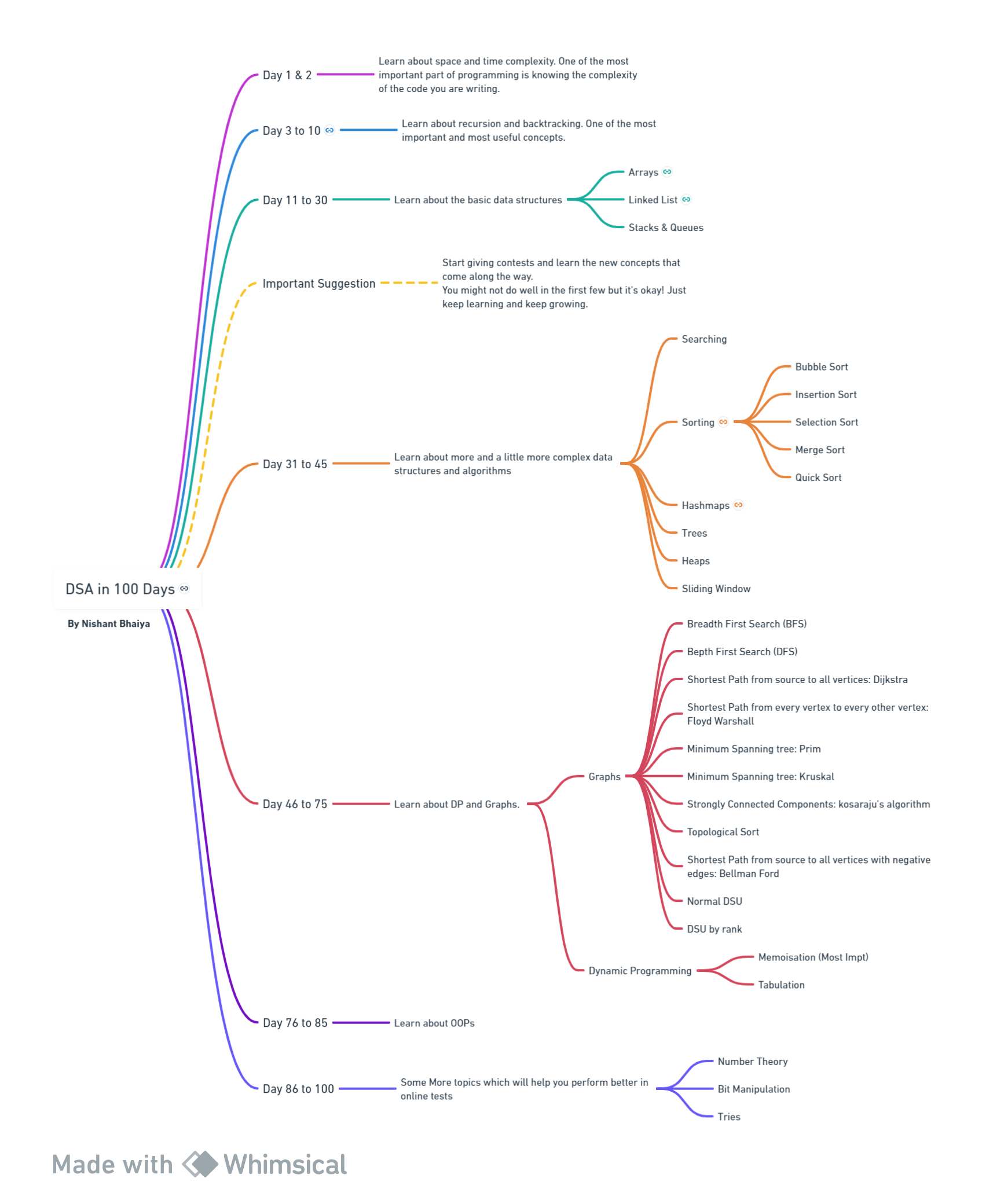
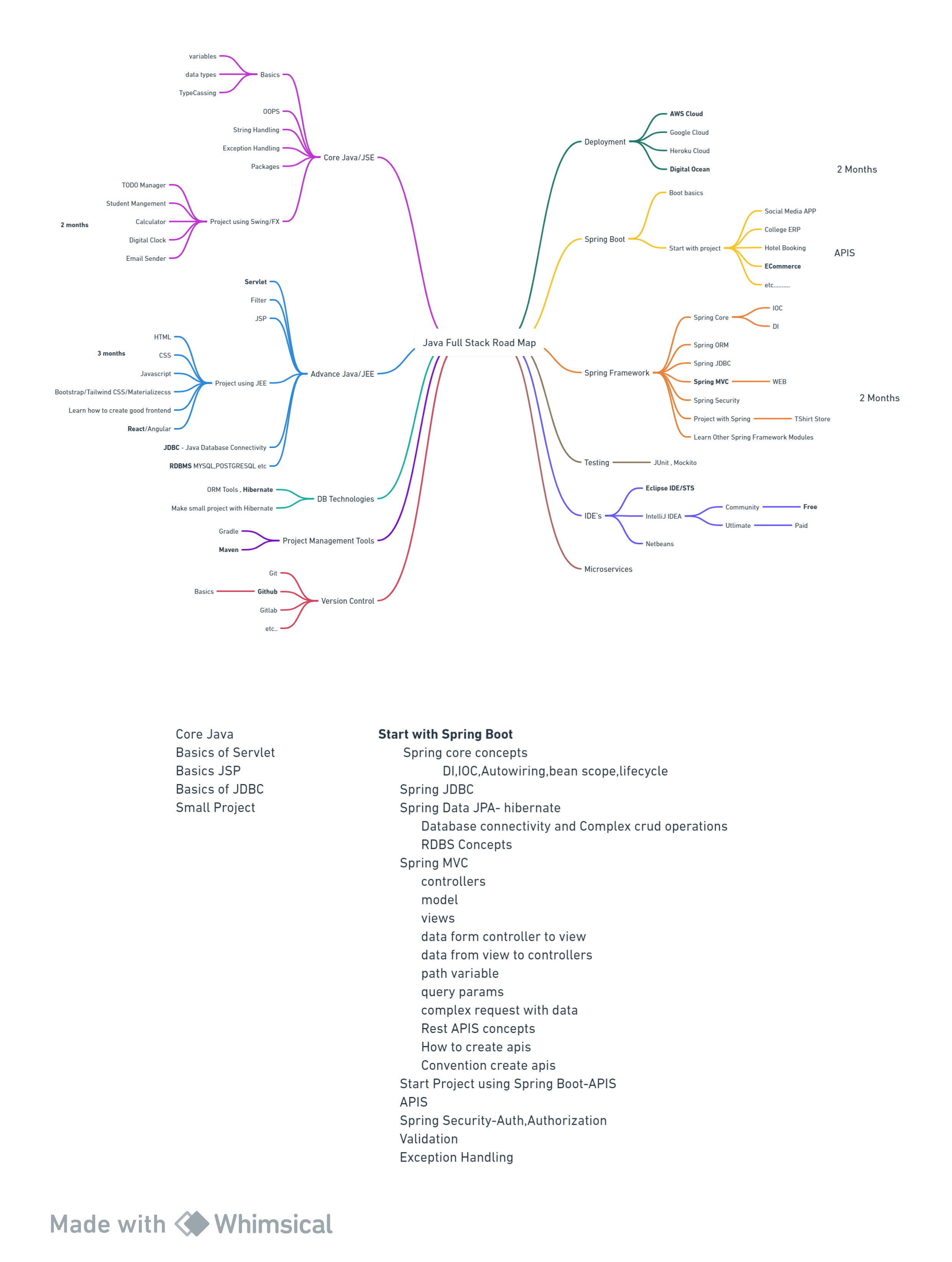
# Core JAVA

## Roadmap to Data Structure and Algorithm:



## Roadmap JAVA Full Stack Development:



**(JSP, HTML, CSS, Angular + JAVA + Spring + Spring Boot + Spring REST, Microservices + Hibernate + GIT + Maven)**

## Internal working of hashmap

Hashmap uses an **Array of Buckets** to store Key Value in Hashmap.

1. **Hashing:** When Key Value added in Hashmap then Hashcode method get called to generate the hashcode for that KEY 🡪 Hashcode is then used to determine the index in the Bucket of Arrays.
2. **Bucket and Collision Resolution:** Each Bucket in Hashmap holds a linked list (Balanced Tree Structure in Latest Version) to store Key Value. If multiple Key Value have same hashcode then collison occurs.
3. **Load Factor and Rehashing:** Rehashing is the process of resizing the internal array when key value exceeds the certain threshold value(Loadfactor \* capacity).
4. **Retrieving Values:** retrieving a value based on a key, the hashCode() method is called on the key to find the corresponding bucket. Then, the equals() method is used to compare the keys within the bucket to find the exact match.

## Hashmap Vs Concurent Hashmap VS Hashtable

In Java, `HashMap`, `ConcurrentHashMap`, and `Hashtable` are all classes that implement the `Map` interface to store key-value pairs. However, they have different characteristics and are suited for different use cases. Here's a comparison of these three classes:

1. **HashMap**:

- `HashMap` is a not-synchronized implementation of the `Map` interface.

- It allows null values for both keys and values.

- It permits one null key.

- It is not thread-safe, which means if multiple threads access and modify the `HashMap` concurrently, it can lead to data corruption or inconsistency.

- It provides good performance for single-threaded scenarios.

- `HashMap` is the most commonly used map implementation when there are no concurrent access requirements.

2. **ConcurrentHashMap**:

- `ConcurrentHashMap` is a synchronized implementation of the `Map` interface that provides thread-safe operations.

- It allows multiple threads to read and write concurrently without external synchronization.

- It is designed to handle concurrent updates efficiently, using a technique called "Segmented Lock" that reduces contention for different parts of the map.

- It supports concurrent retrieval and modification of the map without blocking other threads.

- It does not allow null keys or values. Attempting to add null keys or values will result in a `NullPointerException`.

- When multiple threads need to access and modify the map concurrently, `ConcurrentHashMap` is the recommended choice due to its better concurrency support compared to `Hashtable` and synchronized `HashMap`.

3. **Hashtable**:

- `Hashtable` is a synchronized implementation of the `Map` interface, similar to `ConcurrentHashMap`, but it is considered a legacy class.

- Like `ConcurrentHashMap`, it allows multiple threads to read and write concurrently without external synchronization.

- It is thread-safe but can suffer from performance issues due to a single lock for all operations.

- It does not allow null keys or values. Attempting to add null keys or values will result in a `NullPointerException`.

- Because of its performance limitations, it is generally recommended to use `ConcurrentHashMap` instead of `Hashtable`.

In summary, if you need a basic map implementation and have a single-threaded environment, `HashMap` is the standard choice. If you need a thread-safe map for concurrent access, use `ConcurrentHashMap`. Avoid using `Hashtable` unless you are working with legacy code or have specific compatibility requirements with old systems.

