
| Input | Takes input of type T | Takes no input |
| Output | Returns nothing (void) | Returns a value of type T |
| Method | void accept(T t) | T get() |
| Use Case | Performing operations like logging | Generating or supplying values |
| Stream Use | .forEach() | Used to provide values to collections or APIs |

**📦 Real-World Example**

🎯 **Using Consumer to print list elements:**

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

names.forEach(name -> System.out.println("Name: " + name)); // Consumer

🎯 **Using Supplier to generate random numbers:**

Supplier<Double> randomSupplier = () -> Math.random();

System.out.println("Random: " + randomSupplier.get());

✅ **Summary**

| **Interface** | **Signature** | **Input** | **Output** | **Common Use Case** |
| --- | --- | --- | --- | --- |
| Consumer | accept(T t) | Yes | No | Print, update, store, log |
| Supplier | get() | No | Yes | Supply value, create object |

### **What is the use of Map?**

* answer

### **What is Optional class? What are the advantages of using Optional class?**

* answer

### **What are all functional interfaces present in java?**

* answer

### **Optional case returning null and String, how to handle it? What are the advantages of using it?**

* answer

### **Give an example of Functional Interface in java?**

* Answer

### **Suppose you are using lambda expressions and other than lambda expressions then are we getting any performance benefits?**

* answer

### **Can you elaborate on the Stream API, purpose and use of it?**

* answer

### **Can we have more than one abstract method in a functional interface?**

* answer

### **Have you worked on NullPointerException? How to handle NullPointerException in java 8?**

* answer

### **Difference between Parallel Stream and Serial Stream?**

**🔁 Serial Stream (Sequential Stream)**

Processes elements one by one in a single thread.

Follows the original order of the data source.

Safer for tasks where order matters or there are side effects.

**✅ Example:**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

numbers.stream()

.forEach(System.out::println); // Single-threaded

**⚡ Parallel Stream**

Processes elements concurrently using multiple threads (Fork/Join framework).

Can significantly improve performance for large datasets or CPU-intensive operations.

Order is not guaranteed unless explicitly handled.

Suitable for stateless, independent, and non-blocking operations.

**✅ Example:**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

numbers.parallelStream()

.forEach(System.out::println); // Multi-threaded

**🆚 Key Differences**

| **Feature** | **Serial Stream** | **Parallel Stream** |
| --- | --- | --- |
| Execution | Single-threaded | Multi-threaded (ForkJoinPool) |
| Performance | Slower for large data | Faster for large, CPU-heavy tasks |
| Order Preservation | Maintains order | Order is not guaranteed |
| Thread-Safety Required | Not always | Yes, thread-safety is important |
| Use Case | Small datasets, order matters | Large datasets, independent tasks |
| API Method | .stream() | .parallelStream() |

**⚠️ When to Use Parallel Stream?**

✅ **Use it when:**

* Data set is large
* Operations are stateless, non-blocking, and independent
* You want to utilize multi-core CPU

**❌ Avoid if:**

* Task involves shared mutable state
* Result depends on processing order
* It makes the program slower due to context switching or I/O delays

🔍**Preserve Order in Parallel Stream**

If you still want parallelism with order, use .forEachOrdered():

numbers.parallelStream()

.forEachOrdered(System.out::println); // Keeps order but reduces performance gain

## **Java 11 Features**

### **Explain Java 11 features?**

✅ Java 11 Features Overview

Java 11 (released in September 2018) is a Long-Term Support (LTS) version after Java 8 and introduced several important features and changes.

🔹 **1.** **Local-Variable Syntax for Lambda Parameters**

You can now use var in lambda expressions:

(var a, var b) -> a + b;

✅ Benefit: Useful when you want to add annotations to parameters.

🔹 **2.** New String Methods

Java 11 added several new utility methods to the String class:

| **Method** | **Description** |
| --- | --- |
| isBlank() | Checks if the string is empty or whitespace only |
| lines() | Converts a multiline string into a stream of lines |
| strip() | Removes leading and trailing white space (Unicode-aware) |
| stripLeading() | Removes leading whitespaces |
| stripTrailing() | Removes trailing whitespaces |
| repeat(int) | Repeats the string multiple times |

String str = " Java ";

System.out.println(str.isBlank()); // false

System.out.println("A\nB\nC".lines().count()); // 3

System.out.println(str.strip()); // "Java"

System.out.println("Hello ".repeat(3)); // Hello Hello Hello

🔹 **3.** **HTTP Client (Standardized)**

The new HTTP Client API introduced in Java 9 is now standard.

HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()

.uri(URI.create("https://example.com"))

.build();

HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());

System.out.println(response.body());

🔹 **4. Collection.toArray(IntFunction)**

New method to convert a collection to an array using a generator:

List<String> list = List.of("A", "B", "C");

String[] array = list.toArray(String[]::new);

🔹 **5. Notable Removals**

Java EE and CORBA Modules were removed.

E.g., Removed: javax.xml.bind, javax.activation, java.se.ee, etc.

🔹 **6. Launch Single-File Source Code**

You can run a .java file directly without compiling it manually:

java HelloWorld.java

✅ Useful for quick scripts and testing.

🔹 **7. Flight Recorder and Mission Control**

JDK Flight Recorder and Java Mission Control are now part of the JDK for better performance monitoring and diagnostics.

✅ **Summary Table:**

| **Feature** | **Description** |
| --- | --- |
| var in Lambda | Enables use of var in lambda parameters |
| New String Methods | Utility methods like isBlank(), lines() |
| Standard HTTP Client | Simplified modern HTTP communication |
| Collection.toArray() Overload | Converts collections to arrays efficiently |
| Removed Deprecated Modules | Java EE/CORBA APIs removed |
| Run .java file directly | Java source code launcher |

## **Java 17 Features**

### **Explain Java 17 features?**

✅ Java 17, released in September 2021, is an LTS (Long-Term Support) version like Java 11 and Java 8. It includes many enhancements, new features, and deprecations/removals aimed at improving performance, security, and developer productivity.

🔹 **1. Sealed Classes (Standard Feature)**

Sealed classes restrict which classes can extend or implement them.

public sealed class Vehicle permits Car, Bike {}

final class Car extends Vehicle {}

final class Bike extends Vehicle {}

✅ Use Case: Controlled inheritance for security and design clarity.

🔹 **2. Pattern Matching for instanceof (Standard Feature)**

Simplifies type checks and casting.

if (obj instanceof String s) {

System.out.println(s.toLowerCase());

}

✅ Cleaner and safer casting logic.

🔹 **3. Switch Expressions (Preview earlier, now stable)**

Allows switch to return values and have more concise syntax.

String result = switch (day) {

case MONDAY -> "Start of week";

case FRIDAY -> "Weekend is near";

default -> "Midweek";

};

✅ More readable and expressive control flow.

🔹 **4. Text Blocks (Introduced in Java 15, refined)**

Multi-line string literals with better formatting.

String json = """

{

"name": "ChatGPT",

"type": "AI"

}

""";

✅ Useful for JSON, SQL, XML, etc.

🔹 **5. New Pseudo-Random Number Generators (JEP 356)**

Provides new interfaces and implementations for random number generation.

RandomGenerator rand = RandomGeneratorFactory.of("L64X256MixRandom").create();

int num = rand.nextInt();

✅ More flexible and efficient random generation.

🔹 **6. Foreign Function & Memory API (Incubator)**

Allows Java programs to call native code and allocate memory outside the heap.

✅ Improves native interop without JNI.

🔹 **7. Enhanced NullPointerException Messages**

Helpful exception messages with more context.

Exception in thread "main" java.lang.NullPointerException:

Cannot invoke "String.length()" because "str" is null

✅ Great for debugging.

🔹 **8. Removal/Deprecation of Legacy Features**

Applet API deprecated.

Experimental AOT and JIT compilers removed.

RMI Activation removed.

🔹 **9. JEP 409: Sealed Interfaces**

Interfaces can also be sealed like classes.

public sealed interface Animal permits Dog, Cat {}

✅ **Summary Table:**

| **Feature** | **Description** |
| --- | --- |
| Sealed Classes & Interfaces | Restricts subclassing |
| Pattern Matching for instanceof | Cleaner type checks |
| Switch Expressions | Return values directly from switch |
| Text Blocks | Multiline string literals |
| Enhanced NPE Messages | Better error diagnostics |
| Foreign Function & Memory API | Safe native interop (Incubator) |
| New Random Generators | Flexible PRNG implementations |

## **Java 21 Features**

### **Explain Java 21 features?**

* Answer

✅ **Java 21 Overview**

Java 21, released in September 2023, is a Long-Term Support (LTS) version. It includes powerful new features, especially around pattern matching, virtual threads, and improvements in performance, syntax, and foreign APIs.

🔹 **1. Virtual Threads (JEP 444 - Final)**

Lightweight threads that help with high-throughput concurrent applications.

Thread.startVirtualThread(() -> {

// Non-blocking, lightweight task

System.out.println("Hello from a virtual thread");

});

✅ Great for scalable applications like servers and reactive systems.

🔹 **2. Record Patterns (JEP 440 - Final)**

Deconstruct records more concisely in pattern matching.

record Point(int x, int y) {}

void print(Object obj) {

if (obj instanceof Point(int x, int y)) {

System.out.println("X: " + x + ", Y: " + y);

}

}

✅ Improves destructuring of objects in conditionals.

🔹 **3. Pattern Matching for switch (JEP 441 - Final)**

Enhances switch expressions with type patterns.

static String format(Object obj) {

return switch (obj) {

case Integer i -> "int: " + i;

case String s -> "String: " + s.toUpperCase();

default -> "Unknown";

};

}

✅ Improves readability and reduces boilerplate.

🔹 **4. Sequenced Collections (JEP 431)**

New interface that adds consistent ordering to Collections (first(), last(), etc.).

SequencedCollection<String> names = new SequencedSet<>();

✅ Adds order-sensitive operations.

🔹 **5. String Templates (JEP 430 - Preview)**

Simplifies string construction with embedded expressions.

String name = "Java";

String message = STR."Welcome to \{name} 21!";

✅ Makes dynamic strings concise and readable.

🔹 **6. Scoped Values (JEP 446 - Preview)**

A safer alternative to thread-local variables, designed to work with virtual threads.

✅ Used for managing thread-local-like state safely in a structured way.

🔹 **7. Foreign Function & Memory API (JEP 442 - 3rd Preview)**

Modern and safe access to native code, replacing JNI.

✅ Supports calling native libraries and allocating off-heap memory.

🔹 **8. Deprecations & Removals**

Finalization is deprecated for removal.

Legacy APIs are cleaned and marked deprecated for future removal.

✅ **Summary Table:**

| **Feature** | **Description** |
| --- | --- |
| Virtual Threads | Lightweight threads for scalable concurrency |
| Record Patterns | Deconstruction of record types |
| Pattern Matching for switch | Type-safe, expressive switch |
| Sequenced Collections | Ordered collections with first/last semantics |
| String Templates (Preview) | Interpolated strings like ${} syntax |
| Scoped Values (Preview) | Safe alternative to ThreadLocal |
| Foreign Function & Memory API | Call native functions, work with memory safely |

### **What is the added version after Java 17 to Java 21?**

* answer

## **Multithreading**

### **What is multithreading in java?**

* answer

### **Difference between Process and Thread?**

* answer

### **Explain the Thread Life Cycle?**

* Answer

### **What is a deadlock? Explain with an example?**

* answer

### **What is a Synchronized Thread?**

* answer

### **Difference between synchronized and volatile keyword?**

* Answer

### **What happens if a thread calls notify() instead of notifyAll()?**

* Answer

### **How to create threads using java 1.7 and java 1.8?**

* answer

### **Difference between wait and sleep?**

* answer

### **What is Thread.sleep() method, and what does it do internally?**

* Answer

### **Difference between Thread and Runnable in Java?**

* Answer

### **What is significance of Executors in Java?**

* Answer

### **Explain the Producer Consumer problem and how would you implement it using wait(), notify() or BlockingQueue?**

* Answer

### **Difference between Callable and Runnable. When should you use each?**

* Answer

### **What is ForkJoinPool in Java and how does it differ from normal ExecutorService?**

* Answer

### **Explain Deadlock and how do you avoid it in Java Multithreading?**

* Answer

### **What are the ways of blocking the threads in java?**

* answer

### **What is thread synchronization in java?**

* answer

### **What are the methods of thread?**

* Answer

### **What is thread pool? What are the types of thread pool?**

* Answer

### **In Thread Pool Executor which data structure is used to store the request data?**

* In a **ThreadPoolExecutor**, the data structure used to store the tasks (requests) that need to be executed is typically a **blocking queue**. Specifically, the **BlockingQueue** is used to hold the tasks before they are picked up by available threads for execution.

**Common Data Structures for BlockingQueue:**

**LinkedBlockingQueue**:

This is the most commonly used implementation of a BlockingQueue in ThreadPoolExecutor.

It has an optional capacity limit, meaning it can grow dynamically (if the capacity is not specified) or be bounded with a fixed capacity.

It supports efficient thread-safe operations and allows multiple threads to enqueue and dequeue tasks safely.

If the queue is full and threads are available, the tasks will be added to the queue, and threads will continue to execute the tasks.

BlockingQueue<Runnable> queue = new LinkedBlockingQueue<>();

**ArrayBlockingQueue**:

This is a bounded blocking queue where the capacity is fixed and set during initialization.

It is backed by an array, and it is useful when you want to limit the number of pending tasks.

It can block threads when the queue is full and will prevent new tasks from being added if the capacity is reached.

BlockingQueue<Runnable> queue = new ArrayBlockingQueue<>(100); // Fixed capacity of 100 tasks

**SynchronousQueue**:

This is a special kind of blocking queue that does not store any tasks. It directly hands over the task from the submitter to the thread.

It is useful for high-throughput situations where you don’t need to store the tasks, and each task is handled by a thread immediately.

SynchronousQueue has a capacity of zero, so tasks can only be executed when threads are available to consume them.

BlockingQueue<Runnable> queue = new SynchronousQueue<>();

**PriorityBlockingQueue**:

This is a special kind of blocking queue where tasks are ordered based on their priority rather than the order in which they are submitted.

If tasks are submitted with priority information, this queue will process higher-priority tasks first.

BlockingQueue<Runnable> queue = new PriorityBlockingQueue<>();

**How It Works:**

**When a task is submitted to the executor**: The ThreadPoolExecutor puts the task into the queue (like a BlockingQueue).

