Company Interview Question

---------------------------------Neosofttech----------------------------------

Q1) Write a program to count and print the number of occurrences of each unique character in the below String.

"I live in India"

Ans:-

```
        //count stream using java8 Stream
        public static void Java8Stream()
        {
        String str= "I live in India";
        Map<String, Long> result =
Arrays.stream(str.split("")).map(String::toLowerCase).collect(Collectors.gr
oupingBy(s -> s, LinkedHashMap::new, Collectors.counting()));
        System.out.println(result);
        }
==========================================================

        public static void forLoop() {

                String input = "I live in India";

                Map<Character,Integer> hash = new HashMap<Character,Integer>();

                  for(int i=0;i<input.length();i++)
                  {
                      if(hash.containsKey(input.charAt(i)))
                      hash.put(input.charAt(i), hash.get(input.charAt(i))+1);

                      else
                      hash.put(input.charAt(i), 1);
                  }
                  System.out.println(hash);
        }
```

Q2) How many variables and objects will be created in the following block of code:

String s1 = "Welcome";

String s2 = new String("Welcome");

String s3 = "Welcome";

Ans)-

Variables:

- 3 variables are created: `s1`, `s2`, and `s3`. Each variable holds a reference to a String object.

Objects:

- 2 String objects are created:

1. "Welcome" in the String pool:
   - When `s1 = "Welcome";` is executed, the JVM checks the String pool for a string with the value "Welcome". Since it doesn't exist yet, a new String object is created in the pool and `s1` refers to it.
   - When `s3 = "Welcome";` is executed, the JVM finds the existing "Welcome" string in the pool and assigns a reference to it to `s3`. No new object is created.
2. New String object in the heap:
   - When `s2 = new String("Welcome");` is executed, the `new` keyword forces the creation of a new String object in the heap, even though the same content exists in the pool. `s2` refers to this new object.

Summary:

- Variables: 3 (s1, s2, s3)
- Objects: 2 (one in the String pool, one in the heap)

Key points:

- The String pool is a memory area where the JVM stores string literals to optimize memory usage and string comparisons.
- Using the `new` keyword with strings bypasses the pool and creates a new object in the heap.
- To check if two strings refer to the same object in memory, use the `==` operator (for reference equality) or the `equals()` method (for content equality).

Q3) How many types of relationship in spring data jpa?

There can be 4 types of association mapping in hibernate/JPA.

1. One to One

2. One to Many
3. Many to One
4. Many to Many

Q4) write a program for one to many relationship?

Ans)In unidirectional association, the source entity has a relationship field that refers to the target entity and the source entity's table contains the foreign key.

In a bidirectional association, each entity (i.e. source and target) has a relationship field that refers to each other and the target entity's table contains the foreign key. The source entity must use the mappedBy attribute to define the bidirectional one-to-one mapping.

1 Unidirectional Mapping:-

```
@Entity
@Table(name="orders")
public class Order {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="id")
    private Long id;

    @Column(name="order_tracking_number")
    private String orderTrackingNumber;

    // one to many unidirectional mapping
    // default fetch type for OneToMany: LAZY
    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    @JoinColumn(name = "order_id", referencedColumnName = "id")
    private Set<OrderItem> orderItems = new HashSet<>();
}
```

```
@Entity
@Table(name="order_items",schema = "ecommerce")
public class OrderItem {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="id")
    private Long id;

    @Column(name="image_url")
    private String imageUrl;
}
```

2 Bidirectional Mapping:-

```
@Entity
@Table(name="orders")
public class Order {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="id")
    private Long id;

    @Column(name="order_tracking_number")
    private String orderTrackingNumber;

    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER, mappedBy =
"order")
    private Set<OrderItem> orderItems = new HashSet<>();
```

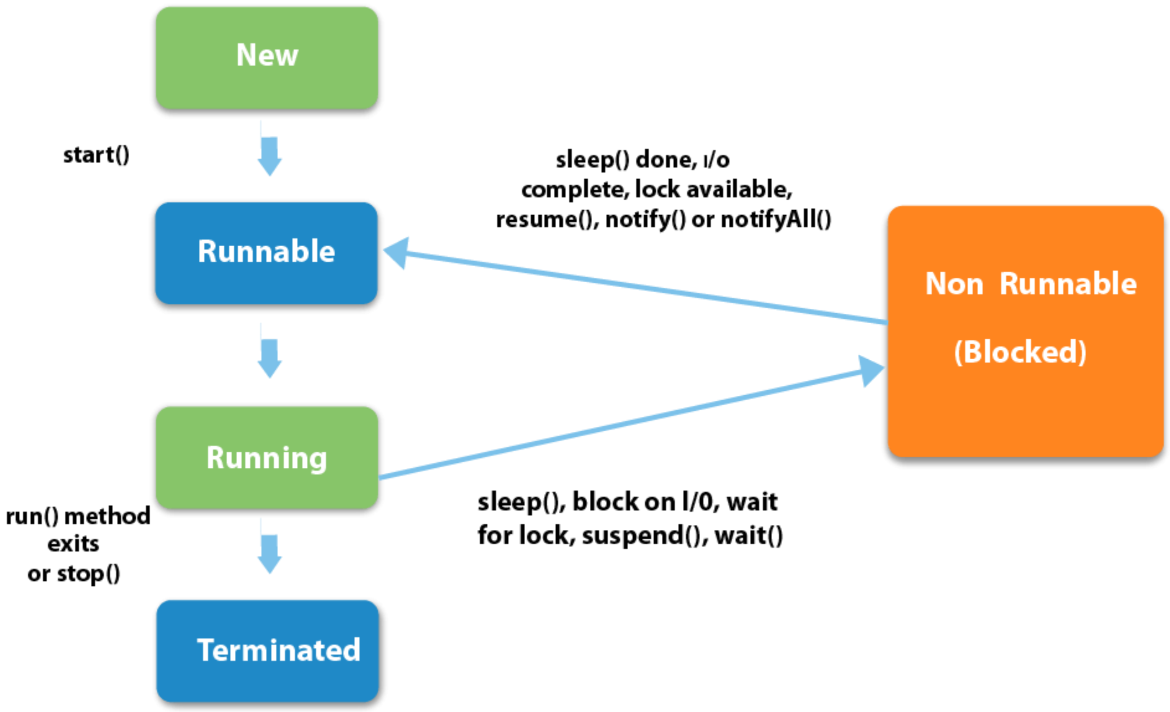
```
@Entity
@Table(name="order_items",schema = "ecommerce")
public class OrderItem {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="id")
    private Long id;

    @Column(name="image_url")
    private String imageUrl;


 // default fetch type for ManyToOne: EAGER
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "order_id")
    private Order order;
```

Q5) What are the states in the lifecycle of a Thread?

Ans) A thread can have one of the following states during its lifetime:

1. **New:** In this state, a Thread class object is created using a new operator, but the thread is not alive. Thread doesn't start until we call the start() method.

2. **Runnable:** In this state, the thread is ready to run after calling the start() method. However, the thread is not yet selected by the thread scheduler.
3. **Running:** In this state, the thread scheduler picks the thread from the ready state, and the thread is running.
4. **Waiting/Blocked:** In this state, a thread is not running but still alive, or it is waiting for the other thread to finish.
5. **Dead/Terminated:** A thread is in terminated or dead state when the run() method exits.


New
start()
Runnable
sleep() done, I/o complete, lock available, resume(), notify() or notifyAll()
Non Runnable (Blocked)
Running
sleep(), block on I/0, wait for lock, suspend(), wait()
run() method exits or stop()
Terminated

Q6) What is the Collection framework in Java?

Collection Framework is a combination of classes and interface, which is used to store and manipulate the data in the form of objects. It provides various classes such as ArrayList, Vector, Stack, and HashSet, etc. and interfaces such as List, Queue, Set, etc. for this purpose.

Q7) What is the cache mechanism in java?

Ans) Caching is a technique used to store and manage frequently accessed data in a way that allows future requests for that data to be served more quickly. In Java, there are several mechanisms and libraries available for implementing caching. Here are some commonly used ones:

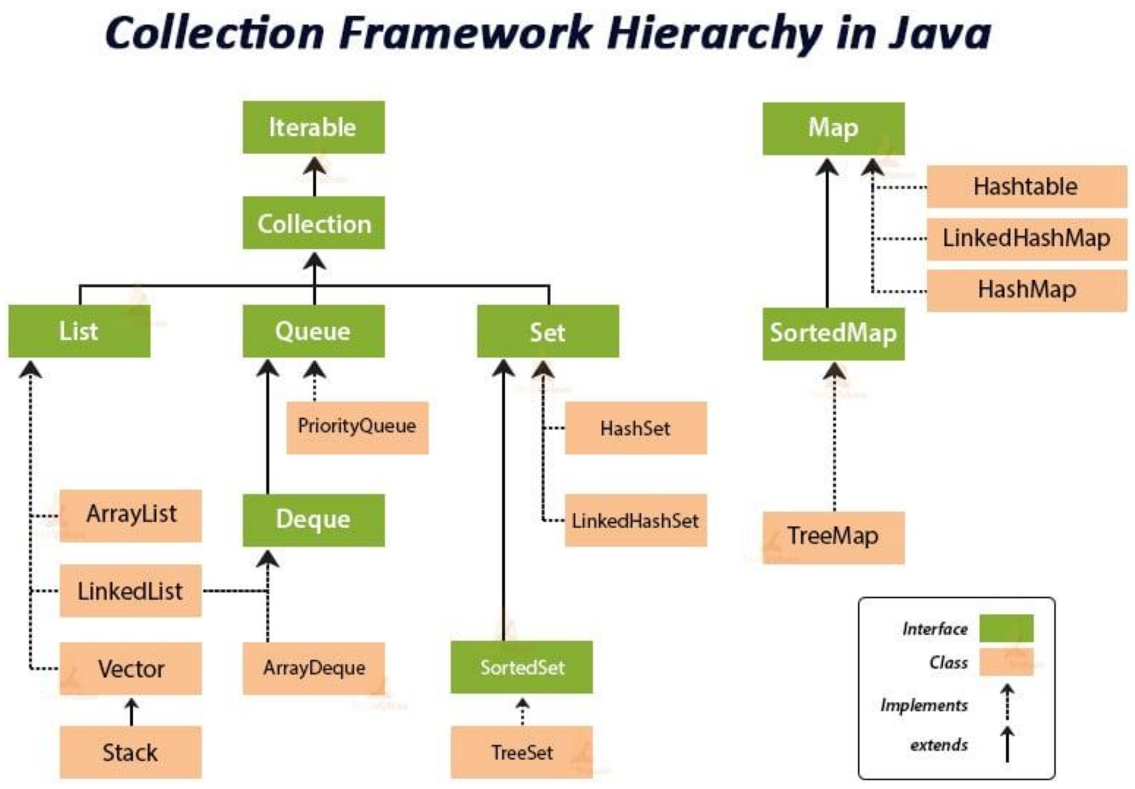
Company Interview Question

1. **Java Caching API (JCache):**
    - JCache is a standard caching API introduced in Java EE 7 (JSR-107) and is also available in Java SE 8. It provides a set of standard annotations and APIs for caching.
    - Popular implementations of JCache include Ehcache, Hazelcast, and Infinispan.
2. **Ehcache:**
    - Ehcache is an open-source, widely used caching library for Java. It supports in-memory caching and can be configured to use disk storage as well.
    - Ehcache can be used standalone or integrated with other frameworks, such as Spring.
3. **Guava Cache:**
    - Google Guava provides a caching library that is easy to use and integrates well with other Guava utilities. It supports features like time-based expiration, size-based eviction, and asynchronous loading.
    - Guava Cache is suitable for in-memory caching within a single JVM.
4. **Caffeine:**
    - Caffeine is a modern caching library for Java 8 and later. It offers high-performance, automatic eviction policies, and supports features like expiration, refresh, and asynchronous loading.
    - Caffeine is designed to be lightweight and memory-efficient.
5. **Spring Caching:**
    - The Spring Framework provides a caching abstraction that can be easily integrated into Spring-based applications. It supports various caching providers, including Ehcache, Caffeine, and others.
    - Spring Caching can be configured using annotations, such as `@Cacheable`, `@CachePut`, and `@CacheEvict`.
6. **Redis:**
    - Redis is an in-memory data structure store that can be used as a distributed cache. It is often used as a caching solution in microservices architectures.
    - The Jedis library or the Spring Data Redis project can be used to interact with Redis as a caching provider.

7. **Memcached:**
   - Memcached is a distributed memory caching system. It allows you to store key-value pairs in memory and is often used in web applications to cache frequently accessed data.
   - There are Java client libraries like Spymemcached for interacting with Memcached.
8. **Hazelcast:**
   - Hazelcast is an open-source distributed caching and computing platform. It provides an in-memory data grid that can be used for caching and distributed computing.
   - Hazelcast can be integrated with Java applications to provide distributed caching capabilities.

The choice of a caching mechanism depends on the specific requirements of your application, such as performance, scalability, and the need for distributed caching. It's important to carefully select and configure a caching solution based on your use case.

Q8) Collection classes and interfaces?



Q9) What is ORM?

Ans:- ORM is an acronym for Object/Relational mapping. It is a programming strategy to map object with the data stored in the database. It simplifies data creation, data manipulation, and data access.

Q10) What is Spring Data JPA?

Ans:- It is a library/framework that adds an extra layer of abstraction on top of our JPA provider (like Hibernate). We can use Spring Data JPA to reduce the amount of boilerplate code required to implement the data access object (DAO) layer.

Q11) What is dispatcher servlet?

Ans:- The Dispatcher Servlet Class Works as the front controller. It dispatches the request to the appropriate controller and manages the flow of the application.

Q12) How to communicate Microservices?

Ans) There are two styles of Microservices Communications:

1. Synchronous Communication
2. Asynchronous Communication

Synchronous Communication

In the case of Synchronous Communication, the client sends a request and waits for a response from the service. The important point here is that the protocol (HTTP/HTTPS) is synchronous and the client code can only continue its task when it receives the HTTP server response.

For example, **Microservice1 acts as a client that sends a request and waits for a response from Microservice2.**

We can use **RestTemplate** or **WebClient** or **Spring Cloud Open Feign library** to make a Synchronous Communication multiple microservices.

Company Interview Question

Asynchronous Communication

In the case of Asynchronous Communication, The client sends a request and does not wait for a response from the service. The client will continue executing its task - It doesn’t wait for the response from the service.

For example, **Microservice1 acts as a client that sends a request and doesn't wait for a response from Microservice2.**

We can use Message brokers such as **RabbitMQ** and **Apache Kafka** to make Asynchronous Communication between multiple microservices.

---------------------------------Capgemini----------------------------------

Q1) How to sort list of employee first age in asc then salary in desc order?

```
Comparator<Employee> comparator =
Comparator.comparing(Employee::getAge)
.thenComparing(Employee::getSalary, Comparator.reverseOrder());

Only sort
        employees.sort(comparator);

for collecting new list
List<Employee> sortedEmployees = employees.stream()
                                          .sorted(comparator)
                                          .collect(Collectors.toList());
```

Q2) difference between ClassNotFoundException and NoClassDefFoundError in java?

The **ClassNotFoundException** occurs when you try to load a class at runtime using **Class.forName()** or **loadClass()** methods and requested classes are not found in classpath. Most of the time this exception will occur when you try to run an application without updating the classpath with JAR files. This exception is a **checked Exception** derived from **java.lang.Exception** class and you need to provide **explicit handling** for it.

The **NoClassDefFoundError** occurs when the class was present during compile time and the program was compiled and linked successfully but the class was not present during runtime. It is an error that is derived from **LinkageError**.

What is **LinkageError**? If a class is dependent on another class and we made changes in that class after compiling the former class, we will get the **LinkageError**.

Q3) will functional interface replace interface in java?

No, a functional interface will not replace the general concept of an interface in Java. Instead, functional interfaces are a special type of interface introduced in Java 8 to support the new features related to lambda expressions and the functional programming paradigm.

- Abstract methods:
  - Functional interfaces have exactly one abstract method.
  - Regular interfaces can have any number of abstract methods.
- Purpose:
  - Functional interfaces primarily represent single-method behaviors, enabling lambda expressions and method references.
  - Regular interfaces define contracts for broader types, often with multiple methods and properties.

Q4) how can you make object immutable in java?

To create an immutable class in Java, you need to follow these general principles:

1. Declare the class as `final` so it can't be extended.
2. Make all of the fields `private` so that direct access is not allowed.
3. Don't provide setter methods for variables.
4. Make all mutable fields `final` so that a field's value can be assigned only once.
5. Initialize all fields using a constructor method performing deep copy.
6. Perform cloning of objects in the getter methods to return a copy rather than returning the actual object reference.

```
// Java Program to Create An Immutable Class

// Importing required classes
import java.util.HashMap;
import java.util.Map;

// Class 1
// An immutable class
```

Company Interview Question

```java
final class Student {

    // Member attributes of final class
    private final String name;
    private final int regNo;
    private final Map<String, String> metadata;

    // Constructor of immutable class
    // Parameterized constructor
    public Student(String name, int regNo,
                   Map<String, String> metadata)
    {

        // This keyword refers to current instance itself
        this.name = name;
        this.regNo = regNo;

        // Creating Map object with reference to HashMap
        // Declaring object of string type
        Map<String, String> tempMap = new HashMap<>();

        // Iterating using for-each loop
        for (Map.Entry<String, String> entry :
             metadata.entrySet()) {
            tempMap.put(entry.getKey(), entry.getValue());
        }

        this.metadata = tempMap;
    }

    // Method 1
    public String getName() { return name; }

    // Method 2
    public int getRegNo() { return regNo; }

    // Note that there should not be any setters

    // Method 3
    // User -defined type
    // To get meta data
    public Map<String, String> getMetadata()
    {

        // Creating Map with HashMap reference
        Map<String, String> tempMap = new HashMap<>();

        for (Map.Entry<String, String> entry :
             this.metadata.entrySet()) {
            tempMap.put(entry.getKey(), entry.getValue());
        }
        return tempMap;
    }
}

// Class 2
// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        // Creating Map object with reference to HashMap
        Map<String, String> map = new HashMap<>();

        // Adding elements to Map object
        // using put() method
        map.put("1", "first");
        map.put("2", "second");

        Student s = new Student("ABC", 101, map);

        // Calling the above methods 1,2,3 of class1
        // inside main() method in class2 and
        // executing the print statement over them
        System.out.println(s.getName());
        System.out.println(s.getRegNo());
        System.out.println(s.getMetadata());

        // Uncommenting below line causes error
        // s.regNo = 102;

        map.put("3", "third");
        // Remains unchanged due to deep copy in constructor
        System.out.println(s.getMetadata());
        s.getMetadata().put("4", "fourth");
        // Remains unchanged due to deep copy in getter
        System.out.println(s.getMetadata());
    }
}
```

Company Interview Question

Q5) What is difference between Get and Post method?

GET	POST
1) limited amount of data can be sent because data is sent in header.	large amount of data can be sent because data is sent in body.
2) not secured because data is exposed in URL bar.	secured because data is not exposed in URL bar.
3) can be bookmarked.	cannot be bookmarked.
4) idempotent . It means second request will be ignored until response of first request is delivered	non-idempotent.
5) It is more efficient and used more than Post.	It is less efficient and used less than get.

more details...

Q6) Difference between @RequestParam and @QueryParam?

@RequestParam is associated with the Spring MVC framework, while **@QueryParam** is associated with JAX-RS. Both annotations are used to extract parameters from the URL, but you would choose the one that corresponds to the framework you are working with. If you are working with Spring MVC, use **@RequestParam**; if you are working with JAX-RS, use **@QueryParam**.

-----------------------------------IBM---

Q1) program to Count/Collect Strings whose length is less than 2 in String.

```
String str = "India is my country";

long count = Arrays.stream(str.split("\\s+"))
.filter(word -> word.length() < 3).count();
```

```
List<String> words = Arrays.stream(str.split("\\s+"))
.filter(word -> word.length() < 3).collect(Collectors.toList());
```

Q2) what is the best way to rollback all api in microservice in spring boot?

Ans:- **saga pattern**

Q3) How is garbage collection controlled?

Garbage collection is managed by JVM. It is performed when there is not enough space in the memory and memory is running low. We can externally call the System.gc() for the garbage collection. However, it depends upon the JVM whether to perform it or not.

Q4) What is Garbage Collection?

Garbage collection is a process of reclaiming the unused runtime objects. It is performed for memory management. In other words, we can say that It is the process of removing unused objects from the memory to free up space and make this space available for Java Virtual Machine. Due to garbage collection java gives 0 as output to a variable whose value is not set, i.e., the variable has been defined but not initialized. For this purpose, we were using free() function in the C language and delete() in C++. In Java, it is performed automatically. So, java provides better memory management.

Q5) If you have 1000 record and you need to fetch one value. which one is better arraylist or linkedlist?

In your scenario of needing to fetch only one value from a list of 1000 records, an **ArrayList** is the better choice over a LinkedList. Here's why:

ArrayList:

- **Faster access:** Accessing an element in an ArrayList takes constant time (O(1)), regardless of its position in the list. This is because elements are stored in contiguous memory locations, allowing direct addressing.
- **Simpler implementation:** An ArrayList is a simpler data structure, leading to less overhead and faster execution.

LinkedList:

- **Slower access:** Accessing an element in a LinkedList takes linear time (O(n)) on average, as you need to traverse the list from the beginning until you find the desired element.

- **More complex:** LinkedLists have additional memory overhead due to the pointers holding references to other nodes, contributing to slightly slower performance.

Therefore, for the specific task of fetching a single value, the constant-time access of an ArrayList makes it significantly faster than the linear-time access of a LinkedList, especially for larger datasets like 1000 records.

However, if your frequent operations involve **frequent insertions or deletions in the middle** of the list, a LinkedList might be better suited as it only requires updating pointers, while an ArrayList requires shifting elements, which can be slower.

Remember:

- Consider the most frequent operations you will perform on the list before making a decision.
- For memory-constrained applications, the smaller memory footprint of an ArrayList might be a factor.
- If random access and fast retrieval are priorities, an ArrayList is the clear winner in this case.

Q6) synchronize block or synchronize method in singleton class?

In summary, **choose synchronized blocks for fine-grained control and performance**, while **synchronized methods offer simplicity and reduced deadlock risk**. Weigh the trade-offs based on your specific requirements to select the most appropriate approach.

Feature	Synchronized Block	Synchronized Method
Scope of Locking	Only specific code block within synchronized keyword	Entire object for the duration of the method
Granularity	Fine-grained control	Less granular, entire method locked
Performance	Generally more performant	Less performant due to entire object locking
Complexity	Requires careful identification of critical section	Easier to implement, entire method synchronized
Deadlocks	Higher risk	Reduced risk due to consistent lock order
Use Cases	Specific sections of code needing protection, performance critical	Simpler implementations, entire method needs protection, deadlock concern

Additional Notes:

- Both can be combined in a single class.
- Alternative approaches like `volatile` keyword or concurrency libraries might be suitable depending on the situation.

Q7) what is singleton design pattern?

Singleton Pattern says that just**"define a class that has only one instance and provides a global point of access to it".**

In other words, a class must ensure that only single instance should be created and single object can be used by all other classes.

How to create Singleton design pattern?

To create the singleton class, we need to have static member of class, private constructor and static factory method.

- **Static member:** It gets memory only once because of static, it contains the instance of the Singleton class.
- **Private constructor:** It will prevent to instantiate the Singleton class from outside the class.
- **Static factory method:** This provides the global point of access to the Singleton object and returns the instance to the caller.

Lazy Instantiation

```
// Classical Java implementation of singleton
// design pattern
class Singleton
{
    private static Singleton obj;

    // private constructor to force use of
    // getInstance() to create Singleton object
    private Singleton() {}

    public static Singleton getInstance()
    {
        if (obj==null)
            obj = new Singleton();
        return obj;
    }
}
```

early Instantiation of Singleton Pattern

```
// Static initializer based Java implementation of
// singleton design pattern
class Singleton
{
    private static Singleton obj = new Singleton();
    private Singleton() {}

    public static Singleton getInstance()
    {
        return obj;
    }
}
```

Q8) difference between protected and default keyword in java?

- **Protected** Protected can be accessed by the class of the same package, or by the sub-class of this class, or within the same class.
- **Default** Default are accessible within the package only. By default, all the classes, methods, and variables are of default scope.

---------------------------------LTIMINDTREE---

Q1) How to call singleton class method?

Step 1

Create a Singleton Class.

SingleObject.java

```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new
SingleObject();

   //make the constructor private so that this class
cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

Step 2

Get the only object from the singleton class.

SingletonPatternDemo.java

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor
SingleObject() is not visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

Q2) How to remove duplicate elements from the list./ how to get unique value through list using java 8?

```
List<Integer> originalList2 = new ArrayList<>(Arrays.asList(1, 2, 2, 3, 4,
4, 5));
            List<Integer> newList2 = originalList.stream()
                    .distinct()
                    .collect(Collectors.toList());
-----------------------------------------------------------------------

Set<Integer> hset = new HashSet<>(originalList1);
            List<Integer> newList1 = new ArrayList<>(hset);
```

Company Interview Question

Q3) How to find all the even numbers that exists in the list.

```
List<Integer> list=Arrays.asList(7,3,2,9,5,32);
listInt.stream().filter(n->n%2==0).forEach(System.out::println);
```

Q4) How to produce xml type value in spring boot rest api?

```
@GetMapping(path = "/{id}", produces = MediaType.APPLICATION_XML_VALUE)
```

Q5) what is the output of the below program?

```
class PrintMessage {
	void msg(Object obj) {
		System.out.println("Object");
	}

	void msg(String str) {
		System.out.println("String");
	}

	void msg(Integer itr) {
		System.out.println("Integer");
	}

	public static void main(String[] args) {
		PrintMessage pm = new PrintMessage();
		pm.msg(null);
	}
}
```

Answer:- It will throw ambiguous error**.**

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem: 
	The method msg(Object) is ambiguous for the type PrintMessage

	at com.vk.interview.PrintMessage.main(PrintMessage.java:18)
```

Q6) what is the output of the below program?

```
		try{
			System.out.println("In Try");
			return 1;
		} catch (Exception e){
			System.out.println("In Catch");
		} finally{
			System.out.println("In Finally");
		}
```

Output:- It will print In Try and In Finally.

```
In Try
In Finally
```

Q7) Which component will it call?

```
const routes: Routes = [
{ path: 'first-component', component: FirstComponent },
{ path: 'second-component', component: SecondComponent },
{ path: '**', component: FirstComponent },
];
/third-component ?
```

Answer:- `FirstComponent`

Q8) What is the difference between hashCode() and equals() method?

Solution:-

In Java, the `hashCode()` method and the `equals()` method are two distinct methods that serve different purposes, but they are often related when working with hash-based data structures like HashMap. Here's a tabular representation of the differences between `hashCode()` and `equals()` in Java:

Feature	`hashCode()`	`equals()`
Purpose	Generates a hash code for an object.	Compares two objects for equality.
Return Type	`int`	`boolean`
Usage in	Used in hash-based collections like	Used in various collections for

Feature	`hashCode()`	`equals()`
Collections	HashMap.	comparison.
Contractual Agreement	Objects that are equal must have the same hash code. However, objects with the same hash code are not necessarily equal.	If `a.equals(b)` is `true`, then `a.hashCode()` must be equal to `b.hashCode()`.
Override Requirement	It's a good practice to override `hashCode()` when overriding `equals()`.	If you override `equals()`, you should also override `hashCode()`.
Default Implementation	The default implementation is provided by the `Object` class, which is based on the object's memory address.	The default implementation in the `Object` class compares object references, i.e., it checks whether the two objects refer to the same memory location.
Custom Implementation	You can provide a custom implementation to generate a hash code based on object fields, ensuring that equal objects have the same hash code.	You need to provide a custom implementation based on the equality criteria you want to define for your objects.

It's important to note that while the `hashCode()` method is used to quickly locate a specific object in a hash table, the `equals()` method is used to determine equality between two objects. Overriding these methods correctly is crucial, especially when working with collections that rely on hash codes and equality, to ensure the expected behavior.

Q9) What is SerialVersionUID ?

The serialization process at runtime associates an id with each Serializable class which is known as SerialVersionUID. It is used to verify the sender and receiver of the serialized object. The sender and receiver must be the same. To verify it, SerialVersionUID is used. The sender and receiver must have the same SerialVersionUID, otherwise, **InvalidClassException** will be thrown when you deserialize the object. We can also declare our own SerialVersionUID in the Serializable class. To do so, you need to create a field SerialVersionUID and assign a value to it. It must be of the long type with static and final. It is suggested to explicitly declare the serialVersionUID field in the class and have it private also. For example:

```
private static final long serialVersionUID=1L;
```

Now, the Serializable class will look like this:

Employee.java

```
import java.io.Serializable;
class Employee implements Serializable{
 private static final long serialVersionUID=1L;
 int id;
 String name;
 public Student(int id, String name) {
  this.id = id;
  this.name = name;
 }
}
```

Q10) What is serializable interface in Java?

Serializable interface is a marker interface. The marker interface provides a hint to the Java runtime that the implementing class allows itself to be serialized. The runtime will take advantage of this interface to serialize the object. Serializable interface in java is a special interface to be implemented by data classes in java. When a class implements this interface, it can be persisted in a database. This interface is declared in java.io package. Serializable interface has two methods, readObject() and writeObject() , which are used to read and write object in database.

Q11) What is marker interface?

An interface that does not contain methods, fields, and constants is known as **marker interface**. In other words, an empty interface is known as **marker interface** or **tag interface.** It delivers the run-time type information about an object. It is the reason that the JVM and compiler have additional information about an object.
The **Serializable** and **Cloneable** interfaces are the example of marker interface. In short, it indicates a signal or command to the JVM.

The declaration of marker interface is the same as interface in Java but the interface must be empty. For example:

```
public interface Serializable
{

}

```

Uses of Marker Interface

Marker interface is used as a tag that inform the Java compiler by a message so that it can add some special behavior to the class implementing it. Java marker interface are useful if we have information about the class and that information never changes, in such cases, we use marker interface represent to represent the same. Implementing an empty interface tells the compiler to do some operations.

It is used to logically divide the code and a good way to categorize code. It is more useful for developing API and in frameworks like Spring.

Built-in Marker Interface

In Java, built-in marker interfaces are the interfaces that are already present in the JDK and ready to use. There are many built-in marker interfaces some of them are:

- **Cloneable Interface**
- **Serializable Interface**
- **Remote Interface**

Q12) Can we create custom marker interface?

Yes, We can create custom marker interface.

Q13) What do you mean by Functional Interfaces in Java?/ What is functional interface?

- **Functional Interfaces:** Functional Interfaces are the types of interfaces that can contain only a single abstract method. These interfaces are additionally recognized as Single abstract method interfaces.

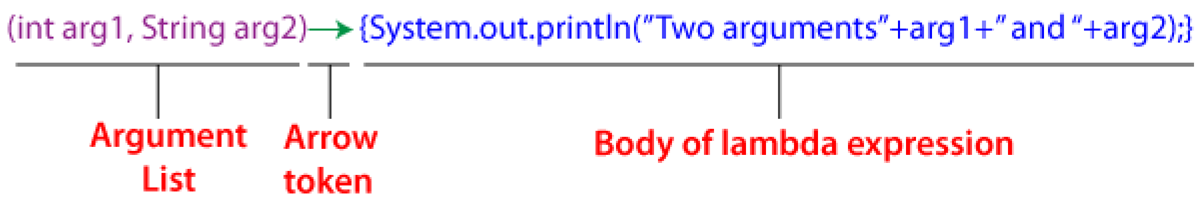
Functional interfaces are included in Java SE 8 with Lambda expressions and Method references in order to make code more readable, clean, and straightforward. These are the interfaces which ensure that they include precisely only one abstract method. It is used and executed by representing the interface with an annotation called **@FunctionalInterface**. As described earlier, functional interfaces can contain only one abstract method. However, they can include any quantity of default and static methods.

