**Interview question**

**Question 1 :-**

**Find first non repeated character using java8**

**public** **static** **void** main(String[] args) {

String name="saurabh";

Character character = name.chars()

.mapToObj(val->(**char**)val)

.collect(Collectors.*groupingBy*(val->val,LinkedHashMap::**new** ,Collectors.*counting*()))

.entrySet().stream().filter(entry->entry.getValue()==1).map(entry->entry.getKey())

.findFirst().get();

System.***out***.println("first non repeated character "+character);

}

}

**Question 2**

**Find character count of each element**

**public** **class** test1 {

**public** **static** **void** main(String[] args) {

String name = "saurabh";

Map<Character, Long> charCount = name.chars().mapToObj(val -> (**char**) val)

.collect(Collectors.*groupingBy*(val -> val, Collectors.*counting*()));

System.***out***.println(charCount);

}

}

Question 3)

Find duplicate character from string

**public** **class** test1 {

**public** **static** **void** main(String[] args) {

String name = "saurabh";

List<Character> dupValue = name.chars().mapToObj(val -> (**char**) val)

.collect(Collectors.*groupingBy*(x -> x, Collectors.*counting*())).entrySet().stream()

.filter(entry -> entry.getValue() > 1L).map(entry -> entry.getKey()).collect(Collectors.*toList*());

System.***out***.println(dupValue);

}

}

Question 4

Find employee salary greater than 35000

**public** **class** test1 {

**public** **static** **void** main(String[] args) {

List<Employee> emp = **new** ArrayList<>();

emp.add(**new** Employee(1, 25000, "bob"));

emp.add(**new** Employee(2, 35000, "alice"));

emp.add(**new** Employee(3, 45000, "john"));

emp.add(**new** Employee(4, 55000, "virat"));

List<Employee> sal = emp.stream().filter(x -> x.getSalary() > 35000).collect(Collectors.*toList*());

System.***out***.println(sal);

System.***out***.println("===============");

List<Employee> desSal = emp.stream().sorted((s1, s2) -> (**int**) (s2.getSalary() - s1.getSalary()))

.collect(Collectors.*toList*());

System.***out***.println(desSal);

}

}

Question 5:) find even number

**public** **class** test1 {

**public** **static** **void** main(String[] args) {

List<Integer> num = Arrays.*asList*(1,2,3,4,5,6,7,8);

Map<Boolean, List<Integer>> collect = num.stream().collect(Collectors.*partitioningBy*(x->x%2==0));

System.***out***.println(collect.get(**true**));

}

}

Question 6

Three categories of design pattern

**A design pattern**

help us identifying recurring problem and provide ready to use solution to solve that problem

**GOF( gangs of four) pattern catalog or core pattern---- three types**

**Creational Pattern :**

provide guidelines to create a single object or group of object. It all about object

 creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

**Structural Pattern:**

They provide a manner to define r/s in classes.

It is used when we want to create diff type of relation among classes

Structural design patterns are concerned with how classes and objects can be composed, to form larger structures. The structural design patterns simplifies the structure by identifying the relationships. These patterns focus on, how the classes inherit from each other and how they are composed from other classes

**Behavioral Patterns**

How communication should happen among classes and object

Question 7:

Create thread safe singleton

Graphical user interface, application

Description automatically generated

**Note :-**

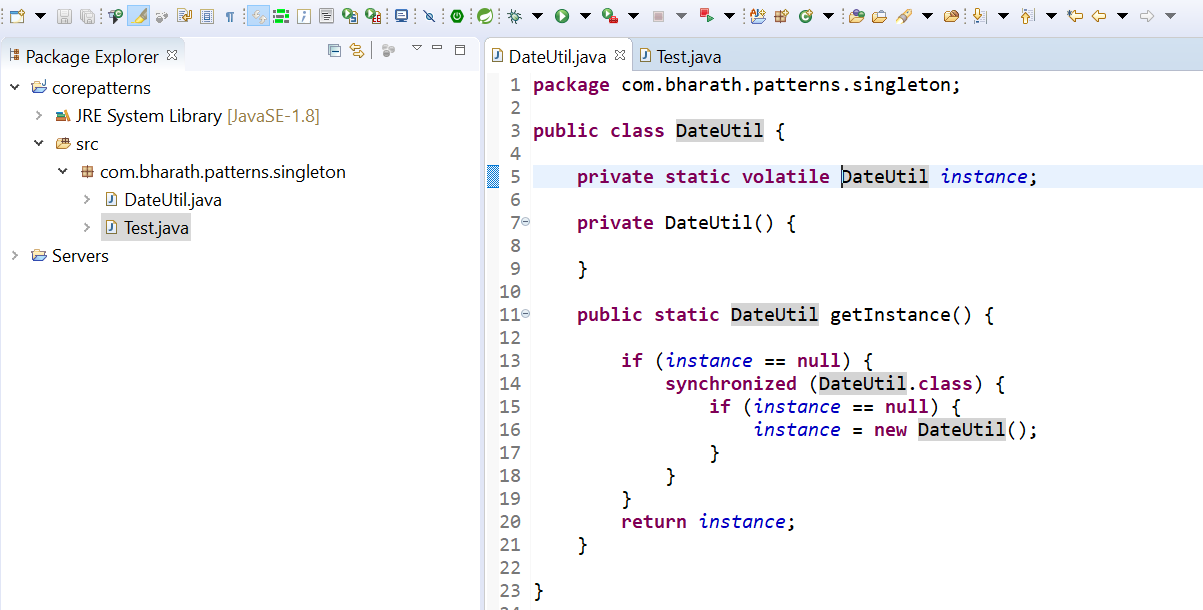
**Acquiring a lock is very expensive process**

In above case we don’t want everytime class level locking even if instance is not null

We want class level locking only when instance is null

**So put one more null check and it recommend to use volatile keyword**

Volatile ensure that multiple threads read the correct instance value. When a variable is declared volatile we suggest the compiler to never store the value of the variable in cache memory.