## Override hashcode and Equals custom object as a key in hashmap

Sure, here's a short coding implementation of a custom object as a unique key in a HashMap with the `hashCode()` and `equals()` methods overridden:

class CustomKey {

private int id;

private String name;

@Override

public int hashCode() {

return Objects.hash(id, name);

}

@Override

public boolean equals(Object obj) {

CustomKey other = (CustomKey) obj;

return id == other.id && Objects.equals(name, other.name);

}

}

```

In this example, we have a `CustomKey` class with `id` and `name` fields. We override the `hashCode()` method using `Objects.hash()` and the `equals()` method to compare the contents of the `CustomKey` objects. This allows the HashMap to recognize different objects with the same `id` and `name` as equal keys.

## Iterate Hashmap using ForEach and by using JAVA 8 Lamba and streams.

**For Each Loop:**

for (Map.Entry<String, Integer> entry : map.entrySet()) {

String key = entry.getKey();

Integer value = entry.getValue();

System.out.println(key + " => " + value);

}

**JAVA 8:**

**// Using forEach and Lambda expression**

map.forEach((key, value) -> System.out.println(key + " => " + value));

**// Using streams**

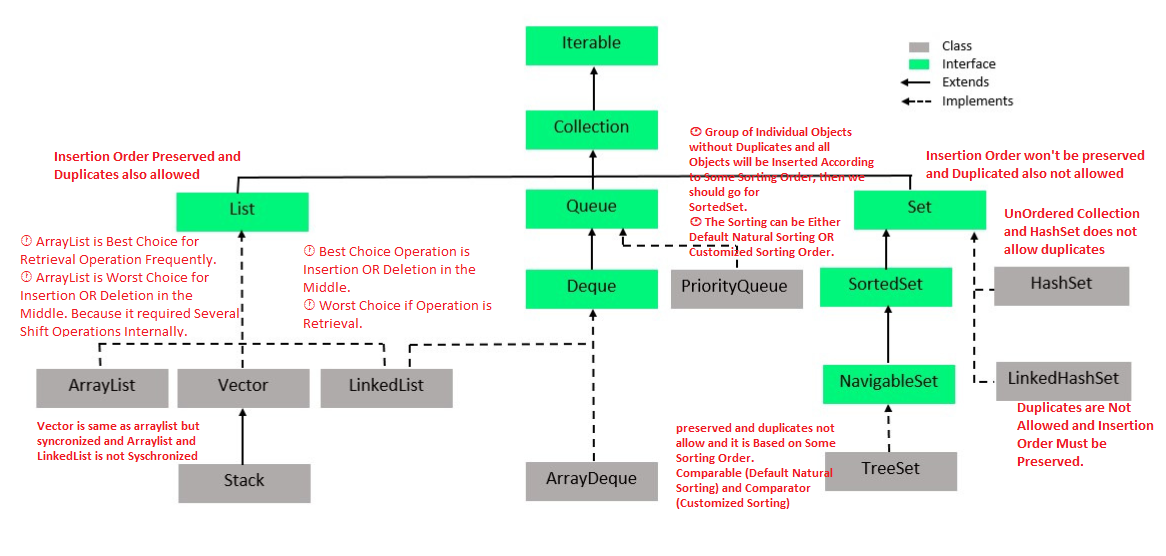
map.entrySet().stream()

.forEach(entry -> System.out.println(entry.getKey() + " => " + entry.getValue()));

## Major changes in Hashmap for JAVA

When a bucket exceeds a certain threshold (8 elements by default), it is converted from a linked list to a balanced binary search tree (Red-Black tree). This change improved the worst-case performance of HashMap from O(n) to O(log n) for some operations.

## Collection Hierarchy:



## Internal working of hashset

It is used to store a unordered collection of unique elements, meaning it does not allow duplicate values. HashSet uses a hash table to store elements internally. Here's a brief overview of the internal working of HashSet in Java:

1. **Hashing:**

- When you add an element to a HashSet, the `hashCode()` method of the object is called to generate a hash code (an integer) representing the object's value.

- The hash code is then used to calculate the index or bucket where the element will be stored in the underlying array.

2. **Bucket and Collision Handling:**

- HashSet internally uses an array of "buckets" to store elements. Each bucket can hold multiple elements due to the possibility of hash code collisions.

- A hash code collision occurs when two different objects have the same hash code. In such cases, the elements are stored in the same bucket.

3. **No Duplicates:**

- HashSet ensures that it does not store duplicate elements by using the `equals()` method of the objects.

- When adding a new element, HashSet checks if an element with the same hash code is already present in the same bucket. If not, it adds the new element to the bucket. If a similar element is already present, the HashSet uses the `equals()` method to determine if it's a duplicate.

4. **Load Factor and Rehashing:**

- HashSet has a load factor, which represents the ratio of the number of stored elements to the total number of buckets.

- When the load factor exceeds a certain threshold, a process called rehashing occurs. During rehashing, the HashSet increases the size of its underlying array and rehashes all the elements to new buckets, reducing the likelihood of collisions and maintaining efficient performance.

5. **Iteration and Ordering:**

- HashSet does not maintain any specific order of elements. Iterating over a HashSet provides elements in an arbitrary order.

- If you need to maintain a specific order, you can use a `LinkedHashSet`, which preserves the order of elements based on their insertion.

In summary, HashSet provides fast and efficient retrieval of elements due to its use of hashing. It avoids duplicate elements and can dynamically resize its underlying array to maintain good performance as the number of elements increases. However, it does not provide any specific ordering of elements, and iterating over a HashSet results in an arbitrary order. If you need a specific order, you can use a `LinkedHashSet`.

// Add elements to the HashSet

numbersSet.add(10);

numbersSet.add(20);

numbersSet.add(30);

numbersSet.add(40);

numbersSet.add(50);

// **Add a duplicate element (UnOrdered Collection and HashSet does not allow duplicates)**

numbersSet.add(30);

## Linked Hashmap in JAVA

LinkedHashMap is just like HashMap. it maintains a doubly linked list in addition to the hash table to maintatin the insertion order of the elements and no duplicate allowed. Here's a brief overview of the internal working of LinkedHashMap in Java:

1. Hashing and Buckets:

- LinkedHashMap uses the same hashing mechanism as HashMap. When you add an element to a LinkedHashMap, its `hashCode()` method is called to generate a hash code.

- The hash code is used to determine the index or bucket where the element will be stored in the underlying array, just like in HashMap.

2. Doubly Linked List for Insertion Order:

- In addition to the hash table, LinkedHashMap maintains a doubly linked list to preserve the order in which elements are inserted.

- Each entry in the linked list represents a key-value pair. The linked list maintains the order of insertion for elements.

3. Bucket and Collision Handling:

- As with HashMap, if two elements have the same hash code (collision), they will be stored in the same bucket.

- However, LinkedHashMap maintains the insertion order of elements within each bucket using the linked list.

4. No Duplicates:

- Like all implementations of the Map interface, LinkedHashMap does not allow duplicate keys. It uses the `equals()` method to check for duplicates.

5. Iteration Order:

- When you iterate over a LinkedHashMap using an iterator or a for-each loop, the elements are returned in the order they were inserted, due to the linked list.

6. Access Order Mode (Optional):

- LinkedHashMap can also be configured to maintain elements in access order rather than insertion order using a constructor with an access-order flag or `accessOrder` parameter.

- In access order mode, the most recently accessed (get or put) element is moved to the end of the linked list, effectively making it the last element in the iteration order.

In summary, LinkedHashMap combines the benefits of a hash table for fast access with a doubly linked list for preserving insertion order. It is a useful data structure when you need both fast retrieval of elements and the ability to iterate over elements in the order they were inserted. Additionally, with access order mode, LinkedHashMap can serve as a simple implementation for implementing a Least Recently Used (LRU) cache.

LinkedHashMap<String, Integer> cityPopulationMap = new LinkedHashMap<>();

// Add elements to the LinkedHashMap

cityPopulationMap.put("New York", 8537673);

cityPopulationMap.put("Los Angeles", 3979576);

cityPopulationMap.put("Chicago", 2693976);

cityPopulationMap.put("Houston", 2132531);

cityPopulationMap.put("Phoenix", 1680992);

// **Print the LinkedHashMap (Maintains insertion order)**

System.out.println("LinkedHashMap: " + cityPopulationMap);

## Internal working of Treemap.

TreeMap maintains the elements in sorted order based on the natural order of keys or a custom comparator. It uses a red-black binary search tree to store and manage elements efficiently. Here's a brief overview of the internal working of TreeMap in Java:

1. Red-Black Binary Search Tree (RB-Tree):

- TreeMap internally uses a self-balancing binary search tree called a red-black tree.

- This tree structure ensures that the height of the tree remains balanced, which guarantees efficient operations like insertion, deletion, and search with a time complexity of O(log n).

2. Sorting:

- TreeMap automatically maintains the keys in sorted order (natural order or custom order).

- For natural ordering, the keys must implement the `Comparable` interface. If not, you can provide a custom `Comparator` during TreeMap construction to define the ordering.

3. Key-Value Pairs:

- Each element in the TreeMap is stored as a key-value pair.

- Keys must be unique within the TreeMap; duplicate keys are not allowed.

- TreeMap allows `null` values, but only one `null` key is allowed since keys must be unique.

4. Insertion and Balancing:

- When inserting a new key-value pair, the TreeMap uses the key to determine its position in the red-black tree based on the natural or custom ordering.

- It then performs rotations and color adjustments as needed to maintain the red-black tree properties and ensure balanced heights.

5. Searching:

- Searching for a specific key in the TreeMap is performed using a binary search within the red-black tree, resulting in a time complexity of O(log n).

6. Iteration:

- TreeMap allows iteration over its elements in sorted order, making it efficient for scenarios where sorted data is required.

7. Performance:

- TreeMap offers efficient operations for basic Map functionality, such as `put`, `get`, and `remove`, with an average time complexity of O(log n).

- However, due to the self-balancing nature of the red-black tree, TreeMap can have slightly slower performance than simpler data structures like HashMap for large datasets.

In summary, TreeMap is a sorted and self-balancing binary search tree implementation that maintains elements in sorted order based on their keys. It provides efficient operations for maintaining sorted data and supports custom ordering through comparators. However, TreeMap may have slightly slower performance than simpler data structures like HashMap due to its self-balancing mechanism.

TreeMap<String, Integer> studentGrades = new TreeMap<>();

// Add elements to the TreeMap

studentGrades.put("Alice", 95);

studentGrades.put("Bob", 87);

studentGrades.put("Charlie", 92);

studentGrades.put("Diana", 78);

studentGrades.put("Eva", 85);

// **Print the TreeMap (Elements sorted in ascending order of keys)**

System.out.println("TreeMap: " + studentGrades);

## Concurrent modification exception

The exception occurs when a collection object, such as an ArrayList or HashMap, is being modified, and at the same time, another thread or iterator tries to make changes to that collection object.

## Internal working of concurrent Hashmap

ConcurrentHashMap is a sychronized (thread-safe) implementation of the Map interface in Java. It is designed to support high-concurrency scenarios and allows multiple threads to read and write concurrently without explicit synchronization. Here's a brief overview of the internal working of ConcurrentHashMap in Java:

1. Segmented Architecture:

- ConcurrentHashMap uses a segmented architecture, where the data is divided into multiple segments (buckets or partitions), each acting as an independent HashMap.

- The number of segments is determined by the concurrency level, which can be specified during ConcurrentHashMap construction.

2. Segment Level Locking:

- Each segment in the ConcurrentHashMap is individually locked using a ReentrantLock or a synchronized block, depending on the Java version.

- This means that different threads can access different segments concurrently, improving concurrency and scalability.

3. Hashing and Bucket Selection:

- ConcurrentHashMap uses the key's hash code to determine the segment in which the key-value pair should be placed.

- Within each segment, a regular hash table data structure is used to store the key-value pairs.

4. Read Operations:

- Read operations like `get` do not require any locking, as different threads can safely read from different segments concurrently.

- Read operations are highly scalable and can be performed concurrently without contention.

5. Write Operations:

- Write operations like `put`, `remove`, and updates require acquiring the lock for the specific segment.

- Only the segment being modified is locked, allowing other threads to operate on other segments concurrently.

- This reduces contention and allows multiple threads to perform write operations simultaneously.

6. Resizing:

- ConcurrentHashMap supports dynamic resizing to maintain a good load factor and reduce collision probabilities.

- When resizing occurs, the ConcurrentHashMap splits the data from the old segments into new segments, allowing for better distribution and load balancing.

7. Iteration:

- Iterating over a ConcurrentHashMap using an iterator or a for-each loop is safe for concurrent modifications, meaning it will not throw a `ConcurrentModificationException`.

- However, the iterator operates on the snapshot of the data at the time of creation, so it may not reflect concurrent modifications.

In summary, ConcurrentHashMap provides a high level of concurrency by dividing its data into multiple segments and allowing concurrent read and write operations on different segments. It effectively reduces contention and provides excellent performance for concurrent scenarios. It is a recommended choice when you need a thread-safe map implementation with high concurrency and scalability requirements.

## Contract between hashcode and equals method

There is a contract between the `hashCode()` and `equals()` methods in Java, which is essential and ensures consistent behavior when using objects as keys in hash-based data structures like HashMap or HashSet. This contract is defined as follows:

1. If two objects are equal according to the `equals()` method, then their `hashCode()` values should also be equal.

2. If two objects have the same `hashCode()` value, it is not necessary for them to be actually equal. In such cases, the `equals()` method should be used to check for actual object equality.

In simpler terms, this contract states that if two objects are considered equal, their `hashCode()` values should also be equal. However, if the `hashCode()` values are equal, it does not necessarily imply that the objects are actually equal. In such cases, the `equals()` method should be used to perform the actual equality check.

I hope this clarifies the contract between the `hashCode()` and `equals()` methods in Java.

## When to use Hashcode and equals methods in Java

The `hashCode()` and `equals()` methods in Java are used for object comparison and identification. Here's a short note on when to use each method:

1. `hashCode()` Method:

- The `hashCode()` method is used to generate a unique integer value (hash code) representing the object's internal state.

- It is commonly used in hash-based data structures like HashMap, HashSet, and ConcurrentHashMap to efficiently store and retrieve objects.

- When using hash-based data structures, it's crucial to override `hashCode()` if you override the `equals()` method. The `hashCode()` and `equals()` methods must be consistent with each other:

- If two objects are equal (according to `equals()`), their hash codes must be equal as well.

- If two objects have the same hash code, it doesn't necessarily mean they are equal, but they might be, and the `equals()` method should be used to confirm.

2. `equals()` Method:

- The `equals()` method is used to compare two objects for logical equivalence based on their internal state.

- It is used in various scenarios to check whether two objects are considered equal.

- Common use cases include searching for objects in collections (e.g., List.contains()) and determining if an object already exists in a set (e.g., HashSet.add()).

In summary, `hashCode()` is mainly used for efficient storage and retrieval in hash-based data structures, while `equals()` is used for logical equivalence checks in various contexts like searching and element comparison in collections. It's crucial to override both methods correctly to ensure consistent behavior when using objects as keys or elements in collections.

## collection and collections in java

In Java, "collection" and "collections" refer to different concepts:

1. Collection (singular):

A "Collection" in Java is an interface that represents a group of objects known as elements. It is a part of the Java Collections Framework and is used to store, retrieve, manipulate, and manage a group of elements. The Collection interface is a root interface that provides common methods for working with collections. Classes like ArrayList, LinkedList, HashSet, TreeSet, etc., implement the Collection interface. Some of the common methods defined by the Collection interface are `add()`, `remove()`, `contains()`, `size()`, `isEmpty()`, etc.

Here's a simple example of using the Collection interface to create an ArrayList and add elements to it:

```java

Collection<String> names = new ArrayList<>();

names.add("Alice");

names.add("Bob");

names.add("Charlie");

System.out.println(names); // Output: [Alice, Bob, Charlie]

```

2. Collections (plural):

Collections in Java refers to a utility class in the `java.util` package. It provides various static methods for working with collections. The `Collections` class contains static methods that operate on or return collections. These methods allow you to perform tasks such as sorting, searching, synchronizing, finding max or min elements, etc. on different types of collections.

Here's an example of using some methods from the `Collections` class:

```java

List<Integer> numbers = new ArrayList<>();

numbers.add(5);

numbers.add(2);

numbers.add(8);

numbers.add(3);

Collections.sort(numbers);

System.out.println(numbers); // Output: [2, 3, 5, 8]

int maxNumber = Collections.max(numbers);

System.out.println("Max number: " + maxNumber); // Output: Max number: 8

```

In summary, "Collection" (singular) is an interface representing a group of elements, while "Collections" (plural) is a utility class providing static methods to operate on collections.

## List and SET in Java

\*\*List and Set in Java - Short Notes:\*\*

\*\*List:\*\*

- Ordered collection that allows duplicate elements.

- Maintains insertion order, accessed by index.

- Examples: `ArrayList`, `LinkedList`, `Vector`.

- Useful when order matters and duplicates are allowed.

Example:

```java

List<String> fruits = new ArrayList<>();

fruits.add("Apple");

fruits.add("Banana");

fruits.add("Orange");

System.out.println(fruits); // Output: [Apple, Banana, Orange]

```

\*\*Set:\*\*

- Unordered collection that enforces uniqueness (no duplicates).

- Examples: `HashSet`, `TreeSet`, `LinkedHashSet`.

- Useful when uniqueness matters, and order is not significant.

Example:

```java

Set<String> colors = new HashSet<>();

colors.add("Red");

colors.add("Green");

colors.add("Blue");

colors.add("Red"); // No duplicate "Red" is added

System.out.println(colors); // Output: [Green, Blue, Red]

```

Both `List` and `Set` are subtypes of `Collection` and provide different behaviors for different use cases. Lists are suitable when order and duplicates matter, while sets are ideal for ensuring uniqueness in an unordered collection.

## Arraylist and Linkedlist

ArrayList:

- Implements a dynamic array to store elements.

- best choice for Retrieval operation because of Random access interface.

- worst choice for insertion and deletion in the middle because it required several shift in memory.

LinkedList:

- Implemented as a doubly-linked list to store elements.

- best choice for insertion and deletion operation in the middle of the elements.

- Worst choice for the retrieval operation.

## TimeComplexity of Hashmap and Linked List in Java

Time Complexity of HashMap:

- Insertion (**put**), Deletion (**remove**), Search (**get**), Contains (**containsKey**, **containsValue**): O(1) average case, O(n) worst case due to hash collisions.

- Iterating through keys or values: O(n) (linear time complexity based on the number of elements).

Time Complexity of LinkedList:

- Insertion (**addFirst**, **addLast**), Deletion (**removeFirst**, **removeLast**): O(1) (constant time) at the beginning or end of the list.

- Search (get, contains), Access by index (get, set): O(n) (linear time) in the worst case.

- Iterating through elements: O(n) (linear time complexity based on the number of elements).

## Collection for Sorting the elements in JAVA

In Java, the `java.util.Collections` class provides utility methods for sorting elements in various collections. The collections that can be used for sorting elements in Java are:

1. **ArrayList**: An ArrayList can be sorted using the `Collections.sort()` method, which sorts the elements in ascending order based on their natural ordering (if they implement the `Comparable` interface) or using a custom comparator (if provided).

Example of sorting an ArrayList:

```java

import java.util.ArrayList;

import java.util.Collections;

public class ArrayListSortingExample {

public static void main(String[] args) {

ArrayList<Integer> numbers = new ArrayList<>();

numbers.add(5);

numbers.add(3);

numbers.add(8);

numbers.add(1);

numbers.add(4);

System.out.println("Before sorting: " + numbers);

Collections.sort(numbers);

System.out.println("After sorting: " + numbers);

}

}

```

2. **TreeSet**: A TreeSet is a set implementation that automatically maintains the elements in sorted order (based on their natural ordering or a custom comparator). TreeSet is always sorted, and elements are automatically placed in their appropriate position as they are added.

Example of using a TreeSet for sorted elements:

```java

import java.util.TreeSet;

public class TreeSetExample {

public static void main(String[] args) {

TreeSet<Integer> numbers = new TreeSet<>();

numbers.add(5);

numbers.add(3);

numbers.add(8);

numbers.add(1);

numbers.add(4);

System.out.println("Sorted elements in TreeSet: " + numbers);

}

}

```

Both ArrayList and TreeSet can be used for sorting elements, but there are differences between them:

- If you need a collection that maintains the order of elements as they are inserted and supports duplicates, you can use an ArrayList and then sort it using `Collections.sort()`.

- If you need a collection that automatically maintains the elements in sorted order and does not allow duplicates, you can use a TreeSet directly.

## Collection. Sort() internal Working in java

\*\*Java `Collection.sort()` - Short Note with Example:\*\*

The `Collection.sort()` method is used to sort elements in a collection. It internally uses TimSort, a hybrid sorting algorithm that combines merge sort and insertion sort.

\*\*Example:\*\*

```java

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class Main {

public static void main(String[] args) {

List<Integer> numbers = new ArrayList<>();

numbers.add(5);

numbers.add(2);

numbers.add(8);

numbers.add(1);

numbers.add(4);

System.out.println("Original list: " + numbers);

Collections.sort(numbers);

System.out.println("Sorted list: " + numbers);

}

}

```

\*\*Output:\*\*

```

Original list: [5, 2, 8, 1, 4]