Functional Interface is additionally recognized as Single Abstract Method Interfaces. In short, they are also known as **SAM** interfaces.

Q14) What are the Java Lambda Expressions?

- **Lambda Expression:** Lambda expressions are the functions and expressions that are shared and pointed to as an object.

Java lambda expressions were Java's initial step towards functional programming. Java Lambda Expression or Lambda function is simply an unnamed expression or function that is rewritten as a parameter for any other function. Lambda Expressions in Java are the functions that are used to be shared as an object. It can also be used for an object to be referenced. Lambda Expressions need more concise coding, include and it also implements a method of executing the functional interfaces of Java 8. Lambda expressions in Java allow the users to express one functioning unit to carry throughout to different code.


(int arg1, String arg2)→{System.out.println("Two arguments"+arg1+" and "+arg2);}
Argument List
Arrow token
Body of lambda expression

Lambda Expressions Syntax:

```
(Argument List)-> {expression;}
```

or

```
(Argument List)-> {statements;}
```

Example of Lambda Expression in Java:

LambdaExpressionExample.java:

```java
public class LambdaExpressionExample{
public static void main(String[] args) {
  // using lambda Expression
  new Thread(():>System.out.println("Thread is started: using Lambda Expressions")).start();
  // old way
  new Thread(new Runnable() {
   @Override
   public void run() {
    System.out.println("Thread is started: using old method");
   }
  }).start();
}
}
```

Output:

```
Thread is started: using the old method

Thread is started: using Lambda Expressions
```

Q15) How HashMap works internally?/Working of HashMap in java?

What is Hashing

It is the process of converting an object into an integer value. The integer value helps in indexing and faster searches.

What is HashMap

HashMap is a part of the Java collection framework. It uses a technique called Hashing. It implements the map interface. It stores the data in the pair of Key and

Value. HashMap contains an array of the nodes, and the node is represented as a class. It uses an array and LinkedList data structure internally for storing Key and Value. There are four fields in HashMap.

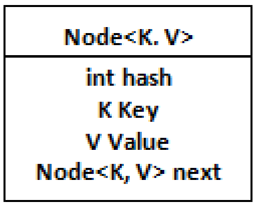


Figure: Representation of a Node

Before understanding the internal working of HashMap, you must be aware of hashCode() and equals() method.

- **equals():** It checks the equality of two objects. It compares the Key, whether they are equal or not. It is a method of the Object class. It can be overridden. If you override the equals() method, then it is mandatory to override the hashCode() method.
- **hashCode():** This is the method of the object class. It returns the memory reference of the object in integer form. The value received from the method is used as the bucket number. The bucket number is the address of the element inside the map. Hash code of null Key is 0.
- **Buckets:** Array of the node is called buckets. Each node has a data structure like a LinkedList. More than one node can share the same bucket. It may be different in capacity.

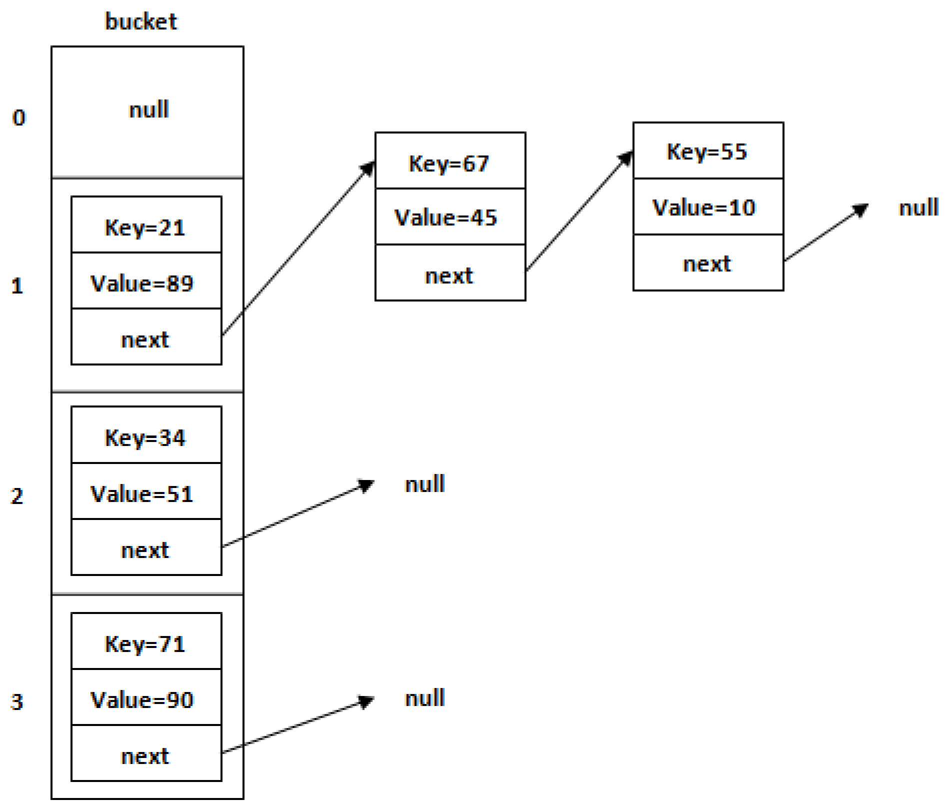


Figure: Allocation of nodes in Bucket

Insert Key, Value pair in HashMap

We use put() method to insert the Key and Value pair in the HashMap. The default size of HashMap is 16 (0 to 15).

Example

In the following example, we want to insert three (Key, Value) pair in the HashMap.

```
1.  HashMap<String, Integer> map = new HashMap<>();
2.  map.put("Aman", 19);
3.  map.put("Sunny", 29);
4.  map.put("Ritesh", 39);
```

Let's see at which index the Key, value pair will be saved into HashMap. When we call the put() method, then it calculates the hash code of the Key "Aman." Suppose the hash code of "Aman" is 2657860. To store the Key in memory, we have to calculate the index.

Calculating Index

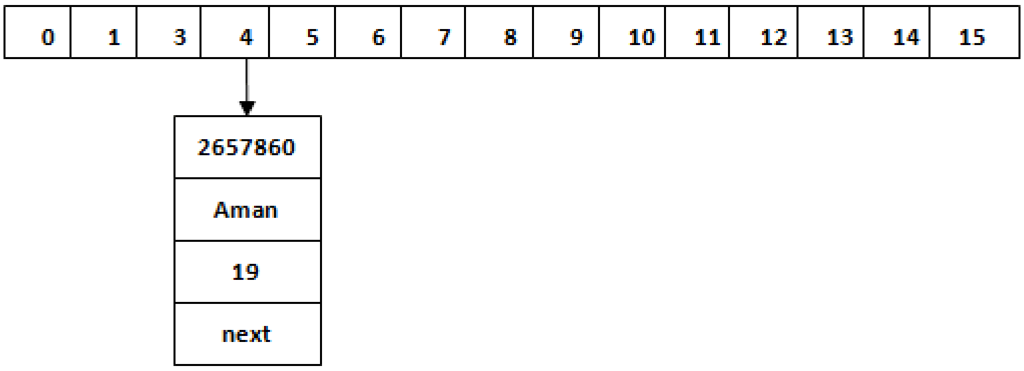
Index minimizes the size of the array. The Formula for calculating the index is:

1. Index = hashcode(Key) & (n-1)

Where n is the size of the array. Hence the index value for "Aman" is:

1. Index = 2657860 & (16-1) = 4

The value 4 is the computed index value where the Key and value will store in HashMap.

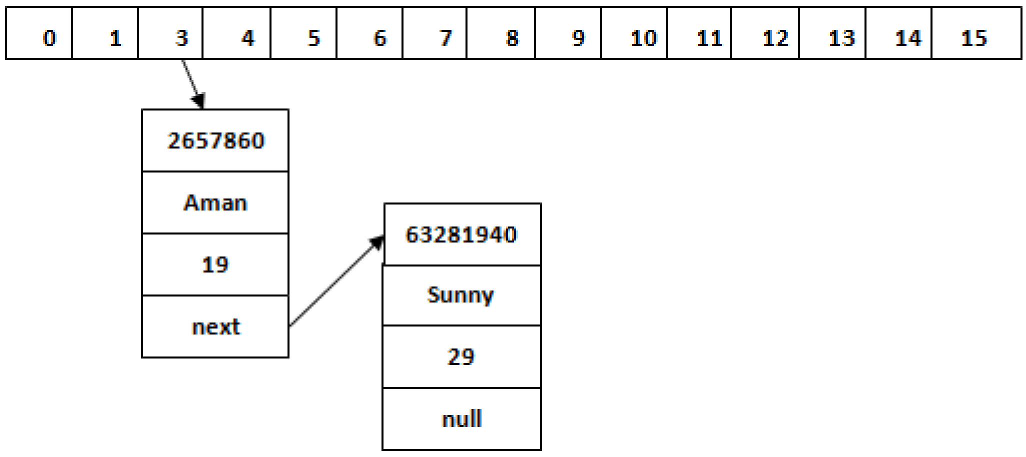


Hash Collision

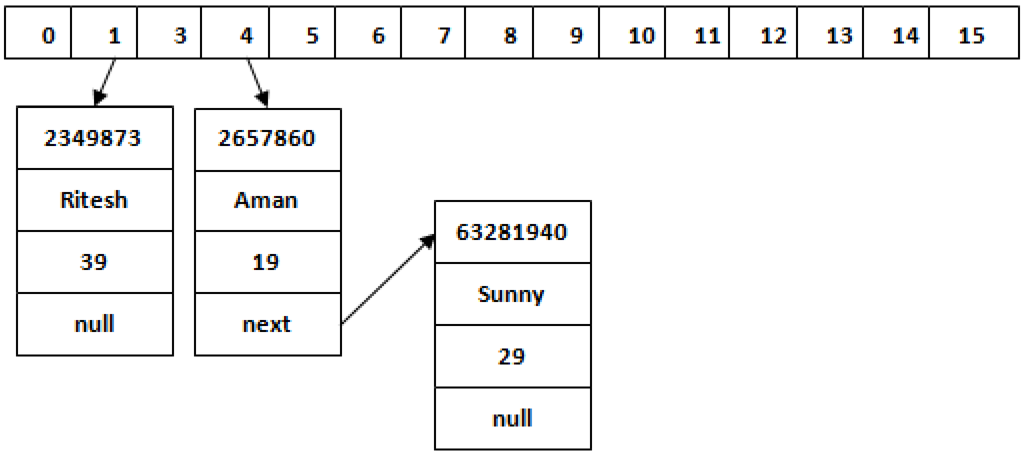
This is the case when the calculated index value is the same for two or more Keys. Let's calculate the hash code for another Key "Sunny." Suppose the hash code for "Sunny" is 63281940. To store the Key in the memory, we have to calculate index by using the index formula.

1. Index=63281940 & (16-1) = 4

The value 4 is the computed index value where the Key will be stored in HashMap. In this case, equals() method check that both Keys are equal or not. If Keys are same, replace the value with the current value. Otherwise, connect this node object to the existing node object through the LinkedList. Hence both Keys will be stored at index 4.



Similarly, we will store the Key "Ritesh." Suppose hash code for the Key is 2349873. The index value will be 1. Hence this Key will be stored at index 1.



get() method in HashMap

get() method is used to get the value by its Key. It will not fetch the value if you don't know the Key. When get(K Key) method is called, it calculates the hash code of the Key.

Suppose we have to fetch the Key "Aman." The following method will be called.

```
1.  map.get(new Key("Aman"));
```

It generates the hash code as 2657860. Now calculate the index value of 2657860 by using index formula. The index value will be 4, as we have calculated above. get() method search for the index value 4. It compares the first element Key with the given Key. If both keys are equal, then it returns the value else check for the next element

in the node if it exists. In our scenario, it is found as the first element of the node and return the value 19.

Let's fetch another Key "Sunny."

The hash code of the Key "Sunny" is 63281940. The calculated index value of 63281940 is 4, as we have calculated for put() method. Go to index 4 of the array and compare the first element's Key with the given Key. It also compares Keys. In our scenario, the given Key is the second element, and the next of the node is null. It compares the second element Key with the specified Key and returns the value 29. It returns null if the next of the node is null.

Q16) what is try-with-resources statement in java?

The try-with-resources statement is a try statement that declares one or more resources. A *resource* is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable, can be used as a resource.

Q17) What is try block?

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

Q18) What is catch block?

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Q19) What is finally block?

The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not. In other words, we can say that finally block is the block which is always executed. Finally block follows try or catch block. If you don't handle the exception, before terminating the program, JVM runs finally block, (if any). The finally block is mainly used to place the cleanup code such as closing a file or closing a connection. Here, we must know that for each try block there can be zero or more catch blocks, but only one finally block. The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

Q20) how many types of Data binding in angular?

In Angular, there are primarily two types of data binding:

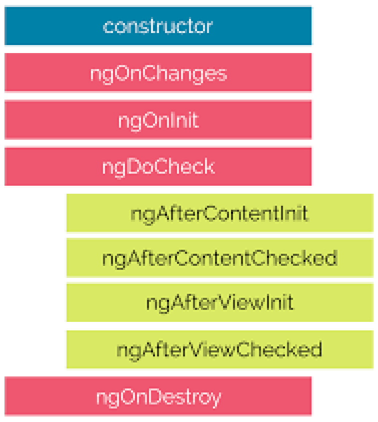
1. **One-Way Data Binding:** This involves binding data from the component to the view (HTML template) or from the view to the component, but not both simultaneously. One-way data binding can be further categorized into:
    - **Interpolation:** This is denoted by double curly braces `{{ }}`. It allows you to bind component data directly into the HTML template.
    - **Property Binding:** This involves binding data from the component to an HTML element property using square brackets `[property]="data"`.
    - **Attribute Binding:** Similar to property binding but for HTML attributes instead of properties.
    - **Class Binding:** This is used to add or remove CSS classes dynamically based on component data.
    - **Style Binding:** It allows you to set inline styles dynamically based on component data.
2. **Two-Way Data Binding:** This involves synchronization of data between the component and the view in both directions. It allows changes in the model (component) to automatically update the view and vice versa. It's denoted by `[(ngModel)]` and requires importing

`FormsModule` or `ReactiveFormsModule` from `@angular/forms`. Two-way data binding combines property binding and event binding.

Angular also provides a form of event binding where you can listen for events such as `(click)`, `(input)`, `(change)`, etc., and react to them in your component code.

Q21)What are lifcycle hooks available?/ angular component lifecycle?

Angular application goes through an entire set of processes or has a lifecycle right from its initiation to the end of the application. The representation of lifecycle in pictorial representation as follows,



The description of each lifecycle method is as below,

i. **ngOnChanges:** When the value of a data bound property changes, then this method is called.
ii. **ngOnInit:** This is called whenever the initialization of the directive/component after Angular first displays the data-bound properties happens.
iii. **ngDoCheck:** This is for the detection and to act on changes that Angular can't or won't detect on its own.
iv. **ngAfterContentInit:** This is called in response after Angular projects external content into the component's view.
v. **ngAfterContentChecked:** This is called in response after Angular checks the content projected into the component.
vi. **ngAfterViewInit:** This is called in response after Angular initializes the component's views and child views.
vii. **ngAfterViewChecked:** This is called in response after Angular checks the component's views and child views.

viii. **ngOnDestroy:** This is the cleanup phase just before Angular destroys the directive/component.

Q22) What is the purpose of async pipe in angular?

The AsyncPipe subscribes to an observable or promise and returns the latest value it has emitted. When a new value is emitted, the pipe marks the component to be checked for changes.

Let's take a time observable which continuously updates the view for every 2 seconds with the current time.

```
@Component({
  selector: 'async-observable-pipe',
  template: `<div><code>observable|async</code>:
       Time: {{ time | async }}</div>`
})
export class AsyncObservablePipeComponent {
  time: Observable<string>;
  constructor() {
    this.time = new Observable((observer) => {
      setInterval(() => {
        observer.next(new Date().toString());
      }, 2000);
    });
  }
}
```

Q23) What is the difference between AngularJs and Angular?

AngularJS is an older JavaScript-based MVC framework, while Angular is a modern TypeScript-based platform with a component-based architecture. AngularJS uses controllers and scopes, whereas Angular relies on components and directives. Angular offers better performance, tooling, and mobile support compared to AngularJS. Additionally, Angular has a more active and well-supported ecosystem, while AngularJS has reached its end-of-life**.**

Q24) What is the difference between @RequestParam and @Param?

`@RequestParam` annotation used for accessing the query parameter values from the request. Look at the following request URL:

```
http://localhost:8080/springmvc/hello/101?param1=10&param2=20
```

In the above URL request, the values for `param1` and `param2` can be accessed as below:

public String getDetails(@RequestParam(value="param1", required=true) String param1,

@RequestParam(value="param2", required=false) String param2){

}

In Spring Framework, @**Param** (org.springframework.data.repository.query.Param) is used to bind the method parameter to Query parameter.
Example:

@Query("select e from Employee e where e.deptId = :deptId")
List<Employee> findEmployeeByDeptId(@**Param**("deptId") Long departmentId);
Here, Employee is JPA Entity, and @Param is used to bind the method parameter departmentId to Query parameter deptId.

In your case, you are trying to fetch URL Parameter value. @RequestParam need to be used. @RequestParam is used to bind method parameter to web URL request parameter.

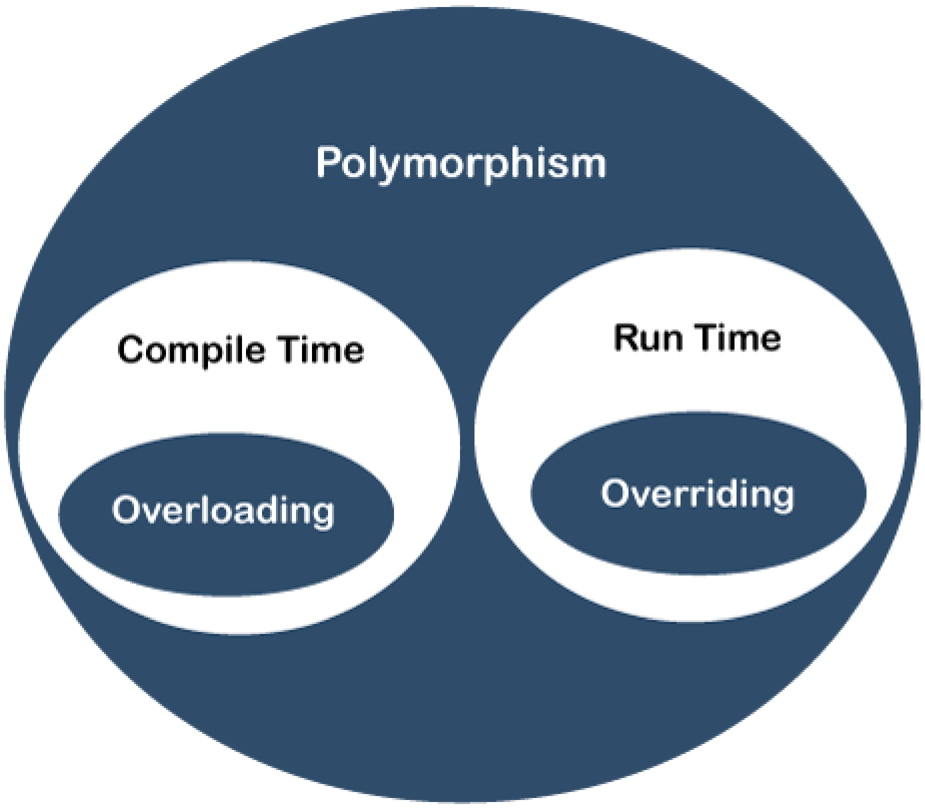
Q25) Explain overloading and overriding with example?/ What is compiletime and runtime polymorphism?

Overloading

Overloading is a concept in OOP when two or more methods in a class with the same name but the method signature is different. It is also known as **compile-time polymorphism**. For example, in the following code snippet, the method **add()** is an overloaded method.

1. public class Sum

```
{
int a, b, c;
public int add();
{
c=a+b;
return c;
}
add(int a, int b);
{
//logic
}
add(int a, int b, int c);
{
//logic
}
add(double a, double b, double c);
{
//logic
}
//statements
}
```



Overriding

If a method with the same method signature is presented in both child and parent class is known as method **overriding**. The methods must have the same number of parameters and the same type of parameter. It overrides the value of the parent class method. It is also known as **runtime polymorphism**. For example, consider the following program.

```
class Dog
{
public void bark()
{
System.out.println("woof ");
}
}
class Hound extends Dog
{
public void sniff()
{
```

```
System.out.println("sniff ");
}
//overrides the method bark() of the Dog class
public void bark()
{
System.out.println("bowl");
}
}
public class OverridingExample
{
public static void main(String args[])
{
Dog dog = new Hound();
//invokes the bark() method of the Hound class
dog.bark();
}
}
```

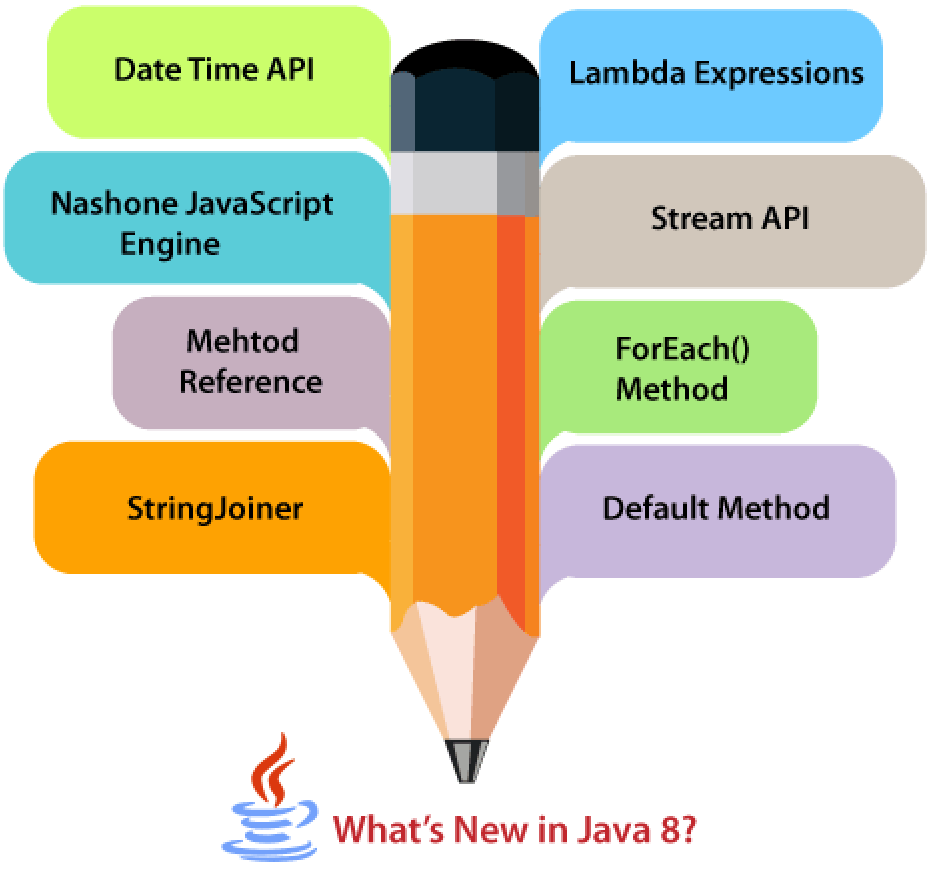
Q26) What is the volatile keyword in java?

Volatile keyword is used in multithreaded programming to achieve the thread safety, as a change in one volatile variable is visible to all other threads so one variable can be used by one thread at a time.

Q27) what is the difference between post and put?

The difference between POST and PUT is that PUT requests are idempotent. That is, calling the same PUT request multiple times will always produce the same result. In contrast, calling a POST request repeatedly have side effects of creating the same resource multiple times.

Q28) What are the new features and characteristics that were introduced in the Java SE 8?



There are several important features and characteristics that are essential for the development using the Java programming language that was implemented in Java SE 8. These features are described as follows:

- **Lambda Expression:** Lambda expressions are the functions and expressions that are shared and pointed to as an object.
- **Functional Interfaces:** Functional Interfaces are the types of interfaces that can contain only a single abstract method. These interfaces are additionally recognized as Single abstract method interfaces.
- **Method References:** Method references are the references that use a function as a parameter to request a method.
- **Default Method:** The default method is a method that provides an implementation of methods within interfaces allowing the Interface development facilities.
- **Stream API:** The Java Stream API is an abstract layer that implements the pipeline processing of the data.
- **Date Time API:** In Java SE 8, the latest, updated Joda-time stimulated APIs are introduced to overcome the disadvantages of the previous versions.

- **Optional**: Optional is a wrapper class included in Java SE 8 that is used to control the imaginary (null) values and assists in additional processing based upon the value.
- **Nashorn:** A JavaScript Engine: Nashorn is a devised variant of JavaScript Engine introduced in Java SE 8 that provides JavaScript performances in Java to succeed Rhino.

Q29) What do you mean by Functional Interfaces in Java?

Functional interfaces are included in Java SE 8 with Lambda expressions and Method references in order to make code more readable, clean, and straightforward. These are the interfaces which ensure that they include precisely only one abstract method. It is used and executed by representing the interface with an annotation called **@FunctionalInterface**. As described earlier, functional interfaces can contain only one abstract method. However, they can include any quantity of default and static methods.

Functional Interface is additionally recognized as Single Abstract Method Interfaces. In short, they are also known as **SAM** interfaces.

Q30) Describe various functional interfaces in the standard library./ Explain some of the functional interfaces which are used in Java?

In Java 8, there are a lot of functional interfaces introduced in the *`java.util.function`* package and the more common ones include but are not limited to:

- *`Function`* – it takes one argument and returns a result
- *`Consumer`* – it takes one argument and returns no result (represents a side effect)
- *`Supplier`* – it takes no argument and returns a result
- *`Predicate`* – it takes one argument and returns a boolean
- *`BiFunction`* – it takes two arguments and returns a result
- BiConsumer - it takes two (reference type) input arguments and returns no result
- *`BinaryOperator`* – it is similar to a BiFunction, taking two arguments and returning a result. The two arguments and the result are all of the same types
- *`UnaryOperator`* – it is similar to a Function, taking a single argument and returning a result of the same type

- *Runnable*: use to execute the instances of a class over another thread with no arguments and no return value.
- *Callable*: use to execute the instances of a class over another thread with no arguments and it either returns a value or throws an exception.
- *Comparator*: use to sort different objects in a user-defined order
- *Comparable*: use to sort objects in the natural sort order

Q31) What are Collectors Class in Java 8?

Collectors is a final class that extends the *Object* class. It provides reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc.

Q32) How will you call a default method of an interface in a class?

Using the *super* keyword along with the interface name.

```
interface Vehicle {
   default void print() {
      System.out.println("I am a vehicle!");
   }
}
class Car implements Vehicle {
   public void print() {
      Vehicle.super.print();
   }
}
```

Q33) How can I start a Spring Boot executable JAR file?

To start a Spring Boot executable JAR file, you can use the `java` command in your terminal or command prompt. Navigate to the directory where your JAR file is located and run the following command:

```

**java -jar your-application.jar**

```

Replace `your-application.jar` with the name of your Spring Boot executable JAR file. This command will execute the JAR file, and your Spring Boot application will start running. Make sure you have Java installed on your system and the `java` command is accessible from your command line interface.

Q34) What is the Spring Boot Actuator and its Features?

Spring Boot Actuator provides production-ready features for monitoring and managing Spring Boot applications. It offers a set of built-in endpoints and metrics that allow you to gather valuable insights into the health, performance, and management of your application.

Here are some key features provided by Spring Boot Actuator:
Health Monitoring: The actuator exposes a `/health` endpoint that provides information about the health status of your application. It can indicate whether your application is up and running, any potential issues, and detailed health checks for different components, such as the database, cache, and message brokers.

Metrics Collection: The actuator collects various metrics about your application's performance and resource utilization. It exposes endpoints like `/metric`s and `/prometheus` to retrieve information about HTTP request counts, memory usage, thread pool statistics, database connection pool usage, and more. These metrics can be integrated with monitoring systems like Prometheus, Graphite, or Micrometer.

Auditing and Tracing: Actuator allows you to track and monitor the activities happening within your application. It provides an `/auditevents` endpoint to view audit events like login attempts, database changes, or any custom events. Additionally, Actuator integrates with distributed tracing systems like Zipkin or Spring Cloud Sleuth to trace requests as they flow through different components.

Environment Information: The actuator exposes an `/info` endpoint that displays general information about your application, such as version numbers, build details, and any custom information you want to include. It is useful for providing diagnostic details about your application in runtime environments.

Configuration Management: Actuator provides an `/configprops` endpoint that lists all the configuration properties used in your application. It helps in understanding the current configuration state and identifying potential issues or inconsistencies.

Remote Management: Actuator allows you to manage and interact with your application remotely. It provides various endpoints, such as `/shutdown` to gracefully shut down the application, /restart to restart the application, and `/actuator` to list all available endpoints. These endpoints can be secured using Spring Security for proper access control.

Enabling the Actuator: The simplest way to enable the features is to add a dependency to the *spring-boot-starter-actuator* dependency.

```
<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

For Gradle, use the following declaration:

```
dependencies {
    compile("org.springframework.boot:spring-boot-starter-actuator")
}
```

Q35) What is the purpose of using @ComponentScan in the class files?

@ComponentScan: The *@ComponentScan* annotation tells Spring where to look for components (such as controllers, services, and repositories) to be managed by the Spring IoC container. It scans the specified packages and registers the annotated classes as beans.

Q36) What is Swagger in Spring Boot?

Swagger in Spring Boot is an open-source project that helps generate documents of REST APIs for RESTful web services via a web browser. It renders the documentation of an API visually using web services.
Open API is the specification, and Swagger is a tool that helps implement the API specification.

Q37) How to create a Spring Boot project using Maven?

Here are two ways to create a Spring Boot project using Maven:

1. Using Spring Initializr:

This is the easiest and most recommended approach for beginners.

- **Visit the Spring Initializr website: https://start.spring.io/**
- **Select "Maven" as the build tool and "Java" as the language.**

- **Choose the Spring Boot starter dependencies you need for your project (e.g., Web, Data JPA).**
- **Click "Generate" to download a ZIP file containing your project structure.**
- **Unzip the downloaded file and open the project in your IDE (e.g., Eclipse, IntelliJ IDEA).**
- **Build the project using Maven (right-click on the project and choose "Maven" -> "Clean" followed by "Maven" -> "Install").**
- **Run the application by right-clicking on the main class (e.g., App.java) and choosing "Run As" -> "Java Application".**

2. Manually creating the project:

This approach gives you more control over the project structure but requires more steps.

- Create a project directory.
- **Create a file named `pom.xml` in the project directory with the following content:**

XML

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>your-group-id</groupId>
    <artifactId>your-artifact-id</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.0.x</version> </parent>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>

</project>
```

Use code with caution.

content_copy

- **Replace `your-group-id` and `your-artifact-id` with your desired values.**
- **Add the Spring Boot starter dependencies you need for your project under the `<dependencies>` tag.**
- **Create the source code for your application, including a main class annotated with `@SpringBootApplication`.**
- **Build and run the application as described in the first approach.**

Remember to adjust the instructions according to your specific needs and chosen Spring Boot version.