**Note**

**readResolve**will be invoke internally by**ObjectInputStream**

**The benefit of enum in java there is no constructor so there will be no access through reflection**

**Question 7**

A factory pattern is a creational pattern that abstracts or hides the object creation process.

**public** **interface** Shape {

**void** draw();

}

**public** **class** Rectangle **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("inside rectangle method");

}

}

**public** **class** Traingle **implements** Shape{

**public** **void** draw() {

System.***out***.println("inside Traingle method");

}

}

**public** **class** ShapeFactory {

**public** Shape getShape(String shape) {

**if** (shape == **null**) {

**return** **null**;

}

**if** (shape.equalsIgnoreCase("Rectangle")) {

**return** **new** Rectangle();

} **else** **if** (shape.equalsIgnoreCase("Triangle")) {

**return** **new** Traingle();

}

**return** **null**;

}

}

**public** **class** FactoryDemoTest {

**public** **static** **void** main(String[] args) {

ShapeFactory sf = **new** ShapeFactory();

Shape shape = sf.getShape("triangle");

shape.draw();

}

}

Abstract factory

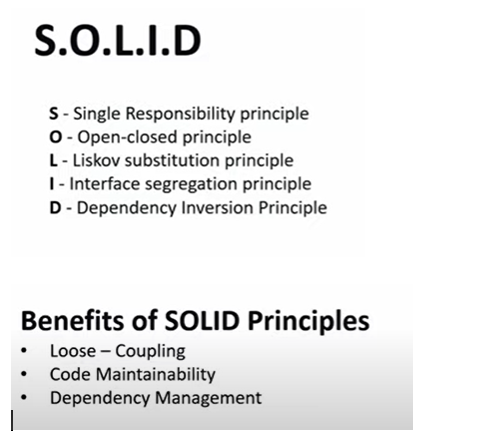
Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

**Refer example**

[**https://www.tutorialspoint.com/design\_pattern/abstract\_factory\_pattern.htm**](https://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm)

**Question 8**

**Explain solid principles**

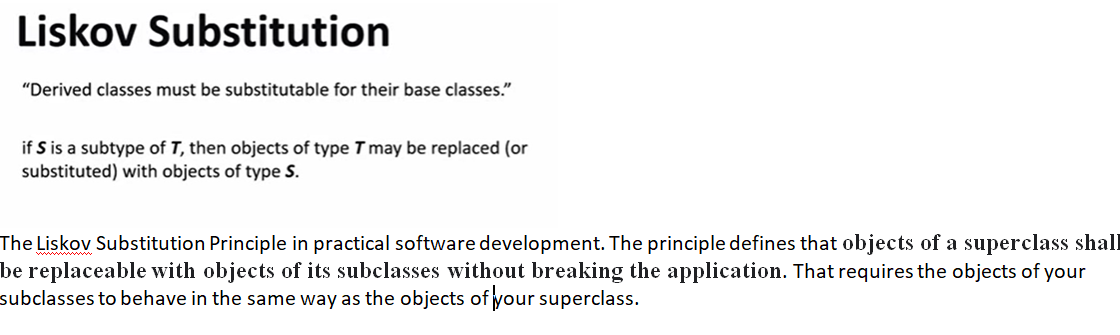
****

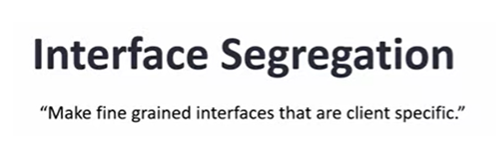
**Text

Description automatically generated with medium confidence**

**Graphical user interface, text

Description automatically generated**

****

****

****

**Question 9**

**Explain immutability and write class.**

**What is Immutability?**

**Immutability** is the ability keep not changing with the modifications. Once it s defined no one can alter it.

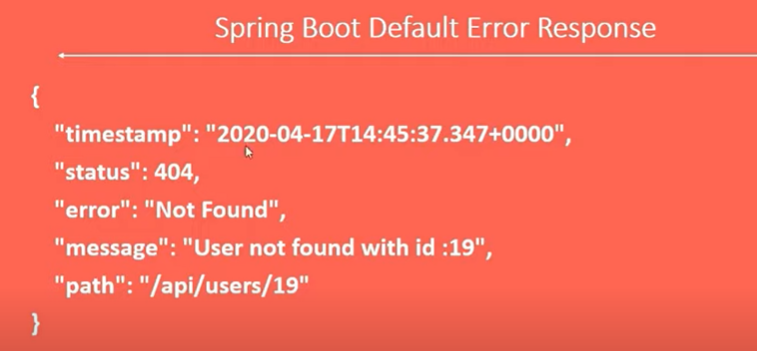
**Immutable class** is a class which is once created, it’s contents can not be changed.

* The class must be declared as final so that child classes can’t be created.
* Data members in the class must be declared private so that direct access is not allowed.
* Data members in the class must be declared as final so that we can’t change the value of it after object creation.
* A parameterized constructor should initialize all the fields performing a deep copy so that data members can’t be modified with an object reference.
* Deep Copy of objects should be performed in the getter methods to return a copy rather than returning the actual object reference)

Question 10

Exception handling in rest

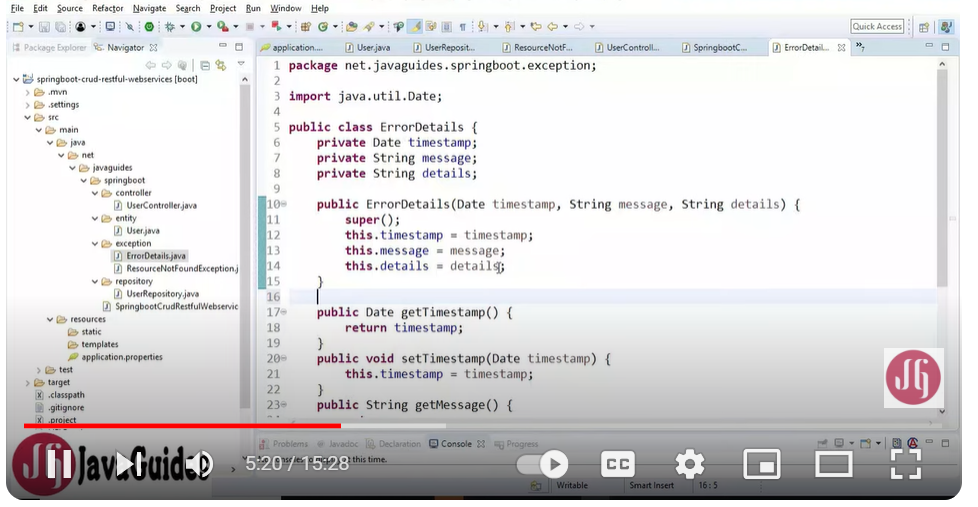
Default one send by springboot



Custom one.