Sorted list: [1, 2, 4, 5, 8]

```

In this example, we use `Collections.sort()` to sort a list of integers in ascending order. The method modifies the original list to produce a sorted list. If the elements do not implement `Comparable`, a custom comparator can be used to specify the sorting order.

## What is Rehashing in JAVA

Rehashing in Java:

- Occurs when the load factor of a hash table exceeds a threshold (default is 0.75).

- Triggers resizing of the hash table to accommodate more elements efficiently.

- Steps: create a larger hash table, recalculate hash codes, and reinsert elements into the new table.

- Maintains a balance between elements and buckets, preventing excessive collisions and maintaining performance.

## Tree set in Collection

\*\*Java `TreeSet` - Short Notes with Example:\*\*

`TreeSet` is a class that implements the `SortedSet` interface, representing a collection of elements stored in sorted order.

\*\*Key Features:\*\*

1. **Sorted Order**: Elements are arranged in ascending order based on natural ordering or a custom comparator.

2. **No Duplicates**: Only unique elements are allowed; duplicates are automatically removed.

3. Efficient Operations: Provides efficient add, remove, and search operations with O(log n) complexity.

4. Backed by Red-Black Tree: Internally uses a Red-Black Tree for efficient sorting and balancing.

\*\*Example:\*\*

```java

import java.util.TreeSet;

public class Main {

public static void main(String[] args) {

TreeSet<Integer> numbers = new TreeSet<>();

numbers.add(10);

numbers.add(5);

numbers.add(15);

numbers.add(2);

numbers.add(8);

System.out.println("TreeSet elements: " + numbers);

numbers.remove(5);

System.out.println("After removing 5: " + numbers);

System.out.println("Smallest element: " + numbers.first());

System.out.println("Largest element: " + numbers.last());

}

}

```

\*\*Output:\*\*

```

TreeSet elements: [2, 5, 8, 10, 15]

After removing 5: [2, 8, 10, 15]

Smallest element: 2

Largest element: 15

```

In this example, we create a `TreeSet` of integers and add elements to it. The elements are automatically sorted in ascending order. We then remove an element and demonstrate how to retrieve the smallest and largest elements using the `first()` and `last()` methods. If elements don't implement `Comparable`, a custom comparator must be provided.

## Comparable and comparator interface

Sure, here are small notes with code examples for `Comparable` and `Comparator` interfaces in Java:

1. **Comparable Interface**:

- Used to define the natural ordering of objects within the class itself.

- Class must implement the `Comparable` interface and override the `compareTo()` method.

- Sorting using `Collections.sort()` or `Arrays.sort()` without specifying a comparator.

Example:

```java

class Student implements Comparable<Student> {

private int rollNumber;

@Override

public int compareTo(Student other) {

return this.rollNumber - other.rollNumber;

}

}

public class ComparableExample {

public static void main(String[] args) {

List<Student> students = new ArrayList<>();

students.add(new Student(101));

students.add(new Student(102));

students.add(new Student(100));

Collections.sort(students);

for (Student student : students) {

System.out.println(student.getRollNumber());

}

}

}

```

2. Comparator Interface:

- Used to define custom sorting logic for objects without modifying the class itself.

- Create a separate class implementing the `Comparator` interface and override the `compare()` method.

- Sorting using `Collections.sort()` or `Arrays.sort()` with the custom comparator.

Example:

```java

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

class Student {

private int rollNumber;

// Constructors, getters, and setters...

}

class RollNumberComparator implements Comparator<Student> {

@Override

public int compare(Student student1, Student student2) {

return student1.getRollNumber() - student2.getRollNumber();

}

}

public class ComparatorExample {

public static void main(String[] args) {

List<Student> students = new ArrayList<>();

students.add(new Student(101));

students.add(new Student(102));

students.add(new Student(100));

Collections.sort(students, new RollNumberComparator());

for (Student student : students) {

System.out.println(student.getRollNumber());

}

}

}

```

In both examples, we have a `Student` class representing students with a `rollNumber`. The first example uses the `Comparable` interface to define natural ordering based on roll numbers within the class itself. The second example uses the `Comparator` interface with a custom `RollNumberComparator` class to sort students based on their roll numbers externally.

## New Features in JAVA 8.

Java 8, released in March 2014, introduced several significant features and enhancements to the Java programming language. Some of the key features introduced in Java 8 are:

1. **Lambda Expressions**: Lambda expressions allow you to write concise and functional-style code. They enable you to treat functionality as a method argument, making it easier to work with collections and functional interfaces.

Example of a simple lambda expression:

```java

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

names.forEach(name -> System.out.println(name));

```

2. **Functional Interfaces**: Java 8 introduced functional interfaces, which are interfaces with only one abstract method. They are used in conjunction with lambda expressions to enable functional programming in Java.

Example of a functional interface:

```java

@FunctionalInterface

interface MyFunctionalInterface {

void doSomething();

}

```

3. **Stream API:** The Stream API provides a more declarative way to work with collections in Java. It allows you to process collections in a functional-style, enabling operations like filtering, mapping, and reducing.

Example of using Stream API:

```java

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

int sum = numbers.stream()

.filter(n -> n % 2 == 0)

.mapToInt(n -> n)

.sum();

System.out.println("Sum of even numbers: " + sum);

```

4. **Default Methods**: Java 8 introduced the concept of default methods in interfaces. These methods provide a way to add new methods to existing interfaces without breaking the implementing classes.

Example of a default method in an interface:

```java

interface MyInterface {

void doSomething();

default void doSomethingElse() {

System.out.println("Doing something else.");

}

}

```

5. **Method References**: Method references allow you to reference methods without invoking them. They are a shorthand notation for lambda expressions when calling existing methods.

Example of method reference:

```java

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

names.forEach(System.out::println);

```

6. **Date and Time API**: Java 8 introduced a new Date and Time API (`java.time`) that provides a more comprehensive and flexible way to work with dates and times, overcoming the limitations of the old `java.util.Date` and `java.util.Calendar` classes.

Example of using the Date and Time API:

```java

LocalDate currentDate = LocalDate.now();

System.out.println("Current date: " + currentDate);

```

7. **Optional**: The `Optional` class provides a way to handle null values more effectively, avoiding potential `NullPointerException`.

Example of using Optional:

```java

Optional<String> optionalValue = Optional.ofNullable(someValue);

optionalValue.ifPresent(value -> System.out.println("Value is present: " + value));

```

These are some of the major features introduced in Java 8. These additions brought significant improvements to the Java language, making it more expressive, flexible, and aligned with modern programming paradigms.

## Streams and method in Streams

Java 8 mein "Streams" ek powerful feature hai jo collection processing aur data manipulation ko simplify karta hai. Stream ek sequence of elements represent karta hai, jise hum operations ke chain ke through process kar sakte hain. Stream ek ek element ko process karta hai, usko transform karta hai, filter karta hai, aur saath mein aggregation (sum, average, count, etc.) jaise operations apply kar sakta hai.

Java 8 mein Streams ke kuch important methods hain:

1. \*\*filter\*\*: Is method se hum Stream ke elements ko filter kar sakte hain based on a specific condition.

Example:

```java

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

List<Integer> evenNumbers = numbers.stream()

.filter(n -> n % 2 == 0)

.collect(Collectors.toList());

System.out.println(evenNumbers); // Output: [2, 4, 6, 8, 10]

```

2. \*\*map\*\*: Is method se hum Stream ke elements ko transform kar sakte hain.

Example:

```java

List<String> names = Arrays.asList("John", "Jane", "Mike");

List<String> upperCaseNames = names.stream()

.map(String::toUpperCase)

.collect(Collectors.toList());

System.out.println(upperCaseNames); // Output: [JOHN, JANE, MIKE]

```

3. \*\*collect\*\*: Is method se hum Stream ke elements ko collect kar sakte hain ek collection mein.

Example:

```java

List<String> names = Arrays.asList("John", "Jane", "Mike");

Set<String> nameSet = names.stream()

.collect(Collectors.toSet());

System.out.println(nameSet); // Output: [John, Jane, Mike]

```

4. \*\*reduce\*\*: Is method se hum Stream ke elements ko combine karke ek single result generate kar sakte hain.

Example:

```java

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

int sum = numbers.stream()

.reduce(0, (a, b) -> a + b);

System.out.println(sum); // Output: 15 (1 + 2 + 3 + 4 + 5)

```

5. \*\*distinct\*\*: Is method se hum Stream ke duplicate elements ko remove kar sakte hain.

Example:

```java

List<Integer> numbers = Arrays.asList(1, 2, 2, 3, 3, 3, 4, 4, 4, 4);

List<Integer> distinctNumbers = numbers.stream()

.distinct()

.collect(Collectors.toList());

System.out.println(distinctNumbers); // Output: [1, 2, 3, 4]

```

Is tarah se, Streams ke kai aur methods hain jaise `sorted`, `forEach`, `limit`, `anyMatch`, `allMatch`, `noneMatch` etc. Jo hume data manipulation mein madad karte hain aur collection processing ko easy banate hain.

## Collections Vs stream

Java mein, `Collections` aur `Stream` do alag concepts hain jo data ko manipulate aur process karne ke liye istemal kiye jate hain, lekin inke alag-alag upayog aur gun hote hain.

1. Collections:

- Collections Java mein ek group of elements ko represent karte hain, jaise ki lists, sets, ya maps, aur inme se elements par various operations perform karne ke liye alag-alag methods provide karte hain. Collections data ko structured aur organized tarike se store, retrieve, aur manipulate karne ke liye istemal hote hain.

- Java mein kuch common collections ke udaharan hain `ArrayList`, `HashSet`, `HashMap`, etc.

Collections ka istemal karne ka udaharan:

```java

import java.util.ArrayList;

import java.util.List;

public class Main {

public static void main(String[] args) {

List<String> names = new ArrayList<>();

names.add("John");

names.add("Jane");

names.add("Alice");

// List mein sabhi names ko print karein

for (String name : names) {

System.out.println(name);

}

}

}

```

Output:

```

John

Jane

Alice

```

2. Stream:

- Java mein Streams functional approach provide karte hain collection of data ko process karne ke liye. Streams allows karte hain various operations (e.g., filter, map, reduce) ko collection ke elements par declarative aur concise tarike se perform karne ke liye. Streams khud data ko store nahi karte hain, balki ek sequence of elements represent karte hain jo data source se aata hai, jaise ki collection ya kisi aur source jaise files ya network.

- Streams bulk data processing tasks ko efficiently perform karne ke liye istemal hote hain aur unka parallel processing ke liye upayog bhi kiya ja sakta hai.

Streams ka istemal karne ka udaharan:

```java

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class Main {

public static void main(String[] args) {

List<String> names = Arrays.asList("John", "Jane", "Alice");

// "J" se shuru hone wale names ko filter karein aur unhe ek new list mein collect karein