---------------------------------Au Small Finance Bank ----------------------------------

Q1) Write a java 8 program to count the male and female employees of the organization?

```
LinkedHashMap<String, Long> maleAndFemaleCount =
employeeList.stream().collect(Collectors.groupingBy(s-
>s.getGender(),LinkedHashMap::new,Collectors.counting()));
System.out.println("maleAndFemaleCount:- "+maleAndFemaleCount);

Map<String, Long> maleAndFemaleCountSecond =
employeeList.stream().collect(Collectors.groupingBy(s-
>s.getGender(),Collectors.counting()));
System.out.println("maleAndFemaleCountSecond:- "+maleAndFemaleCountSecond);
```

Output:-
```
maleAndFemaleCount:- {Female=6, Male=11}
maleAndFemaleCountSecond:- {Male=11, Female=6}
```

Q2) Write a Java 8 Program to find the average salary of male and female employees?

```
Map<String, Double> avgSalaryOfMaleAndFemale =
employeeList.stream().collect(Collectors.groupingBy(s-
>s.getGender(),Collectors.averagingDouble(s->s.getSalary())));
System.out.println("avgSalaryOfMaleAndFemale:- "+avgSalaryOfMaleAndFemale);

Map<String, Double> avgSalaryOfMaleAndFemaleSecond =
employeeList.stream().collect(Collectors.groupingBy(s-
>s.getGender(),LinkedHashMap::new,Collectors.averagingDouble(s-
>s.getSalary())));
System.out.println("avgSalaryOfMaleAndFemaleSecond:-
"+avgSalaryOfMaleAndFemaleSecond);
```

Output:-
```
avgSalaryOfMaleAndFemale:- {Male=21300.090909090908, Female=20850.0}
avgSalaryOfMaleAndFemaleSecond:- {Female=20850.0, Male=21300.090909090908}
```

Q3) what is the difference between default and abstract method?

Certainly, here's a comparison of default and abstract methods in a table format:

Aspect	Default Method	Abstract Method
Purpose	Provides a default implementation within an interface.	Defines a method signature without implementation details.
Introduced in	Java 8	Java (since its inception)

Aspect	Default Method	Abstract Method
Implementation	Method has a default implementation.	Method has no implementation; only the signature is defined.
Override	Implementing classes can optionally override the default implementation.	Subclasses or implementing classes must provide their own implementation.
Interface/Abstract Class	Can only be defined within interfaces.	Can be defined within abstract classes or interfaces.
Contract	Intended for evolving interfaces without breaking existing implementations.	Serves as a contract for subclasses or implementing classes to provide specific functionality.

This table provides a concise comparison of the key differences between default and abstract methods.

Q4) what is memcached in javascript?

it's a caching mechanism commonly used in web development to improve performance and scalability by storing frequently accessed data in memory. JavaScript developers may interact with memcache systems through server-side APIs or libraries.

Q5) Print the below output using javascript.

input:-

let data = [{

name: "sunil",

age: 30,

},

{

name: "anil",

age: 30,

},

```
{
name: "urvashi",
age: 30,
},
{
name: "raju",
age: 32,
},
{
name: "vishnu",
age: 33,
},
{
name: "rubina",
age: 33,
}
]
```

output:-

```
// [
// {age: 30,
// names: [sunil, anil, urvashi]
// },
```

// {age: 32,

// names: [raju]

// },

// {age: 33,

// names: [vishnu, rubina]

// },

//]

Ans:-

```
// Function to group data by age
function groupByAge(arr) {
let groupedData = [];
arr.forEach(item => {
    let found = groupedData.find(element => element.age === item.age);
    if (found) {
        found.names.push(item.name);
    } else {
        groupedData.push({
            age: item.age,
            names: [item.name]
        });
    }
});
return groupedData;
}

let groupedByAge = groupByAge(data);
console.log(groupedByAge);
```

Q6) how will you create the basic structure for spring boot project?

Creating the basic structure for a Spring Boot project involves setting up the project directory, configuration files, and necessary dependencies. Below are the general steps to create the basic structure for a Spring Boot project:

Company Interview Question

1. **Create a New Project**: You can create a new project using your preferred IDE (Integrated Development Environment) or build tool. Popular choices include IntelliJ IDEA, Eclipse, and Spring Initializr.
2. **Use Spring Initializr** (Optional): If you prefer Spring Initializr, you can visit the website https://start.spring.io/ to generate a new Spring Boot project with the desired configurations. You can specify dependencies, project metadata, and download the generated project zip file.
3. **Project Directory Structure**: Once you have the project created, you'll typically see a structure similar to the following:

```
my-spring-boot-project/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/
│   │   │       └── example/
│   │   │           └── myproject/
│   │   │               ├── controller/
│   │   │               │   └── MainController.java
│   │   │               ├── model/
│   │   │               │   └── MyModel.java
│   │   │               └── MySpringBootApplication.java
│   │   └── resources/
│   │       └── application.properties
│   └── test/
│       └── java/
│           └── com/
│               └── example/
│                   └── myproject/
│                       └── MainControllerTest.java
```

24. `└── pom.xml (if using Maven)`

4. **Java Source Code**:
   - `MainController.java`: This is a sample controller class where you define your REST endpoints.
   - `MyModel.java`: This is a sample model class representing data.
   - `MySpringBootApplication.java`: This is the main Spring Boot application class with the `@SpringBootApplication` annotation, which serves as the entry point for the application.
5. **Resources**:
   - `application.properties` or `application.yml`: Configuration file where you can specify properties for your application, such as database configurations, server port, etc.
6. **Test Directory**:
   - `MainControllerTest.java`: This is a sample test class where you write tests for your controller endpoints.
7. **Dependency Management**: If you're using Maven, you'll find a `pom.xml` file where you can specify dependencies for your project. If you're using Gradle, you'll find a `build.gradle` file.
8. **Build and Run**: Finally, you can build and run your Spring Boot application using your IDE or command line tools like Maven or Gradle.

This basic structure provides a starting point for developing Spring Boot applications. Depending on your project requirements, you may add additional packages, classes, and configuration files as needed.

---HCL ---------------------------------------

Q1) what is javascript closure?

Ans:-

A **closure** is the combination of a function bundled together (enclosed) with references to its surrounding state (the **lexical environment**). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

Lexical scoping

Consider the following example code:

```
function init() {
  var name = "Mozilla"; // name is a local variable created by init
  function displayName() {
    // displayName() is the inner function, that forms the closure
    console.log(name); // use variable declared in the parent function
  }
  displayName();
}
init();
```

init() creates a local variable called name and a function called displayName(). The displayName() function is an inner function that is defined inside init() and is available only within the body of the init() function. Note that the displayName() function has no local variables of its own. However, since inner functions have access to the variables of outer functions, displayName() can access the variable name declared in the parent function, init().

Q2) what is the difference between restful service vs microservices?

Ans:-

REST APIs are a communication mechanism, whereas Microservices represent an architectural style. REST APIs are commonly used within Microservices architectures. The choice between the two depends on the specific needs, requirements, and context of your application.

Q3) what is the difference between Promise.race() vs Promise.any()?

Promise. any() fulfills with the first promise to fulfill, even if a promise rejects first. This is in contrast to Promise. race() , which fulfills or rejects with the first promise to settle.

---TCS --------------------------------------

Q1) Remove Duplicate Elements from the list without using java8?

```
List<String> list = Arrays.asList("Apple", "Mango", "Banana", "Apple",
"Mango", "Strawberry");
            List<String> newList = new ArrayList<String>();
            for (String s : list) {
                  if (!newList.contains(s)) {
                        newList.add(s);
                  }
            }
```

Q2) what is the difference between interface & abstract class?

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) **Abstract class can** have abstract and non-abstract **methods.**	**Interface can have** only abstract **methods. Since Java 8, it can have** default and static methods **also.**
2) **Abstract class** doesn't support multiple inheritance**.**	**Interface** supports multiple inheritance**.**
3) **Abstract class** can have final, non-final, static and non-static variables**.**	**Interface has** only static and final variables**.**

4) Abstract class can provide the implementation of interface**.**	**Interface** can't provide the implementation of abstract class**.**
5) The abstract keyword **is used to declare abstract class.**	**The** interface keyword **is used to declare interface.**
6) An abstract class **can extend another Java class and implement multiple Java interfaces.**	**An** interface **can extend another Java interface only.**
7) An abstract class **can be extended using keyword "extends".**	**An** interface **can be implemented using keyword "implements".**
8) A Java abstract class **can have class members like private, protected, etc.**	**Members of a Java interface are public by default.**

9)Example:
**public abstract class Shape{
public abstract void draw();
}**
	Example:
	**public interface Drawable{
	void draw();
	}**

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Example of abstract class and interface in Java

Let's see a simple example where we are using interface and abstract class both.

```
//Creating interface that has 4 methods
interface A{
void a();//bydefault, public and abstract
void b();
void c();
void d();
}

//Creating abstract class that provides the implementation of one method of A interface
abstract class B implements A{
```

```
public void c(){System.out.println("I am C");}
}

//Creating subclass of abstract class, now we need to provide the implementation of rest of the methods
class M extends B{
public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}
public void d(){System.out.println("I am d");}
}

//Creating a test class that calls the methods of A interface
class Test5{
public static void main(String args[]){
A a=new M();
a.a();
a.b();
a.c();
a.d();
}}
```

Q3) what is dependecy injection in spring and types of dependency injection?

DI (Dependency Injection) is a design pattern to provide loose coupling. It removes the dependency from the program.

Let's write a code without following DI.

```
public class Employee{
Address address;
Employee(){
address=new Address();//creating instance
}
}
```

Now, there is dependency between Employee and Address because Employee is forced to use the same address instance.

Company Interview Question

Let's write the IOC or DI code.

```
public class Employee{
Address address;
Employee(Address address){
this.address=address;//not creating instance
}
}
```

Now, there is no dependency between Employee and Address because Employee is not forced to use the same address instance. It can use any address instance.

There are three kinds of Spring Dependency Injection: Setter, Constructor, and Field or Property-Based.

1. Field or Property-Based.

```
public                    class                    ProductService                    {

    @Autowired
    private                ProductRepository                productRepository;
}
```

2. Setter based.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RestController;
@ResstController
public class SchoolMasterDetails{
 private TeacherMasterDetails teachmastdetails;

 @Autowired
 public void setTeachMasterDetails(TeacherMasterDetails teachmastdetails){
    this.teachmastdetails = teachmastdetails;
 }

 @Override
 public String toString(){
    return "SchoolMasterDetails [teachmastdetails="+teachmastdetails+"]";
 }
}
```

3. Constructor based

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RestController;
@ResstController
```

```
public class SchoolMasterDetails{
  private TeacherMasterDetails teachmastdetails;

  @Autowired
  public SchoolMasterDetails(TeacherMasterDetails teachmastdetails){
     this.teachmastdetails = teachmastdetails;
  }

  @Override
  public String toString(){
     return "SchoolMasterDetails [teachmastdetails="+teachmastdetails+"]";
  }
}
```

Q4) what is @Component & @Component scan?

@Component

The @Component annotation in Spring is used to mark a class as a Spring-managed component, allowing it to be automatically discovered and instantiated by the Spring container.

@ComponentScan

This annotation enables component-scanning so that the web controller classes and other components you create will be automatically discovered and registered as beans in Spring's Application Context. All the @Component, @Service, @Repository, and @Controller annotated classes are automatically discovered by this annotation.

Q5) How to fetch Second highest salary record through my sql?

Select max(salary) From Employee where salary not in (Select max(salary) From Employee);

Select max(salary) From Employee where salary < (Select max(salary) From Employee);

Company Interview Question

--**Nagarro**---------------------------------------

Q1) How to fetch 3rd highest salary of each department?

```
WITH employee_Value AS(

SELECT

    department,

    salary,

    RANK() OVER (PARTITION BY

    department

    order by salary desc) salary_rank

    from employee)

    select * from employee_Value where salary_rank= 3;
```

Q2) How to find 3rd highest salary of each department using java 8?

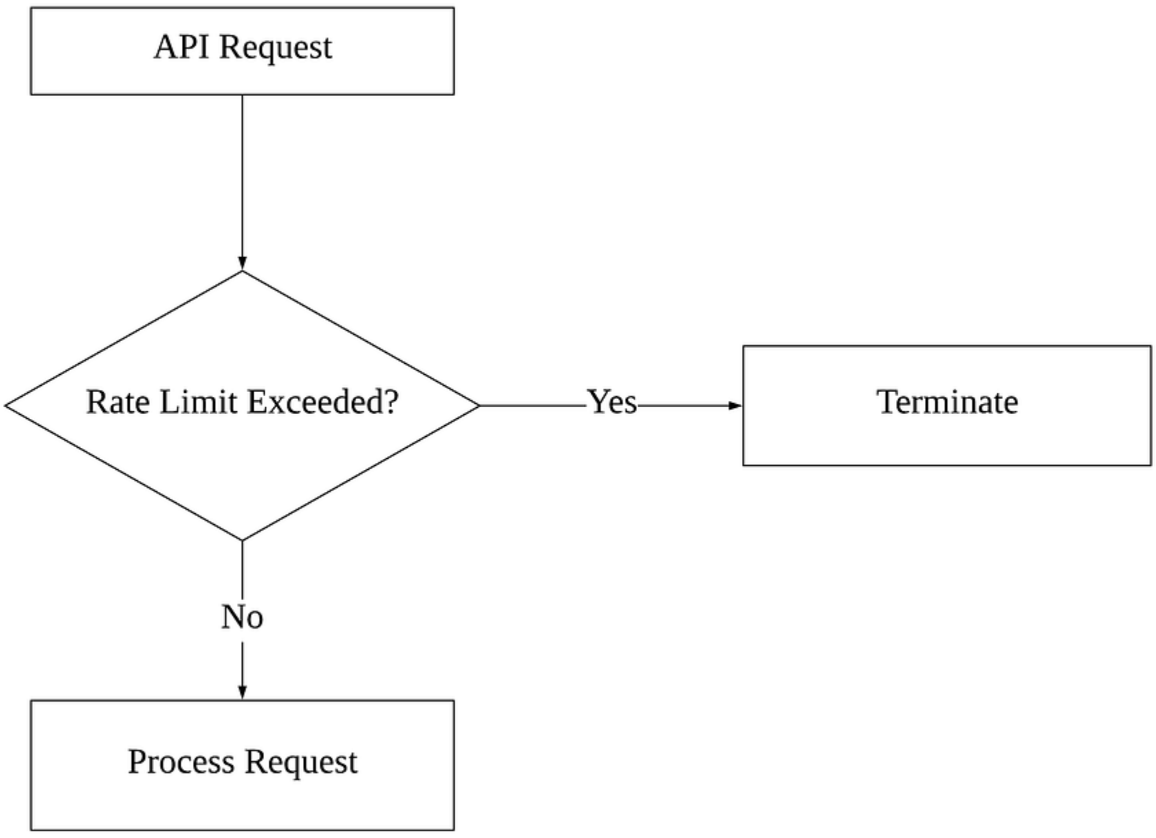
```
Map<String, Integer> thirdHighestSalaries = employees.stream()
                .collect(Collectors.groupingBy(Employee::getDepartment,
                        Collectors.mapping(Employee::getSalary,
Collectors.toList())))
                .entrySet().stream()
                .collect(Collectors.toMap(
                        Map.Entry::getKey,
                        entry -> entry.getValue().stream()
                                .sorted(Comparator.reverseOrder())
                                .skip(2)
                                .findFirst()
                                .orElse(null)
                ));
```

Q3) how can you prevent incoming request in spring boot microservice? How to implement throttling in java spring boot microservice?

Company Interview Question

What is Throttling?

Throttling is the process of limiting the rate that an API is being used in a server. It limits the number of service requests which can be executed in a unit time (for a second, minute..).



Simple flow of throttling

We can use resilience4j circuit breaker or rate limiter design pattern.

Q4) how can you implement authentication and authorization in spring security?

Implementing authentication and authorization in Spring Security involves several steps. Here's a high-level overview:

1. **Add Spring Security Dependency**: First, you need to include the Spring Security dependency in your project. You can do this by adding the appropriate Maven or Gradle dependency in your `pom.xml` or `build.gradle` file.

2. **Configure Security**: Configure Spring Security by creating a configuration class that extends `WebSecurityConfigurerAdapter`. This class allows you to override methods to configure authentication, authorization, and other security settings. You can specify which endpoints require authentication, define custom login/logout pages, etc.

3. **Authentication Configuration**: Configure how users are authenticated. You can use various authentication mechanisms such as in-memory authentication, JDBC authentication (connecting to a database), LDAP authentication, OAuth2, etc. Configure authentication providers, user details services, password encoders, etc.

4. **Authorization Configuration**: Configure access control rules to restrict access to certain parts of your application. You can define which roles or authorities are required to access specific URLs or methods. Spring Security provides annotations (`@PreAuthorize`, `@Secured`, `@RolesAllowed`) for method-level security, as well as configuration-based security (`http.authorizeRequests()`) for URL-level security.

5. **User Management**: Depending on your application requirements, you may need to implement user management features such as registration, password reset, account locking, etc. Spring Security provides hooks for integrating with your user management system or database.

6. **Customization**: Spring Security is highly customizable. You can customize various aspects such as login/logout behavior, session management, CSRF protection, remember-me functionality, etc., based on your application's requirements.

Here's a simplified example of how you might configure authentication and authorization in a Spring Security application:

@Configuration

@EnableWebSecurity

```
public class SecurityConfig extends WebSecurityConfigurerAdapter {



    @Autowired

    public void configureGlobal(AuthenticationManagerBuilder auth) throws
Exception {

        auth

            .inMemoryAuthentication()

            .withUser("user").password("{noop}password").roles("USER")

            .and()

            .withUser("admin").password("{noop}password").roles("ADMIN");

    }



    @Override

    protected void configure(HttpSecurity http) throws Exception {

        http

            .authorizeRequests()

                .antMatchers("/admin/**").hasRole("ADMIN")

                .antMatchers("/user/**").hasRole("USER")

                .anyRequest().authenticated()

                .and()

            .formLogin()

                .loginPage("/login")
                .permitAll()

                .and()

            .logout()

                .logoutUrl("/logout")

                .logoutSuccessUrl("/login?logout")

                .permitAll();

    }

}
```

In this example:

- We configure in-memory authentication with two users: "user" with role "USER" and "admin" with role "ADMIN".
- We define authorization rules using `authorizeRequests()`. URLs starting with "/admin" require the "ADMIN" role, URLs starting with "/user" require the "USER" role, and any other URLs require authentication.
- We configure a custom login page ("/login") and a logout URL ("/logout").
- Passwords are stored in plain text for simplicity (`{noop}`). In a real-world scenario, you should use password hashing.

This is just a basic example. In a real-world application, you'd likely use a more secure authentication mechanism, such as database-backed authentication, LDAP, OAuth2, etc., and you'd also handle user management features and customize security settings according to your application's requirements.

Q5) how to use @Transactional annotation for in case of particular exception?

@Transactional(rollbackFor = Exception.class)

Q6) what is saga pattern and how to implement saga pattern?

➢The Saga pattern provides a way to manage transactions that involve multiple microservices. It is used to ensure that a series of transactions across multiple services are completed successfully, and if not, to roll back or undo all changes that have been made up to that point.

➢The Saga pattern consists of a sequence of local transactions, each of which updates the state of a single service, and a corresponding set of compensating transactions that are used to undo the effects of the original transactions in case of a failure.

➢Suppose you have two microservices, one responsible for processing orders and another responsible for shipping orders.

➢When a new order is placed, the order processing service is responsible for validating the order and ensuring that the items are in stock, while the shipping service is responsible for packaging the order and sending it to the customer.

➢If the order processing service determines that the order is valid and all items are in stock, it sends a message to the shipping service to initiate the shipping process. At this point, the Saga pattern comes into play.

➢The shipping service will create a new transaction to package and ship the order, and if the transaction is successful, it will mark the order as shipped.

➢If, on the other hand, the transaction fails (perhaps due to a problem with the shipping provider), the shipping service will initiate a compensating transaction to undo the effects of the original transaction, such as canceling the shipment and restocking the items.

➢Meanwhile, the order processing service is also using the Saga pattern to manage its own transaction. If the shipping service reports that the order has been shipped successfully, the order processing service will mark the order as completed.

➢If the shipping service reports a failure, the order processing service will initiate a compensating transaction to cancel the order and return any funds that were paid.

➢Overall, the Saga pattern provides a way to manage complex transactions across multiple microservices in a way that ensures consistency and reliability. If you have to just learn one patter, you better learn SAGA patterns as its immensely helpful in Microservice applications.

There are two ways of coordination sagas:

- Choreography - each local transaction publishes domain events that trigger local transactions in other services
- Orchestration - an orchestrator (object) tells the participants what local transactions to execute

Q7) how to write a junit5, mockito test case for Controller,Service and dao?

Certainly! JUnit 5 provides an improved testing experience over JUnit 4 with better support for modern Java features. Mockito can be easily integrated with JUnit 5 to write effective tests. Let's see how to write JUnit 5 tests with Mockito for Controller, Service, and DAO layers:

Controller Layer

For the Controller layer, you will mock the Service layer and test the behavior of your controller methods.

```
import static org.junit.jupiter.api.Assertions.*;

import static org.mockito.Mockito.*;



@ExtendWith(MockitoExtension.class)

public class YourControllerTest {



    @Mock

    private YourService yourService;



    @InjectMocks

    private YourController yourController;



    @Test

    public void testControllerMethod() {
        // Prepare test data or mocks
        when(yourService.someMethod()).thenReturn("expectedResult");

        // Call the controller method
        String result = yourController.yourControllerMethod();

        // Verify the result or behavior
        assertEquals("expectedResult", result);
    }
}
```

Service Layer

For the Service layer, you will mock the DAO layer and test the business logic of your service methods.

```
@ExtendWith(MockitoExtension.class)
public class YourServiceTest {

    @Mock
    private YourDAO yourDAO;

    @InjectMocks
    private YourService yourService;

    @Test
    public void testServiceMethod() {
        // Prepare test data or mocks

        when(yourDAO.findById(1)).thenReturn(new YourEntity());


        // Call the service method

        YourEntity result = yourService.findById(1);


        // Verify the result or behavior

        assertNotNull(result);

        // Add more assertions as needed

    }

}
```

DAO Layer

For the DAO layer, you want to test the database interactions. In this case, we don't mock anything because we want to test the actual data access.

```
@ExtendWith(SpringExtension.class)

@SpringBootTest

@Transactional

public class YourDAOTest {


    @Autowired

    private YourDAO yourDAO;


    @Test

    public void testSave() {
        // Prepare test data

        YourEntity entity = new YourEntity();

        entity.setName("Test");



        // Save the entity

        yourDAO.save(entity);



        // Retrieve the entity

        YourEntity savedEntity = yourDAO.findById(entity.getId());



        // Verify the result or behavior

        assertNotNull(savedEntity);

        assertEquals("Test", savedEntity.getName());

        // Add more assertions as needed

    }

}
```

Ensure you have the necessary annotations (`@ExtendWith(MockitoExtension.class)`, `@ExtendWith(SpringExtension.class)`, `@SpringBootTest`, etc.) and dependencies set up correctly in your test classes. Mockito is used for mocking dependencies, and Spring's testing support is utilized for testing Spring components. Adjust the test cases according to your specific application and requirements.

Q8) what is aws cloud?

Amazon Web Services (AWS) Cloud is a comprehensive cloud computing platform provided by Amazon.com. It offers a wide range of services that enable individuals, businesses, and organizations to build and deploy various types of applications and services in the cloud.

Here are some key aspects of AWS Cloud:

1. **Infrastructure Services**: AWS provides foundational infrastructure services such as compute (e.g., Amazon EC2 - Elastic Compute Cloud), storage (e.g., Amazon S3 - Simple Storage Service), networking (e.g., Amazon VPC - Virtual Private Cloud), and databases (e.g., Amazon RDS - Relational Database Service).

2. **Managed Services**: AWS offers managed services that handle administrative tasks, such as patching, provisioning, and scaling, allowing users to focus more on building their applications. Examples include Amazon DynamoDB (NoSQL database), Amazon Aurora (MySQL and PostgreSQL-compatible relational database), and Amazon Elastic Beanstalk (platform-as-a-service).

3. **Developer Tools**: AWS provides a variety of tools and SDKs to help developers build, test, and deploy applications efficiently. This includes AWS SDKs for various programming languages, AWS CLI (Command Line Interface), AWS CloudFormation (infrastructure as code), and AWS CodePipeline (continuous integration and continuous delivery).

4. **AI/ML and Analytics**: AWS offers a suite of services for artificial intelligence, machine learning, and analytics, including Amazon SageMaker (machine learning platform), Amazon Redshift (data warehouse), Amazon EMR (managed Hadoop framework), and Amazon Kinesis (real-time data streaming).

5. **Security and Compliance**: AWS prioritizes security and compliance, providing a range of tools and features to help customers secure their applications and data. This includes AWS Identity and Access Management (IAM), AWS Key Management Service (KMS), AWS WAF (Web Application Firewall), and AWS Certificate Manager.

6. **Global Infrastructure**: AWS operates a global network of data centers (regions) and availability zones, allowing customers to deploy applications close to their end-users for low-latency performance and high availability. As of my last update, AWS had 25 regions worldwide, with multiple availability zones within each region.

7. **Scalability and Elasticity**: AWS enables users to scale their resources up or down dynamically based on demand. This elasticity allows businesses to handle fluctuations in traffic and workload without over-provisioning or under-provisioning resources.

8. **Cost Management**: AWS provides various pricing models and cost management tools to help users optimize their cloud spending. This includes services like AWS Cost Explorer, AWS Budgets, and AWS Trusted Advisor, as well as pricing options like pay-as-you-go, reserved instances, and spot instances.

Overall, AWS Cloud offers a flexible, scalable, and cost-effective platform for building and running virtually any type of application or workload in the cloud. It has become one of the leading cloud computing platforms, serving millions of customers worldwide, ranging from startups and small businesses to large enterprises and government agencies.

Q9) what is azure?

Microsoft Azure is a cloud computing platform and services offered by Microsoft. Similar to AWS, Azure provides a wide range of cloud services to help individuals, businesses, and organizations build, deploy, and manage applications and services in the cloud.

Here are some key aspects of Azure:

1. **Infrastructure Services**: Azure offers a comprehensive set of infrastructure services, including virtual machines (Azure VMs), storage (Azure Blob Storage, Azure Files), networking (Azure Virtual Network), and databases (Azure SQL Database, Azure Cosmos DB).

2. **Platform Services**: Azure provides platform services that abstract away the underlying infrastructure and allow developers to focus more on building and deploying applications. Examples include Azure App Service (platform-as-a-service), Azure Functions (serverless compute), Azure Kubernetes Service (managed Kubernetes), and Azure Logic Apps (workflow automation).

3. **Developer Tools**: Azure offers a range of developer tools and SDKs to help developers build, debug, and deploy applications on the Azure platform. This includes Azure DevOps (formerly known as Visual Studio Team Services), Azure CLI (Command Line Interface), Azure PowerShell, and Azure SDKs for various programming languages.

4. **AI/ML and Analytics**: Azure provides a suite of services for artificial intelligence, machine learning, and analytics, including Azure Machine Learning, Azure Cognitive Services, Azure Databricks (Apache Spark-based analytics platform), and Azure Synapse Analytics (formerly Azure SQL Data Warehouse).

5. **Security and Compliance**: Azure prioritizes security and compliance, offering a range of tools and features to help customers secure their applications and data. This includes Azure Active Directory (identity and access management), Azure Security

Center, Azure Key Vault (secure key management), and Azure Sentinel (cloud-native security information and event management).

6. **Global Infrastructure**: Azure operates a global network of data centers (regions) and availability zones, allowing customers to deploy applications close to their end-users for low-latency performance and high availability. As of my last update, Azure had over 60 regions worldwide, with multiple availability zones within each region.

7. **Scalability and Elasticity**: Azure enables users to scale their resources up or down dynamically based on demand. This elasticity allows businesses to handle fluctuations in traffic and workload without over-provisioning or under-provisioning resources.

8. **Cost Management**: Azure provides various pricing models and cost management tools to help users optimize their cloud spending. This includes services like Azure Cost Management, Azure Budgets, and Azure Advisor, as well as pricing options like pay-as-you-go, reserved instances, and spot instances.

Overall, Azure is a powerful and flexible cloud platform that offers a wide range of services to meet the needs of developers, IT professionals, and businesses of all sizes. It competes with other leading cloud providers like AWS and Google Cloud Platform in the rapidly growing cloud computing market.

Q10) do you have any idea about jenkins,docker,kubernatives?

Absolutely, I can explain Jenkins, Docker, and Kubernetes, and how they work together in relation to cloud-native development and deployment:

1. Docker:

- Docker is a platform for developing, deploying, and running applications using containers.
- Containers are lightweight, self-contained units of software that package code and all its dependencies together. This ensures applications run consistently across different environments.
- Benefits of Docker:
  - **Portability:** Containers run seamlessly across different operating systems.
  - **Isolation:** Applications in containers are isolated from each other and the host system.
  - **Reproducibility:** Docker guarantees a consistent environment for your application.
- You can create Docker images, which are blueprints for containers, and store them in a registry for easy sharing and deployment.

2. Jenkins:

- Jenkins is an open-source automation server widely used for continuous integration and continuous delivery (CI/CD) in software development.

- It automates the building, testing, and deployment of applications, allowing for faster development cycles and more reliable releases.
- Here's how Jenkins fits in the CI/CD pipeline:
  - Developers push code changes to a version control system (VCS) like Git.
  - Jenkins triggers a build job upon code commit.
  - The build job can leverage Docker to build and test the application within a containerized environment.
  - Upon successful build and testing, Jenkins can automate deployment to various environments (staging, production) using tools like Kubernetes.

3. Kubernetes:

- Kubernetes (often referred to as K8s) is an open-source container orchestration platform.
- It automates the deployment, scaling, and management of containerized applications.
- While Docker focuses on the individual container, Kubernetes manages the entire lifecycle of containerized applications across a cluster of machines.
- Benefits of Kubernetes:
  - **Scalability:** Easily scale applications up or down based on traffic or resource needs.
  - **High Availability:** Kubernetes ensures application availability by automatically restarting failed containers and scheduling them on healthy nodes.
  - **Load Balancing:** Distribute traffic across multiple container instances for better performance.

Working Together:

- Docker provides a standardized way to package and ship applications.
- Jenkins automates the CI/CD pipeline, potentially using Docker containers for building and testing.
- Kubernetes manages the deployment and lifecycle of containerized applications in a production environment.

By combining these tools, you can achieve a streamlined and automated development workflow for building, testing, and deploying cloud-native applications.

Q11) do you know how to write dockerfile?

A Dockerfile is a text document that contains instructions for building a Docker image. This image serves as a blueprint for creating a containerized application. Here's a breakdown of the key elements:

Structure:

A Dockerfile consists of lines containing instructions, each starting with a specific keyword followed by arguments. Here are some common instructions:

- **FROM:** Specifies the base image from which your image will inherit. You can use official images from Docker Hub (e.g., ubuntu:22.04) or custom base images.
- **RUN:** Executes commands within the container during the image build process. This is typically used to install dependencies, copy files, or configure the environment.
- **COPY:** Copies files or directories from the host machine (where you're building the image) into the container's filesystem.
- **WORKDIR:** Sets the working directory for subsequent instructions within the container.
- **CMD:** Defines the command to be executed when a container starts up. This specifies the default application to run.
- **ENTRYPOINT:** Similar to CMD, but allows specifying arguments to be passed when running the container. One or both of CMD or ENTRYPOINT is typically used.

Below is an example of a Dockerfile for a Java application using a JDK base image:

```
# Use an official OpenJDK runtime as the base image
FROM openjdk:11-jre-slim


# Set the working directory in the container
WORKDIR /app


# Copy the JAR file into the container at /app
COPY target/your-application.jar /app/your-application.jar


# Expose the port the application runs on
EXPOSE 8080


# Run the application when the container starts
CMD ["java", "-jar", "your-application.jar"]
```

In this Dockerfile:

- `FROM`: Specifies the base image to use. Here, we're using the official OpenJDK 11 JRE slim image.
- `WORKDIR`: Sets the working directory inside the container. All subsequent instructions will be executed from this directory.
- `COPY`: Copies the JAR file of your Java application from the host machine into the container at `/app`.
- `EXPOSE`: Exposes port 8080 on the container. This doesn't actually publish the port, but it serves as documentation for which ports are intended to be published.