Step 1 :

First create class whatever we want to send to client.



Step 2:

Create class **GlobalExceptionHandler** where we want to handle all exception

Mark it with **@ControllerAdvice : which**  handle exception globally

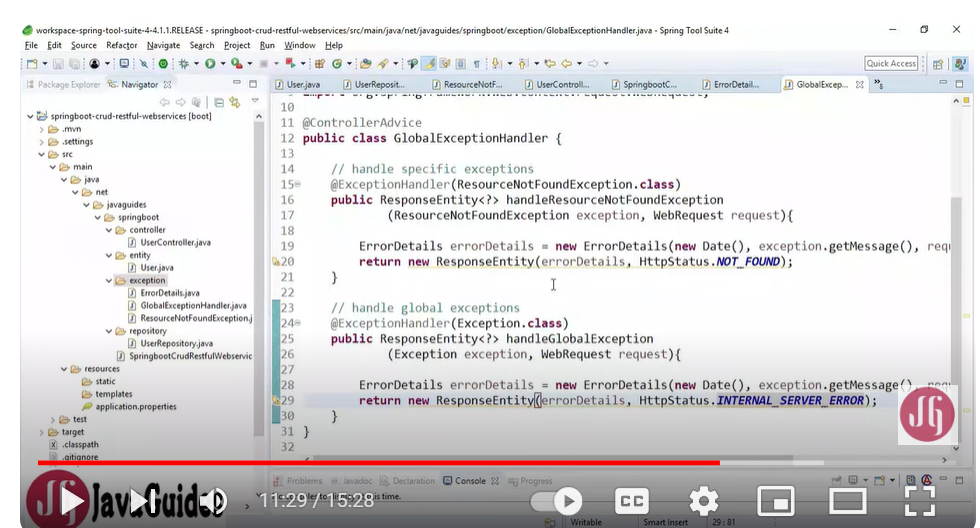
@ControllerAdvice : is use to handle exception globally.

@ExceptionHandler: handle specific exception and send custom response to client.

The **@ControllerAdvice** annotation is used to define a class that will be called whenever an exception is thrown in your application. This class can contain multiple methods, each of which is annotated with the @ExceptionHandler annotation and is responsible for handling a specific exception

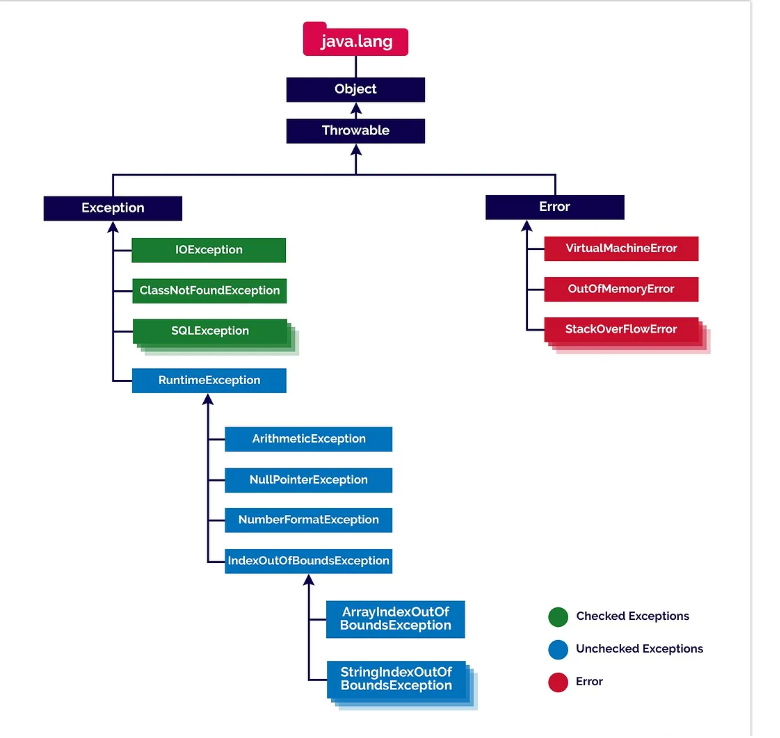
Spring Boot provides the **@ExceptionHandler** annotation to handle exceptions thrown by a specific controller method. This annotation can be used to provide customized error responses for specific exceptions.

**Handle specific and global exception**



Question 11

Difference between checked and unchecked exception



## Checked Exceptions in Java

Theseare the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using the [*throws*keyword](https://www.geeksforgeeks.org/throw-throws-java/). In checked exceptions, there are two types: fully checked and partially checked exceptions. A fully checked exception is a checked exception where all its child classes are also checked, like IOException, and InterruptedException. A partially checked exception is a checked exception where some of its child classes are unchecked, like an Exception.

## Unchecked Exceptions in Java

These are the exceptions that are not checked at compile time.

In short unchecked exceptions are runtime exceptions that are not required to be caught or declared in a throws clause. These exceptions are usually caused by programming errors, such as attempting to access an index out of bounds in an array or attempting to divide by zero.

Unchecked exceptions include all subclasses of the RuntimeException class, as well as the Error class and its subclasses.

**Here are some examples of unchecked exceptions in Java:**

***1. ArrayIndexOutOfBoundsException:****This exception is thrown when you attempt to access an array index that is out of bounds.****2. NullPointerException:****This exception is thrown when you attempt to access a null object reference.****3. ArithmeticException:****This exception is thrown when you attempt to divide by zero or perform an invalid arithmetic operation.*

Question 12

Difference b/w comparable and comparator.