List<String> filteredNames = names.stream()

.filter(name -> name.startsWith("J"))

.collect(Collectors.toList());

// Filtered names ko print karein

filteredNames.forEach(System.out::println);

}

}

```

Output:

```

John

Jane

```

Saransh mein, Collections data ko store aur manipulate karne ke liye istemal hote hain, providing methods jaise `add`, `remove`, aur `get`, jabki Streams declarative aur functional data processing ke liye istemal hote hain, allowing operations jaise `filter`, `map`, aur `collect`. Dono Collections aur Streams Java mein alag-alag upayog karne ke liye valuable tools hain, aur inme se chunav data ke prakriti aur karno par nirbhar karta hai jo aapko perform karne wale operations hain.

## Filter, Map streams in JAVA 8

\*\*Java 8 Stream API: `filter()`\*\*

The `filter()` method in the Stream API is used to filter elements in a stream based on a specified condition. It takes a predicate as an argument, which is a functional interface that evaluates whether an element satisfies the given condition. Elements that pass the condition are included in the new filtered stream.

Example:

```java

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

Stream<Integer> evenNumbersStream = numbers.stream().filter(n -> n % 2 == 0);

```

In this example, we have a list of numbers, and we create a stream from it. We then use the `filter()` method to filter out even numbers. The resulting stream `evenNumbersStream` will contain only the even numbers from the original list.

\*\*Java 8 Stream API: `map()`\*\*

The `map()` method in the Stream API is used to transform each element in a stream into another element. It takes a `Function` as an argument, which is a functional interface that defines how to convert an element of type `T` to an element of type `R`. The resulting stream will contain the transformed elements.

Example:

```java

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

Stream<Integer> nameLengthsStream = names.stream().map(String::length);

```

In this example, we have a list of names, and we create a stream from it. We then use the `map()` method to transform each name into its length (an integer). The resulting stream `nameLengthsStream` will contain the lengths of the names.

You can use these intermediate operations in combination with other stream operations, such as `forEach()`, `collect()`, or another intermediate operation, to perform more complex data processing tasks. By chaining multiple intermediate operations, you can create a pipeline that efficiently processes data in a functional and declarative style.

## Internal working of Streams

The internal working of Streams in Java involves several key concepts and mechanisms that enable efficient and flexible data processing. The Stream API is designed to work in a functional and declarative style, allowing you to perform various operations on the data in a streamlined and parallelizable manner. Here are the main aspects of the internal working of Streams:

1. Source: A Stream is created from a data source, which can be a collection, an array, or an I/O channel. The source provides the elements that will be processed by the stream.

2. Intermediate Operations: Intermediate operations are the functional-style operations that can be applied to the stream. These operations include `filter()`, `map()`, `sorted()`, `distinct()`, and more. Intermediate operations do not modify the original data source but produce a new stream with the modified elements. These operations are typically lazy, meaning they are not executed until a terminal operation is invoked.

3. Pipeline: A sequence of intermediate operations forms a pipeline. When you chain multiple intermediate operations together, you create a data processing pipeline that will be applied sequentially to the elements in the stream.

4. Short-circuiting: Some intermediate operations, like `limit()` and `findFirst()`, are short-circuiting. It means that they can stop processing the data as soon as the desired condition is met, without processing all the elements in the stream.

5. Terminal Operation: A terminal operation is an operation that triggers the processing of the stream and produces a non-stream result, such as a value, a collection, or a side effect. Terminal operations are eagerly evaluated, and they are the ones that initiate the data processing. Common terminal operations include `forEach()`, `collect()`, `reduce()`, and `count()`.

6. Lazy Evaluation: Streams use lazy evaluation, meaning that intermediate operations do not execute their functionality until a terminal operation is called. This allows streams to optimize the processing and avoid unnecessary computations.

7. Parallel Execution: The Stream API allows for parallel execution of operations on streams. When applicable, streams can be processed in parallel, taking advantage of multiple cores and improving performance for large datasets. Parallel processing is achieved using a fork/join pool, which automatically divides the stream into substreams and processes them concurrently.

By combining these concepts, the Stream API in Java provides a powerful and expressive way to work with data in a functional style. The internal mechanisms ensure that data processing is efficient, optimized, and can be easily parallelized for improved performance when dealing with large datasets.

## Flatmap vs Map JAVA 8

\*\*Java 8 Stream API: `map()`\*\*

The `map()` method in the Stream API is used to transform each element in a stream into another element based on a given function. It takes a `Function` as an argument, which maps an element of type `T` to an element of type `R`. The resulting stream will contain the transformed elements of type `R`.

Example:

```java

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

Stream<Integer> nameLengthsStream = names.stream().map(String::length);

```

In this example, the `map()` method transforms each name in the stream to its length (an integer).

\*\*Java 8 Stream API: `flatMap()`\*\*

The `flatMap()` method in the Stream API is used to flatten nested streams or to transform each element in a stream into multiple elements (zero or more) of another stream. It takes a `Function` as an argument, which maps an element of type `T` to a stream of elements of type `R`. The resulting stream will contain all the elements from all the individual streams returned by the function.

Example:

```java

List<List<Integer>> listOfLists = Arrays.asList(Arrays.asList(1, 2), Arrays.asList(3, 4), Arrays.asList(5, 6));

Stream<Integer> flattenedStream = listOfLists.stream().flatMap(List::stream);

```

In this example, the `flatMap()` method flattens the nested list of lists into a single stream containing all the individual integers.

\*\*Difference between `map()` and `flatMap()`\*\*

- The key difference lies in their output types:

- `map()` produces a one-to-one mapping of elements, where each input element is transformed into one output element.

- `flatMap()` produces a one-to-many mapping of elements, where each input element can be transformed into zero, one, or multiple output elements.

- Use `map()` when you want to transform each element individually into another element.

- Use `flatMap()` when you want to transform each element into a stream and then flatten the resulting streams into a single stream.

\*\*Example combining `map()` and `flatMap()`\*\*

```java

List<List<String>> listOfLists = Arrays.asList(

Arrays.asList("apple", "orange"),

Arrays.asList("banana", "grape"),

Arrays.asList("cherry", "kiwi")

);

Stream<String> flattenedFruitsStream = listOfLists.stream()

.flatMap(list -> list.stream().map(String::toUpperCase));

// Output: APPLE, ORANGE, BANANA, GRAPE, CHERRY, KIWI

```

In this example, we use `flatMap()` to transform each inner list of strings into a stream and then use `map()` to convert each string to uppercase. The resulting stream contains all the uppercase strings from all the inner lists.

## Sorting and chain sorting JAVA 8

\*\*Java 8 Stream API: Sorting with `sorted()`\*\*

The `sorted()` method in the Stream API is an intermediate operation used to sort the elements of a stream based on their natural order (for elements that implement the `Comparable` interface) or using a custom comparator.

Example of sorting using `sorted()`:

```java

List<String> names = Arrays.asList("Alice", "Charlie", "Bob");

List<String> sortedNames = names.stream()

.sorted()

.collect(Collectors.toList());

// Output: [Alice, Bob, Charlie]

```

In this example, the `sorted()` method sorts the names in the natural order, resulting in the list `[Alice, Bob, Charlie]`.

\*\*Java 8 Stream API: Chaining Sorting with `sorted()` and `thenComparing()`\*\*

You can chain multiple sorting criteria using the `sorted()` method and the `thenComparing()` method. This allows you to perform secondary and tertiary sorting on top of the initial sorting.

Example of chaining sorting using `sorted()` and `thenComparing()`:

```java

class Person {

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

public String getName() {

return name;

}

public int getAge() {

return age;

}

}

List<Person> people = Arrays.asList(

new Person("Alice", 25),

new Person("Charlie", 20),

new Person("Bob", 30)

);

List<Person> sortedPeople = people.stream()

.sorted(Comparator.comparing(Person::getName)

.thenComparing(Person::getAge))

.collect(Collectors.toList());

// Output: [Alice (25), Bob (30), Charlie (20)]

```

In this example, we have a custom class `Person` with name and age attributes. We sort the list of `Person` objects first by their names using `Comparator.comparing()`, and then for people with the same name, we sort them by their ages using `thenComparing()`. The resulting list `sortedPeople` contains people sorted by name and then by age.

Chaining sorting is powerful and allows you to sort elements based on multiple criteria, giving you more control over the sorting order.

## Functional Interface and Different types of Functional interface in JAVA

In Java, a functional interface is an interface that contains only one abstract method. Functional interfaces are used to enable functional programming in Java and to work seamlessly with lambda expressions and method references. The `@FunctionalInterface` annotation is often used to mark a functional interface explicitly, although it is not strictly required.

Functional interfaces provide a way to treat functionality as a method argument, allowing for a more expressive and concise coding style when working with lambdas and streams.

Different types of functional interfaces in Java include:

1. `Runnable`: This is a functional interface that represents a task that can be executed concurrently. It has a single abstract method `run()` without any arguments and returns void.

2. `Supplier<T>`: Represents a supplier of results, providing a single abstract method `get()` that takes no arguments and returns a value of type `T`.

3. `Consumer<T>`: Represents an operation that takes an input of type `T` and returns no result. It has a single abstract method `accept(T t)`.

4. `Function<T, R>`: Represents a function that takes an argument of type `T` and returns a result of type `R`. It has a single abstract method `apply(T t)`.

5. `Predicate<T>`: Represents a predicate (a boolean-valued function) of one argument. It has a single abstract method `test(T t)`.

6. `BiFunction<T, U, R>`: Represents a function that takes two arguments of types `T` and `U` and returns a result of type `R`. It has a single abstract method `apply(T t, U u)`.

7. `BinaryOperator<T>`: Represents an operation upon two operands of the same type, producing a result of the same type as the operands. It has a single abstract method `apply(T t1, T t2)`.

8. `UnaryOperator<T>`: Represents an operation upon a single operand of type `T`, producing a result of the same type. It has a single abstract method `apply(T t)`.

9. `BiPredicate<T, U>`: Represents a predicate that takes two arguments of types `T` and `U` and returns a boolean result. It has a single abstract method `test(T t, U u)`.

10. `BiConsumer<T, U>`: Represents an operation that takes two input arguments of types `T` and `U`, and returns no result. It has a single abstract method `accept(T t, U u)`.

These are some of the common functional interfaces provided by Java. Additionally, developers can define their custom functional interfaces with a single abstract method to support specific functional requirements in their code.

## Immutable class and how we can define our own immutable class in JAVA.

An immutable class in Java is a class whose instances cannot be modified after creation. To create an immutable class, follow these guidelines:

1. Declare the class as `final`: Prevents the class from being subclassed and modified.

2. Declare the instance variables as `final`: Ensures that their values cannot be changed after object creation.

3. Provide no setter methods: Avoids the ability to modify the object's state.

4. Ensure that mutable objects (like lists) are not directly accessible from outside the class: If the class contains references to mutable objects, ensure they are safely copied during construction to prevent modification.

Example of an immutable class:

```java

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public final class ImmutablePerson {

private final String name;

private final int age;

private final List<String> hobbies; // Mutable list is not directly accessible

public ImmutablePerson(String name, int age, List<String> hobbies) {

this.name = name;

this.age = age;

this.hobbies = new ArrayList<>(hobbies); // Safely copy the list during construction

}

public String getName() {

return name;

}

public int getAge() {

return age;

}

public List<String> getHobbies() {

return Collections.unmodifiableList(hobbies); // Return an unmodifiable view of the list

}

}

```

In this example, we have created an immutable class `ImmutablePerson`. The instance variables `name` and `age` are declared as `final` to make them immutable. The `hobbies` list is safely copied during object construction to prevent external modification, and an unmodifiable view of the list is returned through the `getHobbies()` method.

By following these guidelines, you can ensure that instances of `ImmutablePerson` cannot be changed once created, leading to a more robust and predictable program. Immutable classes are useful in multithreaded environments, caching, and scenarios where you want to ensure the integrity and stability of objects.

## Singleton Design Pattern and How to write singleton class in JAVA.

The Singleton Design Pattern is a creational design pattern that ensures a class has only one instance and provides a global point of access to that instance. To implement the Singleton pattern in Java:

1. Create a private static instance variable of the class.

2. Make the constructor private.

3. Provide a public static method to access the instance and ensure lazy initialization.

Example:

```java

public class Singleton {

private static Singleton instance;

private Singleton() {

// Private constructor to prevent instantiation from outside the class.

}

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton(); // Lazy initialization

}

return instance;

}

}

```

Usage:

```java

public class SingletonUsage {

public static void main(String[] args) {

Singleton instance1 = Singleton.getInstance();

Singleton instance2 = Singleton.getInstance();

System.out.println(instance1 == instance2); // Output: true (both instances are the same)

}

}

```

In this example, the `Singleton` class ensures that only one instance is created. When calling `getInstance()`, it returns the same instance throughout the application, as demonstrated by the comparison of `instance1` and `instance2`, which are indeed the same object.

The Singleton Design Pattern is useful when you need to maintain a single instance for resource management, configuration settings, or a shared central object that multiple parts of the application access. It ensures that the class is instantiated only once, providing a simple and efficient way to manage global resources in your Java applications.

## Break singleton and also prevention

The Singleton Design Pattern ensures that a class has only one instance and provides a global access point to that instance. However, the pattern is not immune to being broken or violated. There are several ways in which the Singleton pattern can be broken:

1. Reflection: Reflection allows accessing private constructors and fields, enabling the creation of multiple instances in a singleton class. A new instance can be created, bypassing the singleton mechanism.

2. Serialization and Deserialization: When a singleton class implements Serializable, it can be serialized and deserialized. During deserialization, a new instance can be created, breaking the singleton pattern.

3. Cloning: If a singleton class implements the Cloneable interface and overrides the clone() method, it is possible to create a copy of the instance, effectively breaking the singleton.

4. Multithreading: In a multithreaded environment, race conditions may occur, leading to the creation of multiple instances if lazy initialization is not handled correctly.

To protect the Singleton pattern from being broken, consider the following techniques:

1. Reflection Prevention: Modify the private constructor to throw an exception if it is called more than once, preventing instantiation via reflection.

2. Serialization Control: Override the readResolve() method to return the same instance during deserialization.

3. Cloning Prevention: Throw an exception in the clone() method to prevent cloning and make the class non-clonable.

4. Thread-Safe Initialization: Use synchronization mechanisms like double-checked locking or static initialization to ensure only one instance is created, even in multithreaded environments.

Example of thread-safe singleton with double-checked locking:

```java

public class ThreadSafeSingleton {

private static volatile ThreadSafeSingleton instance;

private ThreadSafeSingleton() {

// Private constructor to prevent instantiation from outside the class.

}

public static ThreadSafeSingleton getInstance() {

if (instance == null) {

synchronized (ThreadSafeSingleton.class) {

if (instance == null) {

instance = new ThreadSafeSingleton();

}

}

}

return instance;

}

}

```

By applying these techniques, you can protect the Singleton pattern from being broken and ensure that the class remains a true singleton with a single instance throughout the application's lifecycle.

## Throw and throws Difference

1. `throw`:

- Used to explicitly throw an exception within a method.

- Syntax: `throw new ExceptionType("Optional message");`

- Exception will be caught and handled by an appropriate catch block in the calling method or up the call stack.

Example:

```java

public class ThrowExample {

public static void main(String[] args) {

int age = -5;

if (age < 0) {

throw new IllegalArgumentException("Age cannot be negative.");

}

}

}

```

2. `throws`:

- Used in method signatures to indicate that the method may throw checked exceptions.

- Syntax: `void methodName() throws ExceptionType { ... }`

- Caller must handle the declared exception or propagate it further up the call stack.

Example:

```java

import java.io.IOException;

public class ThrowsExample {

public void doSomething() throws IOException {

// Code that may throw IOException

}

public static void main(String[] args) {

ThrowsExample example = new ThrowsExample();

try {

example.doSomething();

} catch (IOException e) {

// Handle the exception here

}

}

}

```

In both examples, `throw` is used to raise an `IllegalArgumentException` when the age is negative, and `throws` is used in the method signature to indicate that the `doSomething()` method may throw an `IOException`.

## New things added to JVM in java 8

PermGen Removal (Metaspace):

In Java 8, the PermGen (Permanent Generation) space was removed from the JVM and replaced with the Metaspace memory region.

Metaspace is more flexible and automatically resizes as needed, avoiding issues related to OutOfMemoryError due to PermGen space exhaustion.

## What is exception table in jvm

- The "exception table" in the JVM tracks mappings between bytecode instructions and exception handlers.

- Created by the Java compiler during bytecode generation.

- Stores ranges of bytecode where exceptions can be thrown and the corresponding exception handler locations.

- Helps the JVM identify the appropriate handler to manage thrown exceptions during execution.

## What are memory block in jvm

- Heap Memory: Largest memory block, used for dynamic memory allocation, stores objects and arrays.

- Young Generation: Part of the heap for new object allocation, divided into Eden Space and Survivor Spaces.

- Old Generation: Part of the heap for long-lived objects, survivors from Young Generation.

- PermGen (Java 7 and earlier) / Metaspace (Java 8 and later): Stores class metadata and JVM-specific data.

- Stack Memory: Each thread has its own stack for method call frames and local variables.

- Native Memory: Memory outside JVM's control, used for JVM itself and native code or libraries.

## String, stringbuffer, stringbuilder

1. `String`:

- Immutable class, value cannot be changed after creation.

- Creates a new `String` object for each operation like concatenation.

- Thread-safe due to immutability.

Example:

```java

String str = "Hello";

str = str + " World"; // A new String object is created with the concatenated value.

```

2. `StringBuffer`:

- Mutable class for handling character sequences.

- Modifies, appends, inserts, and deletes characters in place without creating new objects.

- Synchronized and thread-safe.

Example:

```java

StringBuffer sb = new StringBuffer("Hello");

sb.append(" World"); // The original StringBuffer is modified in place.

```

3. `StringBuilder`:

- Similar to `StringBuffer` but not synchronized.

- Efficient for single-threaded scenarios or when external synchronization is handled.

- Faster than `StringBuffer`.

Example:

```java

StringBuilder sb = new StringBuilder("Hello");

sb.append(" World"); // The original StringBuilder is modified in place.

```

In summary:

- Use `String` for immutability and infrequent modifications.

- Use `StringBuffer` for thread-safety and concurrent access.

- Use `StringBuilder` for single-threaded performance or external synchronization.

## OOPS concepts – abstraction, encapsulation, inheritance, polymorphism

Object-Oriented Programming (OOP) is a programming paradigm that revolves around the concept of objects, which represent real-world entities and their interactions. The four main pillars of OOP are abstraction, encapsulation, inheritance, and polymorphism. Let's explore each of these concepts with examples:

1. Abstraction:

Abstraction is the process of simplifying complex real-world entities by modeling them as objects with essential characteristics and behaviors, while hiding unnecessary details. It allows you to focus on the relevant aspects of an object and ignore the irrelevant ones.

Example:

```java

abstract class Shape {

public abstract double calculateArea();

}

class Circle extends Shape {

private double radius;

public Circle(double radius) {

this.radius = radius;

}

@Override

public double calculateArea() {

return Math.PI \* radius \* radius;

}

}

class Rectangle extends Shape {

private double length;

private double width;

public Rectangle(double length, double width) {

this.length = length;

this.width = width;

}

@Override

public double calculateArea() {

return length \* width;

}

}

public class AbstractionExample {

public static void main(String[] args) {

Shape circle = new Circle(5.0);

Shape rectangle = new Rectangle(4.0, 3.0);

System.out.println("Area of Circle: " + circle.calculateArea());

System.out.println("Area of Rectangle: " + rectangle.calculateArea());

}

}

```

In this example, we use abstraction to represent different shapes (Circle and Rectangle) as objects of the abstract class `Shape`. We only define the common behavior of calculating the area in the `Shape` class, while the specific implementation details are hidden in the concrete subclasses.

2. Encapsulation:

Encapsulation is the bundling of data (attributes) and methods (behaviors) that operate on the data within a single unit (an object). It hides the internal details of an object and exposes only the necessary functionalities through methods, ensuring data security and integrity.

Example:

```java

class BankAccount {

private String accountNumber;

private double balance;

public BankAccount(String accountNumber, double balance) {

this.accountNumber = accountNumber;

this.balance = balance;

}

public void deposit(double amount) {

if (amount > 0) {

balance += amount;

}

}

public void withdraw(double amount) {

if (amount > 0 && amount <= balance) {

balance -= amount;

}

}

public double getBalance() {

return balance;

}

}

public class EncapsulationExample {

public static void main(String[] args) {

BankAccount account = new BankAccount("12345", 1000.0);

account.deposit(500.0);

account.withdraw(200.0);

System.out.println("Account Balance: " + account.getBalance());

}

}

```

In this example, we encapsulate the BankAccount details (accountNumber and balance) and the associated behaviors (deposit, withdraw) within the `BankAccount` class. The access to the data is controlled through methods, ensuring that the account balance cannot be directly modified from outside the class.

3. Inheritance:

Inheritance is a mechanism in which a new class (subclass or derived class) is created based on an existing class (superclass or base class). The subclass inherits the properties and behaviors of the superclass, allowing code reuse and hierarchical organization of classes.

Example:

```java

class Animal {

void makeSound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

@Override

void makeSound() {

System.out.println("Dog barks");

}

void wagTail() {

System.out.println("Dog wags its tail");