- `CMD`: Specifies the command to run when the container starts. In this case, we're running `java -jar your-application.jar` to start our Java application.

Replace `your-application.jar` with the name of your actual JAR file. Also, make sure to place the JAR file in the correct location relative to the Dockerfile.

To build an image from this Dockerfile, navigate to the directory containing the Dockerfile and run:

docker build -t your-image-name .

Replace `your-image-name` with the desired name for your Docker image.

Once the image is built, you can run a container from it using the `docker run` command:

docker run -p 8080:8080 your-image-name

This command runs a container from the image, mapping port 8080 on the host to port 8080 on the container. Adjust the port numbers as needed based on your application's configuration.

Q12) how to check docker logs?

To check Docker container logs, you can use the `docker logs` command followed by the container name or ID. Here's how:

1. First, identify the name or ID of the container whose logs you want to check. You can use the `docker ps` command to list all running containers:

```
docker ps
```

This will display a list of running containers along with their names and IDs.

2. Once you have the container name or ID, use the `docker logs` command followed by the container name or ID to view the logs:

```

docker logs container_name_or_id

```

Replace `container_name_or_id` with the actual name or ID of the container you want to check.

For example, if the container name is `my-container`, you would run:

```

docker logs my-container

```

This command will display the logs of the specified container, showing the output produced by the application running inside the container. You can use this information for troubleshooting, debugging, or monitoring purposes.

Q13) what is the difference between future and completablefuture?

The terms "Future" and "CompletableFuture" both relate to asynchronous programming in Java, particularly when dealing with concurrent tasks and executing operations in parallel. However, they have some differences in their usage and capabilities:

Future:

1. **Java Standard Library**: `Future` is part of the Java standard library, introduced in Java 5 as part of the `java.util.concurrent` package.
2. **Asynchronous Computation**: `Future` represents the result of an asynchronous computation. It allows you to submit a task for execution and retrieve the result at some point in the future, possibly blocking until the result becomes available.
3. **Blocking Operations**: The `get()` method of `Future` is blocking, meaning it waits until the computation is complete and the result is available. If the result is not yet available, the `get()` method will block the calling thread until it is.

4. **Limited Callback Support**: `Future` does not provide built-in support for callbacks or chaining asynchronous operations.

CompletableFuture:

1. **Introduced in Java 8**: `CompletableFuture` is introduced in Java 8 as part of the `java.util.concurrent` package. It's an extension of the `Future` interface and provides more features and flexibility for asynchronous programming.

2. **Completion Callbacks**: `CompletableFuture` supports completion callbacks, allowing you to specify actions to be performed when the future completes, either successfully or exceptionally. This allows for more flexible and non-blocking programming patterns.

3. **Chaining Operations**: `CompletableFuture` supports chaining of asynchronous operations using methods like `thenApply()`, `thenAccept()`, `thenCombine()`, etc. This enables composing complex asynchronous workflows in a fluent and concise manner.

4. **Combining Multiple Futures**: `CompletableFuture` provides methods to combine multiple `CompletableFuture` instances, such as `thenCombine()`, `thenCompose()`, and `allOf()`, allowing for parallel execution and composition of asynchronous tasks.

5. **Non-blocking Operations**: `CompletableFuture` offers non-blocking methods like `join()` and `getNow()` in addition to blocking `get()` method, providing more options for handling future results.

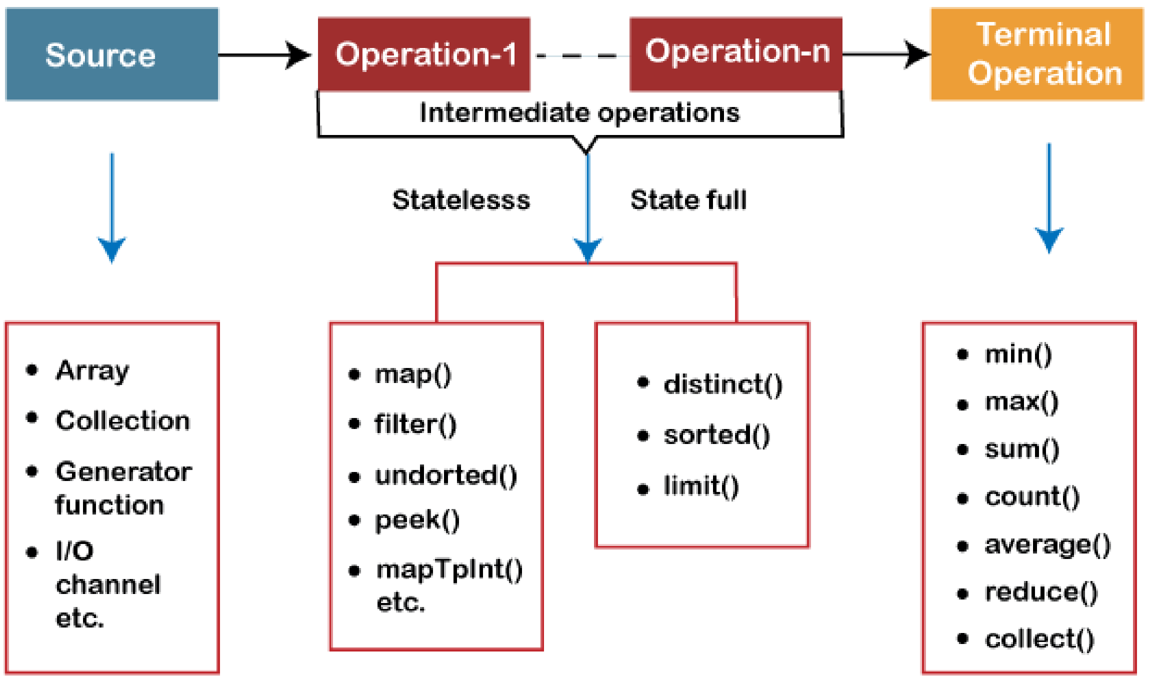
In summary, while both `Future` and `CompletableFuture` represent asynchronous computations, `CompletableFuture` offers more features and flexibility for composing, combining, and handling asynchronous tasks in Java. It's often the preferred choice for writing asynchronous and non-blocking code in modern Java applications.

Q14) What is Intermediate and terminal operations in java 8?

Intermediate operations: Intermediate operations are the operations that return a stream so that the user can chain various intermediate operations without using semicolons, as we do in other programming languages like Scala.

Terminal operations: The terminal operations are the operations that are mainly void and null, and if not null, these operations return a non-stream as a result.

Company Interview Question



---TCS---------------------------------------

Q1) Sort the list in this order department, id, name?

```
employeeList.sort(Comparator.comparing(Employee::getDepartment).thenCompari
ng(Employee::getId).thenComparing(Employee::getName));
```

Q2) How to get the unique list of employee?

List<Student> newStudentList = list.stream().distinct().collect(Collectors.toList());

Q3) What is the difference between string builder and string buffer?

No.	StringBuffer	StringBuilder
1)	StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
2)	StringBuffer is *less efficient* than StringBuilder.	StringBuilder is *more efficient* than StringBuffer.
3)	StringBuffer was introduced in Java 1.0	StringBuilder was introduced in Java 1.5

Q4) what is functional interface?

Functional Interfaces are the types of interfaces that can contain only a single abstract method. These interfaces are additionally recognized as Single abstract method interfaces. *Runnable*, *ActionListener*, ***and*** *Comparable* **are some of the examples of functional interfaces.**

Q5) what is the singleton class?

Singleton class is the class which can not be instantiated more than once. To make a class singleton, we either make its constructor private or use the static getInstance method.

Q6) what is the difference between iterator and enumerator?

No.	Iterator	Enumeration
1)	The Iterator can traverse legacy and non-legacy elements.	Enumeration can traverse only legacy elements.
2)	The Iterator is fail-fast.	Enumeration is not fail-fast.
3)	The Iterator is slower than Enumeration.	Enumeration is faster than Iterator.
4)	The Iterator can perform remove operation while traversing the collection.	The Enumeration can perform only traverse operation on the collection.

Q7) what is predicate and supplier?

Supplier: Supplier is a type of functional interface in Java that does not accept any argument and still returns the desired result.

Predicate: The type of functional interface in Java that accepts one argument and returns a boolean value is known as Predicate functional interface.

Q8) How to get list of salary from employee list whose salary is greater than 12700?

```
List<Double> collect = employeeList.stream().filter(p ->
p.getSalary()>12700).map(e-> e.getSalary()).collect(Collectors.toList());
```

Q9) what is @EnableEurekaServer in spring boot?

@EnableEurekaServer — used for implementing microservices with spring. This class level Annotation makes your service Eureka discovery service.

Q10) what is @EnableDiscoverClient in spring boot?

@EnableDiscoverClient — used for registering as a discoverable service for others.

Q11) What is RestTemplate?

RestTemplate is a class provided by Spring Framework that simplifies the process of making HTTP requests and consuming RESTful web services in Java applications.

Q12) What is the difference between stub and mock?

Stub

Stubs are the objects that hold predefined data and uses it to give responses during tests. In other words, a stub is an object that resembles a real object with the minimum number of methods needed for a test.

Mock

Mocks are the objects that store method calls. It referred to as the dynamic wrappers for dependencies used in the tests. It is used to record and verify the interaction between the Java classes.

Parameters	Stub	Mock

Data Source	The data source of stubs is hardcoded. It is usually tightly coupled to the test suite.	Data on mocks is set up by the tests.
Created by	Stubs are usually handwritten, and some are generated by tools.	Mocks are usually created by using the third-party library such as Mockito, JMock, and WireMock.
Usage	Stubs are mainly used for simple test suites.	Mocks are mainly used for large test suites.
Graphics User Interface (GUI)	Stubs do not have a GUI.	Mocks have a GUI.

Q13) how to enable logger in spring boot?

Enabling logging in a Spring Boot application is typically straightforward due to Spring Boot's opinionated defaults and auto-configuration capabilities. Here's a step-by-step guide on how to enable and configure logging:

1. Dependencies

Ensure you have the necessary dependencies in your `pom.xml` (if using Maven) or `build.gradle` (if using Gradle) for logging. Spring Boot generally uses SLF4J (Simple Logging Facade for Java) with Logback by default.

For Maven, add the following dependencies in your `pom.xml`:

```
xml
Copy code
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
</dependency>
```

2. Configuration

Spring Boot will automatically configure logging based on dependencies and properties specified in your `application.properties` or `application.yml` file.

2.1 application.properties

In your `src/main/resources/application.properties`, you can configure logging levels and appenders (output destinations) like this:

```
properties
Copy code
# Set logging level for root logger and other specific loggers
logging.level.root=INFO
logging.level.org.springframework=DEBUG

# Log file location and pattern
logging.file=myapp.log
logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} - %msg%n
logging.pattern.file=%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36}
- %msg%n
```

2.2 application.yml

Alternatively, you can use `application.yml` for configuration:

```
logging:
  level:
    root: INFO
    org.springframework: DEBUG
  file: myapp.log
  pattern:
    console: "%d{yyyy-MM-dd HH:mm:ss} - %msg%n"
    file: "%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n"
```

3. Logging Levels

- DEBUG: **Detailed information, typically useful only for debugging.**
- INFO: **Informational messages about the application's execution.**
- WARN: **Indicates potentially harmful situations.**
- ERROR: **Indicates error events that might still allow the application to continue running.**
- FATAL: **Severe errors that may lead the application to abort.**
- TRACE: **Very detailed information, typically used for more verbose debugging.**

4. Using Loggers in Your Code

In your Java classes, use SLF4J logger to log messages. For example:

```
java
Copy code
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class MyClass {
    private static final Logger logger =
LoggerFactory.getLogger(MyClass.class);

    public void someMethod() {
        logger.debug("This is a debug message");
        logger.info("This is an info message");
        logger.warn("This is a warning message");
        logger.error("This is an error message");
    }
}
```

5. Customizing Logback (Optional)

If you need more advanced logging configuration (e.g., rotating log files, log rolling policies), you can create a `logback.xml` or `logback-spring.xml` in `src/main/resources` to customize Logback's behavior.

6. External Logging Configuration

For more complex setups or when integrating with external logging systems like Logstash, Elasticsearch, or Splunk, refer to Spring Boot's documentation for additional configuration options.

By following these steps, you should be able to enable and configure logging effectively in your Spring Boot application, ensuring you capture the necessary information for debugging and monitoring purposes.

Q14) how to write custom exception in spring boot?

In Spring Boot, writing custom exceptions involves creating your own exception classes and possibly defining custom handlers to manage these exceptions gracefully within your application. Here's a step-by-step guide on how to write and use custom exceptions in a Spring Boot application:

1. Create Custom Exception Class

First, create a custom exception class by extending either `RuntimeException` (if you want an unchecked exception) or `Exception` (if you want a checked exception).

```
java
Copy code
public class CustomException extends RuntimeException {

    public CustomException() {
        super();
    }

    public CustomException(String message) {
        super(message);
    }

    public CustomException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

2. Throwing Custom Exceptions

You can throw your custom exception wherever appropriate in your application logic:

```
java
```

```
Copy code
public class MyService {

    public void someMethod() {
        // Example of throwing custom exception
        if (someCondition) {
            throw new CustomException("Custom exception occurred");
        }
    }
}
```

3. Handling Custom Exceptions

To handle custom exceptions in your Spring Boot application, you can use `@ExceptionHandler` to define how specific exceptions should be handled globally or within specific controllers.

Global Exception Handling

Create a global exception handler using `@ControllerAdvice` and `@ExceptionHandler` annotations:

```
java
Copy code
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<String> handleCustomException(CustomException ex)
{
        // Create a custom error response
        String errorMessage = "Custom exception handled: " +
ex.getMessage();
        return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(errorMessage);
    }
}
```

Controller-Specific Exception Handling

Alternatively, handle exceptions within a specific controller:

```
java
Copy code
@RestController
@RequestMapping("/api")
public class MyController {

    @GetMapping("/test")
    public ResponseEntity<String> testMethod() {
        try {
            // Business logic that may throw CustomException
            // For example:
            throw new CustomException("Custom exception occurred in
testMethod");
        } catch (CustomException ex) {
            // Handle the exception locally if needed
            return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Handled
locally: " + ex.getMessage());
        }
    }
}
```

4. Custom Error Responses

You can customize the response returned when handling exceptions by creating a custom error message or encapsulating the error details in a custom object. This ensures that your API responses are informative and useful for clients consuming your API.

5. Testing Custom Exceptions

Make sure to test your custom exceptions in unit tests to ensure they are thrown and handled correctly.

Summary

Creating custom exceptions in Spring Boot involves defining your exception class, throwing it where appropriate in your application logic, and optionally defining global or local handlers to manage these exceptions effectively. This approach helps in maintaining clean and organized error handling within your application.

Q15) you have list of string with integer value. you need to sort. how can you do that?

```
List<String> list = Arrays.asList("2","1","5","7","13","6");
List<Long> collect =
list.stream().map(Long::parseLong).sorted().collect(Collectors.toList());
System.out.println(collect);
```

```
==========================================================================

List<Person> person = new ArrayList<>();
            person.add(new Person("23", "Vishal", "56000"));
            person.add(new Person("21", "Vishal", "25000"));
            person.add(new Person("19", "Vishal", "40000"));
            person.add(new Person("25", "Vishal", "66000"));

            // Sort the list based on id converted to long
        List<Person> sortedPersons = person.stream()
                .sorted(Comparator.comparingLong(p ->
Long.parseLong(p.getId())))
                .collect(Collectors.toList());
```

--**Accenture**-------------------------------------

Q1) Is there any way we can access private methods in java?

Yes, with the help of reflection API.

Q2) What is AOP?

AOP is an acronym for Aspect Oriented Programming. It is a methodology that divides the program logic into pieces or parts or concerns.

It increases the modularity and the key unit is Aspect.

More details...

Q3) What is Reflection API in java?

Java Reflection is a *process of examining or modifying the run time behavior of a class at run time*.

The **java.lang.Class** class provides many methods that can be used to get metadata, examine and change the run time behavior of a class.

Q4) How to write custom annotation in java?

Java Custom annotations or Java User-defined annotations are easy to create and use. The *@interface* element is used to declare an annotation. For example:

1. **@interface** MyAnnotation{}

Define the Annotation Type: Use the `@interface` keyword to declare a new annotation type. Annotations can include elements that can have default values.

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Target(ElementType.METHOD) // Specifies where this annotation can be used (method, field, etc.)

@Retention(RetentionPolicy.RUNTIME) // Specifies how long the annotation is retained (runtime, compile-time, etc.)

public @interface MyAnnotation {

String value() default "default value"; // Example of an annotation element with a default value

int count() default 1;

}

- `@Target`: Specifies where the annotation can be applied (method, field, class, etc.). `ElementType.METHOD` means this annotation can only be used on methods.

- `@Retention`: Specifies how long the annotation information is retained. `RetentionPolicy.RUNTIME` means the annotation information is available at runtime via reflection.

- **Using the Custom Annotation:** Once defined, you can use your custom annotation in your code. Here's an example of applying the `@MyAnnotation` to a method:

public class MyClass {

@MyAnnotation(value = "Hello", count = 5)

public void myMethod() {

// Method implementation

}

}

- In this example, `@MyAnnotation(value = "Hello", count = 5)` applies the `MyAnnotation` to the `myMethod()` with specified values for its elements (`value` and `count`).

https://www.javatpoint.com/java-annotation

Q5) How to write custom exception in java?

We can create a custom exception class by extending either `RuntimeException` (if you want an unchecked exception) or `Exception` (if you want a checked exception).

```
public class CustomException extends RuntimeException {

    public CustomException() {
        super();
    }

    public CustomException(String message) {
        super(message);
    }

    public CustomException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Q6) What is Abstract Class in java?

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Q7. What is the internal working of @SpringBootApplication annotation?

Ans: It is a combination of three annotations @ComponentScan, @EnableAutoConfiguration and @Configuration. @Configuration: It is a class level annotation which indicates that a class can be used by the Spring IOC container as a source of bean definition. The method annotated with @Bean annotation will return an object that will be registered as a Spring Bean in IOC. @ComponentScan: It is used to scan the packages and all of its sub-packages which registers the classes as spring bean in the IOC container. @EnableAutoConfiguration: This annotation tells how Spring should configure based on the jars in the classpath. For eg , if H2 database jars are added in classpath , it will create datasource connection with H2 database.

Q8) How to reverse string in java?

```
String str = "hello";
String reversedStr = new StringBuilder(str).reverse().toString();
```

--without java8

```
String reversedStr = "";
      for (int i = str.length() - 1; i >= 0; i--) {
        reversedStr += str.charAt(i);
      }
```

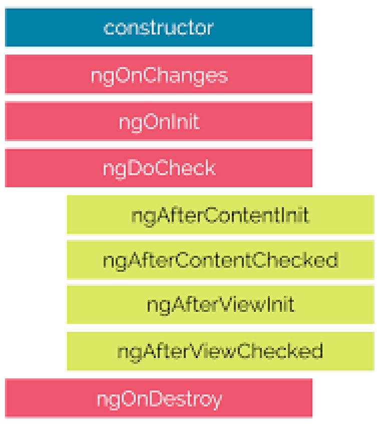
Q9) Difference between @Controller and @RestController annotations.

Ans: A class which is annotated with @Controller indicates that it is a controller class which is responsible to handle the requests and forwards the request to perform business logic. It returns a view which is then resolved by ViewResolver after performing business operations.

@RestController is used in REST Webservices and is combination of @Controller and @ResponseBody.It returns the object data directly to HTTP Response as a JSON or XML.

Q10)What are lifcycle hooks available?/ angular component lifecycle?

Angular application goes through an entire set of processes or has a lifecycle right from its initiation to the end of the application. The representation of lifecycle in pictorial representation as follows,



The description of each lifecycle method is as below,

i. **ngOnChanges:** When the value of a data bound property changes, then this method is called.
ii. **ngOnInit:** This is called whenever the initialization of the directive/component after Angular first displays the data-bound properties happens.
iii. **ngDoCheck:** This is for the detection and to act on changes that Angular can't or won't detect on its own.
iv. **ngAfterContentInit:** This is called in response after Angular projects external content into the component's view.
v. **ngAfterContentChecked:** This is called in response after Angular checks the content projected into the component.
vi. **ngAfterViewInit:** This is called in response after Angular initializes the component's views and child views.
vii. **ngAfterViewChecked:** This is called in response after Angular checks the component's views and child views.
viii. **ngOnDestroy:** This is the cleanup phase just before Angular destroys the directive/component.

Q11) What is @input and @output annotation in angular?

In Angular, `@Input` and `@Output` are decorators used to facilitate communication between components.

1. **@Input()**
    - Used to pass data from a parent component to a child component.
    - It allows a parent component to bind data to a child component property.
    - Syntax example:

      ```
      typescript
      Copy code
      @Input() propertyName: type;
      ```

    - The parent component can then bind to this property in the child component's template using property binding (`[propertyName]="parentData"`).
2. **@Output()**
    - Used to emit events from a child component to a parent component.
    - It allows a child component to send data or events to its parent component.
    - Syntax example:

      ```
      typescript
      Copy code
      @Output() eventName: EventEmitter<type> = new EventEmitter();
      ```

    - The child component can then emit events using `this.eventName.emit(data)`, and the parent component can listen to these events using event binding (`(eventName)="handleEvent($event)"`).

Usage Example:

Consider a parent component (`parent.component.ts`) and a child component (`child.component.ts`):

child.component.ts:

```
import { Component, Input, Output, EventEmitter } from '@angular/core';

@Component({
  selector: 'app-child',
  template: `
    <button (click)="sendMessage()">Send Message</button>
  `
})
export class ChildComponent {
  @Input() message: string;
  @Output() messageSent: EventEmitter<string> = new EventEmitter();

  sendMessage() {
    this.messageSent.emit('Message from child');
  }
}
```

parent.component.ts:

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-parent',
  template: `
    <app-child [message]="parentMessage"
(messageSent)="handleMessage($event)"></app-child>
    <p>Received Message: {{ receivedMessage }}</p>
  `
})
export class ParentComponent {
  parentMessage = 'Message from parent';
  receivedMessage: string;

  handleMessage(message: string) {
    this.receivedMessage = message;
  }
}
```

In this example:

- `@Input() message: string;` in `ChildComponent` allows the parent component (`ParentComponent`) to pass data (`parentMessage`) to `ChildComponent`.
- `@Output() messageSent: EventEmitter<string> = new EventEmitter();` in `ChildComponent` allows `ChildComponent` to emit events (`messageSent`) to the parent component (`ParentComponent`).
- `(messageSent)="handleMessage($event)"` in `<app-child>` tag within `ParentComponent` listens for events emitted by `ChildComponent` and calls `handleMessage($event)` in `ParentComponent` when the event occurs.

These decorators (`@Input` and `@Output`) facilitate the flow of data and events between components in Angular applications, enabling effective component communication.

Q12) What is child component?

Child Component:

- In Angular, a child component refers to a component that is encapsulated within another component, known as its parent component.
- Child components are used to break down the UI into smaller, reusable parts, each responsible for specific functionality or presentation.
- They inherit and extend the behavior of their parent components and can communicate with them using `@Input()` and `@Output()` decorators.

Q13) What is self join or inner join?

"Self join" refers to a SQL operation where a table is joined with itself. This is achieved by referencing the table with different aliases within the same SQL query. The primary use case for a self join is when you need to compare rows within the same table.

Q14) How to find first even integer from the array?

Int arc[] = {1,2,3,4,5,6};

Ans:-

```
int arr[] = {1, 4, 3, 2, 5, 6};
// Convert array to IntStream
int firstEven = Arrays.stream(arr)
        // Filter even numbers
        .filter(num -> num % 2 == 0)
        // Find the first even number
        .findFirst()
        // Return 0 if no even number is found (not recommended)
        .orElse(0);
// Print the result
System.out.println("First even number: " + firstEven);
```

---PWC-------------------------------------

Q1) Write a group by query?

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country;
```

SELECT product_id, SUM(quantity) AS total_quantity FROM orders GROUP BY product_id;

Q2) How to write add example using java8?

```
interface Addable{
    int add(int a,int b);
}

public class LambdaExpressionExample5{
    public static void main(String[] args) {

        // Multiple parameters in lambda expression
        Addable ad1=(a,b)->(a+b);
        System.out.println(ad1.add(10,20));

        // Multiple parameters with data type in lambda expression
        Addable ad2=(int a,int b)->(a+b);
14.     System.out.println(ad2.add(100,200));
15.   }
16. }
```

--**Volksvagon**-------------------------------------

Q1) Find the sum of all even present in List of Integer using Stream APIs and without using sum method?

```
int sumOfEvens = numbers.stream()
                    .filter(num -> num % 2 == 0) // Filter even numbers
                    .reduce(0, (subtotal, num) -> subtotal + num); // Accumulate
```

Q2) Write a program to find first nonrepeating character in the given string using java8?

```
        String str= "aabbccdeeffgghhi";
        String nonrepeatingChar =
Arrays.stream(str.split("")).map(String::toLowerCase).collect(Collectors.gr
oupingBy(s -> s, LinkedHashMap::new,
Collectors.counting())).entrySet().stream()
        .filter(entry -> entry.getValue() == 1)
        .map(Map.Entry::getKey)
        .findFirst()
        .orElseThrow(() -> new RuntimeException("No non-repeating character
found"));
        System.out.println(nonrepeatingChar);
```

Q3) How to call thread using java8?

```
// lambda expression to create the object
        new Thread(() -> {
            System.out.println("New thread created");
        }).start();
```

Q4) functional interface chaining example?

Certainly! Functional interface chaining in Java involves using multiple functional interfaces together to perform a sequence of operations. Here's an example that demonstrates chaining of functional interfaces using Java 8 features:

Suppose we have a functional interface `Operation` with a method `int operate(int a, int b)` that performs some operation on two integers and returns an integer result. We'll define a couple of implementations of this interface using lambda expressions:

```
@FunctionalInterface
interface Operation {
    int operate(int a, int b);
}

public class FunctionalInterfaceChaining {

    public static void main(String[] args) {
        // Example 1: Addition operation
        Operation add = (a, b) -> a + b;

        // Example 2: Multiplication operation
        Operation multiply = (a, b) -> a * b;

        // Chaining example: (2 + 3) * 5
        int result = add.andThen(multiply).operate(2, 3, 5);
        System.out.println("(2 + 3) * 5 = " + result);
    }
}
```

Explanation:

1. **Functional Interface `Operation`:**
   - `Operation` is a functional interface with a single abstract method `int operate(int a, int b)`.
2. **Lambda Expressions:**
   - Two instances of `Operation` are created using lambda expressions:
     - `add` performs addition (`(a, b) -> a + b`).
     - `multiply` performs multiplication (`(a, b) -> a * b`).
3. **Chaining Functional Interfaces:**
   - We demonstrate chaining using the `andThen` method provided by `Operation`. This method allows chaining of operations in sequence.
   - In the example, `add.andThen(multiply)` first performs addition and then performs multiplication.
4. **Execution:**
   - The statement `int result = add.andThen(multiply).operate(2, 3, 5);` computes `(2 + 3) * 5`.
   - `operate(2, 3, 5)` passes `2` and `3` to the `add` operation, then passes the result (`5`) and `5` to the `multiply` operation.
5. **Output:**
   - The program prints `(2 + 3) * 5 = 25`, showing the result of the chained operations.

Notes:

- Functional interfaces in Java 8 enable us to treat lambda expressions as instances of a single-method interface, allowing for concise and expressive functional programming.
- Chaining of functional interfaces allows us to compose complex behaviors by combining simpler operations in a sequence.
- This example demonstrates the power and flexibility of functional programming paradigms in Java, particularly useful in scenarios where operations need to be applied sequentially or conditionally.

Q5) How to write tri function in java8?

Java 8 doesn't have a built-in `TriFunction` interface that takes three arguments and returns a value. However, there are ways to achieve a similar functionality:

1. Custom TriFunction Interface:

You can define your own `TriFunction` interface:

Java

```
@FunctionalInterface
public interface TriFunction<T1, T2, T3, R> {
  R apply(T1 t1, T2 t2, T3 t3);
}
```

Use code with caution.

content_copy

Then, implement this interface for your specific use case. Here's an example that multiplies the first two arguments and adds the third:

Java

```
TriFunction<Integer, Integer, Integer, Integer> multiplyThenAdd = (x, y, z)
-> x * y + z;

int result = multiplyThenAdd.apply(2, 3, 5); // result will be 11
```

Use code with caution.

content_copy

2. Nested Functions or Currying:

You can achieve a similar effect using nested functions or currying. This involves creating a function that returns another function. Here's an example that calculates the volume of a box:

Java

```
Function<Integer, Function<Integer, Function<Integer, Integer>>> volume =
  (width) -> (length) -> (height) -> width * length * height;
```

```
int boxVolume = volume.apply(5).apply(4).apply(3); // boxVolume will be 60
```
Use code with caution.

Q6) Difference between Promise and Observable ?

Here's a comparison of `Promise` and `Observable` in Angular presented in a table format, highlighting their key differences and characteristics:

Feature	Promise	Observable
Basic Usage	**Represents a single async operation that will resolve to a value or fail with an error once.**	**Represents a stream of data that can emit multiple values over time, including completion or error events.**
Creation	**Created using `new Promise((resolve, reject) => { ... })`.**	**Created using various methods (`Observable.create()`, operators like `of`, `from`, etc.).**
Execution	**Starts execution immediately upon creation.**	**Lazy by nature; starts only when subscribed to.**
Handling	**Resolves with `.then()` and rejects with `.catch()`.**	**Subscribed to with `.subscribe()` method to handle emitted values, errors, and completion.**
Cancellation	**Cannot be canceled once created.**	**Can be canceled using `unsubscribe()` method.**
Multiple Values	**Returns a single value or fails with an error.**	**Emits multiple values over time, including zero, one, or many.**
Operators	**Does not have built-in operators.**	**Supports operators (`map`, `filter`, `merge`, etc.) for transforming and managing streams of data.**
Typical Use Cases	**Simple async operations where a single result is expected.**	**Handling streams of data over time (e.g., user interactions, data from server).**
Error Handling	**Uses `.catch()` for error handling.**	**Uses `error` handler in `subscribe()` for error handling.**
Angular HTTP Client	**Returns `Promise` by default.**	**Returns `Observable` for HTTP requests, providing flexibility in handling responses over time.**

Choosing Between Promise and Observable in Angular:

- **Use `Promise`:**
    - **For simple async operations where you expect a single result or error.**
    - **When interacting with APIs that return promises, or when integrating with libraries that work primarily with promises.**
- **Use `Observable`:**
    - **For handling streams of data over time, especially useful in scenarios like handling user interactions or real-time data.**
    - **When using Angular's HTTP client, which returns observables by default and offers powerful operators for data transformation and manipulation.**

Understanding these differences helps in choosing the appropriate asynchronous handling mechanism based on the specific requirements and use cases within an Angular application.

Q7) Difference between Pure and Impure pipes?

Pure pipe	*Impure pipe*
The pipe is executed only when it detects a change in primitive value or object reference	The pipe is executed on every change detection cycle irrespective of the change in the input value.
A single instance is created.	Multiple instances are created
It uses pure function	It uses an impure function
Pure pipe optimizes application performances.	Impure pipe may slow down your application

---Hexaware-------------------------------------

Q1) Shallow Copy Vs. Deep Copy in Java?

Shallow Copy	**Deep Copy**

It is fast as no new memory is allocated.	It is slow as new memory is allocated.
Changes in one entity is reflected in other entity.	Changes in one entity are not reflected in changes in another identity.
The default version of the clone() method supports shallow copy.	In order to make the clone() method support the deep copy, one has to override the clone() method.
A shallow copy is less expensive.	Deep copy is highly expensive.
Cloned object and the original object are not disjoint.	Cloned object and the original object are disjoint.