Comparable :

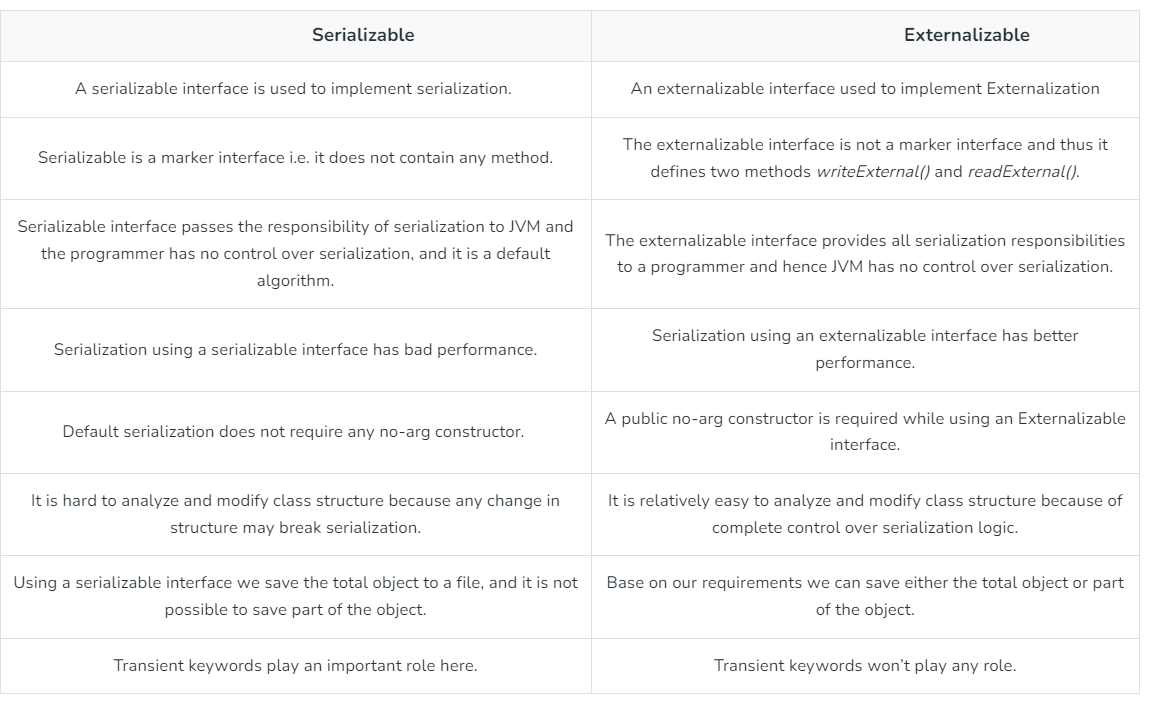
1. We can use it to define default natural sorting order.
2. This interface present in java.lang package
3. Define one method compareTo
4. All wrapper class and string implements comparable interface.

Comparator.

1. We can use to define customized sorting order
2. Present in java.util package
3. Define compare, equal method
4. No predefined class implements comparator interface

Question 13

**Difference between serialization and externalization**



Question 14

Generics : ---- helpful for type safety and type casting

Question 15 ( for more check in copy)

What is cloning , shallow copy and deep copy

**Shallow copy :-** one object and two reference

Issue : if we change any one it will effect other

**Deep copy :-** creating two object and copying each value

Issue: if 50 variables are there then code will increase

**Cloning** : it will create a new object ( internally using deep cloning)

Implements cloneable interface

The **object cloning** is a way to create exact copy of an object. The clone() method of Object class is used to clone an object.

The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.

The **clone() method** is defined in the Object class. Syntax of the clone() method is as follows:

1. **protected** Object clone() **throws** CloneNotSupportedException
2. **class Student18 implements Cloneable{**
3. **int rollno;**
4. **String name;**
6. **Student18(int rollno,String name){**
7. **this.rollno=rollno;**
8. **this.name=name;**
9. **}**
11. **public Object clone()throws CloneNotSupportedException{**
12. **return super.clone();**
13. **}**
15. **public static void main(String args[]){**
16. **try{**
17. **Student18 s1=new Student18(101,"amit");**
19. **Student18 s2=(Student18)s1.clone();**
21. **System.out.println(s1.rollno+" "+s1.name);**
22. **System.out.println(s2.rollno+" "+s2.name);**
24. **}catch(CloneNotSupportedException c){}**
26. **}**
27. **}**

**Question 16**

**Two**  pointer algorithm

Find the sum of two element which is equal to target value

Let’s first see a naive solution:

**public** **boolean** **twoSumSlow**(**int**[] input, **int** targetValue) {

**for** (**int** i = 0; i < input.length; i++) {

**for** (**int** j = 1; j < input.length; j++) {

**if** (input[i] + input[j] == targetValue) {

**return** true;

}

}

}

**return** false;

}

public boolean twoSum(int[] input, int targetValue) {

int pointerOne = 0;

int pointerTwo = input.length - 1;

while (pointerOne < pointerTwo) {

int sum = input[pointerOne] + input[pointerTwo];

if (sum == targetValue) {

return true;

} else if (sum < targetValue) {

pointerOne++;

} else {

pointerTwo--;

}

}

return false;