**Thread Selection**: When a worker thread is idle or available, it takes the task from the queue and executes it.

**Blocking Behavior**: If the queue is full, the BlockingQueue will block the submitter until space is available (in the case of a bounded queue). Similarly, threads will wait if the queue is empty.

**Example:**

int cpuCount = Runtime.getRuntime().availableProcessors();

BlockingQueue<Runnable> queue = new LinkedBlockingQueue<>();

ThreadPoolExecutor executor = new ThreadPoolExecutor(

cpuCount \* 2, cpuCount \* 3, 60L, TimeUnit.SECONDS, queue

);

// Submit tasks

executor.submit(() -> System.out.println("Task 1"));

executor.submit(() -> System.out.println("Task 2"));

**Summary:**

The **BlockingQueue** (like LinkedBlockingQueue, ArrayBlockingQueue, SynchronousQueue, etc.) is the **data structure** used to store the tasks waiting to be executed in a ThreadPoolExecutor.

The exact implementation of the queue may vary depending on whether you want to limit the queue size, prioritize tasks, or handle concurrent task submissions efficiently.

### **When we setup a Thread Pool Executor so what will be the ideal count, how many thread we should keep in it?**

* The **ideal thread count** for a **ThreadPoolExecutor** depends on several factors, such as the nature of the tasks you're executing, the system's hardware capabilities (like the number of CPU cores), and the type of workload you're handling (I/O-bound vs CPU-bound). There's no one-size-fits-all answer, but I can help guide you to an optimal setup.

**Key Considerations:**

**CPU-bound tasks**:

These are tasks that perform intensive computations (e.g., number-crunching tasks).

Since CPU-bound tasks require significant CPU time and only one thread can execute on a single core at a time, the ideal number of threads is usually **equal to the number of CPU cores** available.

For example, if you have a 4-core CPU, a thread pool of **4 threads** is ideal, since adding more threads would not increase the throughput and could actually reduce performance due to excessive context switching.

**I/O-bound tasks**:

These tasks spend more time waiting for I/O operations, like reading files, making network requests, or waiting for database queries.

I/O-bound tasks are less dependent on the number of CPU cores because the threads often spend time in a waiting state rather than actively using the CPU. Therefore, the thread pool can have **more threads than the number of CPU cores**.

A common guideline is to have a thread pool size of **N + (N / 2)**, where N is the number of CPU cores. This accounts for the threads waiting on I/O and helps utilize CPU cores while waiting.

**General Guidelines:**

**For CPU-bound tasks**: Set the pool size to the **number of available CPU cores** (e.g., Runtime.getRuntime().availableProcessors()).

int cpuCount = Runtime.getRuntime().availableProcessors();

ExecutorService executor = new ThreadPoolExecutor(cpuCount, cpuCount, 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue<>());

**For I/O-bound tasks**: You can increase the thread pool size. A common rule is to set the pool size to **2 to 3 times the number of CPU cores**, especially if tasks are blocking on I/O.

int cpuCount = Runtime.getRuntime().availableProcessors();

ExecutorService executor = new ThreadPoolExecutor(cpuCount \* 2, cpuCount \* 3, 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue<>());

**Additional Factors:**

**Task Characteristics**:

**Short-running tasks**: If the tasks are short-lived, you might get away with a smaller pool.

**Long-running tasks**: For tasks that take a long time, increasing the number of threads might help to keep the CPU busy.

**System Resources**:

Having too many threads can lead to context switching overhead and can strain the system resources (memory, CPU, etc.).

The optimal number of threads should also consider available system resources. Too many threads can degrade performance.

**Example Setup:**

int cpuCount = Runtime.getRuntime().availableProcessors();

int poolSize = cpuCount \* 2; // For I/O-bound tasks, or you can adjust based on testing

ExecutorService executor = new ThreadPoolExecutor(

poolSize, poolSize, 60L, TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>()

);

**Monitoring and Tuning:**

**Benchmarking**: It's essential to test and benchmark different thread pool sizes for your specific workload. Sometimes, tuning parameters like keepAliveTime and queue size also matter.

**Load Testing**: Under high loads, you may need to adjust the pool size dynamically or adjust based on the system's behavior (e.g., monitoring queue size or thread utilization).

**Conclusion:**

For **CPU-bound** tasks: Set the thread pool size to the number of available CPU cores (cpuCount).

For **I/O-bound** tasks: Set the thread pool size to around **2 to 3 times the number of CPU cores**.

Test and monitor your system to find the most efficient configuration based on the task type and workload.

### **What are thread pool methods?**

* Answer

### **How to create threads in java?**

* answer

### **How to start a thread in java?**

* answer

### **How to create threads in two different ways?**

* answer

### **What is the necessity of thread synchronization?**

* answer

### **How to create a Thread pool in java?**

* answer

### **Which method you will override in order to create a thread?**

* answer

### **What are all methods present in Thread class?**

* answer

### **How do notify methods work in multithreading?**

* answer

### **What is synchronized in multithreading?**

* answer

### **Thread safety with Singleton class?**

* answer

### **Explain Thread executors?**

* answer

### **How to create three threads so that it will execute simultaneously?**

* Answer

### **What can we replace instead of this keyword in the statement? synchronized(this) {?**

* Answer

### **What is runAsync vs supplyAsync?**

* The difference between **supplyAsync()** and **runAsync()** in Java's CompletableFuture lies in whether the task you're executing returns a result or not.

**1. supplyAsync()**

**Purpose**: Used when you want to execute an asynchronous task **that returns a result**.

**Return Type**: CompletableFuture<T> where T is the type of the result returned by the task.

**Input**: Takes a Supplier<T> — a functional interface that provides a result (of type T).

**Usage**: Use this method when you need to perform an asynchronous operation that **computes a value** or **produces a result**.

**Example**:

CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {

// Simulate some computation

return 42;

});

future.thenAccept(result -> System.out.println("Result: " + result)); // Output: Result: 42

Here, supplyAsync runs the task in a separate thread and returns the result 42, wrapped in a CompletableFuture.

**2. runAsync()**

**Purpose**: Used when you want to execute an asynchronous task **that does not return a result**.

**Return Type**: CompletableFuture<Void> since there's no result being returned (void).

**Input**: Takes a Runnable — a task that performs some action without returning any result.

**Usage**: Use this method when you want to execute a task asynchronously without needing a result back from the task.

**Example**:

CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {

// Simulate some work

System.out.println("Task is running asynchronously");

});

future.thenRun(() -> System.out.println("Task is completed")); // Output: Task is running asynchronously

Here, runAsync executes the task asynchronously, but it doesn't return any value, and the CompletableFuture just represents the task’s completion.

**Summary of Differences:**

**supplyAsync()**:

* Used when you need to execute an asynchronous task that **returns a result**.
* It returns a CompletableFuture<T>, where T is the result of the task.

**runAsync()**:

* Used when you need to execute an asynchronous task that does not return a result.
* It returns a CompletableFuture<Void>, since the task doesn't return anything.

Both methods run tasks asynchronously, but **supplyAsync** handles tasks that **produce a result**, while **runAsync** handles tasks that perform some side effect without returning any value.

### **What is ReentrantLock in java?**

* A **ReentrantLock** in Java is a **mutual exclusion lock (mutex)** that is part of the **java.util.concurrent.locks** package. It provides **explicit locking** with advanced features compared to the traditional synchronized keyword.

**✅ Key Features of ReentrantLock:**

**Reentrant Behavior:**

A thread that **already holds the lock** can acquire it again **without getting blocked**.

This is the same behavior as the synchronized block.

Internally, the lock maintains a **hold count**, which tracks how many times the current thread has acquired the lock.

ReentrantLock lock = new ReentrantLock();

lock.lock();

try {

// critical section

lock.lock(); // Reentrant call (same thread)

try {

// nested critical section

} finally {

lock.unlock();

}

} finally {

lock.unlock();

}

**Manual Lock/Unlock:**

Unlike synchronized, you must **manually acquire and release** the lock using lock() and unlock().

Failing to call unlock() (e.g., in case of exceptions) can lead to deadlocks, so use it with try-finally.

**Try Lock (Non-blocking):**

tryLock() lets a thread **try to acquire the lock** without blocking. It returns immediately with true or false.

if (lock.tryLock()) {

try {

// do something

} finally {

lock.unlock();

}

} else {

// could not get the lock

}

**Timed Locking:**

tryLock(long timeout, TimeUnit unit) allows a thread to **wait for a certain period** to acquire the lock.

if (lock.tryLock(2, TimeUnit.SECONDS)) {

try {

// critical section

} finally {

lock.unlock();

}

}

**Interruptible Lock Acquisition:**

lockInterruptibly() allows a thread to be **interrupted** while waiting to acquire the lock.

lock.lockInterruptibly();

**Fair Locking:**

You can create a **fair ReentrantLock** that gives access to the longest-waiting thread:

ReentrantLock fairLock = new ReentrantLock(true); // fairness = true

**🔄 Difference between synchronized and ReentrantLock:**

| **Feature** | **synchronized** | **ReentrantLock** |
| --- | --- | --- |
| Reentrant | Yes | Yes |
| Try lock without blocking | No | Yes (tryLock()) |
| Timed lock | No | Yes (tryLock(timeout)) |
| Interruptible lock waiting | No | Yes (lockInterruptibly()) |
| Fairness | No (non-deterministic order) | Yes (optional fair mode) |
| Explicit unlock required | No | Yes (unlock()) |

**✅ Use Case:**

Use **ReentrantLock** when:

* You need **more control** over the locking mechanism.
* You need **try-lock**, **timed-lock**, or **interruptible-lock**.
* You want **fair thread scheduling** for lock acquisition.

### **Difference between lock and synchronized?**

* answer

### **What is Callable interface?**

* **Callable** is a functional interface used to execute concurrent tasks that return a result and can throw exceptions.
* It's most commonly used with ExecutorService and Future.
* The **Callable** interface is part of the **java.util.concurrent** package and is similar to the Runnable interface but with some key differences.

**✅ Key Features of Callable:**

* **Returns a result** — Unlike Runnable, which returns void, Callable returns a result of a specified type.
* **Can throw checked exceptions** — Callable.call() method can throw checked exceptions.
* **Used with ExecutorService** — It is typically used with ExecutorService to submit tasks asynchronously.

**Example:**

import java.util.concurrent.\*;

public class CallableExample {

public static void main(String[] args) throws Exception {

ExecutorService executor = Executors.newSingleThreadExecutor();

Callable<String> task = () -> {

Thread.sleep(1000);

return "Task completed!";

};

Future<String> future = executor.submit(task);

System.out.println("Result: " + future.get()); // Blocks until result is ready

executor.shutdown();

}

}

### **What is Runnable interface?**

* The **Runnable** interface is a **functional interface** in Java, designed to represent a task that can be executed by a thread. It is part of the **java.lang** package.
* Runnable is used to execute code concurrently in threads.
* It does **not return a value** and **cannot throw checked exceptions**.
* Often used when the task is simple and doesn't need a return result.

**✅ Key Features of Runnable:**

1. **Single Abstract Method**:
   * void run();
   * Contains no parameters and does not return any result.
2. **Used to Define a Thread Task**:
   * Runnable is often used to define the code that should execute in a separate thread.
3. **Does Not Return a Value**:
   * Unlike Callable, it doesn’t return any result or throw checked exceptions.
4. **Can Be Passed to a Thread**:
   * You can pass an instance of Runnable to a Thread constructor.

**Example:**

public class MyRunnable implements Runnable {

@Override

public void run() {

System.out.println("Running in a thread: " + Thread.currentThread().getName());

}

public static void main(String[] args) {

Thread thread = new Thread(new MyRunnable());

thread.start();

}

}

### **Difference between Future and CompletableFuture?**

* The main difference between Future and CompletableFuture in Java lies in their capabilities and flexibility when working with asynchronous tasks.

**1. Basic Overview:**

**Future:**

* Represents a task that will eventually complete and produce a result.
* It is **blocking**: You can get the result using methods like get(), but it will block the calling thread until the task completes.

**CompletableFuture:**

* A subclass of Future that allows you to write **non-blocking** asynchronous code.
* It can be **explicitly completed** (via complete()) and can be used to **compose asynchronous tasks** without blocking.

**2. Asynchronous Composition:**

**Future:**

* **Does not support chaining or combining asynchronous tasks**. It only allows the retrieval of the result via blocking calls (e.g., get()).

**CompletableFuture:**

* **Supports chaining** of multiple asynchronous tasks via methods like thenApply(), thenAccept(), and thenCombine().
* You can **combine multiple futures** into one using methods like thenCombine(), thenCompose(), and handle exceptions with exceptionally() or handle().
* **Non-blocking** methods that return a CompletableFuture allow you to write cleaner asynchronous workflows.

**3. Completion:**

**Future:**

* You cannot manually complete a Future. The task it represents is managed internally by a thread pool or an executor.
* It provides the get() method to wait for the completion of the task, but it is **blocking**.

**CompletableFuture:**

* You can **complete a CompletableFuture manually** using the complete() method. This is useful for controlling when a task is marked as completed.
* It supports more advanced features like **handling exceptions** and propagating them properly in the asynchronous chain.

**4. Exception Handling:**

**Future:**

* Future provides get() method, which throws **checked exceptions** like ExecutionException or InterruptedException.
* Exception handling is not as flexible as in CompletableFuture.

**CompletableFuture:**

* CompletableFuture allows you to handle exceptions in a more flexible way through methods like exceptionally(), handle(), or whenComplete().
* You can specify fallback logic if the task fails, without blocking the thread.

**5. Blocking vs Non-blocking:**

**Future:**

* The get() method is **blocking**: it will wait for the task to complete and then return the result.

**CompletableFuture:**

* Can be used in **non-blocking** ways, with methods like thenApply(), thenAccept() to perform computations asynchronously without blocking.