}

}

public class InheritanceExample {

public static void main(String[] args) {

Animal animal = new Animal();

animal.makeSound(); // Output: Animal makes a sound

Dog dog = new Dog();

dog.makeSound(); // Output: Dog barks

dog.wagTail(); // Output: Dog wags its tail

}

}

```

In this example, we create an `Animal` class with a `makeSound()` method. The `Dog` class is a subclass of `Animal` and inherits the `makeSound()` method. The `Dog` class also has its specific method, `wagTail()`, which is not present in the `Animal` class.

4. Polymorphism:

Polymorphism allows an object to take on multiple forms. It refers to the ability of a method to behave differently based on the object calling it or the arguments passed to it. Polymorphism can be achieved through method overloading and method overriding.

Example:

```java

class Calculator {

int add(int a, int b) {

return a + b;

}

double add(double a, double b) {

return a + b;

}

}

public class PolymorphismExample {

public static void main(String[] args) {

Calculator calculator = new Calculator();

System.out.println("Sum of integers: " + calculator.add(2, 3)); // Output: Sum of integers: 5

System.out.println("Sum of doubles: " + calculator.add(2.5, 3.5)); // Output: Sum of doubles: 6.0

}

}

```

In this example, we have a `Calculator` class with two `add()` methods, one for integers and another for doubles. The method overloading allows the `add()`

method to behave differently based on the argument types, demonstrating polymorphism.

These four OOP concepts—abstraction, encapsulation, inheritance, and polymorphism—are fundamental to object-oriented programming and enable the creation of modular, maintainable, and flexible software systems.

## Final, Finally and Finalize

1. `final`:

- Used to declare immutable variables, unmodifiable methods, or non-inheritable classes.

- Variable: `final int number = 42;`

- Method: `public final void printMessage() { ... }`

- Class: `public final class FinalClass { ... }`

2. `finally`:

- Used in exception handling along with `try` and `catch` blocks.

- Executes regardless of whether an exception occurs or not.

- Used for cleanup tasks like closing resources.

Example:

```java

import java.io.\*;

public class FinallyExample {

public static void main(String[] args) {

BufferedReader reader = null;

try {

reader = new BufferedReader(new FileReader("file.txt"));

// Code to read data from the file

} catch (FileNotFoundException e) {

System.err.println("File not found: " + e.getMessage());

} finally {

// Close the file reader in the 'finally' block to ensure it is always closed.

try {

if (reader != null) {

reader.close();

}

} catch (IOException e) {

System.err.println("Error while closing file: " + e.getMessage());

}

}

}

}

```

3. `finalize`:

- Deprecated and not recommended for resource cleanup in modern Java versions.

- Called by the garbage collector before an object is garbage collected.

- Overridable to perform cleanup actions before object disposal.

Example:

```java

public class FinalizeExample {

@Override

protected void finalize() throws Throwable {

// Perform cleanup actions before the object is garbage collected.

System.out.println("Finalize method is called for this object.");

super.finalize();

}

}

```

Note: Instead of using `finalize`, prefer try-with-resources or other resource management techniques for critical cleanup tasks.

## Throws Vs Try catch Block

1. `throws`:

- Used in method signatures to indicate that the method may throw checked exceptions.

- Caller must handle the declared exception or propagate it further up the call stack.

Example:

```java

import java.io.IOException;

public class ThrowsExample {

public void readFile() throws IOException {

// Code that reads a file and may throw an IOException

}

public static void main(String[] args) {

ThrowsExample example = new ThrowsExample();

try {

example.readFile();

} catch (IOException e) {

// Handle the IOException here or propagate it up the call stack

}

}

}

```

2. `try-catch` Block:

- Used to catch and handle exceptions locally within a block of code.

- Enclose the code that may throw an exception in the `try` block and handle the exception in the `catch` block.

Example:

```java

import java.io.IOException;

public class TryCatchExample {

public void readFile() {

try {

// Code that reads a file and may throw an IOException

} catch (IOException e) {

// Handle the IOException here or provide a fallback behavior

}

}

public static void main(String[] args) {

TryCatchExample example = new TryCatchExample();

example.readFile();

}

}

```

In both examples, `throws` is used in the method signature to indicate that the `readFile()` method may throw an `IOException`. In the first example, the `IOException` is handled in the `main` method using a `try-catch` block, while in the second example, the method itself handles the `IOException` with a `try-catch` block, providing specific error handling.

## == and equals method

1. "==" operator:

- Used to compare object references or actual values for primitive types.

- Checks if two object references point to the same memory location.

- Example: `String str1 = "Hello"; String str2 = "Hello"; str1 == str2; // true`

2. "equals()" method:

- Used to compare the content or meaningful equality of objects.

- Default implementation compares object references, but many classes override it.

- Example: `String str1 = "Hello"; String str2 = "Hello"; str1.equals(str2); // true`

3. Overriding "equals()":

- In custom classes, "equals()" can be overridden to provide specific content-based comparison.

- Example:

```java

class Person {

private String name;

public Person(String name) {

this.name = name;

}

@Override

public boolean equals(Object obj) {

if (this == obj) return true;

if (obj == null || getClass() != obj.getClass()) return false;

Person person = (Person) obj;

return Objects.equals(name, person.name);

}

}

```

By using "==" and "equals()" appropriately, you can perform the desired comparisons in Java based on your requirements.

## Design Patterns

1. \*\***Factory Method Pattern**\*\*:

- Definition: The Factory Method Pattern is a creational design pattern that provides an interface for creating objects but allows subclasses to decide which class to instantiate.

- Explanation: This pattern promotes loose coupling by delegating the responsibility of object creation to subclasses. The client code uses the factory method to create objects, but it does not need to know the exact class that will be instantiated.

- Example: Consider a payment processing system. We have multiple payment methods like credit card and PayPal. The Factory Method Pattern can be used to create instances of these payment methods without the client code knowing the exact class.

@Component

public interface PaymentMethod {

void processPayment(double amount);

}

@Component

public class CreditCardPayment implements PaymentMethod {

public void processPayment(double amount) {

// Process credit card payment logic

}

}

@Component

public class PayPalPayment implements PaymentMethod {

public void processPayment(double amount) {

// Process PayPal payment logic

}

}

@Component

public class PaymentFactory {

public PaymentMethod createPaymentMethod(String paymentType) {

if ("creditCard".equals(paymentType)) {

return new CreditCardPayment();

} else if ("paypal".equals(paymentType)) {

return new PayPalPayment();

}

throw new IllegalArgumentException("Invalid payment type");

}

}

2. \*\***Observer Pattern**\*\*:

- Definition: The Observer Pattern is a behavioral design pattern that establishes a one-to-many dependency between objects, so when one object (subject) changes its state, all its dependents (observers) are notified and updated automatically.

- Explanation: This pattern helps in building a reactive system, where multiple objects are interested in changes to a particular object's state. The subject maintains a list of its observers and notifies them when its state changes.

- Example: In a shopping application, whenever a new order is placed, the OrderService (subject) notifies all registered EmailNotification and SMSNotification (observers) about the new order.

@Component

public class OrderService {

private List<OrderObserver> observers = new ArrayList<>();

public void addObserver(OrderObserver observer) {

observers.add(observer);

}

public void placeOrder(Order order) {

// Place the order logic

// Notify all observers after placing the order

observers.forEach(observer -> observer.update(order));

}

}

@Component

public interface OrderObserver {

void update(Order order);

}

@Component

public class EmailNotification implements OrderObserver {

public void update(Order order) {

// Send email notification logic

}

}

3. \*\***Proxy Pattern**\*\*:

- Definition: The Proxy Pattern is a structural design pattern that provides a surrogate or placeholder for another object to control access to it.

- Explanation: The proxy acts as an intermediary, allowing additional actions to be performed when accessing the real object. It can be used to add extra functionality like lazy initialization, access control, or logging.

- Example: Suppose you have a large image that takes time to load from disk. The ProxyImage acts as a placeholder for the RealImage and loads it from disk only when needed, avoiding unnecessary loading and improving performance.

public interface Image {

void display();

}

public class RealImage implements Image {

private String fileName;

public RealImage(String fileName) {

this.fileName = fileName;

loadFromDisk(fileName);

}

public void display() {

// Display the image logic

}

private void loadFromDisk(String fileName) {

// Load image from disk logic

}

}

public class ProxyImage implements Image {

private RealImage realImage;

private String fileName;

public ProxyImage(String fileName) {

this.fileName = fileName;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(fileName);

}

realImage.display();

}

}

These are brief explanations and examples of three design patterns. Each design pattern has specific use cases, and they help in solving common design problems and improving the overall design of software systems. By using design patterns, you can create more maintainable, scalable, and flexible software solutions.

## Difference between abstract class and interface

Abstract Class:

- Use an abstract class when you want to provide a common base implementation for related classes.

- Use when sharing code among multiple classes with a strong "is-a" relationship.

- Can have both abstract and non-abstract methods, allowing partial implementation.

- Subclasses extend the abstract class using the `extends` keyword.

Example of an abstract class:

```java

abstract class Shape {

public void displayArea() {

System.out.println("Area: " + calculateArea());

}

public abstract double calculateArea();

}

```

Interface:

- Use an interface when you want to define a contract for unrelated classes.

- Use when achieving multiple inheritance-like behavior, as a class can implement multiple interfaces.

- Contains only method signatures, defining what a class should do, but not how.

- A class implements an interface using the `implements` keyword.

Example of an interface:

```java

interface Drawable {

void draw();

}

```

Using Both Abstract Class and Interface:

```java

abstract class Vehicle {

public void start() {

System.out.println("Vehicle is starting");

}

public abstract void stop();

}

interface AirTransport {

void fly();

}

class Airplane extends Vehicle implements AirTransport {

@Override

public void stop() {

System.out.println("Airplane is stopping");

}

@Override

public void fly() {

System.out.println("Airplane is flying");

}

}

```

In this example, `Vehicle` is an abstract class providing common behavior, and `AirTransport` is an interface defining the `fly()` method. The `Airplane` class extends the `Vehicle` abstract class and implements the `AirTransport` interface, showing how both abstract classes and interfaces can be used together to achieve flexibility and code reusability.

## Singleton in case of thread

Sure! Here's a short note with code examples on two commonly used approaches to create a thread-safe singleton:

1. Using Double-Checked Locking (DCL):

```java

public class Singleton {

private static volatile Singleton instance;

private Singleton() {

// Private constructor

}

public static Singleton getInstance() {

if (instance == null) {

synchronized (Singleton.class) {

if (instance == null) {

instance = new Singleton();

}

}

}

return instance;

}

}

```

In this approach, we use double-checked locking with the `synchronized` keyword inside the `getInstance()` method to ensure that only one thread creates the instance while allowing subsequent calls to access it without synchronization.

2. Using Initialization-on-Demand Holder (Bill Pugh Singleton):