}

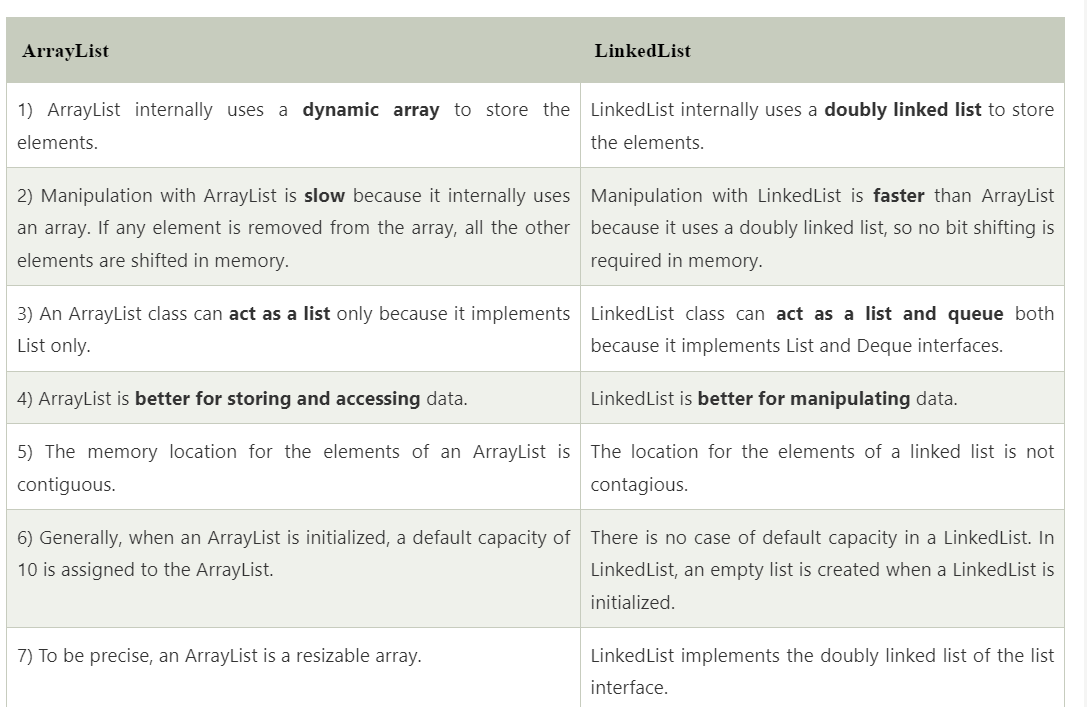
Question 17

What is Jackson used for in Java?

Jackson is one such Java Json library used for parsing and generating Json files. It has built in Object Mapper class which parses json files and deserializes it to custom java objects. It helps in generating json from java objects

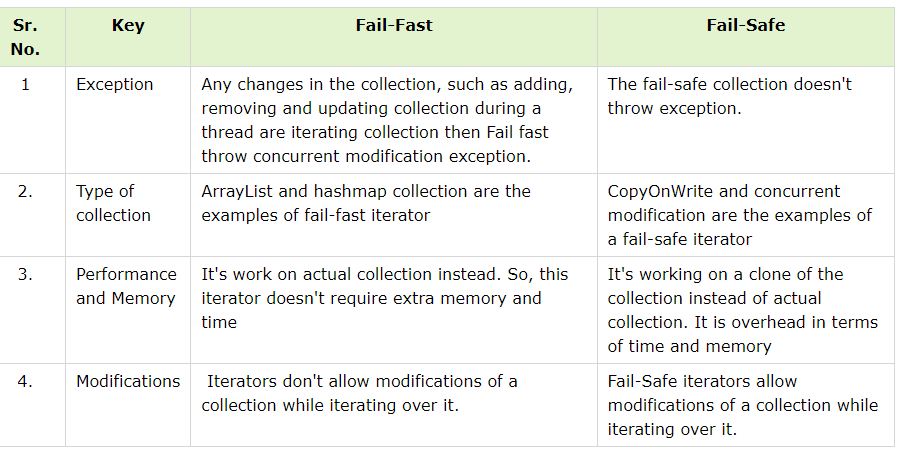
Question 18

Difference between ArrayList and LinkedList



Question 19

Difference between fail fast and fail safe



Question 20

Contract between equals and hashcode

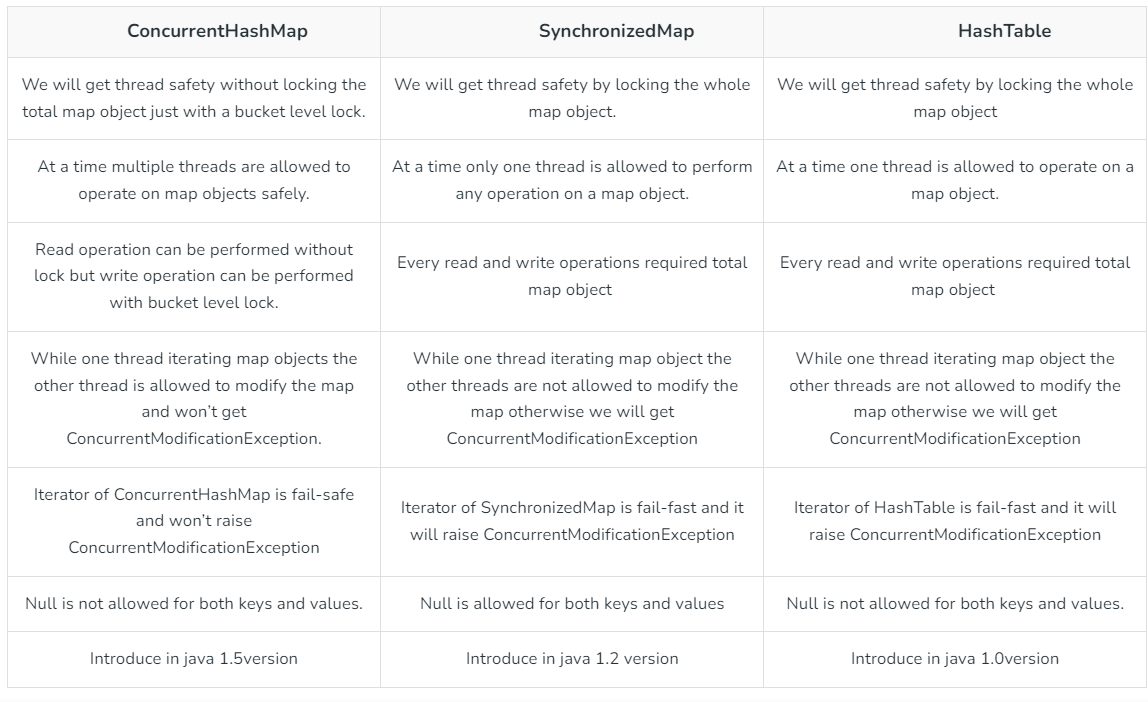
The basic rule of the contract states that if two objects are **equal to each other based on equals() method, then the hash code must be the same**, but if the hash code is the same, then equals() can return false.