Q2)Difference between Overloading vs Overriding?

Method Overloading	Method Overriding
1) Method overloading increases the readability of the program.	Method overriding provides the specific implementation of the method that is already provided by its superclass.
2) Method overloading occurs within the class.	Method overriding occurs in two classes that have IS-A relationship between them.
3) In this case, the parameters must be different.	In this case, the parameters must be the same.
4) Method overloading is the example of compile time polymorphism.	Method overriding is the example of run time polymorphism.

5) In Java, method overloading can't be performed by changing return type of the method only. Return type can be same or different in method overloading. But you must have to change the parameter.	Return type must be same or covariant in method overriding.

Q3) In functional interface can we keep more than one default and static method?

Yes, functional interfaces can contain any quantity of default and static methods.

Q4)What is the Difference between ArrayList vs LinkedList?

No.	ArrayList	LinkedList
1)	ArrayList uses a dynamic array.	LinkedList uses a doubly linked list.
2)	ArrayList is not efficient for manipulation because too much is required.	LinkedList is efficient for manipulation.
3)	ArrayList is better to store and fetch data.	LinkedList is better to manipulate data.
4)	ArrayList provides random access.	LinkedList does not provide random access.
5)	ArrayList takes less memory overhead as it stores only object	LinkedList takes more memory overhead, as it stores the object as well as the address of that object.

Q5) Internal working of ConcurrentHashMap?

The ConcurrentHashMap class provides a concurrent version of the standard HashMap. So its functionality is similar to a HashMap, except that it has internally maintained concurrency. Depending upon the level of concurrency required the concurrent HashMap is internally divided into segments. If the level of concurrency required is not specified then it is takes 16 as the default value. So internally the ConcurrentHashMap will be divided into 16 segments. Each Segment behaves independently.

https://www.javainuse.com/java/javaConcurrentHashMap

Q6) write a Java 8 Program to find/check the given Strings are anagrams or not?

Without Java8

```
            String str1 = "RaceCar";
            String str2 = "CarRace";

            char[] charArray1 = str1.toLowerCase().toCharArray();
            char[] charArray2 = str2.toLowerCase().toCharArray();
            Arrays.sort(charArray1);
            Arrays.sort(charArray2);
            boolean equals = Arrays.equals(charArray1, charArray2);
            System.out.println(equals);
```

With Java8 stream

```
String str1 = "RaceCar";
String str2 = "CarRace";

String nstr1 =
Arrays.stream(str1.split("")).map(String::toLowerCase).sorted().collect(Col
lectors.joining());
            String nstr2 =
Arrays.stream(str2.split("")).map(String::toLowerCase).sorted().collect(Col
lectors.joining());
            boolean eq = nstr1.equals(nstr2)?true:false;
            System.out.println(eq);
```

Q7) What is the difference between Put vs Patch?

- **PUT**: Used to completely replace a resource on the server with the request payload.
- **PATCH**: Used to apply a partial update to a resource, sending only the data that has changed.

PUT is idempotent (repeating the same request has the same effect), while PATCH is not necessarily idempotent.

Q8) Best Practices to writing Rest API?

Writing a good REST API involves following best practices that ensure clarity, consistency, usability, and maintainability. Here are some key practices:

1. **Use Nouns for Resources**: Represent resources as nouns (e.g., `/users`, `/products`) rather than verbs or actions.
2. **Use HTTP Methods Correctly**:
   - **GET**: Retrieve a resource or collection.
   - **POST**: Create a new resource.
   - **PUT**: Replace an existing resource or create if not exists.
   - **PATCH**: Partially update a resource.
   - **DELETE**: Remove a resource.
3. **Use Plural Nouns for Collections**: Use plural nouns for collections (`/users`) and singular nouns for individual resources (`/users/{id}`).
4. **Versioning**: Use versioning in your URLs (e.g., `/api/v1/users`) to manage changes without breaking existing clients.
5. **Consistent Endpoint Naming**: Use consistent naming conventions for endpoints (`/users/{id}/orders`), avoiding unnecessary complexity or abbreviations.
6. **HTTP Status Codes**: Use appropriate HTTP status codes to indicate the result of an operation (`200 OK`, `201 Created`, `400 Bad Request`, `404 Not Found`, etc.).
7. **Error Handling**: Provide meaningful error messages and details in response bodies, especially for client errors (`4xx`) and server errors (`5xx`).
8. **Pagination**: Use pagination (`page`, `limit`) for large collections to improve performance and usability.
9. **Filtering, Sorting, and Searching**: Support filtering (`?filter={criteria}`), sorting (`?sort={field}`), and searching (`?q={query}`) where applicable.
10. **Security**: Implement proper authentication (e.g., JWT tokens) and authorization mechanisms (e.g., roles, scopes).
11. **Documentation**: Provide clear and concise documentation for your API endpoints, including usage examples and response schemas (e.g., Swagger/OpenAPI).
12. **Consistency in Responses**: Ensure consistency in response formats (e.g., JSON), naming conventions, and data structures across endpoints.
13. **Statelessness**: Design your API to be stateless where possible, meaning each request should contain all the information needed to process it.
14. **Testing**: Test your API thoroughly, including edge cases, error handling, and performance considerations.
15. **Version Control**: Use version control (e.g., Git) for managing changes to your API codebase.

By following these best practices, you can create a well-designed and maintainable REST API that is easy to understand, use, and integrate with other systems.

Q9) how to allow cross origin in spring boot?

In a Spring Boot application, you may need to allow Cross-Origin Resource Sharing (CORS) if your frontend application (which runs on a different origin) needs to make requests to your Spring Boot backend API. Here’s how you can enable CORS in Spring Boot:

1. Using `@CrossOrigin` Annotation (for specific controllers or methods)

You can enable CORS for specific controllers or controller methods by using the `@CrossOrigin` annotation.

```
java
Copy code
@RestController
@RequestMapping("/api")
public class YourController {

    @GetMapping("/endpoint")
    @CrossOrigin(origins = "http://localhost:3000") // Replace with your
frontend URL
    public ResponseEntity<?> yourMethod() {
        // Your code here
    }
}
```

- `origins`: **Specify the origins (URLs) that are allowed to access the resource. You can use wildcard `*` to allow access from any origin, but it's recommended to specify specific origins for security reasons.**

2. Global CORS Configuration

Alternatively, you can configure CORS globally for all endpoints in your Spring Boot application.

Create a Configuration Class

Create a class annotated with `@Configuration` that implements `WebMvcConfigurer`.

```
java
Copy code
@Configuration
public class CorsConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**") // Allow CORS for all paths
                .allowedOrigins("http://localhost:3000") // Replace with
your frontend URL
                .allowedMethods("GET", "POST", "PUT", "PATCH", "DELETE",
"OPTIONS") // Allowed HTTP methods
                .allowedHeaders("*") // Allowed headers
                .allowCredentials(true); // Allow credentials (cookies,
authorization headers)
    }
}
```

- `allowedOrigins`: **List of origins that are allowed to access the resource.**
- `allowedMethods`: **List of HTTP methods that are allowed (e.g., `GET`, `POST`, etc.).**
- `allowedHeaders`: **List of headers that are allowed in a CORS request.**

- `allowCredentials`: **Whether the browser should include credentials such as cookies or authorization headers in CORS requests.**

3. Using `application.properties` (or `application.yml`)

You can also configure CORS using properties in your `application.properties` or `application.yml` file:

application.properties:

```
properties
Copy code
# CORS configuration
cors.allowed-origins=http://localhost:3000
cors.allowed-methods=GET,POST,PUT,PATCH,DELETE,OPTIONS
cors.allowed-headers=*
cors.allow-credentials=true
```

application.yml:

```
yaml
Copy code
cors:
  allowed-origins: "http://localhost:3000"
  allowed-methods: "GET,POST,PUT,PATCH,DELETE,OPTIONS"
  allowed-headers: "*"
  allow-credentials: true
```

Accessing Properties in Configuration Class

You can access these properties in your `CorsConfig` class as follows:

```
java
Copy code
@Configuration
@ConfigurationProperties(prefix = "cors")
public class CorsConfig implements WebMvcConfigurer {

    private String allowedOrigins;
    private String allowedMethods;
    private String allowedHeaders;
    private boolean allowCredentials;

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
                .allowedOrigins(allowedOrigins.split(","))
                .allowedMethods(allowedMethods.split(","))
                .allowedHeaders(allowedHeaders.split(","))
                .allowCredentials(allowCredentials);
    }

    // Getters and setters for properties
}
```

Notes:

- Security Considerations**: Be cautious about allowing `*` for `allowedOrigins` as it opens your API to potential cross-site request forgery (CSRF) attacks. Always specify exact origins when possible.**
- Testing**: Ensure to thoroughly test your CORS configuration to verify it behaves as expected with your frontend application.**

By implementing one of these methods, you can configure CORS in your Spring Boot application to allow cross-origin requests from your frontend application securely.

Q10) how to implement spring caching in spring boot?

Implementing caching in a Spring Boot application can significantly improve performance by storing frequently accessed data in memory. Spring Framework provides robust support for caching through various annotations and configurations. Here's how you can implement caching in a Spring Boot application:

1. Add Dependencies

Make sure you have the necessary dependencies in your `pom.xml` (for Maven) or `build.gradle` (for Gradle):

Maven:

```
xml
Copy code
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
```

Gradle:

```
gradle
Copy code
implementation 'org.springframework.boot:spring-boot-starter-cache'
```

This dependency includes the necessary caching libraries along with your Spring Boot application.

2. Enable Caching

In your Spring Boot main class or configuration class, enable caching by annotating with `@EnableCaching`:

```
java
Copy code
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```
@SpringBootApplication
@EnableCaching
public class YourApplication {

    public static void main(String[] args) {
        SpringApplication.run(YourApplication.class, args);
    }
}
```

3. Configure Caching Provider (Optional)

By default, Spring Boot uses a simple in-memory cache manager (`ConcurrentMapCacheManager`). If you want to use a different caching provider (e.g., EhCache, Redis), you can configure it in your application properties (`application.properties` or `application.yml`):

application.properties:

```
properties
Copy code
# Use EhCache as the caching provider
spring.cache.type=ehcache
```

application.yml:

```
yaml
Copy code
# Use EhCache as the caching provider
spring:
  cache:
    type: ehcache
```

4. Add Cacheable Annotations

Use the `@Cacheable` annotation on methods that should be cached. When the method is called with the same arguments, Spring checks the cache first and returns the cached result if available:

```
java
Copy code
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;

@Service
public class YourService {

    @Cacheable(value = "yourCacheName", key = "#key")
    public Object getCachedData(String key) {
        // This method will only be executed if there's no cached value
with the provided key
        // Fetch and return data here
    }
}
```

- `@Cacheable`**: Marks a method for caching. It takes parameters like `value` (cache name), `key` (cache key, using SpEL expressions), and other attributes to customize caching behavior.**

5. Cache Eviction and Expiration

You can control cache eviction (removing items from the cache) and expiration (time-to-live) using additional annotations like `@CacheEvict`, `@CachePut`, and configuration properties:

- `@CacheEvict`: **Removes items from the cache.**
- `@CachePut`: **Updates the cache with the result of the method invocation.**
- Configuration: **Configure eviction and expiration policies in your cache configuration or through annotations.**

Example Configuration for EhCache

If you choose to use EhCache, add its dependency and configure it in your `pom.xml` or `build.gradle`. Here's a basic example of configuring EhCache:

pom.xml (add EhCache dependency):

```
xml
Copy code
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
<dependency>
    <groupId>org.ehcache</groupId>
    <artifactId>ehcache</artifactId>
</dependency>
```

application.properties (configure EhCache):

```
properties
Copy code
spring.cache.type=ehcache
```

ehcache.xml (EhCache configuration file in `src/main/resources`):

```
xml
Copy code
<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="http://www.ehcache.org/ehcache.xsd"
         updateCheck="true"
         monitoring="autodetect"
         dynamicConfig="true">

    <cache name="yourCacheName"
           maxEntriesLocalHeap="1000"
           eternal="false"
           timeToIdleSeconds="300"
           timeToLiveSeconds="600"/>

</ehcache>
```

This XML configuration sets up a cache named `yourCacheName` with specific properties like `maxEntriesLocalHeap` (maximum entries in the cache), `eternal` (whether entries expire or not), `timeToIdleSeconds` (how long an entry can be idle before it expires), and `timeToLiveSeconds` (maximum lifespan of an entry).

Summary

Implementing caching in Spring Boot involves enabling caching, annotating methods for caching, configuring caching providers (if necessary), and optionally configuring cache eviction and expiration policies. This approach helps improve application performance by reducing the need to repeatedly fetch data from slower data sources.

Q11) different design pattern in spring boot like cqrs, saga, rate limiter?

CQRS(Common Querry Responsibility Segregation) is a type of design pattern that separates the responsibility of handling commands and queries into different components. CQRS architectural pattern mainly focuses on separating the way of reading and writing the data. It separates the read and update operations on a datastore into two separate models: Queries and Commands, respectively.

https://www.geeksforgeeks.org/cqrs-command-query-responsibility-segregation/

Rate Limiter Design Pattern

Rate limiting is a technique used in system design to control the rate at which incoming requests or actions are processed or served by a system. It imposes constraints on the frequency or volume of requests from clients to prevent overload, maintain stability, and ensure fair resource allocation.

We can use **Resilience4j-Ratelimiter** or rate limiter design pattern.

The **Saga pattern** provides a way to manage transactions that involve multiple microservices. It is used to ensure that a series of transactions across multiple services are completed successfully, and if not, to roll back or undo all changes that have been made up to that point.

The Saga Pattern can be implemented in two different ways.

Choreography: In this pattern, the individual microservices consume the events, perform the activity, and pass the event to the next service. There is no centralized coordinator, making communication between the services more difficult:

Orchestration: In this pattern, all the microservices are linked to the centralized coordinator that orchestrates the services in a predefined order, thus completing the application flow. This facilitates visibility, monitoring, and error handling:

https://www.baeldung.com/orkes-conductor-saga-pattern-spring-boot

Q12) how to use HashMap with multithreading in java?

Using a HashMap with multithreading in Java requires careful consideration to ensure thread safety, especially if multiple threads are accessing and possibly modifying the HashMap concurrently. Here are several approaches to achieve thread safety when using HashMap in a multithreaded environment:

1. ConcurrentHashMap

The ConcurrentHashMap class is designed for concurrent access from multiple threads without the need for external synchronization. It provides thread safety and efficient concurrency:

Map<KeyType, ValueType> concurrentMap = new ConcurrentHashMap<>();

- **Advantages**: Built-in thread safety, high concurrency, and good performance for most use cases.
- **Usage**: Use ConcurrentHashMap when you need a thread-safe map with frequent reads and writes from multiple threads.

2. SynchronizedMap

You can use Collections.synchronizedMap() to create a synchronized wrapper around a standard HashMap. This approach uses external synchronization to make the HashMap thread-safe:

Map<KeyType, ValueType> synchronizedMap = Collections.synchronizedMap(new HashMap<>());

- **Advantages**: Provides thread safety by synchronizing access to the underlying HashMap.
- **Usage**: Useful when you already have code using HashMap and need a quick way to make it thread-safe.

3. Using Locks

You can use explicit locking mechanisms like ReentrantLock to synchronize access to a HashMap. This approach gives you more control over locking:

```
Map<KeyType, ValueType> map = new HashMap<>();

ReentrantLock lock = new ReentrantLock();



// Example usage in a method

public void addToMap(KeyType key, ValueType value) {

    lock.lock();

    try {

        map.put(key, value);

    } finally {

        lock.unlock();

    }

}
```

- **Advantages**: Fine-grained control over locking, can handle complex synchronization requirements.
- **Usage**: Suitable when you need more control over locking or when integrating with existing locking mechanisms.

Best Practices

- **Choose the Right Implementation**: Use ConcurrentHashMap for most scenarios requiring concurrent access unless you have specific reasons to use synchronized maps or custom locking.
- **Minimize Lock Contention**: Avoid holding locks for extended periods and ensure that critical sections are kept as short as possible to reduce contention.
- **Use Immutable or Thread-Local Objects**: Consider using immutable objects or thread-local variables in conjunction with HashMap to reduce the need for synchronization.
- **Concurrent Modification**: Be cautious with iterators and other operations that can lead to concurrent modification exceptions. ConcurrentHashMap handles this more gracefully than a synchronized HashMap.

In summary, ConcurrentHashMap is generally the preferred choice for thread-safe HashMap usage due to its built-in concurrency support and good performance characteristics. However, the choice depends on your specific concurrency requirements and performance considerations.

Q13) what is transactional in spring boot?

Ans: A database transaction is a sequence of statements/actions which are treated as a single unit of work. These operations should execute completely without any exception or should show no changes at all. The method on which the @Transactional annotation is declared, should execute the statements sequentially and if any error occurs, the transaction should be rolled back to its previous state. If there is no error, all the operations need to be committed to the database. By using @Transactional, we can comply with ACID principles.

E.g.: If in a transaction, we are saving entity1, entity2 and entity3 and if any exception occurs while saving entity3, then as enitiy1 and entity2 comes in same transaction so entity1 and entity2 should be rolledback with entity3.

A transaction is mainly implied on non-select operations (INSERT/UPDATE/DELETE).

https://nikhilsukhani.medium.com/transactional-annotation-in-spring-boot-ae76307fcd26

https://docs.spring.io/spring-framework/reference/data-access/transaction/declarative/annotations.html

https://www.scaler.com/topics/spring-boot/transaction-management-in-spring-boot/

Q14) What is the difference between AngularJs and Angular?

AngularJS is an older JavaScript-based MVC framework, while Angular is a modern TypeScript-based platform with a component-based architecture. AngularJS uses controllers and scopes, whereas Angular relies on components and directives. Angular offers better performance, tooling, and mobile support compared to AngularJS. Additionally, Angular has a more active and well-supported ecosystem, while AngularJS has reached its end-of-life**.**

Company Interview Question

---Capgemini-------------------------------------

Q1) Write a thread program using java8?

```
public static void main(String[] args) {
  // using lambda Expression
new Thread(()->System.out.println("Thread is started: using Lambda
Expressions")).start();
}
```

Q2) What are the different bean scopes in spring?

There are 5 bean scopes in spring framework.

No.	Scope	Description
1)	singleton	The bean instance will be only once and same instance will be returned by the IOC container. It is the default scope.
2)	prototype	The bean instance will be created each time when requested.
3)	request	The bean instance will be created per HTTP request.
4)	session	The bean instance will be created per HTTP session.
5)	globalsession	The bean instance will be created per HTTP global session. It can be used in portlet context only.

Q3) What is DAO design Pattern?

The DAO (Data Access Object) design pattern is a structural pattern that provides a way to separate data access logic and business logic in an application. It abstracts and encapsulates all access to the data source, such as a database or a web service, into a single object or set of objects known as Data Access Objects. The DAO pattern helps to achieve separation of concerns by isolating the application/business layer from the persistence layer (data storage and retrieval).

Key Concepts and Components of DAO Pattern:

1. **DAO Interface**: Defines the standard operations to be performed on a model object or entity. This interface typically includes methods for CRUD operations (Create, Read, Update, Delete) and other specific queries related to the entity.

```
public interface UserDao {

    User findById(long id);

    void save(User user);

    void update(User user);

    void delete(User user);

    List<User> findAll();

    // Other specific queries

}
```

2. **DAO Implementation**: Provides concrete implementation of the DAO interface, which interacts with the underlying data source (e.g., database, web service). This class is responsible for executing queries and managing connections to the data source.

```
public class UserDaoImpl implements UserDao {


    @Override

    public User findById(long id) {

        // Implementation to retrieve user from database

        return null;

    }


    @Override

    public void save(User user) {

        // Implementation to save user into database

    }


    // Other CRUD operations and specific queries

}
```

3. **Entity or Model**: Represents the object that is being persisted or retrieved from the data source. It typically corresponds to a database table or a domain object.

```
public class User {
    private long id;
    private String username;
    private String email;
    // Getters and setters
}
```

4. **Benefits of DAO Pattern**:
   - **Separation of Concerns**: Separates the database code from the rest of the application, promoting cleaner and more maintainable code.
   - **Centralized Access**: Provides a centralized location for managing all data access logic, which can be reused across the application.
   - **Testability**: Easier to unit test DAO implementations without needing to mock out the database or data access logic.
5. **Usage in Enterprise Applications**:
   - In large enterprise applications, DAO pattern is commonly used alongside other patterns like Service Layer pattern to structure the application into layers, each with a distinct responsibility.
   - Helps in achieving scalability and flexibility by abstracting the database interactions, allowing changes to the database schema or technology without affecting the rest of the application.

Example Scenario:

In a web application, suppose you have a User entity and you need to perform CRUD operations on users. The DAO pattern would encapsulate all database interactions related to users into a UserDao interface and its implementations (UserDaoImpl).

```
public interface UserDao {
    User findById(long id);
    void save(User user);
    void update(User user);
    void delete(User user);
    List<User> findAll();
    // Other methods
}

public class UserDaoImpl implements UserDao {

    @Override
    public User findById(long id) {
        // Implementation to retrieve user from database
        return null;
    }

    @Override
    public void save(User user) {
        // Implementation to save user into database
    }

    @Override
    public void update(User user) {
        // Implementation to update user in database
    }

    @Override
    public void delete(User user) {
        // Implementation to delete user from database
    }

    @Override
    public List<User> findAll() {
        // Implementation to retrieve all users from database
        return null;
    }
}
```

In summary, the DAO design pattern provides a structured way to manage data access by encapsulating database-related code into separate DAO objects. This separation helps in achieving modularity, maintainability, and testability in applications that interact with persistent data sources.

https://www.digitalocean.com/community/tutorials/dao-design-pattern

https://www.javatpoint.com/dao-class-in-java

Q4) What is a Spring Bean?

Ans: A java class which is managed by the IOC container is called as Spring Bean.The life cycle of the spring bean are taken care by the IOC container.

A spring bean can be represented by using the below annotations.

- @Component
- @Service
- @Repository
- @Configuration
- @Bean
- @Controller
- @RestControlle

Q5) What is dispatcher servlet?

Ans:- The Dispatcher Servlet Class Works as the front controller. It dispatches the request to the appropriate controller and manages the flow of the application.

Q6) how many ways we can configure spring bean?

Spring Framework provides three ways to configure beans to be used in the application.

1. **Annotation Based Configuration** - By using @Service or @Component annotations. Scope details can be provided with @Scope annotation.
2. **XML Based Configuration** - By creating Spring Configuration XML file to configure the beans. If you are using Spring MVC framework, the xml based configuration can be loaded automatically by writing some boiler plate code in web.xml file.
3. **Java Based Configuration** - Starting from Spring 3.0, we can configure Spring beans using java programs. Some important annotations used for java based configuration are @Configuration, @ComponentScan and @Bean.

Q7) What is the difference between BeanFactory and Application context?

BeanFactory is the **basic container** whereas ApplicationContext is the **advanced container**. ApplicationContext extends the BeanFactory interface. ApplicationContext provides more facilities than BeanFactory such as integration with spring AOP, message resource handling for i18n etc.

Q8) What is mvvm architecture?

MVVM (Model-View-ViewModel) is a software design pattern used to separate the different aspects of an application: the data (model), the user interface (view), and the glue that holds them together (view model). Here's a breakdown of each component and how they interact:

Model:

- Represents the data of your application. This could include things like user information, product data, or any other data your application needs to manage.
- The model typically doesn't contain any UI logic or presentation details.
- It might interact with databases, APIs, or other data sources to retrieve and store data.

View:

- Responsible for displaying information and handling user interactions. This is the visual representation of the data in your application.
- The view doesn't contain any business logic or data access code.

- It displays data provided by the view model and sends user actions (like button clicks) back to the view model.

ViewModel:

- Acts as an intermediary between the view and the model.
- It prepares data from the model in a way that's easily consumable by the view.
- Handles user interactions received from the view and updates the model accordingly.
- Often uses data binding techniques to automatically update the view whenever the underlying data changes.

Benefits of MVVM:

- Separation of Concerns: Each component has a clear responsibility, making the code more maintainable and easier to test.
- Testability: The view model can be easily unit tested without relying on the actual UI or data access layer.
- Flexibility: The view and view model can be easily reused for different UI presentations or with different data sources.
- Improved Maintainability: Changes to the UI or data model are less likely to impact other parts of the application.

Here's an analogy:

Think of MVVM like a restaurant. The model is the kitchen, where the food (data) is prepared. The view is the dining room, where the customer (user) sees the beautifully presented dishes. The view model is the waiter, who takes the customer's order (user interaction), communicates it to the kitchen (model), and delivers the prepared food (formatted data) to the customer (view).

While MVVM offers many advantages, it's not always the best choice for every situation. Here are some additional points to consider:

- MVVM can introduce some additional complexity compared to simpler architectures.
- It might be overkill for very small or simple applications.

If you're building a complex application with a rich UI and data interactions, MVVM is a great design pattern to consider. It promotes cleaner code, better maintainability, and easier testing.

Q9) How to create custom pipe?

In Angular, pipes are used to transform data displayed in the template. They are a great way to format and manipulate data before displaying it to the user. Angular provides several built-in pipes such as DatePipe, UpperCasePipe, LowerCasePipe, etc. However, you can also create custom pipes to suit specific formatting or transformation needs. Here's how you can create a custom pipe in Angular:

Step-by-Step Guide to Create a Custom Pipe

Let's create a custom pipe that truncates a string to a specified length and adds ellipsis (...) at the end if the string exceeds that length.

1. Generate a New Pipe

First, use the Angular CLI to generate a new pipe. Open your terminal or command prompt and run:

bash

Copy code

```
ng generate pipe truncate
```

This command generates a new file truncate.pipe.ts in the src/app directory, and it also registers the pipe automatically in the AppModule.

2. Implement the Pipe Logic

Open truncate.pipe.ts file. The generated pipe file will look like this:

typescript

Copy code

```
import { Pipe, PipeTransform } from '@angular/core';


@Pipe({
  name: 'truncate'
})
export class TruncatePipe implements PipeTransform {


  transform(value: string, limit: number = 100): string {
    if (!value) return '';
    return value.length > limit ? value.substring(0, limit) + '...' : value;
  }


}
```

- @Pipe decorator: This decorator marks the class as an Angular pipe and specifies the name by which you will refer to the pipe in your templates (name: 'truncate').
- PipeTransform interface: The TruncatePipe class implements the PipeTransform interface, which requires you to implement a transform method. This method takes an input value (value: string) and any optional parameters (limit: number in this case) and returns the transformed value.
- Transform logic: In this example, the transform method truncates the input string (value) to the specified limit length and adds ellipsis (...) if the string exceeds that length. If the value is null, undefined, or an empty string, it returns an empty string.

3. Use the Custom Pipe in Your Component

Now that you have created the custom pipe, you can use it in your Angular components' templates.

typescript

Copy code

```
import { Component } from '@angular/core';


@Component({
  selector: 'app-root',
  template: `
    <div>
      <h2>Original Text</h2>
      <p>{{ originalText }}</p>
      <h2>Truncated Text</h2>
      <p>{{ originalText | truncate:50 }}</p>
    </div>
  `,
})
```

```
export class AppComponent {
  originalText = 'Lorem ipsum dolor sit amet, consectetur adipiscing elit.';
}
```

In this example:

- The AppComponent uses the originalText property, which contains a long string.
- The truncate pipe is applied to originalText with a parameter (50), indicating that the string should be truncated to 50 characters.

4. Register the Pipe Manually (if necessary)

If the Angular CLI does not automatically register the pipe in your AppModule, make sure to add it manually in the declarations array:

typescript

Copy code

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { TruncatePipe } from './truncate.pipe'; // Import your custom pipe here

@NgModule({
  declarations: [
    AppComponent,
    TruncatePipe // Add your custom pipe to the declarations array
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

export class AppModule { }

Summary

Creating custom pipes in Angular allows you to encapsulate data transformation logic and reuse it across your application's templates. It's a powerful feature that enhances code readability, maintainability, and reusability by abstracting common formatting or transformation tasks into reusable components.

Q10) What is observable?

Observables are a technique for event handling, asynchronous programming, and handling multiple values emitted over time.

In Angular (and JavaScript in general), an Observable is a powerful concept used for handling asynchronous data streams. It acts like a push-based mechanism for delivering data over time.

Here's a breakdown of what observables are and how they work:

What is an Observable?

Imagine a stream of water flowing from a pipe. You don't know exactly when or how much water will come out, but you can place a bucket underneath to collect it. An Observable is similar. It represents a stream of data (like numbers, strings, objects, etc.) that can be emitted over time.

Key Points about Observables:

- **Emits Values:** An Observable doesn't hold the data itself; it emits (pushes) values to interested parties.
- **Asynchronous:** Observables deal with asynchronous data, meaning the values might arrive at different times.
- **Observer Pattern:** Observables follow the observer pattern. You define an observer that specifies how to handle the emitted values (like collecting water in the bucket).
- **Subscription:** To receive data from an Observable, you need to subscribe to it. This creates a connection between the Observable and the observer.

Benefits of Observables:

- **Handling Asynchronous Operations:** Observables are ideal for handling asynchronous operations like HTTP requests, user interactions, or data fetched from a server.

- **Improved Code Readability:** They can improve code readability by separating the data source (Observable) from the logic that handles the data (observer).
- **Error Handling:** Observables allow for proper error handling mechanisms to deal with issues that might arise during data emission.
- **Chaining Operations:** You can chain operators on Observables to transform, filter, or manipulate the data stream before it reaches the observer.

Using Observables in Angular:

Angular heavily utilizes Observables for various purposes. Here are some common examples:

- **HTTP Requests:** The HttpClient service in Angular returns Observables for HTTP requests, allowing you to react to the response data asynchronously.
- **Form Events:** Angular forms provide Observables for user interactions like input changes or form submissions.
- **Asynchronous Data Updates:** You can use Observables to manage data updates from external sources (like websockets) and update your components accordingly.

Learning More:

Observables are a fundamental concept in reactive programming. Here are some resources to learn more:

- Official Angular Documentation on Observables: https://angular.io/guide/observables-in-angular
- RxJS (the library used for Observables in Angular): https://rxjs.dev/

By understanding Observables, you can write more robust, asynchronous code in your Angular applications.

Q11) What is merge map and merge switch?

In RxJS (Reactive Extensions for JavaScript), mergeMap and switchMap are operators used to handle asynchronous operations and manage streams of data emitted by Observables. Both operators are commonly used for scenarios where you want to map each emitted value to another Observable, and then flatten these inner Observables into a single Observable.

mergeMap Operator:

The mergeMap operator is used to merge the emissions of multiple Observables into a single Observable. It takes each value emitted by the source Observable and applies a function to it, which returns an Observable. It then subscribes to this returned Observable and emits the values emitted by it. It continues this process for each emitted value, resulting in a merged stream of values from all inner Observables.

Syntax:

typescript

Copy code

```
import { of } from 'rxjs';
import { mergeMap } from 'rxjs/operators';


// Example using mergeMap
of('hello', 'world').pipe(
  mergeMap(value => of(`Mapped ${value}!`))
).subscribe(result => console.log(result));
```

In this example:

- of('hello', 'world') creates an Observable that emits two values: 'hello' and 'world'.
- mergeMap applies a function (value => of(Mapped ${value}!)) to each emitted value ('hello' and 'world'). This function returns an Observable (of(Mapped ${value}!)) that emits a mapped value.
- The subscribe method logs each mapped value (Mapped hello! and Mapped world!) to the console.

switchMap Operator:

The switchMap operator is similar to mergeMap, but it has a distinct behavior: whenever a new value is emitted by the source Observable, it cancels (or unsubscribes) from any previous inner Observables that were still processing and switches to the new inner Observable. This means that only the values from the most recent inner Observable are emitted, while the values from previous inner Observables are discarded.