**6. Example:**

**Future Example:**

ExecutorService executor = Executors.newSingleThreadExecutor();

Future<Integer> future = executor.submit(() -> {

// Simulating long-running task

Thread.sleep(1000);

return 123;

});

try {

// Blocking call

Integer result = future.get();

System.out.println("Result: " + result);

} catch (Exception e) {

e.printStackTrace();

}

**CompletableFuture Example:**

CompletableFuture<Integer> completableFuture = CompletableFuture.supplyAsync(() -> {

// Simulating long-running task

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

return 123;

});

completableFuture

.thenApply(result -> result \* 2) // Chaining another task

.thenAccept(result -> System.out.println("Processed Result: " + result))

.exceptionally(ex -> {

System.out.println("Exception: " + ex.getMessage());

return null;

});

**7. Summary of Key Differences:**

| **Feature** | **Future** | **CompletableFuture** |
| --- | --- | --- |
| **Blocking vs Non-blocking** | Blocking (via get()) | Non-blocking (via methods like thenApply()) |
| **Manual Completion** | Cannot manually complete | Can be manually completed using complete() |
| **Exception Handling** | Limited and blocking | Flexible, with methods like exceptionally() |
| **Chaining/Composition** | No chaining support | Supports chaining of asynchronous tasks |
| **Usage** | Simple use cases | Complex asynchronous workflows with composition |

**When to Use:**

**Use Future:**

* For simple, blocking task results, where you need the result of an asynchronous task and can afford to wait.

**Use CompletableFuture:**

* When you want to write complex, non-blocking asynchronous logic that can be composed, chained, or handled with custom exception handling.

## **Scenario based Questions**

### **Scenario based question - 1?**

String s1 = “abc”; and String s2 = new String(“abc”); Where will s1 and s2 be stored?

* answer

### **Scenario based question - 2?**

What is the output of the given Java code?

public class Test

{

public static void main(String[] args)

{

method(null);

}

public static void method(Object o)

{

System.out.println("Object method");

}

public static void method(Integer i)

{

System.out.println("Integer method");

}

}

* Answer

### **Scenario based question - 3?**

What is the output?

Map<Integer, String> aMap = new HashMap<>();

Integer a = new Integer(20);

Integer b = 20;

aMap.put(a, "Blume");

aMap.put(b, "BlumeGlobal");

System.out.println(aMap.get(20));

System.out.println(aMap.get(new Integer(20)));

System.out.println(aMap.get(b));

System.out.println(aMap.get(a));

* answer

### **Scenario based question - 4?**

Modify below code in order to print the value of a and b?

class Account {

int a;

int b;

public void setData(int a, int b) {

a = a;

b = b;

}

public void showData() {

System.out.println("Value of A ="+a);

System.out.println("Value of B ="+b);

}

public static void main(String args[]) {

Account obj = new Account();

obj.setData(2,3);

obj.showData();

}

}

* answer

### **Scenario based question - 5?**

What will happen if we declare the list below?

List<Object> var = new ArrayList<String>;

* answer

## **JDBC**

### **What is JDBC?**

* answer

# ***Java Programs***

### **Program – 1**

Write a program in java to return max sum of contiguous subarray of size 3?

(Write algorithm steps as well)

Example. [2,1,5,1,3,2], where k=3 (max size of sub array)?

* answer

### **Program – 2**

Print even and odd numbers using thread

* Answer

### **Program – 3**

Write a method to return the maximum value from an integer array passed as an input parameter

public class FindLargestNumber

{

public static int returnLargetNumber(int arr[])

{

int i;

int largestNumber = arr[0];

for (i = 1; i < arr.length; i++)

if (arr[i] > largestNumber) {

largestNumber = arr[i];

return largestNumber;

}

public static void main(String[] args)

{

int arr[] = {55, 12, 0, 786, 98};

System.out.println("Largest number in given array : " + returnLargetNumber(arr));

}

}

### **Program – 4**

Problem Definition: Write a program to implement Singleton class?

* Answer

### **Program – 5**

Problem Definition: Write a program - method to return missing number from array of length n=3

Example: n=3, [2,0,3] -> Output -> 1

* Answer

### **Program – 6**

Problem Definition: Write a program in java to check if two strings are Anagram or not. Return boolean method

LISTEN - SILENT

Implement using collections.

import java.util.Arrays;

import java.util.HashMap;

public class Anagram {

public static boolean checkAnagramUsingArray(String str1, String str2) {

str1 = str1.replaceAll("\\s", "");

str2 = str2.replaceAll("\\s", "");

if (str1.length() != str2.length()) {

return false;

}

char[] str1Array = str1.toLowerCase().toCharArray();

char[] str2Array = str2.toLowerCase().toCharArray();

Arrays.sort(str1Array);

Arrays.sort(str2Array);

return Arrays.equals(str1Array, str2Array);

}

public static boolean checkAnagramUsingCollection(String str1, String str2) {

str1 = str1.replaceAll("\\s", "");

str2 = str2.replaceAll("\\s", "");

if (str1.length() != str2.length()) {

return false;

}

HashMap<Character, Integer> hm1 = new HashMap<Character, Integer>();

HashMap<Character, Integer> hm2 = new HashMap<Character, Integer>();

char[] str1Array = str1.toCharArray();

char[] str2Array = str2.toCharArray();

for (char value : str1Array) {

if (hm1.get(value) == null) {

hm1.put(value, 1);

} else {

int c = hm1.get(value);

hm1.put(value, ++c);

}

}

for (char c : str2Array) {

if (hm2.get(c) == null) {

hm2.put(c, 1);

} else {

int d = hm2.get(c);

hm2.put(c, ++d);

}

}

return hm1.equals(hm2);

}

public static void main(String[] args) {

String str1 = "LISTEN";

String str2 = "SILENT";

System.out.println(checkAnagramUsingArray(str1, str2));

System.out.println(checkAnagramUsingCollection(str1, str2));

}

}

### **Program – 7**

Problem Definition: Write a program to Swap two numbers without using a third variable

* Answer

### **Program – 8**

Problem Definition: Write a program to print even and odd numbers without using the modulus operator?

* Answer

### **Program – 9**

Problem Definition: Write a program to return the third largest element from an array using java 8 features?

* Answer

### **Program – 10**

Problem Definition: Write a program to print even and odd numbers using java 8 stream api?

* Answer

### **Program – 11**

Problem Definition: Write a program to print fibonacci series?

* Answer

### **Program – 12**

Problem Definition: Write a program to perform Reverse an array in groups of given size?

**Example:** [1,2,3,4,5,6,7] where k=3

**Output:** [5,6,7,1,2,3,4]

* Answer

### **Program – 13**

Problem Definition: Write a program to print factorials of natural numbers from 1 to 10

* Answer

### **Program – 14**

Problem Definition: Write a program to reverse the integer array without using loops? Using recursion?

* Answer

### **Program – 15**

Problem Definition: Write a program to perform Overriding methods in parent child

* Answer

### **Program – 16**

Problem Definition: Write a Java 8 code to return the student object if gender is male? (filter)

* Answer

### **Program – 17**

Problem Definition: Write a program using string, use only single for loop?

Input - String str = "Siddhant Patni";

Output - SiindtdahPata

* Answer

### **Program – 18**

Problem Definition: Write a program to print all the subsets of the given set with sum equal to given sum

set of numbers = {3, 35, 56, 2, 95, 10, 65, 150, 165, 23, 65, 18, 57}

sum = 28

* Answer

### **Program – 19**

Problem Definition: Write a program to count set bits in an integer

Example : 13 -> 1101 Print number of 1’s in the given binary number

* Answer

### **Program – 20**

Problem Definition: Write a program to balancing of the brackets

Example : Input - {[(a+b)+c]+x+y]}

* Answer

### **Program – 21**

Problem Definition: Write a program to print prime factors for given numbers?

* Answer

### **Program – 22**

Problem Definition: Write a program to print prime numbers between two numbers? Example: 24 to 100

* Answer

### **Program – 23**

Problem Definition: Write code to iterate ArrayList in different ways

* Answer

### **Program – 24**

Problem Definition: Write a program in java to reverse a singly linked list

* Answer

### **Program – 25**

Problem Definition: Write a program to print the first occurrence of a repeating element from integer Array?

Input: [6,10,7,8,9,7,11,12]

Output: 7

* Answer

### **Program – 26**

Problem Definition: Write a program to print occurrence of each character in the given string

Example: String str = "This is an interview is going on with Amdocs";

* Answer

### **Program – 27**

Problem Definition: Merge two sorted linked lists

Input: list1 = 10, 20,30,40;

list2 = 11,13,14,21,22,33,35;

Output: Mergedlist = 10,11,13,20,21,22,30,33,35,40

* Answer

### **Program – 28**

Problem Definition: Sort array with decreasing frequency of element

Input - 9,5,6,9,6,1,.2,9

Output - 9,9,9,6,6,1,2

* Answer

### **Program – 29**

Problem Definition: Write a program to remove the duplicate values from HashMap and return the max key from the HashMap.

* Answer

### **Program – 30**

Problem Definition: Write a program to return the maximum occurrence element from the array?

Input: int[] arr = {1,5,3,5,6,5};

Output: 5

* Answer

### **Program – 31**

**Problem Definition:** You are given a list of student test scores, where each entry contains a student’s name and their test score as a string array: [student\_name, test\_score]. Each student may appear multiple times in the list with different scores.

Your task is to write a Java function that calculates the **highest average score** among all students. If a student appears multiple times, their average score should be computed by averaging all of their scores. In case the average is a floating-point number, return the **floor value** of the average (i.e., round down to the nearest integer). If the input list is empty, return 0

* Answer

public class Solution {

public static int bestAverageGrade(String[][] scores) {

if (scores == null || scores.length == 0) return 0;

Map<String, List<Integer>> studentScores = new HashMap<>();

// Step 1: Populate the map with scores per student

for (String[] entry : scores) {

String name = entry[0];

int score = Integer.parseInt(entry[1]);

studentScores.putIfAbsent(name, new ArrayList<>());

studentScores.get(name).add(score);

}

int bestAverage = Integer.MIN\_VALUE;

// Step 2: Calculate averages and track the max

for (Map.Entry<String, List<Integer>> entry : studentScores.entrySet()) {

List<Integer> marks = entry.getValue();

int sum = 0;

for (int score : marks) {

sum += score;

}

int average = (int) Math.floor((double) sum / marks.size());

bestAverage = Math.max(bestAverage, average);

}

return bestAverage == Integer.MIN\_VALUE ? 0 : bestAverage;

}

// Optional main method for testing

public static void main(String[] args) {

String[][] input = {

{"Bobby", "87"},

{"Charles", "100"},

{"Eric", "64"},

{"Charles", "22"}

};

System.out.println(bestAverageGrade(input)); // Output: 87

}

}

### **Program – 32**

Problem Definition: Write a program to return the number of occurrences for a given word in the string.

Accept the inputs from the user.

* Answer

### **Program – 33**

Problem Definition: Write a java 8 program to print words which start with 'S'?

Example: List<String> list = Arrays.asList("Siddhant", "Patni", "Kotak", "Mahindra");

* Answer

### **Program – 34**

Problem Definition: Write a program to Swap two numbers without using a third variable?

* Answer

### **Program – 35**

Problem Definition: Write a program to check Palindrome number. Input : 59095

* Answer

### **Program – 36**

Problem Definition: Write a program to reverse String without using reverse function (Use brute force approach)

* Answer

### **Program – 37**

Problem Definition: Write a program to display the count of each word in a string?

Input: str = "w1 w2 w2 w3 w3 w4”

* Answer

### **Program – 38**

Problem Definition: Write a java 8 program to make all elements of the list to uppercase and create a new ArrayList

* Answer

public static void main(String[] args) {

//String str = "w1 w2 w2 w3 w3 w4";

List<String> list = Arrays.asList("w1", "w2", "w2","w3","w4");

List<String> newList = list.stream().map(str -> str.toUpperCase()).forEach(str ->

System.out.println(str + " "));

}

### **Program – 39**

Problem Definition: Write a program in java to print PreOrder Traversal

Input: Given the root of a binary tree, return the preorder traversal of its nodes' values.

* Answer

### **Program – 40**

Problem Definition: Write a program to calculate average marks of students and return the student object with average marks

* Answer

### **Program – 41**

Problem Definition: Write a program to get the count of each character in string

* Answer

### **Program – 42**

Problem Definition: Write a java 8 code to get the age greater than 15 from the Employee object?

Input - Employee

id

name

Age

* Answer

### **Program – 43**

Problem Definition: Write a program to get a list of files and directories by passing the path of the directory

* Answer

### **Program – 44**

Problem Definition: What is the output of the below snippet if we print str?

String str =”India”;

str = “Bharat”;

* Answer

### **Program – 45**

Problem Definition: Write a program to print Fibonacci series of nth number?

* Answer

### **Program – 46**

Problem Definition: Write a program to get the count of each word in a given string?

* Answer

### **Program – 47**

Problem Definition: Write a program - How to check whether a number is an Armstrong or not?

* Answer

### **Program – 48**

Problem Definition: Write a program to print a possible set of palindromes of a given string?

Input: ABBAAAAABBA

Output: B, BB, AA, AAA, AAAA, AAAAA, ABBA, BAAAAAB, BBAAAAABB,

ABBAAAAABBA

* Answer

### **Program – 49**

Problem Definition: Write a program to print the reverse of a given number? Input - 1234

* Answer

### **Program – 50**

Problem Definition: Write a program to return the minimum number of platforms required for a train?

Input:

12:05 12:50

13:10 14:25

05:30 07:25

05:35 05:55

02:15 02:20

* Answer

public static int getPlatformDetails(int[] arrival, int[] departure)

{

Arrays.sort(arrival);

Arrays.sort(departure);

int platform=0, finalPlatform=INT\_MIN, i=0, j=0, count =0;;

while(i <arrival.length)

{

if(arrival[i] < departure[j])

{

Platform = Integer.max(platform, ++count);

i++;

} else

{

Count --;

j++;

}

finalPlatform = Integer.max(platform, ++count);;

//here update result

}

return finalPlatform;

for(int i=0; i<num; i++)

{

for(int j= i +1; j< num;j++)

{

if(arrival[i] >= arrival[j] && arrival[i] <= departure[j] )

}

}

}

Public static void main(String[] args)

{

int arrival[] = {1205, 1310, 0530,0535 0215};

int departure[] = {1250, 1425, 0725, 0555, 0220};

int num = 5;

System.out.println(“Platform required are :”+getPlatformDetails(arrival, departure, num));

}

### **Program – 51**

Problem Definition: Write a Java program to count islands in a boolean 2D matrix.

Input :

0 0 1 10

0 0 1 0 0

1 0 0 0 0

0 1 0 0 0

1 1 0 1 0

3 connected island

* Answer

### **Program – 52**

Problem Definition: Write a program to shorten the given URL?

Input: https://docs.google.com/document/d/1g1CwT8dhCGgjh1LtXSaRBZ1ctREqkldD76lAU4KxPOo/edit

* Answer

### **Program – 53**

Problem Definition: Write a java program to find element closest to given target

Input - int arr[] = { 1, 2, 4, 5, 6, 6, 8, 9 };

int target = 11;

Output - 9

* Answer

### **Program – 54**

Problem Definition: If I want to remove the duplicate values from ArrayList then how can we achieve it using java 8?