Question 21

Time complexity in inserting an element in arraylist, linkedlist and hashmap

The average time complexity to add an item to the end of an ArrayList is O(1). 2. The worst case time complexity to add an item to the end of an ArrayList is O(n), which happens when the list has to be reallocated due to its size.

The advantage of a HashMap is that the time complexity to insert and retrieve a value is O(1) on average



Question 22

**CopyOnWriteArray List**

**Here are few points about CopyOnWriteArrayList:**

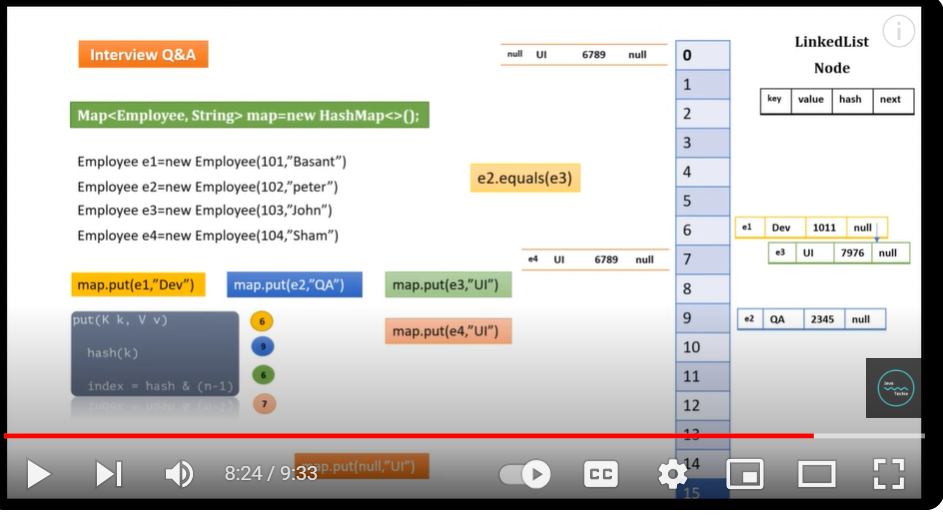
* As the name indicates, CopyOnWriteArrayList creates a Cloned copy of underlying ArrayList, for every update operation at a certain point both will be synchronized automatically, which is taken care of by JVM. Therefore, there is no effect for threads that are performing read operation.
* It is costly to use because for every update operation a cloned copy will be created. Hence, CopyOnWriteArrayList is the best choice if our frequent operation is read operation
* The underlined data structure is a grow-able array.
* It is a thread-safe version of ArrayList.
* Insertion is preserved, duplicates, null, and heterogeneous Objects are allowed.
* The main important point about CopyOnWriteArrayList is the [Iterator](https://www.geeksforgeeks.org/iterators-in-java/) of CopyOnWriteArrayList can not perform remove operation otherwise we get Run-time exception saying **UnsupportedOperationException.**add() and set() methods on CopyOnWriteArrayList iterator also throws **UnsupportedOperationException.**Also Iterator of CopyOnWriteArrayList will never throw **ConcurrentModificationException**.

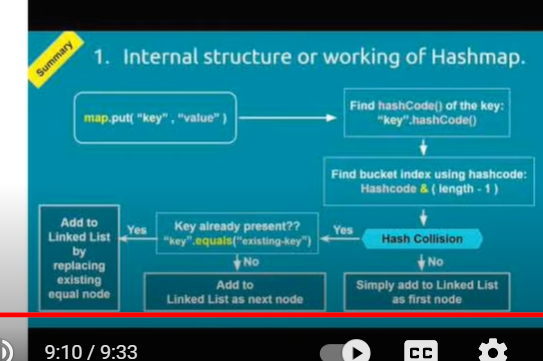
The most common cause for this exception is trying to make changes to an immutable list, set, map, etc.

To fix the issue, please ensure that the data structure you are trying to operate on is mutable and allows for changes to be made to its contents.

Question 23

Internal working of hashmap





Question 24

Difference between intermediate operation and terminal operation

**A screenshot of a computer

Description automatically generated**

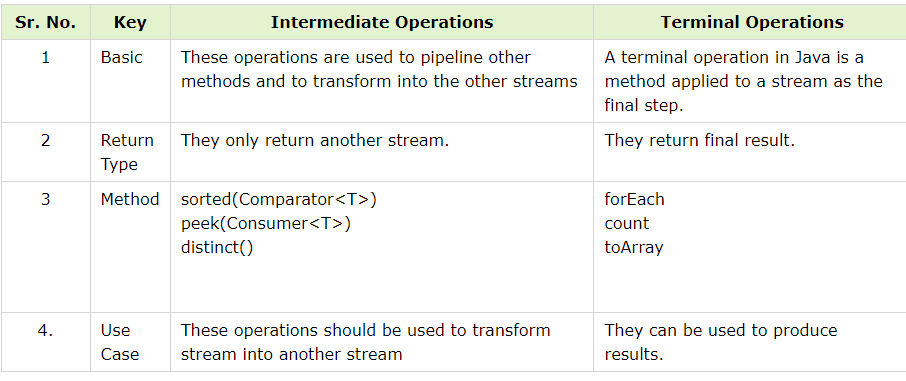
**Intermediate operation :**  convert stream into another stream