Syntax:

typescript

Copy code

```
import { of, interval } from 'rxjs';
import { switchMap, take } from 'rxjs/operators';


// Example using switchMap
of('hello', 'world').pipe(
  switchMap(value => interval(1000).pipe(take(3)))
).subscribe(result => console.log(result));
```

In this example:

- of('hello', 'world') creates an Observable that emits two values: 'hello' and 'world'.
- switchMap applies a function (value => interval(1000).pipe(take(3))) to each emitted value ('hello' and 'world'). This function returns an Observable (interval(1000).pipe(take(3))) that emits numbers every second (interval(1000)) and completes after emitting 3 values (take(3)).
- The subscribe method logs the numbers emitted by the inner Observable (0, 1, 2, 0, 1, 2) to the console. Note how the inner Observable restarts for each emitted value from the source.

Comparison:

- mergeMap:
  - Merges the emissions of multiple inner Observables into a single Observable.
  - Emits values from all inner Observables in the order they are emitted.
  - Does not cancel or unsubscribe from inner Observables when a new value is emitted from the source Observable.
- switchMap:
  - Maps each value from the source Observable to an inner Observable, but only emits values from the most recent inner Observable.
  - Cancels and unsubscribes from previous inner Observables whenever a new value is emitted from the source Observable.
  - Useful for scenarios where you want to switch to a new inner Observable whenever the source emits a new value, such as handling user inputs or search queries.

Use Cases:

- mergeMap: Use when you want to concurrently handle multiple inner streams and merge their emissions into a single stream. For example, handling multiple HTTP requests concurrently.
- switchMap: Use when you want to cancel ongoing operations and switch to a new operation whenever the source emits a new value. For example, autocomplete search functionality where you only care about the latest search term entered by the user.

In summary, mergeMap and switchMap are powerful operators in RxJS for mapping values emitted by Observables to inner Observables and managing their emissions. Understanding their behavior helps in effectively handling asynchronous operations and managing streams of data in reactive programming.

Here's an analogy to illustrate the difference:

- MergeMap: Imagine you have multiple mailboxes receiving letters (inner Observables). MergeMap delivers all the letters from all mailboxes in any order, potentially getting letters from different mailboxes at the same time.
- SwitchMap: Think of a single mailbox that only accepts one letter at a time. SwitchMap throws away any letter currently in the mailbox when a new letter arrives (cancels previous inner Observable) and only delivers the latest letter.

Q12) how ngOnDestroy work?

ngOnDestroy: This is the cleanup phase just before Angular destroys the directive/component.

In Angular, ngOnDestroy is a lifecycle hook that is called when a component is about to be destroyed. It's part of Angular's component lifecycle management, providing an opportunity to clean up resources or perform any necessary cleanup before a component instance is destroyed and removed from the DOM.

Q13) Print count of character occurence in descending order?

String input = "java programming test";

Output: a=3, g=2,m=2,r=2,t=2 ...s=1

Sol:-

LinkedHashMap<String,Long> charCount = Arrays.*stream*(input.split("")).map(String::toLowerCase)

.collect(Collectors.*groupingBy*(s->s,Collectors.*counting*()))

.entrySet().stream()

```
.sorted(Map.Entry.comparingByValue(Comparator.reverseOrder()))
                                          .collect(Collectors.toMap(
                                              Map.Entry::getKey,
                                              Map.Entry::getValue,
                                              (oldValue, newValue) -> oldValue,
LinkedHashMap<String, Long>::new));


charCount.forEach((ch, count) -> System.out.print(ch + "=" + count+" ,"));
```

Q14) {11,22,30,87,15,42,75,98}

Find largest odd number from above list by using stream API

Output: 87

Sol:-

```
Integer arr[] = {11,22,30,87,15,42,75,98};
Integer integer = Stream.of(arr).filter(a-
>a%2!=0).max(Comparator.comparing(Integer::valueOf)).get();
                         System.out.println(integer);
```

Q15) Find the total expense using javascript?

```
const transactions = [
  { name: 'Pika', expense: 90, expenseType: 'personal' },
  { name: 'Gika', expense: 100, expenseType: 'general' },
  { name: 'Fika', expense: 200, expenseType: 'personal' },
  { name: 'Mika', expense: 400, expenseType: 'general' },
  { name: 'Zika', expense: 60, expenseType: 'personal' },
];
```

Sol:-

```
let totalGExpense = 0;
let totalPExpense = 0;

transactions.forEach(e => {
  if (e.expenseType === 'general') {
    totalGExpense += e.expense;
  } else {
    totalPExpense += e.expense;
  }
});

console.log("total general expense: " + totalGExpense);
console.log("total personal expense: " + totalPExpense);
```

Q16) write a custom pipe for returning the length of string?

Sol:-

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'stringLength'
})
export class StringLengthPipe implements PipeTransform {

  transform(value: string): number {
    if (!value) return 0;
    return value.length;
  }

}
```

<p>The length of "Hello, world!" is {{ 'Hello, world!' | stringLength }}</p>

Q17) What is the output of below code?

```
String s1 = "Java";
String s2 = "Java";
String s3 = new String("Java");
System.out.println(s1 == s2);
System.out.println(s1.equals(s2));
System.out.println(s1 == s3);
System.out.println(s1.equals(s3));
```

Ans:-

```
true
true
false
true
```

Q18) What is the output of below code?

```
public class Block
{
        static
        {
                System.out.println("Static Block-1");
        }
        public static void main(String args[])
        {
                System.out.println("Main Method");
        }
        static
        {
```

System.*out*.println("Static Block-2");

}

}

Ans:-

Static Block-1

Static Block-2

Main Method

--KPMG INDIA------------------------------------

Q1)Write a program using Java 8, to find the no of occurence of the last name in the list.

Input - Full name list -> {Ravi Kumar, Neha Gupta, Arti Gupta, Kamal Rai}

Output - Kumar - 1, Gupta - 2, Rai – 1

Sol:-

```
List<String> list = Arrays.asList("Ravi Kumar", "Neha Gupta", "Arti Gupta",
"Kamal Rai");
Map<String, Long> collect = list.stream().map(s -> s.split("\\s")[1])
                .collect(Collectors.groupingBy(s->s,Collectors.counting()));
                collect.forEach((k,v)->{
                        System.out.print(k+"-"+v);
                });
```

Q2) Move all zeroes to end of array.

Input : arr[] = {1, 2, 0, 0, 0, 3, 6} {1,3,0,0,6,2}

Output : 1 2 3 6 0 0 0

Sol:-

```
                Integer arr[] = {1,3,0,0,6,2};
                int count = 0;
                for(int i = 0 ; i<arr.length; i++) {
                        if(arr[i]!=0) {
                                arr[count] = arr[i];
                                count++;
                        }
                }
                for(int i = count; i<arr.length; i++) {
                        arr[count] = 0;
                        count++;
                }
                for (Integer integer : arr) {
                        System.out.print(integer +" ");
                }
```

Company Interview Question

Q3) Check the given string of brackets is balanced or not : input = "{{}}(){[()]}"?

```
import java.util.Stack;

public class BalancedBrackets {

    public static boolean isBalanced(String s) {
        Stack<Character> stack = new Stack<>();

        for (char c : s.toCharArray()) {
            if (c == '(' || c == '{' || c == '[') {
                stack.push(c);
            } else if (c == ')' || c == '}' || c == ']') {
                if (stack.isEmpty()) {
                    return false; // Closing bracket with no matching
opening bracket
                }
                char top = stack.pop();
                if ((c == ')' && top != '(') ||
                    (c == '}' && top != '{') ||
                    (c == ']' && top != '[')) {
                    return false; // Mismatched brackets
                }
            }
        }

        return stack.isEmpty(); // True if stack is empty (all opened
brackets have a matching closing bracket)
    }

    public static void main(String[] args) {
        String input1 = "{{}}(){[()]}";
        String input2 = "{{}}(){[()]";
        String input3 = "{[()]}";

        System.out.println("Input 1 is balanced: " + isBalanced(input1));
// true
        System.out.println("Input 2 is balanced: " + isBalanced(input2));
// false
        System.out.println("Input 3 is balanced: " + isBalanced(input3));
// true
    }
}
```

https://www.javatpoint.com/balanced-parentheses-in-java

---Altimetrik--------------------------------------

Q1) Write a program to count vowel from string?

```
String str = "Hello";

Map<String,Long> vowelCount = Arrays.stream(str.split("")).filter(s ->
s.contains("a") || s.contains("e") || s.contains("i") || s.contains("o") ||
s.contains("u")).collect(Collectors.groupingBy(s->s,
Collectors.counting()));
```

```
        vowelCount.forEach((k,v)-> {
              System.out.println(k+"-"+v);
        });
```

Q2) difference between stream vs parallel stream?

Sequential Stream	**Parallel Stream**
Runs on a single-core of the computer	Utilize the multiple cores of the computer.
Performance is poor	The performance is high.
Order is maintained	Doesn't care about the order,
Only a single iteration at a time just like the for-loop.	Operates multiple iterations simultaneously in different available cores.
Each iteration waits for currently running one to finish,	Waits only if no cores are free or available at a given time,
More reliable and less error,	Less reliable and error-prone.
Platform independent,	Platform dependent

```
List numbers = new ArrayList<>();
        System.out.println("Start Adding Numbers.....");
        for (int i = 0; i < 100000; i++) {
            numbers.add(i);
        }
        System.out.println(".............................Added Numbers");
        List numbersSync = new ArrayList();
        List numbersAsync = new ArrayList();


        System.out.println("Start Sync.....");
        long startTime = System.currentTimeMillis();
        numbers.stream()
                .forEach(n -> {
                    numbersSync.add(n);
                });
        long endTime = System.currentTimeMillis();
        System.out.println("Sync: " + (endTime-startTime));


        System.out.println("Start Async.....");
        long startTime1 = System.currentTimeMillis();
        numbers.parallelStream()
                .forEach(n -> {
                    numbersAsync.add(n);
                });
        long endTime1 = System.currentTimeMillis();
        System.out.println("Async: " + (endTime1-startTime1));
```

Q3)write a query for total marks for student?

select sid,name,Sum(subject) from student group by sid;

Q4) write a query to get each student name and contact no.?

select s.sid,s.sname.c.contact from student s

inner join contact c on s.sid = c.sid;

Q5) Write multi select dropdown example in angular?

<div> <label>Select Options:</label>

<select formControlName="selectedOptions" multiple>

<option *ngFor="let option of options" [value]="option">{{ option }}</option>

</select> </div>

Q6) *ng if example in angular?

<h2 *ngIf="showTitle">Conditional Title</h2>

Q7) How will you call a default method of an interface in a class?

Using the *super* keyword along with the interface name.

```
interface Vehicle {
   default void print() {
      System.out.println("I am a vehicle!");
   }
}
class Car implements Vehicle {
   public void print() {
      Vehicle.super.print();
   }
}
```

Note:- We can't override the default method.

Company Interview Question

Q8) Functional Interface Example?

```
@FunctionalInterface
interface Operation {
    int operate(int a, int b);
}

public class FunctionalInterfaceChaining {

    public static void main(String[] args) {
        // Example 1: Addition operation
        Operation add = (a, b) -> a + b;

        // Example 2: Multiplication operation
        Operation multiply = (a, b) -> a * b;

        System.out.println(add. operate(2,3));
    }
}
```

Q9) How to find all the even numbers that exists in the list.

```
List<Integer> list=Arrays.asList(7,3,2,9,5,32);
listInt.stream().filter(n->n%2==0).collect(Collectors.toList());
```

--Accolite--------------------------------------

Q1) Given a staircase of N steps and you can either climb 1 or 2 steps at a given time. The task is to return the count of distinct ways to climb to the top.

https://www.geeksforgeeks.org/count-ways-reach-nth-stair-using-step-1-2-3/

```
     // Returns count of ways to reach
    // n-th stair using 1 or 2 or 3 steps.
    public static int findStep(int n)
    {
        if ( n == 0)
            return 1;
        else if (n < 0)
            return 0;

        else
            return findStep(n - 3) + findStep(n - 2)
                + findStep(n - 1);
    }

    // Driver function
    public static void main(String argc[])
    {
        int n = 4;
        System.out.println(findStep(n));
    }
```

Q2) Delete Nth node from the end of the given linked list

Given a linked list and an integer N, the task is to delete the Nth node from the end of the given linked list.

```
public static LinkedList
      deleteAtPosition(LinkedList list, int index)
      {
            // Store head node
            Node currNode = list.head, prev = null;

            //
            //Case 1: remove last node value from linked list
            // Find the second last node
            if (index == -1 && currNode != null) {
        while (currNode.next.next != null)
            currNode = currNode.next;

        // Change next of second last
        currNode.next = null;

        return list;
            }

}
```

Q3)what is functional interface?

Functional Interfaces are the types of interfaces that can contain only a single abstract method. These interfaces are additionally recognized as Single abstract method interfaces. *Runnable*, *ActionListener*, ***and*** *Comparable* **are some of the examples of functional interfaces.**

Q4)What is Default Method?

The default method is a method that provides an implementation of methods within interfaces allowing the Interface development facilities.

the default keyword is used to define default methods in interfaces.

Why Default Methods?

Before Java 8, interfaces could only declare method signatures that classes implementing those interfaces had to provide. This became problematic when new methods needed to be added to interfaces, as it would break existing implementation classes unless those classes were updated to implement the new methods.

To solve this problem, Java 8 introduced default methods. Default methods provide a way to add new functionality to interfaces without forcing all implementing classes to implement these new methods.

Q5) How HashMap works internally?/Working of HashMap in java?

What is Hashing

It is the process of converting an object into an integer value. The integer value helps in indexing and faster searches.

What is HashMap

HashMap is a part of the Java collection framework. It uses a technique called Hashing. It implements the map interface. It stores the data in the pair of Key and Value. HashMap contains an array of the nodes, and the node is represented as a class. It uses an array and LinkedList data structure internally for storing Key and Value. There are four fields in HashMap.

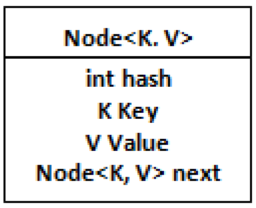


Figure: Representation of a Node

Before understanding the internal working of HashMap, you must be aware of hashCode() and equals() method.

- **equals():** It checks the equality of two objects. It compares the Key, whether they are equal or not. It is a method of the Object class. It can be overridden. If you override the equals() method, then it is mandatory to override the hashCode() method.
- **hashCode():** This is the method of the object class. It returns the memory reference of the object in integer form. The value received from the method is

used as the bucket number. The bucket number is the address of the element inside the map. Hash code of null Key is 0.

- **Buckets:** Array of the node is called buckets. Each node has a data structure like a LinkedList. More than one node can share the same bucket. It may be different in capacity.

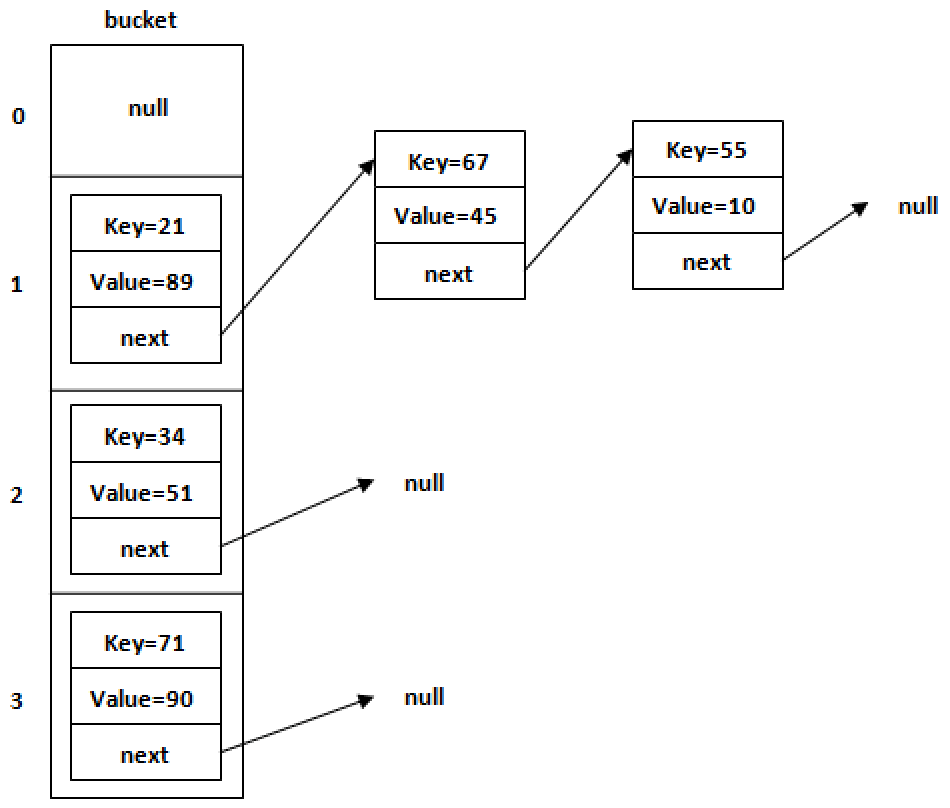


Figure: Allocation of nodes in Bucket

Insert Key, Value pair in HashMap

We use put() method to insert the Key and Value pair in the HashMap. The default size of HashMap is 16 (0 to 15).

Example

In the following example, we want to insert three (Key, Value) pair in the HashMap.

```
5. HashMap<String, Integer> map = new HashMap<>();
6. map.put("Aman", 19);
7. map.put("Sunny", 29);
8. map.put("Ritesh", 39);
```

Let's see at which index the Key, value pair will be saved into HashMap. When we call the put() method, then it calculates the hash code of the Key "Aman." Suppose the hash code of "Aman" is 2657860. To store the Key in memory, we have to calculate the index.

Calculating Index

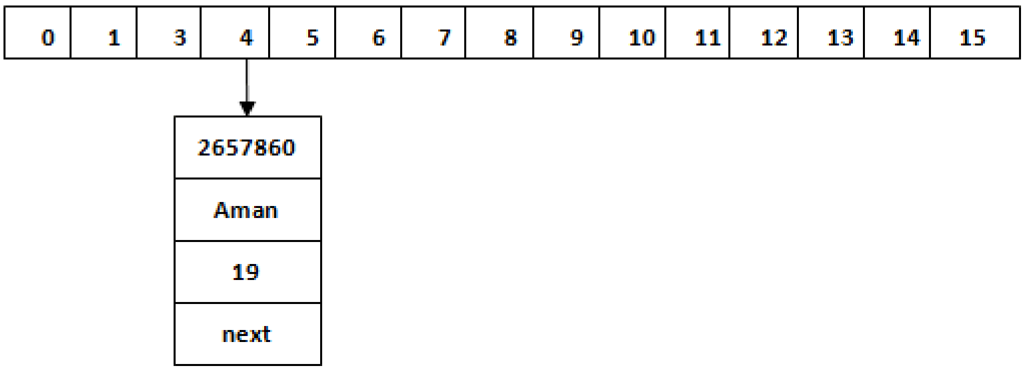
Index minimizes the size of the array. The Formula for calculating the index is:

```
2.  Index = hashcode(Key) & (n-1)
```

Where n is the size of the array. Hence the index value for "Aman" is:

```
2.  Index = 2657860 & (16-1) = 4
```

The value 4 is the computed index value where the Key and value will store in HashMap.

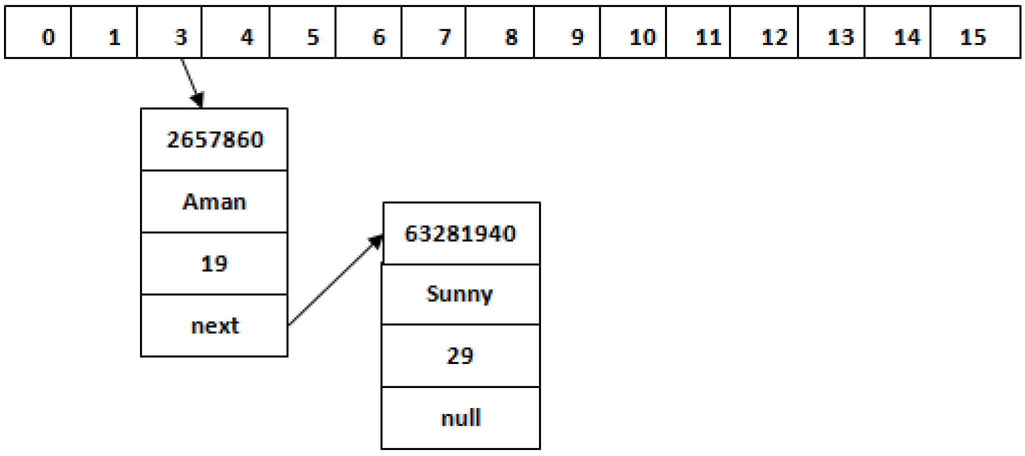


Hash Collision

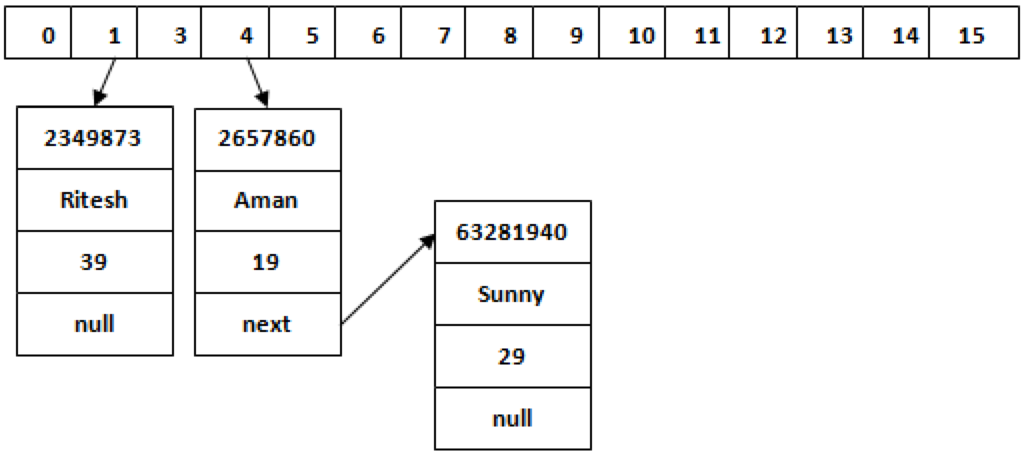
This is the case when the calculated index value is the same for two or more Keys. Let's calculate the hash code for another Key "Sunny." Suppose the hash code for "Sunny" is 63281940. To store the Key in the memory, we have to calculate index by using the index formula.

```
2.  Index=63281940 & (16-1) = 4
```

The value 4 is the computed index value where the Key will be stored in HashMap. In this case, equals() method check that both Keys are equal or not. If Keys are same, replace the value with the current value. Otherwise, connect this node object to the existing node object through the LinkedList. Hence both Keys will be stored at index 4.



Similarly, we will store the Key "Ritesh." Suppose hash code for the Key is 2349873. The index value will be 1. Hence this Key will be stored at index 1.



get() method in HashMap

get() method is used to get the value by its Key. It will not fetch the value if you don't know the Key. When get(K Key) method is called, it calculates the hash code of the Key.

Suppose we have to fetch the Key "Aman." The following method will be called.

```
2.  map.get(new Key("Aman"));
```

It generates the hash code as 2657860. Now calculate the index value of 2657860 by using index formula. The index value will be 4, as we have calculated above. get() method search for the index value 4. It compares the first element Key with the given Key. If both keys are equal, then it returns the value else check for the next element

in the node if it exists. In our scenario, it is found as the first element of the node and return the value 19.

Let's fetch another Key "Sunny."

The hash code of the Key "Sunny" is 63281940. The calculated index value of 63281940 is 4, as we have calculated for put() method. Go to index 4 of the array and compare the first element's Key with the given Key. It also compares Keys. In our scenario, the given Key is the second element, and the next of the node is null. It compares the second element Key with the specified Key and returns the value 29. It returns null if the next of the node is null.

Q6) What is IOC Container?

IOC container is responsible to:

- create the instance
- configure the instance, and
- assemble the dependencies

Q7)what is @Autowired annotation?

The @Autowired annotation in Spring Framework is used to automatically inject dependencies into a Spring bean. It provides a way to achieve dependency injection (DI) in a Spring application without the need for explicit bean wiring in XML configuration files or Java code.

Q8) how many ways we can inject di?

There are three kinds of Spring Dependency Injection: Setter, Constructor, and Field or Property-Based.

1. Field or Property-Based.

```
public class ProductService {

    @Autowired
    private ProductRepository productRepository;
}
```

2. Setter based.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RestController;
@ResstController
public class SchoolMasterDetails{
 private TeacherMasterDetails teachmastdetails;

 @Autowired
 public void setTeachMasterDetails(TeacherMasterDetails teachmastdetails){
   this.teachmastdetails = teachmastdetails;
 }

 @Override
 public String toString(){
   return "SchoolMasterDetails [teachmastdetails="+teachmastdetails+"]";
 }
}
```

3. Constructor based

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RestController;
@ResstController
public class SchoolMasterDetails{
 private TeacherMasterDetails teachmastdetails;

 @Autowired
 public SchoolMasterDetails(TeacherMasterDetails teachmastdetails){
   this.teachmastdetails = teachmastdetails;
 }

 @Override
 public String toString(){
   return "SchoolMasterDetails [teachmastdetails="+teachmastdetails+"]";
 }
}
```

--LSI--------------------------------------

Q1) 3 11 17 25 30 37 45 53 59 65. I need a data from 3 to 7 over so how can i get?

```
int[] arr = {3, 11, 17, 25, 30, 37, 45, 53, 59, 65};
int[] copyOfRange = Arrays.copyOfRange(arr, 2, 7);
System.out.println(Arrays.toString(copyOfRange));
```

Q2) How to get index of highest over score. 3 11 17 25 30 37 45 53 59 65?

```
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		int[] numbers = { 3, 11, 17, 25, 30, 37, 45, 53, 59, 65 };

		// Using Java 8 streams to calculate differences between 
consecutive elements
		int[] array = IntStream.range(1, numbers.length) // Creates a 
stream of indices from 1 to numbers.length-1
				.map(i -> numbers[i] - numbers[i - 1]).toArray();// 
Calculate difference

		int indexOfMaxValue = IntStream.range(0, 
numbers.length).reduce((i, j) -> numbers[i] > numbers[j] ? i : j)
				.orElse(-1);

		// Print the index of max value
		if (indexOfMaxValue != -1) {
			System.out.println("Index of max value: " + 
indexOfMaxValue);
		} else {
			System.out.println("Array is empty.");
		}
	}
```

Q3) Second highest salary from employee ?

select max(salary) from emp where salary < (select max(salary) from emp);

Q4)what is the output of this javascript code?

let x = 10;

function xyz(){ console.log(x) };

xyz()

Ans:- 10;

Q5)what is OOPS?

Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts.

- **Object**
- **Class**

- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Q6)What is Polymorphism?

If *one task is performed in different ways*, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

In Java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

Q7)What is Abstraction?

Hiding internal details and showing functionality is known as abstraction. For example phone call, we don't know the internal processing.

In Java, we use abstract class and interface to achieve abstraction.

Q8) Primitive Data Type in java?

Data Type	**Default Value**	**Default size**
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte

float	0.0f	4 byte
double	0.0d	8 byte

Q9)what is the difference between bind, all and apply in javascript?

In JavaScript, bind, all, and apply are methods that are used in different contexts to manipulate functions, arrays, and objects. Let's delve into each of these methods:

1. bind Method

The bind method in JavaScript creates a new function that, when called, has its this keyword set to a specific value, and arguments are also preset if provided. It's used primarily for setting the context (this value) of a function and partially applying arguments.

Syntax:

javascript

Copy code

```
const boundFunction = someFunction.bind(thisArg, arg1, arg2, ...);
```

- someFunction: The original function whose this context and arguments are to be bound.
- thisArg: The value to be passed as the this parameter when calling the function.
- arg1, arg2, ...: Optional arguments that are preset when the function is called.

Example:

javascript

Copy code

```
const person = {
  firstName: 'John',
  lastName: 'Doe',
  getFullName: function() {
    return this.firstName + ' ' + this.lastName;
  }
};


const logName = function(greeting) {
  console.log(greeting + ', ' + this.getFullName());
};


const boundLogName = logName.bind(person, 'Hello');
boundLogName(); // Outputs: Hello, John Doe
```

2. apply Method

The apply method calls a function with a given this value and arguments provided as an array (or an array-like object). It's similar to call, but apply accepts arguments as an array.

Syntax:

javascript

Copy code

```
someFunction.apply(thisArg, [argsArray]);
```

- someFunction: The function to be called.
- thisArg: The value to be passed as the this parameter when calling the function.
- argsArray: An array or array-like object containing the arguments to pass to the function.

Example:

javascript

Copy code

```
const numbers = [5, 10, 15, 20];


const sum = function(a, b, c, d) {

  return a + b + c + d;

};


const total = sum.apply(null, numbers);

console.log(total); // Outputs: 50 (5 + 10 + 15 + 20)
```

3. all Method

In JavaScript, there isn't a standard all method for arrays like in some other languages. However, if you're referring to a function like Array.prototype.every(), it checks whether all elements in an array pass a test (provided as a function). It returns a boolean value.

Syntax:

javascript

Copy code

```
const allPassed = array.every(function(currentValue, index, array) {

  // Return true or false based on condition

});
```

- currentValue: The current element being processed in the array.
- index (Optional): The index of the current element being processed.
- array (Optional): The array every was called upon.

Example:

javascript

Copy code

```
const numbers = [2, 4, 6, 8, 10];
const allEven = numbers.every(function(num) {

  return num % 2 === 0;

});



console.log(allEven); // Outputs: true (all numbers are even)
```

Summary

- **bind**: Creates a new function with a specified this value and initial arguments.
- **apply**: Calls a function with a specified this value and arguments provided as an array (or array-like object).
- **all (.every())**: Checks if all elements in an array satisfy a condition specified by a callback function.

These methods are fundamental in JavaScript for function manipulation (bind, apply) and array operations (every as a form of all). Understanding and using them effectively can greatly enhance your ability to write concise and expressive JavaScript code.

Q10) what is Promise, async , await in js?

In JavaScript, Promise, async, and await are features that facilitate asynchronous programming, making it easier to manage and handle asynchronous operations such as fetching data from a server, reading files, or executing long-running tasks without blocking the main thread. Let's explore each of these concepts:

1. Promise

A Promise is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. It allows you to handle asynchronous operations more cleanly and avoid callback hell.

Creating a Promise:

javascript

Copy code

```
const fetchData = new Promise((resolve, reject) => {
  // Simulate fetching data asynchronously (e.g., from a server)

  setTimeout(() => {

    const data = { id: 1, name: 'John Doe' };

    resolve(data); // Resolve with data if successful

    // reject(new Error('Failed to fetch data')); // Reject with an error if failed

  }, 2000); // Simulate a delay of 2 seconds

});
```

Consuming a Promise:

javascript

Copy code

```
fetchData.then(data => {

  console.log('Data:', data);

}).catch(error => {

  console.error('Error:', error);

});
```

- **resolve**: Function called when the asynchronous operation completes successfully, passing the result data.
- **reject**: Function called when the operation fails, passing an error object.

2. async Function

The async function declaration defines an asynchronous function, which returns a Promise. It allows you to write asynchronous code as if it were synchronous, making code easier to read and maintain.

Syntax:

javascript

Copy code

```
async function myAsyncFunction() {
  // Async code goes here
  return result; // Returns a Promise that will resolve with 'result'
}
```

Example:

javascript

Copy code

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    return data;
  } catch (error) {
    console.error('Error fetching data:', error);
    throw error; // Rethrow the error to handle it elsewhere
  }
}
```

- **await**: Used inside async functions to pause execution until a Promise settles (either resolves or rejects), and to resume execution after the Promise settles.