List<Integer> list = new ArrayList<>(Arrays.asList(1, 10, 1, 2, 2, 3, 10, 3, 3, 4, 5, 5));

List<Integer> newList = list.stream().distinct().collect(Collectors.toList());

### **Program – 55**

Problem Definition: If you have an array of integers then how will you sort it? Write actual logic without using inbuilt Methods?

### **Program – 56**

Problem Definition: Write a Java function to count the number of distinct ways to reach the nth stair by climbing either 1 or 2 steps at a time, and return the result modulo 10^9+7.

public class StairClimber {

static final int MOD = 1\_000\_000\_007;

public static int countWays(int n) {

if (n <= 1) return 1;

int prev = 1, curr = 1;

for (int i = 2; i <= n; i++) {

int temp = (prev + curr) % MOD;

prev = curr;

curr = temp;

}

  return curr;

}

// Test the function

public static void main(String[] args) {

int n1 = 4;

int n2 = 10;

System.out.println("Ways to climb " + n1 + " stairs: " + countWays(n1)); // Output: 5

System.out.println("Ways to climb " + n2 + " stairs: " + countWays(n2)); // Output: 89

}

}

### **Program – 57**

Problem Definition: There is a stair having total no of stairs as n. Requirement is to reach at the top in minimum hop. Each step has some value associated with it which shows how many steps you can move forward from that step. Find the minimum hop required to reach from 0th step to nth step. n= 6 and stairs = {2,0,1,3,1,1};

* Answer

### **Program – 58**

Problem Definition: How will you sort the Student data based on their Age?

* Answer

### **Program – 59**

Problem Definition: Explain the logic of frequency of each character of String?

* Answer

### **Program – 60**

Problem Definition: Write a code to print the count of unique characters from the String str = “Siddhant Patni”?

* Answer

### **Program – 61**

Problem Definition: Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that:

i, j, and k are distinct indices, nums[i] + nums[j] + nums[k] == 0. The solution set must not contain duplicate triplets. Input: nums = [-1,0,1,2,-1,-4]

Output: [ [-1,-1,2], [-1,0,1] ]

What is Time complexity and Space complexity?

How can you optimize the time complexity?

* Answer

### **Program – 62**

Problem Definition: Input: nums = [2,7,11,15], target = 9 Output: [0,1]

Explanation: nums[0] + nums[1] = 2 + 7 = 9, so return [0, 1].

How will you optimize the time complexity?

What is space complexity?

* Answer

### **Program – 63**

Problem Definition: Explain the output line by line in below code?

class Parent

{

public Parent()

{

System.out.println("in parent Constructor");

}

public void ml()

{

System.out.println("in parent method m1");

}

public static void m2()

{

System.out.println("in parent method m2");

}

}

Class Child extends Parent

{

Public Child()

{

System.out.println("in child Constructor");

}

public void ml()

{

System.out.println("in child method m1");

}

public static void m2()

{

System.out.println("in parent method m2");

}

}

Class Main

{

public static void main (String[] args)

{

Parent p = new Parent();

p.ml();

p.m2();

Child c = new Child();

c.ml();

c.m2();

Child ch = new Parent();

ch.ml();

ch.m2();

Parent pl= new Child();

pl.ml();

pl.m2();

}

}

* Answer

### **Program – 64**

Problem Definition: Write a code to add into the list and it should sort on the basis Address?

class Emp {

int id, str name , Address addr

Emp(id,name, addr){id= this.id, name =this.name, this.addr= addr}

}

class test{

Emp e1 = new Emp(1,'A','pune');

Emp e2 = new Emp(1,'A','nagpur');

.

.

Emp e100 = new Emp(1,'A','zz');

* Answer

### **Program – 65**

Problem Definition: What is the output?

public class Employee {

int id; String name;

Employee(int id, String name){

this.id = id;

this.name = name;

//this.addr = add;

}

}

class test{

public static void main(String[] args)

{

Employee e1 = new Employee(1,"A");

Employee e2 = new Employee(1,"A");

Employee e3 = new Employee(1,"A");

Employee e4 = new Employee(1,"A");

Employee e5 = new Employee(1,"A");

HashMap<Employee,Employee> h = new HashMap();

h.put(e1,e1);

.

.

h.put(e5,e5);

System.out.println("HS size: "+h.size());--1

}

}

* Answer

### **Program – 66**

Problem Definition: What is the output?

public class Employee {

int id; String name;

Employee(int id, String name){

this.id = id;

this.name = name;

//this.addr = add;

}

}

class test{

public static void main(String[] args)

{

Employee e1 = new Employee(1,"A");

Employee e2 = new Employee(1,"A");

Employee e3 = new Employee(1,"A");

Employee e4 = new Employee(1,"A");

Employee e5 = new Employee(1,"A");

HashSet<Employee,Employee> h = new HashSet();

h.add(e1,e1);

.

.

h.add(e5,e5);

System.out.println("HS size: "+h.size());--1

}

}

* Answer

### **Program – 67**

Problem Definition: What will be the output of the below program?

class Simple

{

Simple ( )

{

System.out.println("l am in Simple");

}

public static void main (string[] args)

{

Simple s = new Simple() ;

}

}

* Answer

### **Program – 68**

Problem Definition: Write a program to find the first occurrence of non-repeating character in String

Input: “aabbcdec”

Output: d

* Answer

### **Program – 69**

Problem Definition: Develop a Java program that models a Cinema Booking System with the following

Requirements:

An Auditorium containing multiple Screens.

Each Screen hosts multiple Movie Shows.

Each Movie Show offers seating in three categories: Gold, Silver, and Platinum.

Implement functionality to book tickets, check seat availability, and include additional relevant

methods to enhance the booking system.

* Answer

### **Program – 70**

Problem Definition: Movie booking system related java codes, check availability, book ticket with multiple classes and use of data structures?

* Answer

### **Program – 71**

Problem Definition: Explain the implementation of the User registration method using java? Explain how will you implement it?

* Answer

### **Program – 72**

Problem Definition: Write a method to register and check the available user in repo?

* Answer

### **Program – 73**

Problem Definition: How can you handle WhatsApp notification and mobile notification after user registration?

* Answer

### **Program – 74**

Problem Definition: Write a Java code, we have Product and Feature class with Product having model, color, price, location and List<Feature> and Feature class having picture and resolution fields, Write a code using Java 8 Stream API?

1. Print Product which is iPhone 16

2. get the product and its feature which has highest resolution

3. Print number of iPhone15 and iPhone16 count in each location

* Answer

### **Program – 75**

Problem Definition: We have 2 Employee object and if I add these objects in Set and check for the size then what will be the output 1 or 2?

Employee e1 = new Employee(“Swapnil”);

Employee e2 = new Employee(“Swapnil”);

* Answer

### **Program – 76 – Person Management System, Write model classes**

Problem Definition: You are required to design a basic **Person Management System** in Java that models different types of people within an organization. The system should be able to manage and perform common operations for various person types such as **Managers** and **Drivers**.

Requirements:

Person

- Manager

- Driver

Attributes

- name

- age

- don

- Address

methods

- eat

- doSleep

- doWork

public class PersonManagementSystem {

// Base class

static class Person {

protected String name;

protected int age;

protected String don; // Date of Joining

protected String address;

public Person(String name, int age, String don, String address) {

this.name = name;

this.age = age;

this.don = don;

this.address = address;

}

public void eat() {

System.out.println(name + " is eating.");

}

public void doSleep() {

System.out.println(name + " is sleeping.");

}

public void doWork() {

System.out.println(name + " is working.");

}

}

// Manager class

static class Manager extends Person {

public Manager(String name, int age, String don, String address) {

super(name, age, don, address);

}

@Override

public void doWork() {

System.out.println(name + " is managing the team.");

}

}

// Driver class

static class Driver extends Person {

public Driver(String name, int age, String don, String address) {

super(name, age, don, address);

}

@Override

public void doWork() {

System.out.println(name + " is driving the vehicle.");

}

}

// Main method

public static void main(String[] args) {

Manager manager = new Manager("Alice", 35, "2020-01-01", "New York");

Driver driver = new Driver("Bob", 28, "2022-03-05", "Los Angeles");

System.out.println("--- Manager Actions ---");

manager.eat();

manager.doSleep();

manager.doWork();

System.out.println("\n--- Driver Actions ---");

driver.eat();

driver.doSleep();

driver.doWork();

}

}

### **Program – 77 -** Flatten a List of Lists using Java 8 Streams

List<lnteger> listl = new ArrayList<>();

List<lnteger> 11 = Arrays.asList(1,2,3);

List<lnteger> 12 = Arrays.asList(4,5,6);

List<lnteger> 13 = Arrays.asList(7,8,9);

listl.addA11(11);

listl.addA11(12);

listl.addA11(13);

System. out. println(listl);

* Answer

import java.util.\*;

import java.util.stream.Collectors;

public class FlattenListExample {

public static void main(String[] args) {

List<Integer> l1 = Arrays.asList(1, 2, 3);

List<Integer> l2 = Arrays.asList(4, 5, 6);

List<Integer> l3 = Arrays.asList(7, 8, 9);

List<List<Integer>> listOfLists = new ArrayList<>();

listOfLists.add(l1);

listOfLists.add(l2);

listOfLists.add(l3);

// Flattening using flatMap

List<Integer> flatList = listOfLists.stream()

.flatMap(List::stream)

.collect(Collectors.toList());

System.out.println("Flattened List: " + flatList);

}

}

### **Program – 78 - How to find the second largest element from the list?**

**1. Java 8 Streams Approach (Cleanest):**

import java.util.\*;

public class SecondLargest {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(5, 2, 9, 1, 7, 9);

Optional<Integer> secondLargest = numbers.stream()

.distinct()

.sorted(Comparator.reverseOrder())

.skip(1)

.findFirst();

secondLargest.ifPresentOrElse(

val -> System.out.println("Second Largest: " + val),

() -> System.out.println("Not enough distinct elements")

);

}

}

✅ **2. Classic Iterative Approach (Single Pass, O(n)):**

public class SecondLargest {

public static void main(String[] args) {

int[] arr = {5, 2, 9, 1, 7, 9};

Integer largest = null;

Integer second = null;

for (int num : arr) {

if (largest == null || num > largest) {

second = largest;

largest = num;

} else if ((second == null || num > second) && num != largest) {

second = num;

}

}

if (second != null) {

System.out.println("Second Largest: " + second);

} else {

System.out.println("No second largest element found.");

}

}

}

# Advance Java / J2EE

### **What is servlet?**

* Answer

### **Explain servlet lifecycle?**

* Answer

### **What is a dispatcher servlet?**

* answer

### **What is JSP?**

* answer

### **What is the difference between servlet and JSP?**

* answer

### **What is the JSP lifecycle?**

* answer

### **What is Spring MVC? Explain architecture of spring MVC?**

* Answer

### **What is the singleton design pattern?**

* answer

### **What is the REST API?**

* answer

### **Difference between API and REST?**

* answer

### **Difference between SOAP and REST?**

* answer

### **Which among SOAP and REST is easy?**

* answer

### **Explain REST architecture?**

* answer

### **What do you know about DTD and XSD?**

* answer

### **Explain SOAP Web Service?**

* answer

### **What is Path in XML?**

* answer

### **Write a path for a given xml file?**

* answer

### **What is the use of XSLT?**

* answer

### **How does SOAP work?**

* answer

### **What are the action types of http?**

* answer

### **Explain how you will create a SOAP web service?**

* answer

# Data Structure

### **Find the middle element from the singly linked list without finding the length?**

* Answer

### **Complexity to find length of singly linked list?**

* Answer

### **Do you know how LinkedList works? Can you create a data structure for LinkedList?**

* answer

### **How will you traverse LinkedList? Your List will contains multiple nodes so how will you traverse over it? Can you write a pseudo code? You have your head with you so how will you move to the next node? How will the current node be null?**

* answer

### **Explain Merge Sort??**

* answer

# Design Patterns

### **What is the singleton design pattern? Explain it?**

* answer

### **Explain Factory design pattern?**

* answer

### **What is the Observer design pattern?**

* answer

### **Except for the creational design pattern, can you explain other design patterns?**

* answer

### **Explain Saga design pattern?**

* answer

# Spring

### **Explain Spring MVC architecture?**

* Answer

### **Explain Spring IOC?**

* answer

### **Explain Spring Dependency Injection?**

* answer

### **What are the spring bean scopes?**

* answer

### **Explain any scope of spring bean?**

* Answer

### **How to add scope in bean configuration?**

* Answer

### **Difference between session and prototype scope?**

* Answer

### **What is the Spring batch? Explain where you have used the spring batch in your project?**

* answer

### **How spring batch works?**

* answer

### **How to implement the spring batch?**

* Answer

### **What is the difference between SOAP and REST? Which is more efficient?**

* Answer

### **How REST API works?**

* Answer

### **Difference between Spring and Spring Boot?**

* Answer

### **Difference between application context and beanfactory in spring?**

* Answer

### **Explain Spring MVC architecture?**

* Answer

### **Difference between Spring and Spring Boot?**

* Answer

### **What is the use of @Transactional annotation in spring?**

* Answer

### **What are different stereotypes annotations in spring?**

* Answer

### **What is the front controller?**

* answer

### **Difference between applicationcontext and beanfactory?**

* answer

### **Explain spring bean life cycle?**

* answer

### **If I have prototype bean and singleton bean, then how can we call prototype beans inside singleton bean?**

* answer

### **What are containers present in the spring?**

* answer

### **How transactions work in spring?**

* answer

### **Do you know the web.xml file? What is the significance of a web.xml file?**

* answer

### **What is the dependency injection?**

* answer

### **What are the different ways of injection?**

* answer

### **Who will take care of objects in Spring?**

* answer

### **What is ApplicationContext and its types in Spring?**

* answer

### **Difference between Filter and Interceptor in Spring MVC?**

* answer

### **Which one has access to spring beans, interceptors or filters?**

* answer

### **Why do interceptors have access to Spring beans?**

* answer

### **Difference between Spring data JPA, Hibernate, JDBC, My batis?**

* answer

### **Why do we use Spring data JPA?**

* answer

# Spring Security

### **What is Spring Security?**

* Spring Security is a **powerful and customizable authentication and access control framework** for Java applications. It provides protection against common threats like:
* Authentication/Authorization
* CSRF
* Session Fixation
* Clickjacking
* URL access restrictions

### **How does Spring Security work internally?**

* It uses a **chain of filters** (FilterChainProxy) to intercept HTTP requests.
* The filters include:
  + UsernamePasswordAuthenticationFilter
  + BasicAuthenticationFilter
  + SecurityContextPersistenceFilter
  + ExceptionTranslationFilter
* Authentication is handled by AuthenticationManager.
* Upon success, an Authentication object is stored in the SecurityContext.