**Terminal operation** : bring the required result.

**Note** : when terminal operation called then only intermediate operation executed

A diagram of a program

Description automatically generated



Question 25

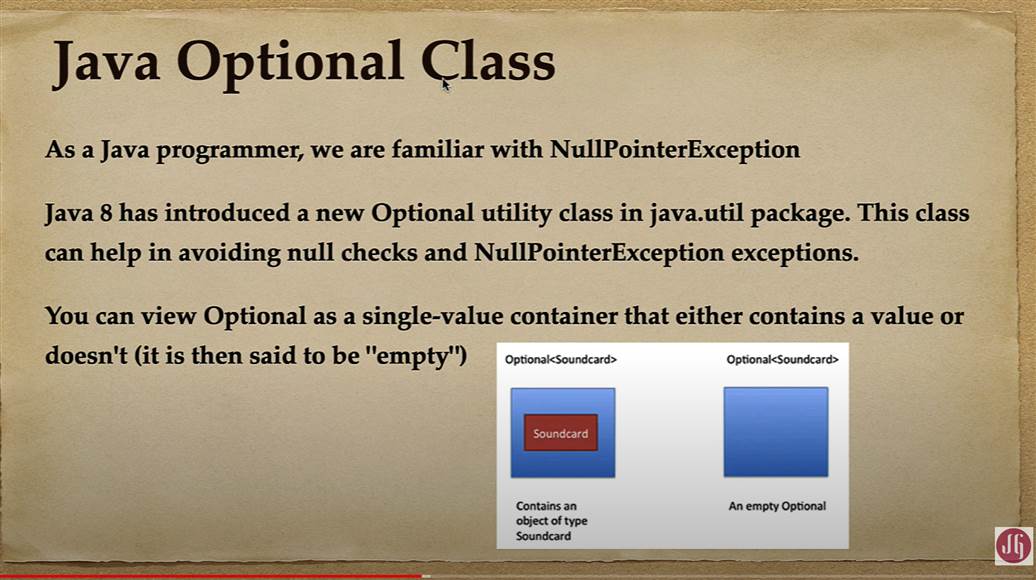
Difference between stream and parallel stream

A sequential stream is executed in a single thread running on one CPU core. The elements in the stream are processed sequentially in a single pass by the stream operations that are executed in the same thread. A parallel stream is executed by different threads, running on multiple CPU cores in a computer

A screenshot of a computer

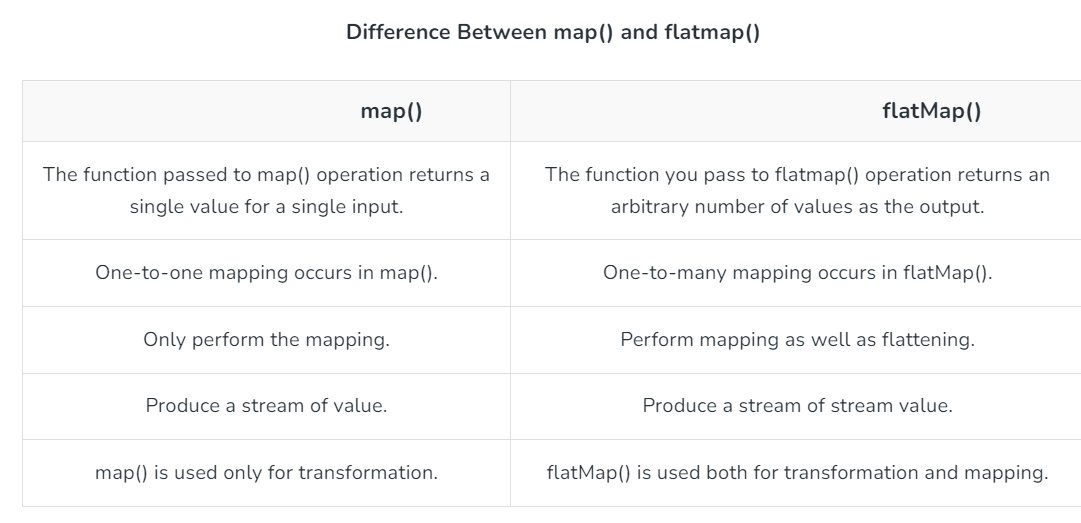
Description automatically generated

Q26 Optional classes (read more in notes)



**Question 27**

**Difference between map and flatmap**

****

**Question 28**

# Java Future Example

In [Java](https://www.javatpoint.com/java-tutorial), **Future** is an [interface](https://www.javatpoint.com/interface-in-java) that belongs to **java.util.concurrent** [package](https://www.javatpoint.com/package). It is used to represent the result of an asynchronous computation. The interface provides the methods to check if the computation is completed or not, to wait for its completion, and to retrieve the result of the computation. Once the task or computation is completed one cannot cancel the computation.

**Syntax:**

1. **public** **interface** Future<V>

## Example of Java Future

The best example of Java Future is **ExecutorService** interface. It produces a Future (from some of their methods) object for tracking progress of one or more asynchronous task.

**Difference between future and completable future**

[**https://www.linkedin.com/pulse/java-8-future-vs-completablefuture-saral-saxena/**](https://www.linkedin.com/pulse/java-8-future-vs-completablefuture-saral-saxena/)

**refer this link**

**Limitations of Future**

* ***It can not be manually completed***

**Multiple Futures can not be chained together**

* ***No Exception Handling***

## Creating a CompletableFuture :-

**CompletableFuture<String> completableFuture = new CompletableFuture<String>();**

## Running asynchronous computation using runAsync()

If you want to run some background task asynchronously and don’t want to return anything from the task, then you can use CompletableFuture.runAsync() method. It takes a Runnable object and returns CompletableFuture<Void>