3. await Operator

The await operator is used inside async functions to wait for a Promise to resolve. It allows asynchronous code to look and behave more like synchronous code, making it easier to write and understand.

Syntax:

javascript

Copy code

```
async function myAsyncFunction() {
  const result = await someAsyncOperation();
  return result;
}
```

- **Usage**: await can only be used inside async functions. It suspends the execution of the async function until the Promise is settled (resolved or rejected).

Example:

javascript

Copy code

```
async function getData() {
  try {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    console.log('Data:', data);
  } catch (error) {
    console.error('Error fetching data:', error);
  }
}


getData();
```

- **Benefits**: async functions and await make asynchronous code more readable and manageable, especially for handling multiple asynchronous operations sequentially or in parallel.

Summary

- **Promise**: Represents the eventual completion or failure of an asynchronous operation, allowing you to handle the result asynchronously.
- **async function**: Declares an asynchronous function that returns a Promise, simplifying the syntax of working with asynchronous operations.
- **await**: Pauses the execution of an async function until a Promise is settled, allowing asynchronous code to be written in a more synchronous style.

These features are powerful tools in modern JavaScript for managing and handling asynchronous operations effectively, improving code readability and maintainability. They are widely used in web development, especially in scenarios involving API calls, data fetching, and other asynchronous tasks.

Q11) difference between let vs const?

var	**let**	**const**
The scope of a *var* variable is functional or global scope.	The scope of a *let* variable is block scope.	The scope of a *const* variable is block scope.
It can be updated and re-declared in the same scope.	It can be updated but cannot be re-declared in the same scope.	It can neither be updated or re-declared in any scope.
It can be declared without initialization.	It can be declared without initialization.	It cannot be declared without initialization.
It can be accessed without initialization as its default value is "undefined".	It cannot be accessed without initialization otherwise it will give 'referenceError'.	It cannot be accessed without initialization, as it cannot be declared without initialization.
These variables are hoisted.	These variables are hoisted but stay in the temporal dead zone untill the initialization.	These variables are hoisted but stays in the temporal dead zone until the initialization.

Q12)What is JIT?

Just-In-Time(JIT) compiler: It is used to improve the performance. JIT compiles parts of the bytecode that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

Q13)What is JVM?

Java Virtual Machine is a virtual machine that enables the computer to run the Java program. JVM acts like a run-time engine which calls the main method present in the Java code. JVM is the specification which must be implemented in the computer system. The Java code is compiled by JVM to be a Bytecode which is machine independent and close to the native code.

---Infogain--

Q1)How to get second highest salary from Employee?

select id,name,max(salary) from emp where salary = (select max(salary) from emp);

Q2)Write a program to check character is equal or not in javascript?

str1 = coder and str2 = cdoer

true

str1 = hello and str2 = kghjhgh

false

Ans:-

```
let str1 = "coder";

let str2 = "cdoer";

const arr1 = str1.split("");

const arr2 = str2.split("");

arr1.sort();

arr2.sort();

console.log(arr1.toString()==arr2.toString());
```

Q3) What is ConcurrentModificationException ?

A ConcurrentModificationException is an error in programming that occurs when a collection (like a list, set, or map) is modified concurrently (i.e., at the same time) while it is being iterated over using an iterator.

In simpler terms, it happens when you try to change a collection while you are in the middle of iterating through it with an iterator. This can lead to inconsistencies or unexpected behavior because the iterator becomes invalid due to changes made outside of its control.

https://www.javatpoint.com/concurrentmodificationexception-in-java

------------------------------------Intense Technologies--------------------------------

Q1) What is Spring actuator and its advantages.

Ans: An actuator is mainly used to provide the production ready features of an application. It helps to monitor and manage our application. It provides various features such as healthcheck, auditing, beans loaded into the application,etc.

Q2) what is application properties?

So in a spring boot application, application.properties file is used to write the application-related property into that file. This file contains the different configuration which is required to run the application in a different environment, and each environment will have a different property defined by it.

In Spring Boot, the application.properties (or application.yml) file is a central configuration file that allows you to configure your Spring Boot application. It provides a convenient way to externalize configuration from your Java code, making it easier to modify and manage various settings without changing the code itself.

https://www.javatpoint.com/spring-boot-properties

Q3) how to combine two list?

List<Integer> combinedList = Stream.concat(list1.stream(), list2.stream()) .collect(Collectors.toList());

--------Merging lists while removing duplicates using streams:

Company Interview Question

List<Integer> mergedList = Stream.concat(list1.stream(), list2.stream()) .distinct() .collect(Collectors.toList());

------------------old way

```
List<String> list1 = new
ArrayList<>(Arrays.asList("Vishal","Tarunu","Yash"));
List<String> list2 =  new
ArrayList<>(Arrays.asList("Somya","Mike","Peter"));
List<String> combined_list = new ArrayList<>();
combined_list.addAll(list1);
combined_list.addAll(list2);
System.out.println(combined_list);
```

------------------------------------Persistent---------------------------------

Q1) All oops into one single class program?

```
public class DemoClass {

  // Encapsulation (private field with getter/setter)
  private String name;

  public String getName() {
    return name;
  }

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

  // Polymorphism (method overloading)
  public int add(int x, int y) {
    return x + y;
  }

  public double add(double x, double y) {
    return x + y;
  }

  // Abstraction (simple method demonstrating functionality)
  public void performAction() {
    System.out.println("Performing action...");
  }

  // Inheritance: Overrides Object class's toString() method
  @Override
  public String toString() {
      return "OopConceptsDemo{" +
              ", name='" + name + '\'' +
              '}';
  }
}
```

Q2)Revere the word in below string?

input: i ma a avaj repoleved

output: i am a java developer

String str = "i ma a avaj repoleved";

Ans:-

```
public class ReverseWords {

    public static void main(String[] args) {
        String input = "i ma a avaj repoleved";
        String output = reverseWords(input);
        System.out.println("Input: " + input);
        System.out.println("Output: " + output);
    }

    public static String reverseWords(String input) {
        String[] words = input.split(" ");
        StringBuilder result = new StringBuilder();

        for (String word : words) {
            // Reverse each word and append to result
            String reversedWord = reverseWord(word);
            result.append(reversedWord).append(" ");
        }

        return result.toString().trim(); // Trim to remove trailing space
    }

    public static String reverseWord(String word) {
        StringBuilder reversed = new StringBuilder(word);
        return reversed.reverse().toString();
    }
}
```

Q3) dept wise emp list in java8?

```
Map<String,List<Employee>>                    collect2              =
employeeList.stream().collect(Collectors.groupingBy(e-
>e.getDepartment(),Collectors.toList()));
```

Q4) difference between ClassNotFoundException and NoClassDefFoundError in java?

The **ClassNotFoundException** occurs when you try to load a class at runtime using **Class.forName()** or **loadClass()** methods and requested classes are not found in classpath. Most of the time this exception will occur when you try to run an application without updating the classpath with JAR files. This exception is a **checked Exception** derived from **java.lang.Exception** class and you need to provide **explicit handling** for it.

The **NoClassDefFoundError** occurs when the class was present during compile time and the program was compiled and linked successfully but the class was not present during runtime. It is an error that is derived from **LinkageError**.

What is **LinkageError**? If a class is dependent on another class and we made changes in that class after compiling the former class, we will get the **LinkageError**.

Q5) how many ways to we can create java object?

- By new keyword
- By newInstance() method
- By clone() method
- By deserialization
- By factory method etc.

https://www.geeksforgeeks.org/different-ways-create-objects-java/

Q6) what is fail fast and fail safe?

Fail fast means changes inside the method while iterating the loop.

Fail safe means handle in a better way to handle safe scenario.

Fail-Fast systems abort operation as-fast-as-possible exposing failures immediately and stopping the whole operation.

Whereas, Fail-Safe systems don't abort an operation in the case of a failure. Such systems try to avoid raising failures as much as possible.

https://www.javatpoint.com/fail-fast-and-fail-safe-iterator-in-java

https://www.baeldung.com/java-fail-safe-vs-fail-fast-iterator#:~:text=Fail%2DFast%20systems%20abort%20operation,failures%20as%20much%20as%20possible.

https://www.javatpoint.com/fail-fast-and-fail-safe-in-java

https://www.geeksforgeeks.org/fail-fast-fail-safe-iterators-java/

Q7) how to make immutable class in java if i have a another class variable that contain getter and setter?

To create an immutable class in Java, you need to follow these general principles:

1. Declare the class as `final` so it can’t be extended.
2. Make all of the fields `private` so that direct access is not allowed.
3. Don’t provide setter methods for variables.
4. Make all mutable fields `final` so that a field’s value can be assigned only once.
5. Initialize all fields using a constructor method performing deep copy.
6. Perform cloning of objects in the getter methods to return a copy rather than returning the actual object reference.

```java
public class ImmutableClass {

  private final MyMutableClass mutableVariable;

  public ImmutableClass(MyMutableClass mutableVariable) {
    // Deep copy (assuming MyMutableClass has a copy constructor)
    this.mutableVariable = new MyMutableClass(mutableVariable);
  }

  public MyMutableClass getMutableVariable() {
    // Return a copy
    return new MyMutableClass(this.mutableVariable);
  }

  // Other methods of the immutable class...
}
```

Q8) what is @Primary and @Qualifier annotation?

@Qualifier

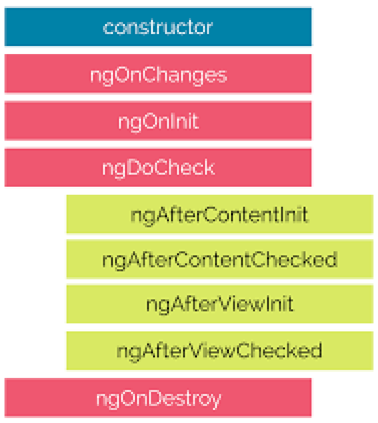
The @Qualifier annotation in Spring is used to specify which bean to inject when there are multiple beans of the same type.

@Primary

We use @Primary to give higher preference to a bean when there are multiple beans of the same type. When a bean is not marked with @Qualifier, a bean marked with @Primary will be served in case on ambiquity.

Q9) angular component life cycle?

Angular application goes through an entire set of processes or has a lifecycle right from its initiation to the end of the application. The representation of lifecycle in pictorial representation as follows,



The description of each lifecycle method is as below,

i. **ngOnChanges:** When the value of a data bound property changes, then this method is called.
ii. **ngOnInit:** This is called whenever the initialization of the directive/component after Angular first displays the data-bound properties happens.
iii. **ngDoCheck:** This is for the detection and to act on changes that Angular can't or won't detect on its own.
iv. **ngAfterContentInit:** This is called in response after Angular projects external content into the component's view.
v. **ngAfterContentChecked:** This is called in response after Angular checks the content projected into the component.
vi. **ngAfterViewInit:** This is called in response after Angular initializes the component's views and child views.
vii. **ngAfterViewChecked:** This is called in response after Angular checks the component's views and child views.
viii. **ngOnDestroy:** This is the cleanup phase just before Angular destroys the directive/component.

Q10)How many ways we can share data in angular?/ data sharing techniques

Ans:- service, @input and @output

Q11) role based authentication in angular?

can active, auth guard, route guard

Q12) any messaging service experience in java?

Twillio,Rabbit MQ, Apache Kafka

Q13)What is transaction management in spring?

https://www.scaler.com/topics/spring-boot/transaction-management-in-spring-boot/

https://nikhilsukhani.medium.com/transactional-annotation-in-spring-boot-ae76307fcd26

Q14) Exceptional Handling in java?

In Spring Boot, writing custom exceptions involves creating your own exception classes and possibly defining custom handlers to manage these exceptions gracefully within your application. Here's a step-by-step guide on how to write and use custom exceptions in a Spring Boot application:

1. Create Custom Exception Class

First, create a custom exception class by extending either `RuntimeException` (if you want an unchecked exception) or `Exception` (if you want a checked exception).

```
java
Copy code
public class CustomException extends RuntimeException {

    public CustomException() {
        super();
    }

    public CustomException(String message) {
        super(message);
    }

    public CustomException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

2. Throwing Custom Exceptions

You can throw your custom exception wherever appropriate in your application logic:

```
java
Copy code
public class MyService {

    public void someMethod() {
        // Example of throwing custom exception
        if (someCondition) {
            throw new CustomException("Custom exception occurred");
        }
    }
}
```

3. Handling Custom Exceptions

To handle custom exceptions in your Spring Boot application, you can use `@ExceptionHandler` to define how specific exceptions should be handled globally or within specific controllers.

Global Exception Handling

Create a global exception handler using `@ControllerAdvice` and `@ExceptionHandler` annotations:

```
java
Copy code
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<String> handleCustomException(CustomException ex)
{
        // Create a custom error response
        String errorMessage = "Custom exception handled: " +
ex.getMessage();
        return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(errorMessage);
    }
}
```

Controller-Specific Exception Handling

Alternatively, handle exceptions within a specific controller:

```
java
Copy code
@RestController
@RequestMapping("/api")
public class MyController {

    @GetMapping("/test")
    public ResponseEntity<String> testMethod() {
        try {
            // Business logic that may throw CustomException
            // For example:
            throw new CustomException("Custom exception occurred in
testMethod");
        } catch (CustomException ex) {
            // Handle the exception locally if needed
            return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Handled
locally: " + ex.getMessage());
        }
    }
}
```

4. Custom Error Responses

You can customize the response returned when handling exceptions by creating a custom error message or encapsulating the error details in a custom object. This ensures that your API responses are informative and useful for clients consuming your API.

5. Testing Custom Exceptions

Make sure to test your custom exceptions in unit tests to ensure they are thrown and handled correctly.

Summary

Creating custom exceptions in Spring Boot involves defining your exception class, throwing it where appropriate in your application logic, and optionally defining global or local handlers to manage these exceptions effectively. This approach helps in maintaining clean and organized error handling within your application.

Q15) what is rank, nvl, joins in sql?

These are all functionalities used in SQL queries to manipulate and retrieve data:

1. **RANK:**
   - Rank is a window function used to assign a position (rank) to rows within a result set based on a specific ordering criteria.
   - It's typically used with the OVER clause to define the window of rows considered for ranking.

Syntax:

SQL

```
SELECT column_name, RANK() OVER (PARTITION BY partition_column ORDER BY order_column) AS rank_alias

FROM table_name;
```

Use code with caution.

- column_name: The column you want to select.
- partition_column: (Optional) Defines a group of rows to be ranked within (e.g., department in an employee table).
- order_column: The column used to determine the order of ranking (e.g., salary in descending order for top earners).
- rank_alias: An alias for the rank column.

Example: Find the top 3 highest-selling products by category:

SQL

```
SELECT product_name, category, RANK() OVER (PARTITION BY category ORDER BY sales_amount DESC) AS rank
```

FROM products

WHERE rank <= 3;

Use code with caution.

2. **NVL:**
    - NVL (Null Value Logic) is a function used to handle null values in SQL.
    - It takes two arguments: the value to check for null and the value to return if the first argument is null.

Syntax:

SQL

SELECT NVL(column_name, default_value) AS alias

FROM table_name;

Use code with caution.

- column_name: The column to check for null values.
- default_value: The value to return if column_name is null.
- alias: An alias for the resulting column (optional).

Example: Replace null values in the "city" column with "Unknown City":

SQL

SELECT customer_name, NVL(city, 'Unknown City') AS city

FROM customers;

Use code with caution.

3. **Joins:**
    - Joins are used to combine data from two or more tables based on a related column.
    - There are different types of joins used for various scenarios:
        - **Inner Join:** Returns rows where there is a match in the join condition from both tables.
        - **Left Join:** Returns all rows from the left table and matching rows from the right table, even if there's no match in the right table. Null values are used for unmatched columns in the right table.
        - **Right Join:** Similar to left join, but keeps all rows from the right table.
        - **Full Join:** Returns all rows from both tables, including unmatched rows with null values.

Syntax (using inner join as an example):

SQL

```
SELECT table1.column1, table2.column2

FROM table1

INNER JOIN table2 ON table1.join_column = table2.join_column;
```

Use code with caution.

- table1, table2: The tables to be joined.
- column1, column2: Columns used for the join condition.
- join_column: The column(s) with the same values in both tables for matching rows.

Example: Join a "customers" table with an "orders" table based on the "customer_id" column:

SQL

```
SELECT c.customer_name, o.order_date, o.total_amount

FROM customers c

INNER JOIN orders o ON c.customer_id = o.customer_id;
```

Use code with caution.

By understanding these concepts (rank, nvl, joins), you can write more powerful and flexible SQL queries to manipulate and retrieve data effectively.

-------------------------------------Wissen Technology---------------------------------

Q1) Find an element in array such that sum of left array is equal to sum of right array

```
import java.util.*;
class HelloWorld {
    public static void main(String[] args) {
        int[] arr={5,-1,4,6,12,0,-4};
        int n=arr.length;
        int result=-1;
        for(int i=0;i<n;i++){
            int left_sum=0;
            int right_sum=0;

            for(int j=0;j<i;j++){
             left_sum+=arr[j];
            }

            for(int j=i+1;j<n;j++){
             right_sum+=arr[j];
            }

            if(left_sum==right_sum)
            result=arr[i];
        }
        System.out.println(result);
    }
}
```

-------------------------------------IBM--------------------------------

Q1) Minimum number of Parentheses to be added to make it valid

```
package test;
// Java Program to find minimum number of '(' or ')'
// must be added to make Parentheses string valid.

public class GFG {

    // Function to return required minimum number
    static int minParentheses(String p)
    {

        // maintain balance of string
        int bal = 0;
        int ans = 0;

        for (int i = 0; i < p.length(); ++i) {

            bal += (p.charAt(i) == '(')||(p.charAt(i) == '{') ? 1 : -1;

            // It is guaranteed bal >= -1
            if (bal == -1) {
                ans += 1;
                bal += 1;
            }
        }

        return bal + ans;
    }

    public static void main(String args[])
    {
        String p = "()){";

        // Function to print required answer
        System.out.println(minParentheses(p));
    }
    // This code is contributed by ANKITRAI1
}
```

Output 2.

Q2) get Bingo Random Number in java?

------------------------------------Capgemini--

Q1) Project -> id, projectName

User -> id,username

Details -> id,userId,projectId;

Ans:-

Select u.username,count(d.projectId) from user u inner join Details d on u.id = d.userId group by d.projectId;

Q2)write a rest controller with api?

```
@RestController

@RequestMapping("/api/users/")

public class user{



        @Autowired

        Private UserService userService;

        @Post

        public ResponseEntity<User> addUser(@RequestBody User u){

        return new ResponseEntity<userService.save(u),HttpStatus.Created>();



}

@Get
```

```
        public ResponseEntity<User> fet(){

        return new ResponseEntity<userService.save(u),HttpStatus.Created>();



}




}
```

Q3) Write a program to print output in this format?

int arr[] = {7,8,4,5,1,2};

output:- 8,1,7,2,5,4

Sol:-

```
        public static void main(String[] args) {

   int arr[] = {7,8,4,5,1,2};

   Arrays.sort(arr);

   for(int i =0; i<arr.length/2; i++){

           System.out.print(arr[arr.length-i-1]+","+arr[i]+",");

      }

        }
```

Q4) Print the value that is containg #in the string?

String str[] = {"My #name is vishal","Attending the #interview"};

Sol:-

```
        String str[] = {"My #name is vishal", "Attending the #interview"};

        // Split each string into words and filter to find words containing
'#'
        Arrays.stream(str)
```

```
                .flatMap(s -> Arrays.stream(s.split("\\s+"))) // Split each
string into words
                .filter(word -> word.contains("#")) // Filter words
containing '#'
                .forEach(System.out::println); // Print each word
```

-------------------------------------Wissen Technology----------------------------------

Q1) Consider a database schema for an online bookstore with the following tables. Write a SQL query to find the top 5 bestselling books (based on the total number of copies sold) along with their titles and the total number of copies sold for each book.

Books: book_id title author genre price

Orders: order_id customer_id book_id quantity order_date

Sol:-

select b.title, sum(o.quantity) as total_copies_sold from Books b

inner join Orders o on b.book_id = o.book_id

GROUP BY b.book_id, b.title ORDER BY total_copies_sold DESC LIMIT 5;

Q2) What is the ouput of this?

```
public class TestCollection {

 public static void main(String args[]) {

  PriorityQueue < String > queue = new PriorityQueue < String > ();

  System.out.println("add Operation in Queue: " + queue.add("Kumar"));

  System.out.println("add Operation in Queue: " + queue.add("Hitesh"));

  System.out.println("Elements in Queue: " + queue);

  System.out.println("element Operation in Queue: " + queue.element());

  System.out.println("Elements in Queue: " + queue);

  System.out.println("peek Operation in Queue: " + queue.peek());
```

```
    System.out.println("Elements in Queue: " + queue);
    System.out.println("offer Operation in Queue: " + queue.offer("Manoj"));
    System.out.println("Elements in Queue: " + queue);
    System.out.println("remove Operation in Queue: " + queue.remove());
    System.out.println("Elements in Queue: " + queue);
    System.out.println("poll Operation in Queue: " + queue.poll());
    System.out.println("Elements in Queue: " + queue);
  }
}
```

Ans:-

```
add Operation in Queue: true
add Operation in Queue: true
Elements in Queue: [Hitesh, Kumar]
element Operation in Queue: Hitesh
Elements in Queue: [Hitesh, Kumar]
peek Operation in Queue: Hitesh
Elements in Queue: [Hitesh, Kumar]
offer Operation in Queue: true
Elements in Queue: [Hitesh, Kumar, Manoj]
remove Operation in Queue: Hitesh
Elements in Queue: [Kumar, Manoj]
poll Operation in Queue: Kumar
Elements in Queue: [Manoj]
```

Explanation and Expected Output:

1. **Creating PriorityQueue and Adding Elements:**
   - queue.add("Kumar"): Adds "Kumar" to the queue. Returns true indicating the operation was successful.
   - Output: add Operation in Queue: true
   - queue.add("Hitesh"): Adds "Hitesh" to the queue. Returns true.
   - Output: add Operation in Queue: true
   - **Elements in Queue:** At this point, the queue contains two elements. The exact order might not be predictable without knowing the internal workings of PriorityQueue, but let's assume it could be ["Hitesh", "Kumar"] due to alphabetical order or other factors.
2. **element Operation:**
   - queue.element(): Retrieves, but does not remove, the head of the queue. Throws NoSuchElementException if the queue is empty.
   - Output: Since the queue is not empty, this would print something like element Operation in Queue: Hitesh.
3. **peek Operation:**
   - queue.peek(): Retrieves, but does not remove, the head of the queue. Returns null if the queue is empty.
   - Output: Again, assuming the queue is ["Hitesh", "Kumar"], this would print peek Operation in Queue: Hitesh.
4. **offer Operation:**
   - queue.offer("Manoj"): Inserts "Manoj" into the queue. Returns true.
   - Output: offer Operation in Queue: true. Now the queue might be ["Hitesh", "Kumar", "Manoj"].
5. **remove Operation:**
   - queue.remove(): Removes and returns the head of the queue. Throws NoSuchElementException if the queue is empty.
   - Output: Assuming ["Hitesh", "Kumar", "Manoj"], this would print remove Operation in Queue: Hitesh.
6. **poll Operation:**
   - queue.poll(): Removes and returns the head of the queue. Returns null if the queue is empty.
   - Output: Assuming ["Kumar", "Manoj"] after the previous operations, this would print poll Operation in Queue: Kumar.

Q3)What is the output of below code?

```
class VehicleException extends Exception {
  public VehicleException(String str) {
    System.out.println("Thrown Vehicle Exception For : " + str);
  }
}
class CarException extends VehicleException {
  public CarException(String str) {
    super(str);
    System.out.println("Thrown Car Exception For : " + str);
  }
}
class HatchbackException extends CarException {
  public HatchbackException() {
    super("Hatchback");
  }
}
class SUVException extends CarException {
  public SUVException() {
    super("SUV");
  }
}
public class TestException {
  public void testCar() throws CarException {
    testHatchback();
    testSUV();
  }
  public void testHatchback() throws VehicleException {
    throw new VehicleException("Hatchback");
  }
  public void testSUV() throws SUVException {
    throw new SUVException();
  }
  public static void main(String[] args) {
    TestException testException = new TestException();
    testException.testCar();
  }
}
```

Ans:- It will give compiler error because in testCar() method for testHatchback() method we are not throwing VehicleException. And according to code VehicleException is the super class of CarException Also.

And for testException.testCar(); method we need to handle with try catch.

Q4)What is the output of below program?

```
class RunnableTask1 implements Runnable {
  private String resource1, resource2;
  public RunnableTask1(String resource1, String resource2) {
    this.resource1 = resource1;
    this.resource2 = resource2;
  }
  @Override public void run() {
    synchronized(resource1) {
      System.out.println("Task 1: Printing : " + resource1);
      try {
        Thread.sleep(100);
      } catch (Exception e) {}
```

```
        synchronized(resource2) {
          System.out.println("Task 1: Printing : " + resource2);
        }
      }
    }
  }
  class RunnableTask2 implements Runnable {
    private String resource1, resource2;
    public RunnableTask2(String resource1, String resource2) {
      this.resource1 = resource1;
      this.resource2 = resource2;
    }
    @Override public void run() {
      synchronized(resource2) {
        System.out.println("Task 2: Printing : " + resource2);
        try {
          Thread.sleep(100);
        } catch (Exception e) {}
        synchronized(resource1) {
          System.out.println("Task 2: Printing : " + resource1);
        }
      }
    }
  }
  public class ThreadTest {
    public static void main(String[] args) {
      final String resource1 = "Test1";
      final String resource2 = "Test2";
      Thread t1 = new Thread(new RunnableTask1(resource1, resource2));
      Thread t2 = new Thread(new RunnableTask2(resource1, resource2));
      t1.start();
      t2.start();
    }
  }
```

Ans:-

Task 2: Printing : Test2

Task 1: Printing : Test1

Q5) First Missing Positive:

Given an unsorted integer array numbers, return the smallest missing positive integer.

You must implement an algorithm that runs in O(n) time and uses O(1) auxiliary space.

Example 1:

Input: nums = [1,2,0]

Output: 3

Explanation: The numbers in the range [1,2] are all in the array.

Example 2:

Input: nums = [3,4,-1,1]

Output: 2

Explanation: 1 is in the array but 2 is missing.

Example 3:

Input: nums = [7,8,9,11,12]

Output: 1

Explanation: The smallest positive integer 1 is missing.

Constraints:

1 <= nums.length <= 105

-231 <= nums[i] <= 231 – 1

Ans:-

```
import java.util.Arrays;

public class Main
{
	public static void main(String[] args) {
		int[] input = new int[]{1,2,4};
		//Expected ans : 2
		int firstMissingPositive = getFirstMissingPositive(input);
```

```
            System.out.println(firstMissingPositive);
        }

        static int getFirstMissingPositive(int[] input){
            Arrays.sort(input);
          int ans = 1;
          for (int i = 0; i < input.length; i++) {
              if (input[i] == ans)
                  ans++;
          }
            return ans;
        }
}
```

Follow the steps below to solve the problem:

- First sort the array and the smallest positive integer is 1.
- So, take ans=1 and iterate over the array once and check whether **arr[i] = ans** (Checking for value from 1 up to the missing number).
- By iterating if that condition meets where **arr[i] = ans** then increment ans by 1 and again check for the same condition until the size of the array.
- After one scan of the array, the missing number is stored in **ans** variable.
- Now return that **ans** to the function.

Time Complexity: O(N*log(N)), Time required to sort the array
Auxiliary Space: O(1)

```
static int getFirstMissingPositive(int[] input){

        Arrays.sort(input);

        if(input[0]>1){

            return 1;

        }

        else{

            int count = 0;

            for(int i=0;i<input.length;i++){
```

```
            if(input[i]==1){
                count = 1;
            }
            if(input[i] != count && input[i]>0){
                return count;
            }
            count++;
        }
    }
    return 0;
}
```

Q6) time complexity and space complexity in java?

Q7)What is package.json in angular?

the package.json file in Angular projects serves as a central configuration file that manages dependencies, scripts, and metadata. It's essential for setting up, building, and managing your Angular application throughout its lifecycle.

package.json file locates in project root and contains information about your web application. The main purpose of the file comes from its name *package*, so it'll contain the information about npm packages installed for the project.

In an Angular app, package.json is a file that contains information about the project and its dependencies. It is a part of the Node.js ecosystem and is used by the npm (Node Package Manager) to manage the packages required by the project.

The package.json is an npm configuration file that lists the third-party packages that your project depends on.

Q8) what is currying in javascript?

Currying is defined as changing a function having multiple arguments into a sequence of functions with a single argument. It is a process of converting a function with more arity into a function having less arity. The term arity means the number of parameters in a function.

https://www.javatpoint.com/currying-in-javascript#:~:text=Currying%20is%20defined%20as%20changing,of%20parameters%20in%20a%20function.

Q9) what is event looping in javascript?

event looping allows JavaScript to handle asynchronous operations efficiently without blocking the main execution thread. This is crucial for building responsive web applications and managing tasks that may take varying amounts of time to complete.

https://medium.com/@kamaleshs48/event-loop-in-javascript-c332b0f81b1e

---------------------------------MPC Cloud Computing---------------------------------

Q1) What is the output of below code?