### **Difference between Authentication and Authorization?**

🔐 **Authentication vs Authorization**

| **Concept** | **Authentication** | **Authorization** |
| --- | --- | --- |
| Definition | Verifying who the user is | Determining what the user can access |
| Example | Logging in with username & password | Checking if the user can access /admin |
| Question Answered | *Are you the right user?* | *Do you have permission?* |
| When it Happens | Before authorization | After authentication |
| Handled via | Credentials (e.g., username, password, token) | Roles, permissions, policies |
| Outcome | User identity is confirmed | Access is granted or denied |

✅ **Authentication Flow (Spring Boot Example)**

1. User logs in via /login (form, basic auth, OAuth, etc.)
2. Spring Security authenticates using a configured provider:
   * UserDetailsService
   * JwtAuthenticationFilter
   * OAuth2Login
3. If successful, a SecurityContext is established.

✅ **Authorization Flow**

1. After authentication, Spring checks what resources the user is allowed to access.
2. This is controlled using:
   * @PreAuthorize("hasRole('ADMIN')")
   * http.authorizeRequests() in the config
   * Role or permission-based policies

🧪 **Example in Spring Boot:**

@Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http

.authorizeRequests()

.antMatchers("/admin/\*\*").hasRole("ADMIN")

.antMatchers("/user/\*\*").hasAnyRole("USER", "ADMIN")

.anyRequest().authenticated()

.and()

.formLogin()

.and()

.httpBasic();

}

@Bean

public UserDetailsService userDetailsService() {

var user = User.withUsername("user")

.password("{noop}password")

.roles("USER")

.build();

var admin = User.withUsername("admin")

.password("{noop}admin")

.roles("ADMIN")

.build();

return new InMemoryUserDetailsManager(user, admin);

}

}

🔐 Authentication Types:

* Username/password
* Token-based (JWT)
* OAuth2 / OpenID Connect (Google, GitHub)
* Biometric / SSO

🔑 Authorization Types:

* Role-based (RBAC)
* Permission-based
* Attribute-based access control (ABAC)

🧠 Real-World Analogy:

* Authentication is showing your ID at the gate to prove your identity.
* Authorization is whether you're allowed to enter the VIP lounge.

### **What is the role of UserDetailsService in Spring Security?**

* It is used to **load user-specific data** during authentication. You override it to fetch user details from a **DB, LDAP**, or any source.

public interface UserDetailsService {

UserDetails loadUserByUsername(String username);

}

### **How do you implement Role-Based Authorization?**

* Use method security or HTTP configuration:

// Method level

@PreAuthorize("hasRole('ADMIN')")

// HTTP configuration

.antMatchers("/admin/\*\*").hasRole("ADMIN")

### **What is CSRF and how does Spring Security protect against it?**

* **CSRF (Cross-Site Request Forgery)** is an attack that forces authenticated users to perform unwanted actions.

Spring Security enables CSRF protection by default for non-GET methods.

To disable (not recommended unless you're using stateless APIs like JWT):

http.csrf().disable();

### **What is the use of SecurityContextHolder?**

* It holds the SecurityContext, which contains the authenticated user's Authentication object.

Authentication auth = SecurityContextHolder.getContext().getAuthentication();

### **How is password encryption handled in Spring Security?**

* Use PasswordEncoder:

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

### **What is the difference between hasRole() and hasAuthority()?**

| **Method** | **Role Format** | **Notes** |
| --- | --- | --- |
| hasRole("USER") | Automatically adds ROLE\_ prefix | Role should be ROLE\_USER in DB |
| hasAuthority("ROLE\_USER") | Explicit | More flexible |

### **How do you secure REST APIs using Spring Security?**

* Use JWT or Basic Auth
* Disable CSRF for stateless APIs
* Example config:

http

.csrf()

.disable()

.authorizeRequests()

.antMatchers("/api/admin/\*\*").hasRole("ADMIN")

.anyRequest().authenticated()

.and()

.httpBasic(); // or JWT filter

### **How to implement JWT in Spring Security?**

* Intercept requests with a custom OncePerRequestFilter
* Validate the token and set the authentication in SecurityContextHolder
* Stateless session

### **What is the difference between stateless and stateful authentication?**

| **Type** | **Description** |
| --- | --- |
| Stateful | Session is stored in server memory |
| Stateless | Token (e.g., JWT) sent with every request |

### **What is OncePerRequestFilter?**

* A base class for filters that should be **invoked only once per request**, often used for:
* Logging
* JWT validation
* Custom security filters

### **How to exclude certain endpoints (like /login, /register) from Spring Security?**

.antMatchers("/login", "/register").permitAll()

### **What is OAuth2 and how is it supported in Spring Security?**

* OAuth2 is an **authorization framework** for token-based access delegation.

Spring Security provides:

* spring-security-oauth2-client for login with Google, GitHub, etc.
* spring-security-oauth2-resource-server for JWT token validation

# Hibernate

### How data is persisted in database by using hibernate**?**

* Hibernate is an ORM (Object-Relational Mapping) framework that simplifies database operations in Java applications. Here's how data gets persisted to a database using Hibernate:

1. Configuration Setup

First, configure Hibernate to connect to your database:

<!-- hibernate.cfg.xml -->

<hibernate-configuration>

<session-factory>

<!-- Database connection settings -->

<property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>

<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/mydb</property>

<property name="hibernate.connection.username">root</property>

<property name="hibernate.connection.password">password</property>

<!-- SQL dialect -->

<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

<!-- Mapping files -->

<mapping class="com.example.Employee"/>

</session-factory>

</hibernate-configuration>

2. Entity Class Definition

Create a Java class annotated with @Entity:

@Entity

@Table(name = "employees")

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(name = "emp\_name")

private String name;

private double salary;

// Getters and setters

}

3. Persistence Process

Here's the step-by-step persistence flow:

a) Create SessionFactory

Configuration config = new Configuration().configure("hibernate.cfg.xml");

SessionFactory sessionFactory = config.buildSessionFactory();

b) Open Session

Session session = sessionFactory.openSession();

c) Begin Transaction

Transaction tx = session.beginTransaction();

d) Create Entity Object

Employee emp = new Employee();

emp.setName("John Doe");

emp.setSalary(50000);

e) Save/Persist the Object

session.save(emp); // or session.persist(emp);

f) Commit Transaction

tx.commit();

g) Close Session

session.close();

4. What Happens Internally

When you call **session.save():**

Hibernate checks the entity state - determines if it's transient (new), persistent (managed), or detached

Generates SQL - Hibernate creates the appropriate INSERT statement based on your entity mappings

Executes SQL - The statement is executed against the database

Updates entity state - The transient object becomes persistent

Handles ID generation - If using auto-increment, retrieves the generated ID

5. Transaction Management

Hibernate operations should be performed within transactions:

Session session = sessionFactory.openSession();

try {

Transaction tx = session.beginTransaction();

// Perform operations

Employee emp = new Employee("Jane Smith", 60000);

session.save(emp);

tx.commit(); // Changes are flushed to database here

} catch (Exception e) {

if (tx != null) tx.rollback();

throw e;

} finally {

session.close();

}

6. Different Persistence Operations

**Operation Description**

save() Persists the object, returns generated ID

persist() Similar to save() but doesn't guarantee immediate ID assignment

update() Updates a detached object

merge() Copies state of detached object to persistent object

saveOrUpdate() Either saves or updates based on object state

7. Hibernate Cache Flow

**First Level Cache (Session cache):**

* Exists per session
* All persistent objects are stored here
* Cleared when session closes

**Second Level Cache (Optional):**

* Shared across sessions
* Needs explicit configuration
* Reduces database hits for frequently accessed data

Example Complete Flow

public class HibernateExample {

public static void main(String[] args) {

// 1. Configure and build SessionFactory

SessionFactory sessionFactory = new Configuration()

.configure("hibernate.cfg.xml")

.buildSessionFactory();

// 2. Open session

Session session = sessionFactory.openSession();

Transaction tx = null;

try {

// 3. Begin transaction

tx = session.beginTransaction();

// 4. Create and persist object

Employee newEmployee = new Employee();

newEmployee.setName("Michael Johnson");

newEmployee.setSalary(75000);

// 5. Save to database

Long employeeId = (Long) session.save(newEmployee);

System.out.println("Employee saved with ID: " + employeeId);

// 6. Commit transaction

tx.commit();

} catch (Exception e) {

if (tx != null) tx.rollback();

e.printStackTrace();

} finally {

// 7. Close session

session.close();

sessionFactory.close();

}

}

}

**Key Benefits of Hibernate Persistence:**

* Object-Oriented Approach - Work with objects rather than SQL
* Automatic SQL Generation - No need to write CRUD queries
* Transaction Management - Built-in support for ACID properties
* Caching - Improves performance
* Database Independence - Switch databases with minimal code changes

This is the fundamental process of how Hibernate persists Java objects to relational databases while handling all the underlying JDBC complexity.

### **How to retrieve select query using Hibernate?**

* Answer

### **How to retrieve data through hibernate?**

* Answer

### **Hibernate caching?**

* Answer

### **Explain L1 and L2 cache in Hibernate?**

* Answer

### **Explain the hibernate sessions?**

* Answer

### **Difference between JDBC and Hibernate?**

* Answer

### **What is the difference between get and load method in hibernate?**

* Answer

### **Explain bean scopes?**

* Answer

### **Can we have beans without bean id?**

* Answer

### **Difference between get and load method?**

* Answer

### **Explain Performance tuning and indexing in hibernate?**

* Answer

### **Explain different classes and annotations in hibernate?**

* Answer

### **How to call stored procedures using hibernate?**

* answer

### **What is Many to many associations in hibernate?**

* answer

### **What is JPA?**

* answer

### **What is the many to many associations in hibernate?**

* answer

### **What is a Session factory in hibernate?**

* answer

### **Explain Many to many associations in hibernate?**

* answer

### **If I want to retrieve a few entries from an employee table, how to retrieve employee details through JPA?**

* answer

### **Explain a built-in function in JPA?**

* answer

### **What is lazy loading in hibernate?**

* answer

### **What is circular dependency?**

* answer

### **What is the second level cache in hibernate?**

* answer

### **Can we use Hibernate without JPA?**

* answer

### **What are the advantages of ORM?**

* answer

# Spring Boot

### **Difference between Spring and Spring Boot?**

* Answer

### **What are the advantages of SprigBoot?**

* Answer

### **What is a yaml file in SpringBoot?**

* Answer

### **What is what is @transient in spring boot?**

* answer

### **What are the types of propagation?**

* Answer

### **What @Qualifier in spring? What is the use of @Qualifier?**

* answer

### **What are the advantages of Spring boot?**

* answer

### **How to create multiple modules using spring boot?**

* answer

### **What are all methods present in the controller?**

* answer

### **Explain annotations in Spring boot?**

* answer

### **What is the query parameter?**

* answer

### **How to change the server in Spring boot?**

* answer

### **Difference between @Controller and @RestController?**

* answer

### **What is the use of @Responsebody annotation?**

* answer

### **Explain @RestController annotation in Spring boot?**

* answer

### **Which two annotations are used in built by @RestController?**

* answer

### **How to access application.properties properties in java code?**

* answer

### **Explain Spring boot application? How does it work?**

* answer

### **What is the use of @Service annotation in spring?**

* answer

### **What is the use of @Component annotation in spring?**

* answer

### **What is the use of @Repository annotation in spring?**

* answer

### **What is the difference between @Service and @Controller annotation?**

* answer

### **What is @ComponentScan?**

* answer

### **What is the use of @Autowired annotation?**

* answer

### **Write a REST API to accept user name as input, return the response with message as "Hello, user" and status as "success" and Endpoint - /api/message?**