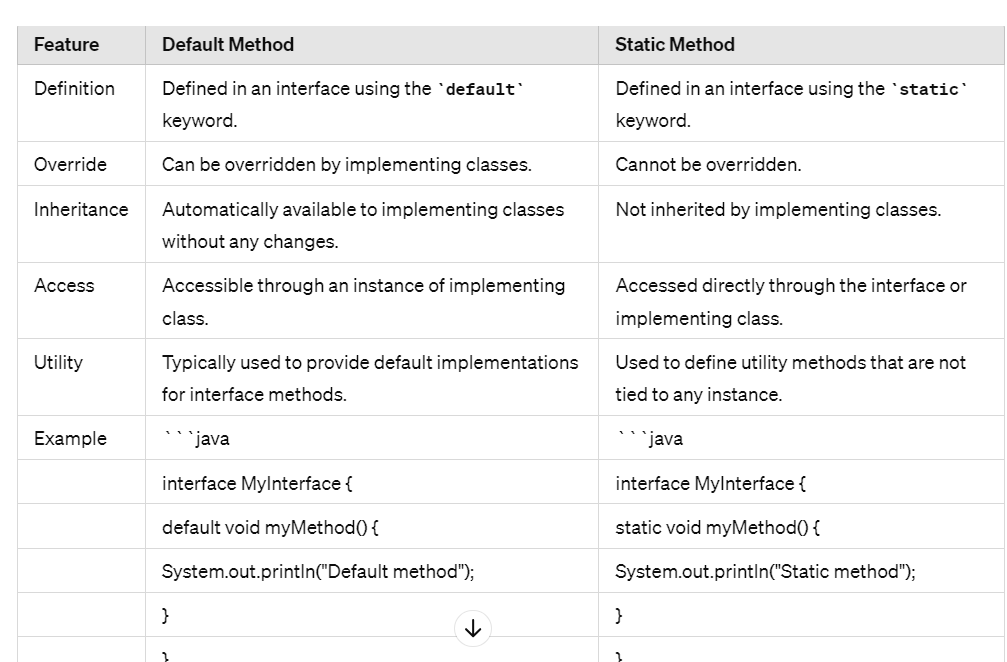
## Running a task asynchronously and return the result using supplyAsync()

CompletableFuture.runAsync() is useful for tasks that don’t return anything. But what if you want to return some result from your background task?

Well, CompletableFuture.supplyAsync() is your companion. It takes a [Supplier<T>](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html) and returns CompletableFuture<T> where T is the type of the value obtained by calling the given supplier

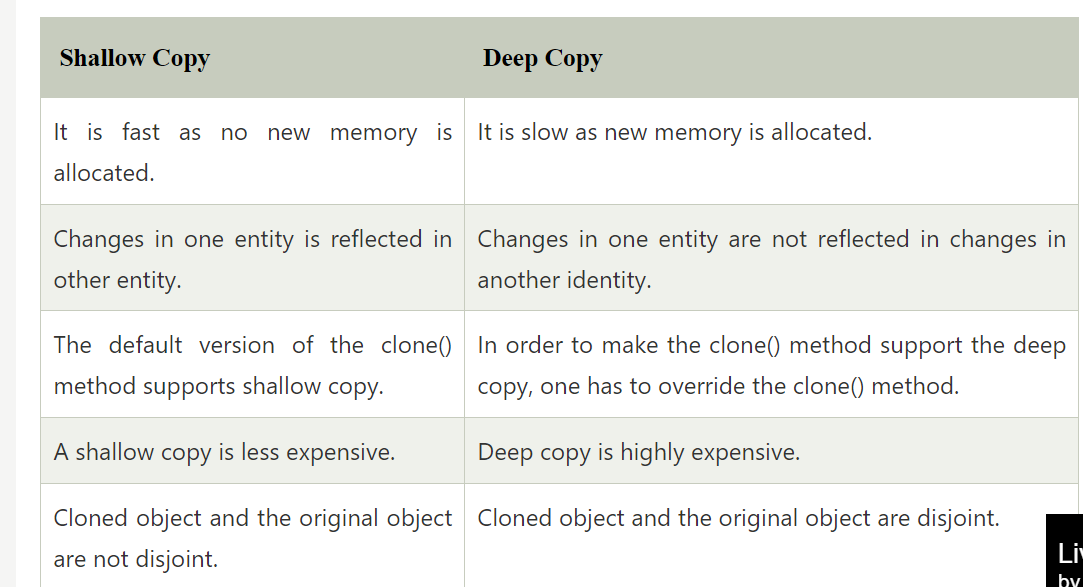
**Question 29**

**Difference between default and static method**

****

Both default and static methods were introduced in Java 8 to provide more flexibility in interfaces without breaking the existing codebase. Default methods allow interfaces to evolve without breaking the classes that implement them, while static methods allow interfaces to have "helper" methods that don't depend on a specific instance of the implementing class.

**Question 30 : difference between deep copy and shallow copy ?**

****

## Shallow Copy

When we do a copy of some entity to create two or more than two entities such that changes in one entity are reflected in the other entities as well, then we can say we have done a shallow copy. In shallow copy, new memory allocation never happens for the other entities, and the only reference is copied to the other entities. The following example demonstrates the same.

1. **class** ABC
2. {
3. // instance variable of the class ABC
4. **int** x = 30;
5. }
6. **public** **class** ShallowCopyExample
7. {
8. // main method
9. **public** **static** **void** main(String argvs[])
10. {
11. // creating an object of the class ABC
12. ABC obj1 = **new** ABC();
14. // it will copy the reference, not value
15. ABC obj2 = obj1;
17. // updating the value to 6
18. // using the reference variable obj2
19. obj2.x = 6;
21. // printing the value of x using reference variable obj1
22. System.out.println("The value of x is: " + obj1.x);
23. }
24. }

## Deep Copy

When we do a copy of some entity to create two or more than two entities such that changes in one entity are not reflected in the other entities, then we can say we have done a deep copy. In the deep copy, a new memory allocation happens for the other entities, and reference is not copied to the other entities. Each entity has its own independent reference. The following example demonstrates the same.

1. **class** ABC
2. {
3. // instance variable of the class ABC
4. **int** x = 30;
5. }
6. **public** **class** DeepCopyExample
7. {
8. // main method
9. **public** **static** **void** main(String argvs[])
10. {
11. // creating an object of the class ABC
12. ABC obj1 = **new** ABC();
14. // it will copy the reference, not value
15. ABC obj2 = **new** ABC();
17. // updating the value to 6
18. // using the reference variable obj2
19. obj2.x = 6;
21. // printing the value of x using reference variable obj1
22. System.out.println("The value of x is: " + obj1.x);
23. }
24. }