```
public class Parent {

}
public class Child extends Parent{

	public void display() {
		System.out.println("Hello");
	}
	public static void main(String[] args) {
		Parent p = new Child();
		p.display();
	}
}
```

Ans:- It will give compile error because display method is not present in parent class.

--------------------------------Persistent---------------------------------

Q1) i have a product in cart and it is waiting sometime. another user try to purchase the same product fast. we have only one product. only one user can purchase this order how to write code for only single user thread?

To handle the scenario where only one user can successfully purchase a product when there is only one available, you typically need to manage this at the application level using locking mechanisms. Here’s a conceptual approach to implement this in a multi-threaded environment, assuming you have multiple users concurrently interacting with your application:

```
import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;



public class ProductPurchaseManager {

    private boolean productAvailable = true;

    private Lock purchaseLock = new ReentrantLock();



    public boolean attemptPurchase() {

        if (productAvailable) {

            if (purchaseLock.tryLock()) {

                try {

                    // Perform purchase operation (e.g., mark product as sold, update inventory)

                    productAvailable = false;

                    return true; // Purchase successful

                } finally {

                    purchaseLock.unlock();
                }
            } else {
                // Another user is currently purchasing, handle accordingly
                return false; // Purchase unsuccessful
            }
        } else {
            // Product is not available (already purchased), handle accordingly
            return false; // Purchase unsuccessful
        }
    }
}
```

Explanation:

1. **Lock Initialization**:
   - purchaseLock is initialized as a ReentrantLock. This lock will be used to synchronize access to the critical section where the purchase operation occurs.
2. **Attempt Purchase Method (attemptPurchase())**:
   - Checks if the product is available (productAvailable).
   - If available, it attempts to acquire the purchaseLock using tryLock(). This method tries to acquire the lock but does not block if the lock is not available immediately.
   - If tryLock() returns true, it means the current thread successfully acquired the lock and can proceed with the purchase operation:
     - Mark the product as sold (productAvailable = false).
     - Perform any necessary operations like updating inventory or marking the product as purchased.
     - Finally, unlock the purchaseLock using unlock().
     - Return true to indicate that the purchase was successful.
   - If tryLock() returns false, another user is currently purchasing the product. In this case, return false to indicate that the purchase attempt was unsuccessful due to concurrency.

  - If the product is not available (productAvailable is false), return false to indicate that the purchase attempt was unsuccessful because the product is no longer available (already purchased).

Usage:

- Each user interaction that attempts to purchase the product should call the attemptPurchase() method of ProductPurchaseManager.
- Depending on the returned value (true or false), you can inform the user about the success or failure of their purchase attempt.

Considerations:

- Ensure that purchaseLock is managed consistently across your application to prevent deadlocks or improper synchronization.
- Handle edge cases such as timeout for lock acquisition (tryLock() with a timeout), and manage product availability updates appropriately (e.g., if a user cancels their purchase or the session expires).

This approach ensures that only one user can successfully purchase the product at any given time, respecting the availability and preventing simultaneous purchases of the same product.

Q2) int a[] = {1,5,4,3,2}; if I need to filter million of records?

Arrays.*stream*(a).parallel().filter(p -> p==2).forEach(System.*out*::println);

-------------------------------------Wissen Technology---------------------------------

Q1) Given a sorted array arr[] of size N, the task is to remove the duplicate elements from the array.

Examples:

Input: arr[] = {2, 2}

Output: arr[] = {2,null}

Explanation: All the elements are 2, So only keep one instance of 2.

Input: arr[] = {1, 2, 2, 3, 4, 4 }

Output: arr[] = {1, 2, 3, 4,null,null}

Ans:-

Company Interview Question

```
import java.util.*;
public class RemoveDuplicatesFromSortedArray
{
      public static void main(String[] args) {
            int arr[] = { 1, 2, 2, 3, 4, 4, 4, 5, 5 };
        int n = arr.length;

        // removeDuplicates() returns new size of array
        n = removeDuplicates(arr, n);

        // Print updated array
        for (int i = 0; i < n; i++)
            System.out.print(arr[i] + " ");
      }
      static int removeDuplicates(int arr[], int n){

          int rd =0;
          for (int i = 1; i < n; i++) {
                  if(arr[rd] != arr[i]) {
                        rd++;
                        arr[rd]=arr[i];
                  }
            }
          return rd+1;

      }


}
```

Q2) Write a Java program to print a sequence of numbers upto N using 3 threads. For example, if we want to print a sequence of numbers upto 10 then it'll look like this:

THREAD-1 : 1
THREAD-2 : 2
THREAD-3 : 3
THREAD-1 : 4
THREAD-2 : 5
THREAD-3 : 6
THREAD-1 : 7
THREAD-2 : 8
THREAD-3 : 9
THREAD-1 : 10

So, here we can see that Thread-1 is printing all those numbers which when divided by number of threads (i.e. 3) leaves remainder as 1 e.g. 1, 4, 7. Thread-2 is printing all those numbers which when divided by number of threads (i.e. 3) leaves remainder as 2 e.g. 2, 5, 8. Thread-3 is printing all those numbers which when divided by number of threads (i.e. 3) leaves remainder as 0 e.g. 3, 6, 9. We'll use this logic to code our solution.

Solution

First, let's create a resource class NumbersGenerator that'll print the numbers and will be shared by all the threads.

```
/*
   - Resource class to print numbers
   - Shared by multiple threads
 */
public class NumbersGenerator {
   private int currNumber = 1;
   private int numOfThreads;
   private int totalNumbersInSeq;

   public NumbersGenerator(final int numOfThreads, final int totalNumbersInSeq) {
      this.numOfThreads = numOfThreads;
      this.totalNumbersInSeq = totalNumbersInSeq;
   }

   public void printNumber(int index) {
      synchronized (this) {
         while(currNumber < totalNumbersInSeq-1) {
            while (currNumber % numOfThreads != index) {
               try {
                  wait();
               } catch (InterruptedException e) {
                  e.printStackTrace();
               }
            }

            System.out.println(Thread.currentThread().getName() + " : " +
currNumber++);
            notifyAll();
         }
      }
   }
}
```

Since, printNumber() will be called by multiple threads we must make its logic synchronized on the current class object which is shared by all the threads. wait() , notify() and notifyAll() methods are final and are present in the Object class. They are used for inter-thread communication and must be used inside a synchronized block/method.

wait() makes the current thread release lock on the synchronized code and go to sleep until some other thread wakes it up by calling notify() or notifyAll() . notify() wakes up a single thread waiting for the Object but it doesn't actually releases the lock. notifyAll() wakes up all the threads but it doesn't actually releases the lock.

Now, let's write our own thread class that implements runnable. We'll use this class to create our threads.

```
public class SequenceGenerator implements Runnable {

   private NumbersGenerator numbersGenerator;
   private int index;
```

```
    public SequenceGenerator(NumbersGenerator numbersGenerator, int index) {
        this.numbersGenerator = numbersGenerator;
        this.index = index;
    }

    @Override
    public void run() {
        numbersGenerator.printNumber(this.index);
    }
}
```

Finally, let's test our code:

```
public class SequenceGeneratorTester {
    private static final int NUMBER_OF_THREADS = 3;
    private static final int TOTAL_NUMBERS_IN_SEQ = 10;

    public static void main(String[] args) {
        NumbersGenerator numbersGenerator = new
NumbersGenerator(NUMBER_OF_THREADS, TOTAL_NUMBERS_IN_SEQ);

        Thread t1 = new Thread(new SequenceGenerator(numbersGenerator, 1),
"THREAD-1");
        Thread t2 = new Thread(new SequenceGenerator(numbersGenerator, 2),
"THREAD-2");
        Thread t3 = new Thread(new SequenceGenerator(numbersGenerator, 0),
"THREAD-3");

        t1.start();
        t2.start();
        t3.start();
    }
}
```

Output

```
THREAD-1 : 1
THREAD-2 : 2
THREAD-3 : 3
THREAD-1 : 4
THREAD-2 : 5
THREAD-3 : 6
THREAD-1 : 7
THREAD-2 : 8
THREAD-3 : 9
THREAD-1 : 10
```

Q3) Print student name total marks in descending order?

Table : Student

Column : Name, Subject, Marks

Rama Math 50

Rama Physics 60

Rama English 45

Hari Math 70

Hari Physics 65

Hari English 85

Gita Math 90

Gita Physics 55

Gita English 80

Expected Output:

Gita 225

Hari 220

Rama 155

Ans:-

Select name, Sum(Marks) as totalMarks from Student group by name order by totalMarks desc;

Q4)What is the output of below code?

```
import java.util.HashSet;

class Student {
    public int id;
    public String name;
    public Student(String name, int id) {
        this.name = name;
        this.id = id;
    }
    public int hashCode() {
        return this.id;
    }
    public String toString() {
        return "Student: " + this.name + "@" +
Integer.toHexString(hashCode());
    }
    public boolean equals(Object o) {
        if (o instanceof Student) {
           Student s = (Student) o;
           return s.id == this.id ? true : false;
        }
        return false;
    }
}
public class UpdateHashSet {
    public static void main(String[] args) {
        HashSet<Student> studentList = new HashSet<>();
        Student st1 = new Student("Nimit", 1);
        Student st2 = new Student("Rahul", 3);
        Student st3 = new Student("Nimit", 2);
        studentList.add(st1);
        studentList.add(st2);
        studentList.add(st3);
        System.out.println(studentList.size());//guess the output here
        st1.id = 3;
        System.out.println(studentList.size());//guess the output here
    }
}
```

Output:-

3

3

Q5) best way to read data from custom date range from millions of records? Which one collection is better?

We have a csv file

student_id, student_name, student_dob

1, Bob, 2020-01-05

2, Alice, 2020-01-31

Ans:- tree set but small doubt

Q6) I have to fetch the data from a millions of record. what is the best way to fetch data through id ? which collection is better?

Ans:- LinkHashmap

Q7) What is Thread dump in java?

In Java, a thread dump is a powerful diagnostic tool for understanding the runtime behavior of threads within a JVM. It provides essential information about thread states, stack traces, and thread interactions, helping developers and administrators troubleshoot and resolve threading-related issues effectively.

Q8) What is the difference between Java Callable interface and Runnable interface?

The Callable interface and Runnable interface both are used by the classes which wanted to execute with multiple threads. However, there are two main differences between the both :

- A Callable <V> interface can return a result, whereas the Runnable interface cannot return any result.
- A Callable <V> interface can throw a checked exception, whereas the Runnable interface cannot throw checked exception.
- A Callable <V> interface cannot be used before the Java 5 whereas the Runnable interface can be used.

Q10) Explain the ExecutorService Interface.

The ExecutorService Interface is the subinterface of Executor interface and adds the features to manage the lifecycle. Consider the following example.

```

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class TestThread {
  public static void main(final String[] arguments) throws InterruptedException
  {
    ExecutorService e = Executors.newSingleThreadExecutor();

    try {
      e.submit(new Thread());
      System.out.println("Shutdown executor");
      e.shutdown();
      e.awaitTermination(5, TimeUnit.SECONDS);
    } catch (InterruptedException ex) {
      System.err.println("tasks interrupted");
    } finally {

      if (!e.isTerminated()) {
        System.err.println("cancel non-finished tasks");
      }
      e.shutdownNow();
      System.out.println("shutdown finished");
    }
  }

  static class Task implements Runnable {

    public void run() {

      try {
        Long duration = (long) (Math.random() * 20);
        System.out.println("Running Task!");
        TimeUnit.SECONDS.sleep(duration);
      } catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

Output

```
Shutdown executor
shutdown finished
```

---------------------------------NeoSoft Pvt Ltd----------------------------------

Q1) Convert into one to many example?

```
import java.util.HashSet;
import java.util.Set;

import jakarta.persistence.CascadeType;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.OneToMany;
import lombok.Getter;
import lombok.Setter;
```

```
@Getter @Setter
@Entity
public class Employee {
      @Id
      @GeneratedValue(strategy = GenerationType.IDENTITY)
      int id;
      @Column
      String name;
      String designation;
      // one to many unidirectional mapping
    // default fetch type for OneToMany: LAZY
      @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
      @JoinColumn(name = "emp_id", referencedColumnName = "id")
      private Set<Address> addresses = new HashSet<>();

}


import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import lombok.Getter;
import lombok.Setter;

@Getter     @Setter
@Entity
public class Address {

      @Id
      @GeneratedValue(strategy = GenerationType.IDENTITY)
      int id;
      String streetName;
      String area;
      String district;
      int pinCode;
      String type;

}
```

Q2) Filter the data according to district?

```
List<Employee> filteredParents = employees.stream()
                  .filter(employee -> employee.getAddresses().stream()
                      .anyMatch(adr ->
"Sangli".equalsIgnoreCase(adr.getDistrict())))
                  .collect(Collectors.toList());
```

Q3) What is Spring Security?

Ans: Spring security is a powerful access control framework. It aims at providing authentication and authorization to java applications. It enables the developer to impose security restrictions to save from common attacks.

Company Interview Question

---------------------------------NTT DATA----------------------------------

Q1) write a javascript function to call alert popup on click of button after 10 seconds?

```
<button type="button" onclick="showAlertAfterDelay()">Click Me!</button>
 <script>
 function showAlertAfterDelay() {
    setTimeout(function() {
        alert('This is a delayed alert!');
    }, 10000); // 10000 milliseconds = 10 seconds
}
</script>
```

Q2) How to fetch Second highest salary record through my sql?

Select max(salary) From Employee where salary not in (Select max(salary) From Employee);

Select max(salary) From Employee where salary < (Select max(salary) From Employee);

Q3) Write a program to find first nonrepeating character in the given string using java8?

```
		String str= "aabbccdeeffgghhi";
		String nonrepeatingChar = 
Arrays.stream(str.split("")).map(String::toLowerCase).collect(Collectors.gr
oupingBy(s -> s, LinkedHashMap::new, 
Collectors.counting())).entrySet().stream()
	        .filter(entry -> entry.getValue() == 1)
	        .map(Map.Entry::getKey)
	        .findFirst()
	        .orElseThrow(() -> new RuntimeException("No non-repeating character 
found"));
		System.out.println(nonrepeatingChar);
```

Q4)What is the output of this?

```
String s1 = "India";
s1 = s1.concat("is my country");
```

Ans:- **India is my country**

Q5) final vs static?

The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. If you make any variable as final, you cannot change the value of final variable(It will be constant).
2. If you make any method as final, you cannot override it.
3. If you make any class as final, you cannot extend it.

The static keyword in Java is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class than an instance of the class.

Java static variable

If you declare any variable as static, it is known as a static variable.

- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Q6) How HashMap works internally?/Working of HashMap in java?

What is Hashing

It is the process of converting an object into an integer value. The integer value helps in indexing and faster searches.

What is HashMap

HashMap is a part of the Java collection framework. It uses a technique called Hashing. It implements the map interface. It stores the data in the pair of Key and Value. HashMap contains an array of the nodes, and the node is represented as a class. It uses an array and LinkedList data structure internally for storing Key and Value. There are four fields in HashMap.

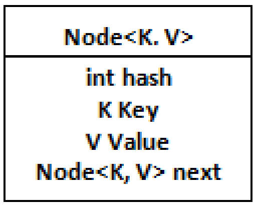


Figure: Representation of a Node

Before understanding the internal working of HashMap, you must be aware of hashCode() and equals() method.

- **equals():** It checks the equality of two objects. It compares the Key, whether they are equal or not. It is a method of the Object class. It can be overridden. If you override the equals() method, then it is mandatory to override the hashCode() method.
- **hashCode():** This is the method of the object class. It returns the memory reference of the object in integer form. The value received from the method is used as the bucket number. The bucket number is the address of the element inside the map. Hash code of null Key is 0.
- **Buckets:** Array of the node is called buckets. Each node has a data structure like a LinkedList. More than one node can share the same bucket. It may be different in capacity.

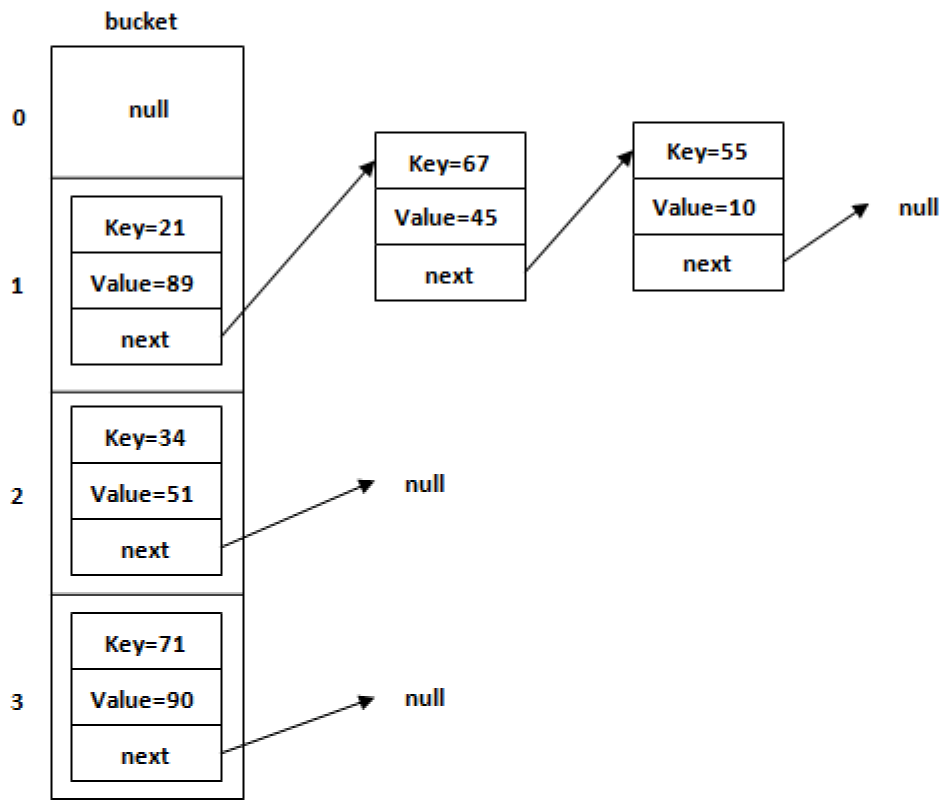


Figure: Allocation of nodes in Bucket

Insert Key, Value pair in HashMap

We use put() method to insert the Key and Value pair in the HashMap. The default size of HashMap is 16 (0 to 15).

Example

In the following example, we want to insert three (Key, Value) pair in the HashMap.

```
9.  HashMap<String, Integer> map = new HashMap<>();
10. map.put("Aman", 19);
11. map.put("Sunny", 29);
12. map.put("Ritesh", 39);
```

Let's see at which index the Key, value pair will be saved into HashMap. When we call the put() method, then it calculates the hash code of the Key "Aman." Suppose the hash code of "Aman" is 2657860. To store the Key in memory, we have to calculate the index.

Calculating Index

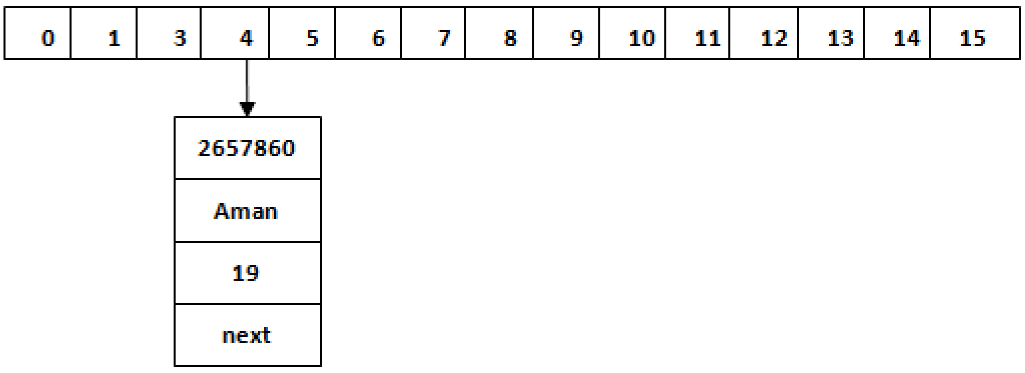
Index minimizes the size of the array. The Formula for calculating the index is:

3. Index = hashcode(Key) & (n-1)

Where n is the size of the array. Hence the index value for "Aman" is:

3. Index = 2657860 & (16-1) = 4

The value 4 is the computed index value where the Key and value will store in HashMap.

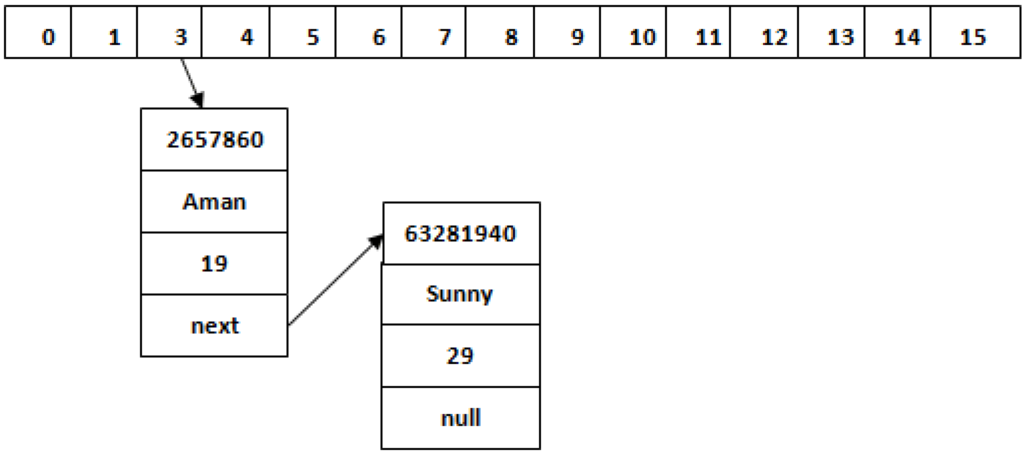


Hash Collision

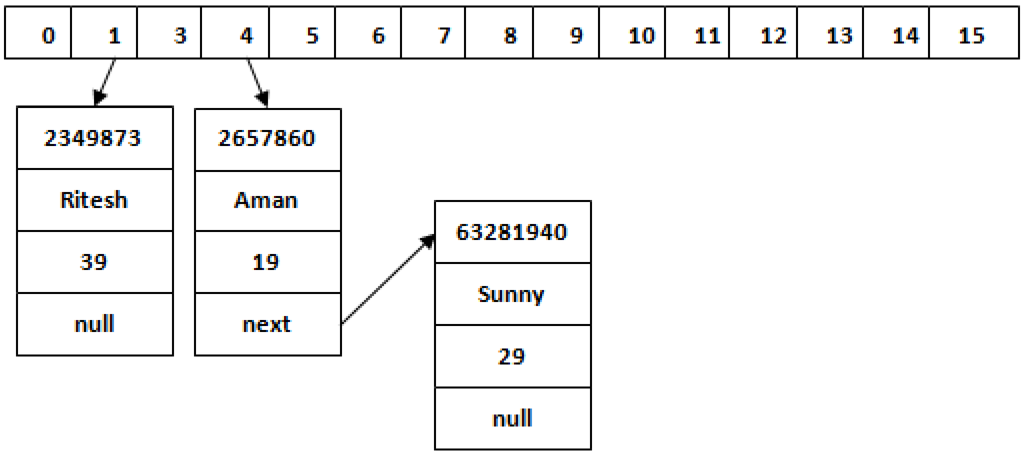
This is the case when the calculated index value is the same for two or more Keys. Let's calculate the hash code for another Key "Sunny." Suppose the hash code for "Sunny" is 63281940. To store the Key in the memory, we have to calculate index by using the index formula.

3. Index=63281940 & (16-1) = 4

The value 4 is the computed index value where the Key will be stored in HashMap. In this case, equals() method check that both Keys are equal or not. If Keys are same, replace the value with the current value. Otherwise, connect this node object to the existing node object through the LinkedList. Hence both Keys will be stored at index 4.



Similarly, we will store the Key "Ritesh." Suppose hash code for the Key is 2349873. The index value will be 1. Hence this Key will be stored at index 1.



get() method in HashMap

get() method is used to get the value by its Key. It will not fetch the value if you don't know the Key. When get(K Key) method is called, it calculates the hash code of the Key.

Suppose we have to fetch the Key "Aman." The following method will be called.

3. map.get(**new** Key("Aman"));

It generates the hash code as 2657860. Now calculate the index value of 2657860 by using index formula. The index value will be 4, as we have calculated above. get() method search for the index value 4. It compares the first element Key with the given Key. If both keys are equal, then it returns the value else check for the next element

in the node if it exists. In our scenario, it is found as the first element of the node and return the value 19.

Let's fetch another Key "Sunny."

The hash code of the Key "Sunny" is 63281940. The calculated index value of 63281940 is 4, as we have calculated for put() method. Go to index 4 of the array and compare the first element's Key with the given Key. It also compares Keys. In our scenario, the given Key is the second element, and the next of the node is null. It compares the second element Key with the specified Key and returns the value 29. It returns null if the next of the node is null.

-------------------------------------Wissen Technology---------------------------------

Q1) how to implement the method till the attempts is not completed

```
public class Main
{
      public static void main(String[] args) {
            //test 1
            try {
            String test = Retry.run(3, () -> {
                if (Math.random() > 0.5) {
                    return "random success";
                } else {
                    throw new Exception ("Random Failure");
                }
      });
            } catch (Exception e) {
                System.out.println("Error : " + e);
            }
      }
}
class Retry <T>{
       public static <T> T run(int noOfAttempts, FunctionThatCanFail <T> 
retriable) throws Exception {


            return null;//TODO -- IMPLEMENT THIS FUNCTIONALITY
      }
}
@FunctionalInterface
interface FunctionThatCanFail<T> {
      T call() throws Exception;
}
```

Solution:-

To implement the Retry.run method to retry a given task a specified number of times, you'll need to handle the retries and exceptions properly. Here's how you can do it:

Company Interview Question

1. **Initialize the Attempt Counter**: Start with the initial number of attempts.
2. **Try Executing the Task**: Attempt to execute the task inside a loop.
3. **Handle Success**: If the task succeeds, return the result immediately.
4. **Handle Failure**: If the task fails, catch the exception and decrement the attempt counter.
5. **Throw Exception on Exhaustion**: If all attempts are exhausted and the task still fails, throw the exception.

```java
class Retry<T> {
    public static <T> T run(int noOfAttempts, FunctionThatCanFail<T> retriable) throws Exception {

            Exception lastException = null; // To hold the last exception encountered
                for (int attempt = 1; attempt <= noOfAttempts; attempt++) {
                    try {
                        // Attempt to run the task
                        return retriable.call();
                    } catch (Exception e) {
                        lastException = e; // Store the exception for the final throw
                        if (attempt == noOfAttempts) {
                            // If this was the last attempt, rethrow the exception
                            throw e;
                        }
                        // Optionally, you could add a delay between attempts here
                    }
                }
                // In practice, this point should never be reached
                throw lastException;
            }

    }
```

Explanation:

1. **Loop through Attempts**: The loop iterates up to noOfAttempts times.
2. **Try-Catch Block**: Inside the loop, attempt to execute the retriable.call(). If it succeeds, return the result immediately.
3. **Catch Exceptions**: If an exception is thrown, catch it and update lastException. If this was the last attempt, throw the exception. Otherwise, proceed to the next attempt.
4. **Throw Final Exception**: If the loop completes without a successful call, throw the last encountered exception.

Notes:

- **Exception Handling**: The loop is designed to keep track of the last exception encountered, which will be rethrown if all attempts fail.
- **Optional Delay**: You might want to add a delay between attempts to avoid immediate retries, which can be done using Thread.sleep() within the catch block if needed.

With this implementation, you have a reusable retry mechanism for any task that might fail intermittently.

Q2) I have a millions of file and i need to take latest file and also i need to check in how many files content is same. if same i need to take latest file. what is the best approach in java?

SoL:-

Handling millions of files and performing operations like identifying the latest file and checking for duplicate content can be quite challenging. Below is a structured approach to achieve this efficiently in Java:

Approach

1. **List All Files:**
   - Use Java's java.nio.file package to list all files in the directory.
2. **Sort Files by Modification Time:**
   - Maintain a data structure to store the files and their last modification times.
   - Use a TreeMap or PriorityQueue to keep track of the latest files efficiently.
3. **Hash File Contents:**
   - Use hashing (e.g., MD5, SHA-256) to identify files with the same content. This involves reading the file contents and generating a hash value.
4. **Check for Duplicates:**
   - Maintain a map to keep track of hashes and their latest file. This helps in quickly identifying duplicates.
5. **Optimize I/O Operations:**
   - Given the volume, consider optimizing I/O operations by using buffering and minimizing disk access.

Sample Code

Here is a Java code snippet demonstrating this approach:

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.HashMap;
```

Company Interview Question

```
import java.util.Map;
import java.util.TreeMap;

public class FileProcessor {

    public static void main(String[] args) {
        File directory = new File("D:\\Java Interview\\Self\\Docx\\");
        Map<String, File> latestFilesMap = new TreeMap<>(); // Key: File
hash, Value: Latest file
        Map<String, File> fileModificationMap = new HashMap<>(); // Key:
File hash, Value: File object

        try {
            File[] files = directory.listFiles();
            if (files != null) {
                for (File file : files) {
                    if (file.isFile()) {
                        String fileHash = getFileChecksum(file);
                        if (fileModificationMap.containsKey(fileHash)) {
                            File existingFile =
fileModificationMap.get(fileHash);
                            if (file.lastModified() >
existingFile.lastModified()) {
                                fileModificationMap.put(fileHash, file);
                            }
                        } else {
                            fileModificationMap.put(fileHash, file);
                        }
                    }
                }

                // Collect the latest files
                for (File file : fileModificationMap.values()) {
                    latestFilesMap.put(getFileChecksum(file), file);
                }

                // Output results
                for (Map.Entry<String, File> entry :
latestFilesMap.entrySet()) {
                    System.out.println("Hash: " + entry.getKey() + " |
File: " + entry.getValue().getAbsolutePath());
                }

            } else {
                System.out.println("Directory does not exist or is
empty.");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static String getFileChecksum(File file) throws
NoSuchAlgorithmException, IOException {
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
        try (FileInputStream fis = new FileInputStream(file)) {
            byte[] buffer = new byte[1024];
            int bytesRead;
            while ((bytesRead = fis.read(buffer)) != -1) {
                digest.update(buffer, 0, bytesRead);
            }
        }
        byte[] hashBytes = digest.digest();
        StringBuilder sb = new StringBuilder();
        for (byte b : hashBytes) {
            sb.append(String.format("%02x", b));
        }
        return sb.toString();
    }
}
```

Explanation

1. **File Listing and Processing:**
   - Lists all files in the specified directory.
   - For each file, calculates its SHA-256 hash using MessageDigest.
2. **Tracking Latest Files:**
   - Uses a HashMap to keep track of the latest file for each unique hash.
   - Updates the map if a file with the same hash but a newer timestamp is found.
3. **Handling Large Volumes:**
   - This approach is efficient but can be optimized further based on specific requirements and system capabilities.

Considerations

- **Performance:** Processing millions of files might require efficient file reading, caching strategies, or parallel processing.
- **Memory Usage:** Ensure sufficient memory allocation, especially if the data size is large.
- **Error Handling:** Implement comprehensive error handling to deal with file access issues, hash computation errors, etc.

This approach should give you a good starting point for managing and processing a large number of files in Java.

------------------------------------NTT DATA SERVICES---------------------------------

Q1) How to get third highest salary of employee?

```
Optional<Employee> first =
employeeList.stream().sorted(Comparator.comparing(Employee::getSalary,Colle
ctions.reverseOrder())).skip(2).findFirst();
```

Q2) How to get highest salary of every department?

```
Map<String,Optional<Employee>>         highestSalariesByDepartment        =
employeeList.stream().collect(Collectors.groupingBy(e-
>e.getDepartment(),Collectors.maxBy(Comparator.comparing(Employee::getSalar
y))));
```

Q3) Output of below java code?

```
public class foo {
	static {
		System.out.println("Static Block-1");
	}

	public static void main(String args[]) {
		System.out.println("Main Method");
	}

	static {
		System.out.println("Static Block-2");
	}
}
```

```
Ans:-

Static Block-1

Static Block-2

Main Method
```

Q4) what is executor service?

The Java ExecutorService is the interface which allows us to execute tasks on threads asynchronously. The Java ExecutorService interface is present in the java.util.concurrent package. The ExecutorService helps in maintaining a pool of threads and assigns them tasks. It also provides the facility to queue up tasks until there is a free thread available if the number of tasks is more than the threads available.

https://www.javatpoint.com/java-executorservice

Q5) What Are Atomic Variables?

an atomic variable is a type of variable that provide a lock-free and thread-safe environment or programming on a single variable.

Atomic variables are part of the java.util.concurrent.atomic package.

It also supports atomic operations. All the atomic classes have the get() and set() methods that work on the volatile variable. The method works the same as read and writes on volatile variables.

There is a branch of research focused on creating non-blocking algorithms for concurrent environments. These algorithms exploit low-level atomic machine instructions such as compare-and-swap (CAS), to ensure data integrity.

A typical CAS operation works on three operands:

1. The memory location on which to operate (M)
2. The existing expected value (A) of the variable
3. The new value (B) which needs to be set

The CAS operation updates atomically the value in M to B, but only if the existing value in M matches A, otherwise no action is taken.

In both cases, the existing value in M is returned. This combines three steps – getting the value, comparing the value, and updating the value – into a single machine level operation.

https://www.javatpoint.com/java-atomic

https://www.baeldung.com/java-atomic-variables

Q6) what is exceptional handling?

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc. so that the normal execution flow of the program can be maintained.

Q7) what is basic response type of spring boot?

JSON

Q8) new memory management in java 8?

PermGen is known as the memory space which is used for collecting class data like byte code, static variables, and many more. The space allocated to PermGen is by default 64Mb. The PermGen can also be attuned with the help of -XXMaxPermSize.

Company Interview Question

In Java SE 8, the PermGen process area was restored with MetaSpace. Java developers have transferred permGem to the unique consciousness in the original OS, which is now commonly known as MetaSpace. MetaSpace can auto-enhance its capacity, by default. In MetaSpace, the classes can store and discharge during the total life sentence of the Java Virtual Machine (JVM).