* answer

### **How to handle Exceptions in Spring boot?**

* answer

### **How to provide security to spring boot applications?**

* answer

### **What is @ControllerAdvice in Spring boot?**

* answer

### **How to retrieve query parameters in Spring boot?**

* answer

### **How is the data persisted in DB with the REST API?**

* answer

### **Explain GET and POST methods?**

* answer

### **Write a SQL or JPQL to retrieve the data from DB?**

* answer

### **How will you add Employee object data in the DB using POST call?**

* answer

### **What is the size of Spring in your project? Who will decide that?**

* answer

### **Explain about spring security?**

* answer

### **Explain the flow of REST api?**

* answer

### **What will happen when we replace @Service instead of @Repository?**

* answer

### **Explain uses of SpringBoot application?**

* answer

### **How is https used in SpringBoot applications?**

* answer

### **Difference between RequestMapping and GetMapping?**

* answer

### **If you have to create a Spring boot application? What steps will you follow?**

* answer

### **What is a starter kit in spring boot application?**

* answer

### **What are the starters in Spring boot application?**

* answer

### **What are Actuators in spring boot applications?**

* answer

### **What are custom exceptions? How do you define your own custom exception?**

* answer

### **Have you worked on Unit testing? Which framework have you used in your project?**

* answer

### **What is the difference between lazy loading and eager loading?**

* answer

### **Difference between put and patch?**

* answer

### **Can we use @RequestMapping at class level and method level both?**

* Answer

### **What will happen if I use @RequestMapping at method level?**

* answer

### **How to map Service class to RestController?**

* answer

### **What is Spring security? Have you used Spring security in your project?**

* answer

### **What are Profiles in Spring boot?**

* answer

### **How will you do CRUD operations using spring boot applications?**

* answer

### **How to change from JAR to WAR in Spring boot?**

* answer

### **Where will you give database details in the spring boot application?**

* answer

### **How will you give database details for the individual environment?**

* answer

### **What is a JPA repository?**

* answer

### **What is the use of @SpringBootApplication annotation in Spring boot?**

* answer

### **How will the Spring boot application know whether its JAR or WAR type?**

* answer

### **How are you providing security to your services?**

* answer

### **As an end user how can I consume your services?**

* answer

### **How to validate a token?**

* answer

### **Explain a few security algorithms?**

* answer

### **What is native queries in Spring boot?**

* answer

### **Where do you use native queries?**

* answer

### **Suppose I am having different microservices, if I want to invoke different microservice so how can we interact?**

* answer

### **What kind of information are you passing to the GET method?**

* answer

### **Suppose I want to start a spring boot application, so what are all the steps?**

* answer

### **How to configure a custom server in a Spring boot application?**

* answer

### **How to configure multiple databases in a Spring boot application?**

* answer

### **How does your application know I am using a particular database?**

* answer

### **What is the use of @Qalifier annotation?**

* answer

### **How are REST calls intercepted in our project? How is the call happening?**

* answer

### **When I hit a URL how data will be invoked with your application?**

* answer

### **How is the DTO getting invoked with REST endpoints?**

* answer

### **I have a URL and I hit the URL, so which is the first class which is invoked in our product?**

* answer

### **If you have to expose a new endpoint, what steps will you follow?**

* answer

### **What is the difference between @Bean and @Configuration?**

* answer

### **What is the difference between RequestParam and path variable? Give me one example with one URL?**

* answer

### **How to change the port of the embedded tomcat server in Spring boot? Write a syntax for that?**

* answer

### **What are Profiles in Spring boot applications? Give one example?**

* answer

### **Which annotations are we using in order to use the Exception?**

* answer

### **What is the default scope for spring beans?**

* answer

### **Which database have you used? How do you integrate databases with JPA?**

* answer

### **How to communicate between two spring boot applications using asynchronous calls?**

* answer

### **Have you worked messaging services?**

* answer

### **How will you handle errors / exceptions in your project?**

* answer

### **What are Actuators?**

* answer

### **How are you keeping track of logs in your project?**

* answer

### **If you have millions of data then how will you manage it?**

* answer

### **What is the difference between REST API and Http?**

* answer

### **Explain @Autowired usage?**

* answer

### **How does the REST API work?**

* answer

### **How can we handle a yaml file when it is exposed?**

* answer

### **How to externalize variables in spring boot?**

* answer

### **What is the Spring reactive framework?**

* answer

### **Why should we use Spring reactive?**

* answer

### **What is the use of block and subscribe?**

* answer

### **What is a novel based server?**

* answer

### **How will the findById or findByName method in the repository work with name? When we go with camelcase then where is it implemented internally?**

* answer

### **How does the JPA Repository know there is a class?**

* answer

### **How does the Spring Boot application work internally?**

* answer

### **How will you authenticate API in Spring Boot?**

* answer

### **In the Spring REST API what is the White label error accessing from the browser? What is the status code you will get?**

* answer

# Microservices

### **Monolith vs Microservices differences and when to choose which?**

**🧱 Monolithic Architecture**

**✅ What is it?**

A single unified application where all modules (UI, business logic, data access) are tightly coupled and deployed together.

**🔍 Characteristics:**

* Single codebase and deployment unit.
* All features share the same memory space and process.
* Easy to develop initially and suitable for small teams.

**🧩 Microservices Architecture**

**✅ What is it?**

An application is divided into a collection of small, loosely coupled services that communicate via APIs (usually REST or messaging systems like Kafka).

**🔍 Characteristics:**

* Each service is independently deployable.
* Each microservice has its own data store (DB per service pattern).
* Services can be written in different programming languages (polyglot).

**🆚 Monolith vs Microservices: Key Differences**

| **Feature** | **Monolithic** | **Microservices** |
| --- | --- | --- |
| Architecture | Single application unit | Distributed system with many services |
| Codebase | One large codebase | Multiple small codebases |
| Deployment | Deployed as a single unit | Each service deployed independently |
| Scalability | Vertical (scale the whole app) | Horizontal (scale individual services) |
| Development Speed | Faster initially | Slower initially, but scales well |
| Technology Stack | Same tech across the app | Each service can use a different stack |
| Testing | Easier end-to-end | Complex due to inter-service dependencies |
| Failure Impact | Entire app may go down | One service failure doesn’t necessarily affect others |
| Database | Shared DB | Decentralized DB per service |
| Communication | In-process calls | API or messaging-based communication |

**📌 When to Choose Monolithic:**

**✅ Go for Monolith when:**

* The team is small (startups, MVPs).
* The application is simple and won’t grow much.
* Quick development and deployment is required.
* You’re in the prototyping or early stage.

**🚀 When to Choose Microservices:**

**✅ Go for Microservices when:**

* Application is large, complex, and will grow over time.
* Different parts of the system need to scale independently.
* You have multiple teams working in parallel.
* You need high availability, fault tolerance, and agility.
* You want flexibility in tech stacks or deployment pipelines.

🛠 **Example Use Case:**

**Monolith:** Blogging platform, early-stage ERP, internal admin tools.

**Microservices:** E-commerce platforms, fintech apps, enterprise SaaS, streaming platforms.

### **How to design a microservice from scratch?**

* Designing a microservice from scratch requires a thoughtful approach that involves domain modeling, technology selection, infrastructure setup, and ensuring principles like loose coupling, scalability, and independent deployment.

🧠 **Step-by-Step: Designing a Microservice from Scratch**

**1️ Understand the Business Requirement**

* Clarify the problem the microservice needs to solve.
* Identify the boundaries of the service (Single Responsibility Principle).
* Use DDD (Domain-Driven Design) to define aggregates and entities.

**2️ Define the Microservice Scope**

**Ask:**

* What is this microservice responsible for?
* What data does it own?
* What other services does it need to communicate with?

📌 **Example:** For an E-commerce app, microservices could be:

* Product Service
* Order Service
* Payment Service
* Inventory Service

**3️ Choose Tech Stack**

**Depending on your ecosystem:**

✅ Common choices:

* **Language:** Java (Spring Boot), Node.js, Go, Python
* **Database:** PostgreSQL, MongoDB, Redis (each service should manage its own DB)
* **Communication:** REST APIs, gRPC, Kafka, RabbitMQ

**4️ Define the API Contract**

Use OpenAPI/Swagger to define:

* Endpoints
* HTTP Methods
* Input/Output models
* Status codes

**Example:**

* POST /orders
* GET /orders/{id}

**5️ Design the Database (DB per service pattern)**

Each microservice should own its own data.

**Example (Order Service DB):**

Order(id, userId, productId, status, createdDate)

Use Liquibase or Flyway for DB migrations.

**6️ Build the Microservice (Spring Boot Example)**

**✅ Common Layers:**

* Controller – REST endpoints
* Service – Business logic
* Repository – DB access (JPA)
* DTOs – For input/output models
* Exception Handling – Custom exceptions + Global handler

**7️ Inter-Service Communication**

* Synchronous: REST, gRPC
* Asynchronous: Kafka, RabbitMQ (for decoupled event-driven designs)

Example: Order Service publishes an event to Kafka after order is placed.

**8️ Add Cross-Cutting Concerns**

* Logging – SLF4J, Logback
* Monitoring – Prometheus + Grafana
* Tracing – OpenTelemetry, Zipkin
* Security – OAuth2/JWT (Spring Security)

**9️ Write Tests**

* Unit tests for services and utils
* Integration tests for REST endpoints
* Use tools like JUnit, Mockito, Testcontainers

**10 Containerize with Docker**

FROM openjdk:21

COPY target/order-service.jar app.jar

ENTRYPOINT ["java", "-jar", "app.jar"]

🔁 **Deploy to Kubernetes (Optional)**

* Create Deployment, Service, and Ingress YAMLs
* Use Helm for templating

🚧 **Best Practices**

* Follow 12-factor app principles
* Avoid shared databases
* Automate build & deploy via CI/CD (GitHub Actions, Jenkins, etc.)
* Use API Gateway (like Spring Cloud Gateway) for routing/authentication
* Apply circuit breakers, retry, and rate limiting

### **API Gateway pattern and its advantages?**

**🧰 API Gateway Pattern**

An API Gateway is a server that acts as a single entry point for all client requests to a system of microservices. It routes requests, handles common functionalities (like authentication, rate limiting, logging), and often aggregates responses from multiple services.

**📌 Why Use an API Gateway?**

In a microservices architecture, clients would otherwise need to:

* Know the locations of multiple services
* Handle different protocols
* Make multiple calls for a single operation

➡️ This leads to complexity, tight coupling, and redundancy.

The API Gateway pattern solves this by centralizing access control and request routing.

**🗂 Common Responsibilities of API Gateway**

| **Responsibility** | **Description** |
| --- | --- |
| Request Routing | Forwards requests to the appropriate backend microservice |
| Authentication & AuthZ | Validates JWT tokens, API keys, OAuth2 credentials |
| Rate Limiting & Throttling | Controls traffic to prevent abuse or DDoS attacks |
| Load Balancing | Distributes load across multiple instances |
| Request/Response Transformation | Modifies headers, URL paths, or payloads |
| Caching | Reduces backend load by returning cached responses |
| Logging and Monitoring | Logs all incoming/outgoing traffic for auditing and monitoring |
| Response Aggregation | Combines results from multiple services into one response |
| Protocol Translation | Converts protocols (e.g., HTTP to WebSocket, gRPC, etc.) |

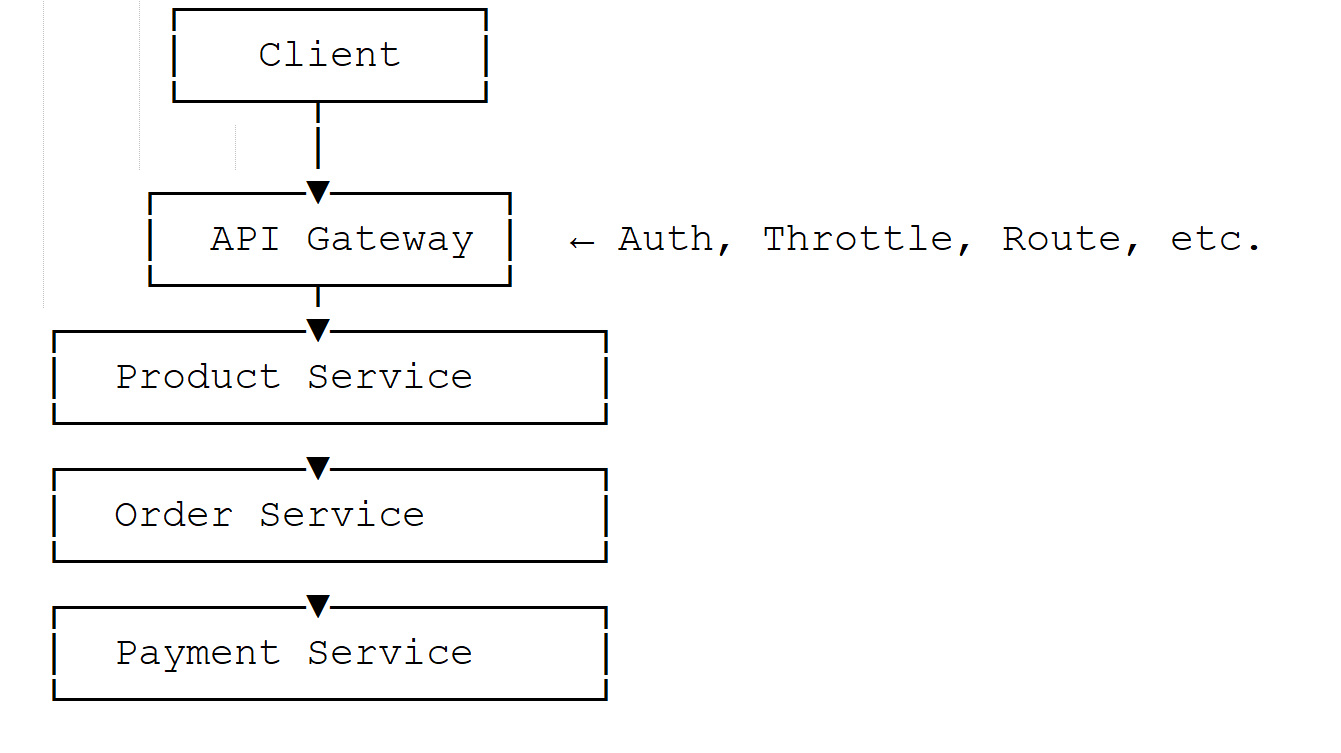
**✅ Advantages of API Gateway Pattern**

| **Advantage** | **Explanation** |
| --- | --- |
| ✅ Centralized Access Point | Clients interact with one endpoint instead of many microservices |
| ✅ Simplifies Client Logic | API Gateway hides microservice topology and protocols |
| ✅ Improved Security | All requests go through a secure, audited, and controlled gateway |
| ✅ Scalability | Can scale API Gateway independently of the backend services |
| ✅ Performance Optimization | Through caching, compression, and rate-limiting |
| ✅ Flexible Routing | Can perform routing based on request data like headers, path, or payload |
| ✅ Decoupling | Helps decouple frontend clients from backend service implementations |

🛠️ **Popular API Gateway Tools**

| **Tool/Framework** | **Description** |
| --- | --- |
| Spring Cloud Gateway | Java-based, for Spring Boot microservices |
| Netflix Zuul (legacy) | Older gateway, succeeded by Spring Cloud Gateway |
| Kong | Open-source, Lua-based, highly extensible |
| NGINX | Widely used, highly performant |
| AWS API Gateway | Managed gateway service for AWS environments |
| Apigee | Google Cloud’s enterprise API management |
| Istio Ingress Gateway | Used with service mesh architecture |

🧱 **Example Architecture with API Gateway**



### **Inter-service communication: REST vs Messaging?**

**🔁 Inter-Service Communication: REST vs Messaging**

**🟦 REST (Synchronous Communication)**

REST uses HTTP protocols (usually over JSON) for communication between microservices.

**✅ Pros:**

* Simple and widely adopted
* Easy to debug and test (Postman, curl)
* Works well when immediate response is needed (e.g., GET /orders/1)

**❌ Cons:**

* Tightly coupled: if Service B is down, Service A fails
* Increased latency with chained service calls
* Harder to scale under heavy load

**🟨 Messaging (Asynchronous Communication)**

Uses message brokers (like Kafka, RabbitMQ, ActiveMQ) to send/receive messages between services.

**✅ Pros:**

* Loose coupling: services don’t need to know each other’s availability
* High scalability and throughput
* Great for event-driven architecture (e.g., “OrderPlaced” event)

**❌ Cons:**

* More complex to set up and monitor
* Eventual consistency (response is not immediate)
* Harder to trace/debug (needs distributed tracing)

🆚 **REST vs Messaging: Comparison Table**

| **Feature** | **REST (HTTP)** | **Messaging (Kafka, RabbitMQ)** |
| --- | --- | --- |
| Communication Type | Synchronous | Asynchronous |
| Coupling | Tight | Loose |
| Availability Dependency | Service must be up | Sender and receiver can be decoupled |
| Data Consistency | Strong consistency | Eventual consistency |
| Reliability | Less reliable (network failure) | More reliable (can retry, durable) |
| Scalability | Limited by sync calls | Highly scalable |
| Latency | Low (if successful) | Can be delayed |
| Error Handling | Needs retries or fallbacks | Built-in retry, dead-letter queues |
| Use Case | Real-time requests/responses | Background jobs, event notification |

**📦 When to Use What?**

**✅ Use REST when:**

* Real-time response is required (e.g., Login, Search)
* Simpler integration with frontends
* Request/response fits a client-server pattern

**✅ Use Messaging when:**

* You want decoupled, event-driven architecture
* Operations can be processed asynchronously (e.g., Order confirmation, Email sending)
* High volume of traffic or batching is expected

**🎯 Hybrid Approach (Best Practice)**

* Most production-grade microservice systems use both:
* Use REST for synchronous, real-time operations
* Use messaging/event streams for decoupled, asynchronous workflows

**⚙️ Example:**

Order Service:

* REST: POST /order → creates order
* Event: publishes OrderCreated to Kafka

Inventory Service:

* Listens to OrderCreated → updates stock

Notification Service:

* Listens to OrderCreated → sends email/SMS

### **Circuit Breaker pattern and its implementation using Resilience4j?**

**🧯 Circuit Breaker Pattern in Microservices**

The Circuit Breaker pattern is a resilience design pattern used to prevent a system from making repeated calls to a failing service, thereby avoiding system overload and allowing time for recovery.

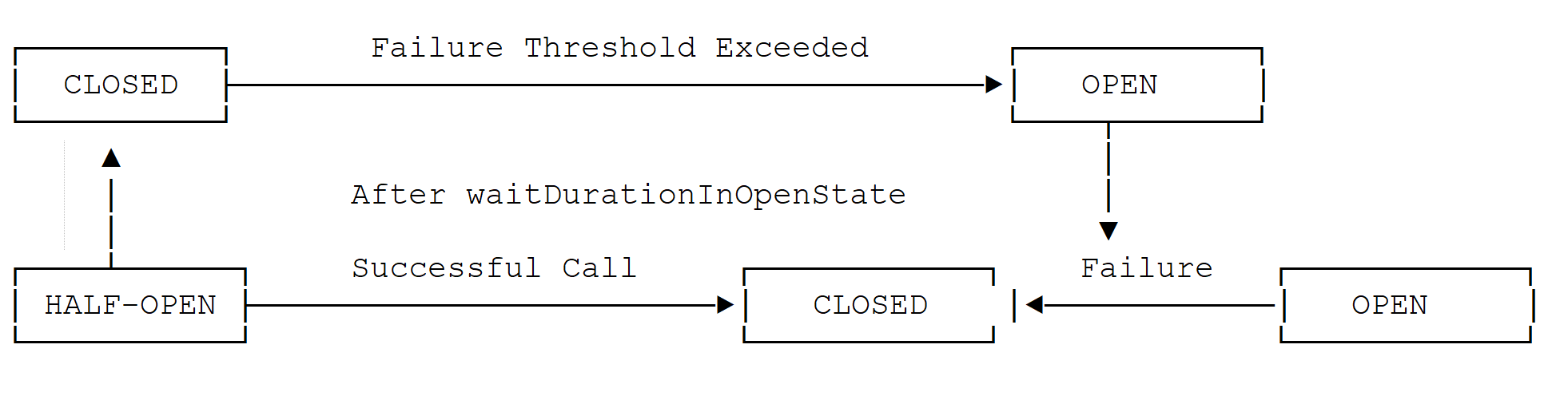
**🧠 Why Circuit Breaker?**

Imagine Service A calls Service B. If B is down or slow, A will keep waiting and retrying, leading to:

* High latency
* Resource exhaustion
* Cascading failures

➡️ Circuit Breaker helps by "breaking" the connection temporarily after a failure threshold is reached.

**🔁 Circuit Breaker States**



**⚙️ Resilience4j Implementation in Spring Boot**

Resilience4j is a lightweight fault tolerance library designed for Java 8+ and functional programming.

✅ Features:

* Circuit Breaker
* Rate Limiter
* Retry
* Bulkhead
* Time Limiter
* Cache

🚀 **Step-by-Step: Circuit Breaker using Resilience4j**

**1️ Add Dependency (Gradle)**

implementation 'io.github.resilience4j:resilience4j-spring-boot3:2.1.0'

implementation 'org.springframework.boot:spring-boot-starter-aop'

***For Maven:***

<dependency>

<groupId>io.github.resilience4j</groupId>

<artifactId>resilience4j-spring-boot3</artifactId>

<version>2.1.0</version>

</dependency>

**2️⃣ Sample Service Layer with Circuit Breaker**

@Service

public class ProductService {

@CircuitBreaker(name = "productServiceCB", fallbackMethod = "fallbackGetProducts")

public List<String> getProducts() {

// Simulate failure or call external service

if (new Random().nextInt(10) < 8) {

throw new RuntimeException("Service is down!");

}

return List.of("Laptop", "Phone", "Tablet");

}

// Fallback method must have same signature + Throwable parameter

public List<String> fallbackGetProducts(Throwable ex) {

return List.of("Fallback Product 1", "Fallback Product 2");

}

}

**3️ Application Properties**

resilience4j.circuitbreaker.instances.productServiceCB:

slidingWindowSize: 5

minimumNumberOfCalls: 3

failureRateThreshold: 50

waitDurationInOpenState: 10s

permittedNumberOfCallsInHalfOpenState: 2

automaticTransitionFromOpenToHalfOpenEnabled: true

**4️ Controller Layer**

@RestController

@RequestMapping("/api/products")

public class ProductController {

@Autowired

private ProductService productService;

@GetMapping

public ResponseEntity<List<String>> getProducts() {

return ResponseEntity.ok(productService.getProducts());

}

}

📊 **Monitoring with Spring Actuator + Resilience4j Dashboard (Optional)**

**Add these to monitor circuit breaker metrics:**

implementation 'org.springframework.boot:spring-boot-starter-actuator'

implementation 'io.github.resilience4j:resilience4j-micrometer'

implementation 'io.micrometer:micrometer-registry-prometheus'

Then you can expose endpoints like /actuator/metrics/resilience4j.circuitbreaker.calls.

✅ **Advantages of Circuit Breaker Pattern**

| **Benefit** | **Description** |
| --- | --- |
| 🔒 Prevents overload | Stops calling a failed service until it recovers |
| ⚡ Improves response time | Returns fallback immediately instead of waiting |
| 🔁 Enables graceful degradation | Users get default or cached response |
| 📊 Helps monitoring | Tracks failure rates and transition states |

### **Load balancing in microservices using Spring Cloud Load Balancer?**

🚀 **What is Load Balancing in Microservices?**

When you have multiple instances of a microservice running (for fault tolerance, scalability, etc.), load balancing ensures that requests are evenly distributed among those instances.

**⚙️ Spring Cloud LoadBalancer**

Spring Cloud LoadBalancer is a lightweight, client-side load balancer that replaces Netflix Ribbon (now deprecated). It allows you to call microservices via service names instead of hardcoded URLs.

**🔧 Setup and Usage**

**1. Add Dependencies**

In your Spring Boot app, include:

// build.gradle

implementation 'org.springframework.cloud:spring-cloud-starter-loadbalancer'

Also make sure you have service discovery (e.g., Eureka):

implementation 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client'

**2. Enable Discovery Client**

@SpringBootApplication

@EnableDiscoveryClient

public class MyApp {

public static void main(String[] args) {

SpringApplication.run(MyApp.class, args);

}

}

**3. Inject LoadBalanced RestTemplate**

@Configuration

public class AppConfig {

@Bean

@LoadBalanced

public RestTemplate restTemplate() {

return new RestTemplate();

}

}

**4. Call Another Microservice**

@RestController

public class MyController {

@Autowired

private RestTemplate restTemplate;

@GetMapping("/get-data")

public String getData() {

String response = restTemplate.getForObject("http://user-service/users", String.class);

return response;

}

}

✅ user-service is the service name registered in Eureka, and Spring Cloud LoadBalancer will resolve its IPs and apply a round-robin strategy.

**⚖️ Supported Load Balancing Strategies**

You can customize the strategy:

spring:

cloud:

loadbalancer:

ribbon:

enabled: false

loadbalancer:

hint:

user-service: round-robin

Or use Java-based configuration with your own strategy (e.g., weighted, random, zone-aware).

**🔄 Alternatives to RestTemplate**

**WebClient (recommended in reactive apps):**

@Bean

@LoadBalanced

public WebClient.Builder webClientBuilder() {

return WebClient.builder();

}

**🧠 Behind the Scenes**

* @LoadBalanced tells Spring to intercept HTTP calls and resolve service names to real IPs.
* It communicates with Eureka to get service instance details.
* It uses round-robin as default load balancing.

**✅ Benefits:**

* Automatic client-side load balancing
* No hardcoded URLs
* Integrates with Eureka, Consul, etc.
* Easy to extend and customize

🛡️ **Summary**

| **Feature** | **Spring Cloud LoadBalancer** |
| --- | --- |
| Type | Client-side |
| Default Strategy | Round-Robin |
| Service Discovery | Required (e.g., Eureka) |
| Replacement for | Netflix Ribbon |
| Works with | RestTemplate, WebClient |

### **How Spring Cloud Config helps in centralized configuration management?**

📦 **What is Spring Cloud Config?**

Spring Cloud Config is a part of the Spring Cloud ecosystem that provides server-side and client-side support for externalized configuration in a distributed system.

**🔑 Core Idea:**

Keep all your configuration files (application.yml/properties) for different microservices in one centralized Git repository, and let microservices fetch their configs from there.

**💡 Why Centralized Configuration?**

**Without centralized config:**

* Each microservice maintains its own config
* Difficult to manage multiple environments (dev, test, prod)
* Changes require service restarts or redeployments

**With centralized config:**

* Easier updates
* Environment separation
* Consistent versioning
* Reloadable configs

**🧩 Components of Spring Cloud Config**

**1. Config Server**

Hosts the configuration files centrally.

**2. Config Client**

Each microservice acts as a client and fetches config from the config server.

**🔧 How to Set It Up**

**🔹 Step 1: Create a Git Repo**

Structure:

my-config-repo/

├── application.yml

├── user-service-dev.yml

├── order-service-prod.yml

**🔹 Step 2: Create a Spring Cloud Config Server**

@SpringBootApplication

@EnableConfigServer

public class ConfigServerApp {

public static void main(String[] args) {

SpringApplication.run(ConfigServerApp.class, args);

}

}

**application.yml:**

server:

port: 8888

spring:

cloud:

config:

server:

git:

uri: https://github.com/my-org/my-config-repo

**🔹 Step 3: Configure Microservices (Config Clients)**

**Add dependency:**

implementation 'org.springframework.cloud:spring-cloud-starter-config'

**Client bootstrap.yml:**

spring:

application:

name: user-service

cloud:

config:

uri: http://localhost:8888

profile: dev

**🔁 Refresh Config at Runtime**

**Add this to your microservice:**

implementation 'org.springframework.boot:spring-boot-starter-actuator'

**Enable refresh:**

@RefreshScope

@RestController

public class SomeController {

@Value("${custom.property}")

private String value;

}

**Call:**

POST http://localhost:{port}/actuator/refresh

✅ **Benefits of Spring Cloud Config**

| **Feature** | **Benefit** |
| --- | --- |
| 🔄 Centralized Config | One place to manage all properties |
| 🌐 Git-based Versioning | Rollbacks & audits made easy |
| 🔁 Dynamic Refreshing | No restarts needed (with @RefreshScope) |
| 🏗️ Environment Specific Config | Supports dev, test, prod environments |
| 🧩 Integrates with Eureka etc. | Works seamlessly with other Spring Cloud modules |

**🔐 Advanced Features**

* Support for Vault for secrets
* Can pull config from file system, Git, SVN
* Can be secured with OAuth2 or JWT
* Works well with Spring Boot Admin and Actuator

**🔚 Conclusion**

Spring Cloud Config helps you centralize and manage configurations across all microservices — making your system more flexible, maintainable, and environment-aware. 💼📁

### **Service discovery using Eureka or Consul?**

* answer

### **Feign Client vs WebClient: Which one to use and why?**

* answer

### **Event-driven architecture and Kafka integration?**

* answer

### **Database per service vs Shared Database: Pros and cons?**

* answer

### **Saga Pattern for distributed transactions in microservices?**

* answer

### **JWT-based authentication and OAuth2 in microservices?**

* answer

### **How to handle security in an API Gateway?**

* answer

### **Observability: Logging, tracing, and monitoring best practices?**

* answer

### **Role of Prometheus and Grafana in microservices monitoring?**

* answer

### **Kubernetes deployment strategies for microservices?**

* answer

### **Blue-Green and Canary deployments in microservices?**

* answer

### **When to use WebFlux for reactive microservices?**

* answer

### **CQRS and Event Sourcing: When and why to use them?**

* Answer

### **Difference between Monolithic and Microservice architecture?**

* answer

### **Explain Microservices architecture?**

* answer

### **Where are you deploying your microservice?**

* answer

### **What are different Aggregation patterns in Microservices?**

* answer

### **What are challenges in Microservice architecture?**

* answer

### **How are you handling load in your microservices?**

* answer

### **How will you handle resilience and fault tolerance in microservice?**

* answer

### **Can you explain Event driven microservice and how you have used it in your project?**

* answer

# SQL

### **Difference between Primary Key and Unique key?**

* Answer

### **What is ordered by and range in the database?**

* Answer

### **Write a SQL query to get the second highest salary of an employee from the employee table?**

* Answer

### **Write a SQL query to print employee id, employee\_name, depatment\_name Employee - employee\_id, employee\_name, department\_id**

### **Department - department\_id, department\_name?**

* Answer

### **SQL query to find department name and number of employees in that department?**

* Answer

**SELECT** d.department\_name, **COUNT**(e.employee\_id) **AS** employee\_count  
**FROM** employees e  
**JOIN** departments d **ON** e.department\_id = d.department\_id  
**GROUP BY** d.department\_name;

### **What is a collision in the oracle database?**

* Answer

### **If you create multiple folders one inside another and store one file into the last folder then how can you implement using database tables in oracle? How many tables are required?**

* answer

### **What are DDL and DML commands?**

* Answer

### **Write a query to update data in the table?**

* Answer

### **What is the use of joins in the database?**

* Answer

### **Explain ACID properties in the database?**

* Answer

### **Explain Normalization?**

* Answer

### **What are the types of Normalization?**

* Answer

### **What is Denormalization?**

* Answer

### **What is the use of join?**

* Answer

### **Write a SQL query to return the 7th highest salary of an employee?**

* Answer

### **What is Union in SQL?**

* Answer

### **Can we perform union operations on one table?**

* Answer

### **Difference between where and let?**

* Answer

### **What is Semaphore?**

* Answer

### **What is View and Index in SQL?**

* Answer

### **What is Trigger in SQL? What is the use of Trigger in SQL?**

* Answer

### **What is inner join in SQL?**

* Answer

### **Write a SQL query to get emp\_sal > 5000 and company\_name = "Kotak"**

Table :

1. Company

- company\_id

- company\_name

2. Employee

- emp\_id

- company\_id

- emp\_name

- emp\_sal?

* Answer

Select e.emp\_sal, c.company\_name

From Employee e

join Company c

On c.company\_id = e.company\_id

Where e.emp\_sal > 5000 and c.company\_name = "Kotak";

### **Write a JPA query to retrieve the above result?**

* answer

### **What is an index in SQL?**

* answer

### **Explain different types of Index?**

* answer

### **Explain different types of joins?**

* answer

### **Difference between Outer join and Full outer join?**

* answer

### **Write a SQL query to find the name of a city which has more than one customer in it?**

Example: Customer

Customer\_name

Customer\_city?

* answer

### **Write SQL query to display 10th highest salary of employee?**

* answer

### **What is the difference between DDL and DML commands?**

* answer

### **Difference between Inner join and Outer join?**

* answer

### **Difference between left outer join and right outer join?**

* answer

### **What are Triggers?**

* answer

### **SQL query to get the youngest employee from the employee table?**

* answer

### **SQL query to get list of employees from Finance department?**

* answer

### **What is Indexing in SQL?**

* answer

### **What are the disadvantages of Index?**

* answer

### **What is the difference between primary key and unique key?**

* answer

### **What is the difference between implicit cursor and explicit cursor?**

* answer

### **What is a correlated subquery?**

* answer

### **What is the left outer join?**

* answer

### **What is the use of stored procedure?**

* answer

### **Explain the database schema for Students and Course table?**

* answer

### **What is UNION ALL?**

* answer

### **SQL query to get second highest salary?**

* answer

### **Write a SQL query to display a list of students which have a Mathematics subject?**

Input - Student

student\_id

student\_name

Subject\_id

Subject

subject\_id

Subject\_name

Where subject=Mathematics?

* answer

### **What is Normalization?**

* answer

### **Can you explain different forms of Normalization with one example?**

* answer

### **What kind of fetching strategy are you using in hibernate?**

* answer

### **What is OneToMany mapping?**

* answer

### **Write a SQL query to get the 3rd highest salary of an employee?**

* answer

### **What is dense and rank in SQL?**

* answer

### **What is the difference between SQL and NoSQL?**

* answer

### **Difference between Stored Procedure and Function?**

* answer

### **What is Transaction in SQL?**

* answer

### **Write a SQL query:**

Consider 2 tables, find out 3 employees who are earning the highest salary in each department?

Employees

employee\_id (primary key)

first\_name

last\_name

department\_id (foreign key to departments.id)

salary

Hire\_date

Departments

id (primary key)

department\_name?

* answer

### **SQL query that categorizes employees based on their salary into different percentage ranges.**

emp

id Name Salary

1 A 10

2 B 100

3 C 4

4 D 60

5 E 70

6 F 40

---------------------

id name sal Percentage

2 B 100 greater than 90

5 E 70 greater than 50

4 D 60 greater than 50

1 A 40 less than 50

3 C 10 less than 10

6 F 5 less than 10

--4 cate--

greater than 90

greater than 50

less than 50

less than 10?

* answer

### **Write a SQL query to find the department name who does not have an employee? (Employee and Department table given)?**

* answer

### **What is the difference between Primary Key and Unique Key?**

* answer

### **What is the Composite key?**

* answer

### **What is Self join?**

* answer

### **We have Table A (2 columns and 4 rows) and Table B (2 columns and 4 rows) and I want to print all 8 rows? Write a query for it? (Use Union All)?**

* answer

### **What is the difference between SQL Union and Union All?**

* answer

### **Write a SQL query and print the employee name which does not have any department?**

* answer

### **Write a query - Employee table and add one extra column in output as category? Let's say if salary is in 1 to 4000 then Category 1, 4000 - 7000 then category 2, 7000 - 10000 then category 3?**

* answer

# Kafka

### **What is Apache Kafka?**

* answer

### **Why does Kafka compare to RabbitMQ?**

* answer

### **How to scale up Kafka?**

* answer

### **What are the default partitions in kafka?**

* answer

### **Difference between topic and partition in Kafka?**

* answer

### **Is there any ordering of messages in partitions in kafka?**

* answer

### **How does partition assignment happen? What method is used?**

* answer

### **What is offset and partition in Kafka?**

* answer

# GitHub

### **What is Version control i.e. Git?**

* Answer

### **How to push and commit code in git using commands?**

* Answer

### **Difference between rebase and merge?**

* answer

### **Which versioning tool are you using? Explain a few Git commands?**

* answer

### **What is the Git command to create a new branch from an existing branch?**

* answer

### **If we have more consumer groups than partitions and how will it manage?**

* answer

# Docker

### **Explain use cases of Docker?**

* answer

### **You have 2 containers 1 and 2, can they communicate with each other?**

* answer

### **If both are with different hosts then whether they will be able to communicate with each other or not?**

* answer

### **What is the host in this case?**

* answer

### **Imagine you have one volume and you want to share it with multiple containers? How will you do it?**

* answer

### **Where does an object get stored when created inside a docker container?**

* answer

### **What is the process of Docker?**

* answer

### **How is Docker helpful?**

* answer

### **What is the use of Docker?**

* answer

# Kubernetes

### **What are Kubernetes Secrets??**

* answer

# AWS

### **Explain AWS services?**

**☁️ What is AWS?**

Amazon Web Services (AWS) is a cloud computing platform offering on-demand resources like servers, databases, storage, and more. It follows a pay-as-you-go model.

**🚀 Core AWS Services (by Category)**

**1. 🖥️ Compute Services**

| **Service** | **Description** |
| --- | --- |
| EC2 (Elastic Compute Cloud) | Virtual servers in the cloud. You can choose OS, size, and performance. |
| Lambda | Run code without provisioning servers (serverless). Supports auto-scaling, pay-per-use. |
| Elastic Beanstalk | Platform-as-a-Service (PaaS) for deploying apps (e.g., Java, Node.js). |
| ECS / EKS | Container services: ECS (Amazon managed), EKS (Kubernetes). |
| Lightsail | Simplified VPS service for small apps/websites. |

**2. 🗃️ Storage Services**

| **Service** | **Description** |
| --- | --- |
| S3 (Simple Storage Service) | Object storage for files, backups, media, etc. |
| EBS (Elastic Block Store) | Block storage for EC2, like virtual hard drives. |
| EFS (Elastic File System) | Shared file storage across multiple EC2 instances. |
| Glacier | Long-term, low-cost archival storage. |

**3. 🛢️ Database Services**

| **Service** | **Description** |
| --- | --- |
| RDS | Managed relational databases (MySQL, PostgreSQL, Oracle, SQL Server, MariaDB). |
| Aurora | High-performance, MySQL/PostgreSQL-compatible cloud DB. |
| DynamoDB | Serverless NoSQL database, key-value store. |
| Redshift | Data warehousing for analytics. |
| ElastiCache | In-memory caching (Redis/Memcached). |

**4. 🌐 Networking & CDN**

| **Service** | **Description** |
| --- | --- |
| VPC (Virtual Private Cloud) | Isolated virtual network for AWS resources. |
| Route 53 | DNS and domain name management. |
| CloudFront | Content Delivery Network (CDN) to serve static and dynamic content globally. |
| API Gateway | Expose REST/HTTP APIs to the internet or internal apps. |
| Load Balancer (ALB/ELB/NLB) | Distribute traffic across EC2 or containers. |

**5. 🛡️ Security & Identity**

| **Service** | **Description** |
| --- | --- |
| IAM (Identity and Access Management) | User/role management and fine-grained permission control. |
| KMS (Key Management Service) | Encryption key management. |
| Secrets Manager / Parameter Store | Store and manage secrets like API keys, DB credentials. |
| Cognito | User authentication and access control for web/mobile apps. |

**6. 📈 Monitoring & DevOps**

| **Service** | **Description** |
| --- | --- |
| CloudWatch | Monitor logs, metrics, and set alerts. |
| CloudTrail | Audit AWS API calls and account activity. |
| CodePipeline | CI/CD pipeline orchestration. |
| CodeBuild / CodeDeploy | Build and deploy automation. |
| CloudFormation | Infrastructure as Code (IaC) using YAML/JSON templates. |
| Elastic Container Registry (ECR) | Container image storage and versioning. |

**7. 🧠 AI/ML Services**

| **Service** | **Description** |
| --- | --- |
| SageMaker | End-to-end machine learning platform. |
| Rekognition | Image and video analysis (face, objects, text). |
| Comprehend | Natural Language Processing (sentiment, entity recognition). |
| Lex | Build conversational interfaces (chatbots). |
| Translate / Polly / Textract | Language translation, speech synthesis, text extraction. |

**8. 🛠️ Developer Tools**

| **Service** | **Description** |
| --- | --- |
| AWS CLI | Command-line tool for managing AWS resources. |
| AWS SDKs | Language-specific libraries (Java, Python, Node.js, etc.). |
| Cloud9 | Online IDE integrated with AWS. |

**9. 📦 Other Popular Services**

| **Service** | **Description** |
| --- | --- |
| SQS (Simple Queue Service) | Message queuing for decoupling systems. |
| SNS (Simple Notification Service) | Pub/Sub for notifications (SMS, email, Lambda, etc.). |
| Step Functions | Workflow orchestration service. |

### **What is cloud computing?**

* answer

### **Explain services provided by cloud computing?**

* answer

### **How to deploy service to AWS?**

* answer

### **Which services have you used?**

* answer

### **What are different services in EKS?**

* answer

### **What is the benefit of Serverless services? Is EKS serverless?**

* answer

# Azure

### **question?**

* answer

# Angular

### **What are the components in the Angular?**

* answer

### **How to pass data from components in Angular?**

* answer

### **What does the component.ts file contain in angular?**

* answer

### **Explain two way data binding in angular?**

* answer

### **How to make backend calls in angular?**

* answer

### **What is authguard in angular?**

* answer

### **What is interpolation in angular?**

* answer

### **How to create components in angular?**

* answer

### **Explain angular directives?**

* answer

### **What are observables in angular?**

* answer

### **How to implement dependency injection in angular?**

* answer

### **What are different components of Angular?**

* answer

### **How to handle caching in Angular?**

* answer

# React

### **Features of ReactJS?**

* answer

### **How does Virtual DOM work?**

* answer

### **Features of Virtual DOM?**

* answer

### **What is reducing modifications in Virtual DOM?**

* answer

### **What do you call the backend API from ReactJS?**

* answer

### **What are other Hooks in React?**

* answer

### **What is the purpose of Hook?**

* answer

### **What is the dependency array in useEffect?**

* answer

### **What is the use and significance of dependency arrays?**

* answer

### **How to make a call to an external API from ReactJS?**

* answer

### **What is an alternative to Axios?**

* answer

# Others

### **Explain Jira?**

* answer

### **What is the output of below program?**

#include<stdio.h>

int main()

{

int a =0;

1 + 1 - 1 + 1

a=a++ + ++a - a++ + ++a;

printf(“%d\n”,a);

return 0;

}

* answer

### **What is https protocol?**

* answer

### **How does https internally work?**

* answer

### **What is a certificate?**

* Answer

### **Difference between http and https?**

* answer

### **What is a Sprint retrospective?**

* answer

### **Do you know anything about CI/CD?**

* answer

### **How will Jenkins' job work? Explain the process?**

* answer

### **What is Node JS?**

* answer

### **How is your application getting deployed? Which platform are you using in your project?**

* answer

### **What do you know about Asynchronous communication?**

* answer

### **What are the measure objects i.e key things in order to make Asynchronous calls?**

* answer

### **If I have two different technologies then how can we achieve asynchronous communication?**

* answer

### **Do you have exposure in AI and ML?**

* answer

### **Examples of Fund Transfer are Synchronous or Asynchronous?**

* answer

### **Is WhatsApp Synchronous or Asynchronous?**

* answer

### **Difference between 2 Tier and 3 Tier Architecture?**

* answer

### **Have you heard about the log4j vulnerability?**

* answer

### **What is https protocol?**

* answer

### **Difference between http and https?**

* answer

### **Suppose in server one job at 9 AM and copy json file and paste it in temp folder another job 6 PM read it and load in db and no one should change the file. How to prevent file from any Change?**

* answer

### **If we increase timeout then how can we resolve it? Deployment related issue and how will you handle it?**

* answer

# Linux

### **Explain Linux commands?**

* answer

### **Command - Read a file and print first two lines from file Ans. head -2 filename?**

* answer

### **Search a word from a file?**

* grep 'word' filename ack 'pattern' /path/to/file.txt

### **Why do you want to change your current organization?**

* answer

# Puzzles

### **Puzzle – 1**

Weighing the 9 balls puzzle, find the heavier ball among 9 balls. How many max iterations will it be required to find the heavier ball?

* answer

### **Puzzle – 2**

You are doing some gardening, and need exactly 4 litres of water to mix up some special formula for your award-winning roses. But you only have a 5-liter and a 3-liter bowl, but do have access to plenty of water.

How would you measure exactly 4 litres?

* Fill the 5-liter bowl. Then fill the 3-liter bowl from the 5-liter bowl. You will now have 2 litres left in the 5 litres bowl. Empty the 3-liter bowl, and then transfer the 2 litres from the 5-liter bowl into it. Now fill the 5-liter bowl again, then pour water carefully from the 5-liter bowl into the 3-liter bowl until it is full - exactly one more litre. The 5-liter bowl now has exactly 4 litres.

### **Puzzle – 3**

Divide square in 5 equal parts?