```java

public class Singleton {

private Singleton() {

// Private constructor

}

private static class SingletonHolder {

private static final Singleton INSTANCE = new Singleton();

}

public static Singleton getInstance() {

return SingletonHolder.INSTANCE;

}

}

```

In this approach, the singleton instance is created as a static final field in a nested static class `SingletonHolder`. The instance is created lazily when the `getInstance()` method is first called, ensuring thread-safe initialization.

Note: In modern Java development, using enums for singletons is also a preferred approach as enums guarantee thread safety and allow only one instance. Here's an example:

```java

public enum SingletonEnum {

INSTANCE;

// Add other methods and properties as needed

}

```

Using any of these approaches, you can create a thread-safe singleton, ensuring that only one instance of the class is created even in a multi-threaded environment.

## How to define thread and Thread Lifecycle in JAVA.

In Java, a thread is a lightweight unit of execution that allows concurrent execution of tasks within a single process. Each thread has its own call stack, program counter, and local variables, allowing it to run independently from other threads. Threads enable multi-threading, which can improve the performance of applications that can benefit from parallelism.

Thread Lifecycle:

The lifecycle of a thread in Java consists of several states, and a thread can transition between these states as it executes its tasks. The different states in the lifecycle are as follows:

1. New: When a thread is created, it is in the new state. It means the thread object is created, but its `start()` method hasn't been called yet.

2. Runnable: When the `start()` method is called on the thread, it enters the runnable state. The thread is ready to run, but the actual execution depends on the CPU scheduler.

3. Running: When the CPU scheduler selects a thread to run, it enters the running state, and its `run()` method starts executing.

4. Blocked: A thread can move to the blocked state when it is waiting for a certain condition, such as I/O operations or acquiring a lock.

5. Waiting: A thread can move to the waiting state when it is waiting for another thread to perform a specific action.

6. Terminated: A thread enters the terminated state when its `run()` method completes execution or if an unhandled exception occurs within the thread.

Here's an example of how to define a thread and understand its lifecycle:

```java

public class MyThread extends Thread {

@Override

public void run() {

for (int i = 1; i <= 5; i++) {

System.out.println("Thread: " + Thread.currentThread().getId() + ", Value: " + i);

try {

Thread.sleep(500); // Sleep for 500 milliseconds

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public static void main(String[] args) {

MyThread thread1 = new MyThread();

MyThread thread2 = new MyThread();

thread1.start();

thread2.start();

}

}

```

Output:

```

Thread: 12, Value: 1

Thread: 13, Value: 1

Thread: 12, Value: 2

Thread: 13, Value: 2

Thread: 12, Value: 3

Thread: 13, Value: 3

Thread: 12, Value: 4

Thread: 13, Value: 4

Thread: 12, Value: 5

Thread: 13, Value: 5

```

In this example, we define a custom thread `MyThread` that extends the `Thread` class. The `run()` method is overridden with the task that the thread will perform. The `main()` method creates two instances of `MyThread` and starts them using the `start()` method. Both threads run concurrently, and their tasks are interleaved based on the CPU scheduler.

Please note that in modern Java development, it is more common to use the `Runnable` interface or the `ExecutorService` framework for creating and managing threads, as they provide more flexibility and separation of concerns. However, understanding the traditional thread creation using the `Thread` class helps in understanding the basics of Java thread management and the thread lifecycle.

## What is ExecutorService Framework

\*\*Executor Service Framework in Java\*\*

The Executor Service Framework in Java provides a high-level API for managing and executing tasks concurrently using a pool of worker threads. It is part of the `java.util.concurrent` package and offers an efficient and flexible alternative to manually creating and managing threads using the `Thread` class.

\*\*Key Components:\*\*

1. `ExecutorService` Interface: Represents an asynchronous execution service that manages and executes tasks. It provides methods to submit tasks for execution and manages the worker threads.

2. `ThreadPoolExecutor` Class: A commonly used implementation of the `ExecutorService` interface, representing a thread pool. It allows customizing thread management settings, such as core pool size, maximum pool size, and thread idle time.

3. `Executors` Utility Class: Provides factory methods to create different types of executor services, such as single-threaded executor, fixed-size thread pool, cached thread pool, etc.

\*\*Advantages:\*\*

- Thread Reuse: The framework manages a pool of threads, avoiding the overhead of creating new threads for each task.

- Load Management: Handles thread pooling and thread creation, preventing system overload with too many threads.

- Task Coordination: Provides ways to coordinate tasks, such as waiting for all tasks to complete or the first completed task.

- Exception Handling: Offers mechanisms to handle exceptions during task execution.

\*\*Example:\*\*

```java

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class ExecutorServiceExample {

public static void main(String[] args) {

ExecutorService executorService = Executors.newFixedThreadPool(5);

for (int i = 1; i <= 10; i++) {

int taskId = i;

executorService.submit(() -> {

System.out.println("Task " + taskId + " is being executed by thread: " + Thread.currentThread().getName());

});

}

executorService.shutdown();

}

}

```

In this example, we create an `ExecutorService` with a fixed thread pool of 5 threads using `Executors.newFixedThreadPool()`. We submit 10 tasks for execution using `submit()`. The tasks are executed concurrently by the threads in the pool. Finally, we shut down the `ExecutorService` after all tasks are submitted to free resources.

The Executor Service Framework simplifies concurrent task execution and thread management, making it easier to develop efficient multi-threaded applications.

## Transient and volatile keyword significance

The `transient` and `volatile` keywords in Java are used to modify the behavior of fields (instance variables) in a class.

1. `transient` keyword:

The `transient` keyword is used to indicate that a field should not be serialized when the object is converted to a byte stream, typically for storage or network transmission using serialization frameworks like Java Object Serialization. When an object is deserialized, the value of a transient field will be initialized to its default value (e.g., 0 for numeric types or `null` for reference types).

Example of using the `transient` keyword:

```java

import java.io.Serializable;

public class MyClass implements Serializable {

private transient int myTransientField;

private int myNormalField;

// Constructor, methods, etc.

}

```

In this example, the `myTransientField` will not be serialized when the `MyClass` object is serialized, while the `myNormalField` will be serialized.

2. `volatile` keyword:

The `volatile` keyword is used to indicate that a field is to be accessed directly from the main memory (shared memory) rather than from a thread's local cache. It ensures that the value of the field is always up-to-date and visible to all threads, making it useful for concurrent programming when multiple threads are accessing the same shared variable.

Example of using the `volatile` keyword:

```java

public class SharedData {

private volatile int count;

// Constructor, methods, etc.

}

```

In this example, the `count` field will be accessed directly from the main memory when multiple threads read or write its value, ensuring visibility and avoiding thread caching issues.

In summary:

- `transient`: Used to exclude a field from the serialization process to prevent it from being stored or transmitted as part of the serialized object. It is mainly used with serialization.

- `volatile`: Used to ensure that a field's value is always read from and written to the main memory, making it visible and up-to-date to all threads in a concurrent environment. It is used for concurrent programming to avoid data inconsistency issues between threads.

## Significance of static keyword

\*\*Java Static Keyword - Key Points:\*\*

1. \*\*Static Variables (Class Variables):\*\*

- Shared among all instances of a class.

- Declared with "static" keyword, initialized once for the class.

- Accessed using the class name.

- Changes are reflected in all instances.

2. \*\*Static Methods:\*\*

- Belong to the class, called without creating instances.

- Access static variables, invoke other static methods.

- Cannot access non-static variables or methods directly.

- Used for utility methods or state-independent operations.

3. \*\*Static Blocks:\*\*

- Initialize static variables or perform static initialization.

- Executed when class is loaded into JVM, before instance creation.

- Declared using "static" keyword within the class.

4. \*\*Static Nested Classes:\*\*

- Defined inside another class, marked as static.

- Accessed without creating an instance of the outer class.

- Used for grouping utility classes or encapsulation.

Static members provide class-level behavior and data, shared among all instances. Static nested classes help maintain code organization and encapsulation.

## Marker Interface

\*\*Marker Interface in Java\*\*

- A marker interface in Java is an interface that doesn't declare any methods or fields.

- Its purpose is to mark or tag a class to indicate that it possesses certain behavior or belongs to a specific category.

- Marker interfaces act as metadata markers, allowing other parts of the code to handle objects of marked classes in a specialized way.

- Example: `Serializable` is a marker interface in Java, used to indicate that a class's objects can be converted into byte streams for serialization.

- Marker interfaces were commonly used in Java before the introduction of annotations, which offer a more flexible and expressive way to provide metadata.

- While less common today, marker interfaces still find application in various scenarios where the need to tag classes with metadata arises.

## What is Class

\*\*Class in Object-Oriented Programming\*\*

- In object-oriented programming, a class is a blueprint or template for creating objects.

- It defines the structure and behavior of objects of a specific type.

- A class encapsulates data (attributes or fields) and methods (functions) representing the characteristics and behaviors of objects.

- Objects are instances of classes, and each object has its own set of data and can perform actions based on class-defined methods.

\*\*Example:\*\*

```java

public class Car {

private String make;

private String model;

private int year;

public Car(String make, String model, int year) {

this.make = make;

this.model = model;

this.year = year;

}

public String getDetails() {

return "Make: " + make + ", Model: " + model + ", Year: " + year;

}

}

```

\*\*Creating and Using Objects:\*\*

```java

public class Main {

public static void main(String[] args) {

Car myCar = new Car("Toyota", "Corolla", 2022);

System.out.println(myCar.getDetails());

}

}

```

\*\*Output:\*\*

```

Make: Toyota, Model: Corolla, Year: 2022

```

- Classes are fundamental to object-oriented programming, promoting code reusability and modeling complex systems.

## Different Types of Class Loaders in Java

In Java, the ClassLoader is a crucial component of the Java Runtime Environment (JRE) responsible for loading Java classes into memory at runtime. Java supports different types of ClassLoaders to facilitate various class-loading scenarios. The primary built-in ClassLoaders in Java are as follows:

1. Bootstrap ClassLoader:

- It is the top-level ClassLoader in the Java class-loading hierarchy.

- It is implemented in native code and is responsible for loading core Java classes from the Java Runtime Environment (JRE) system classes.

- It is not written in Java and cannot be directly accessed or extended by Java code.

2. Extensions ClassLoader (Extension ClassLoader):

- It is responsible for loading classes from the Java extension directory (`jre/lib/ext`).

- It is a child of the Bootstrap ClassLoader and loads classes from the extension JAR files.

- It is also written in native code and is not directly accessible or extendable by Java code.

3. System ClassLoader (Application ClassLoader):

- It is responsible for loading classes from the classpath provided to the Java application using the `-classpath` (or `-cp`) option or the `CLASSPATH` environment variable.

- It is a child of the Extensions ClassLoader and loads application-specific classes from the application's classpath.

- This is the ClassLoader used by default for loading user-defined classes.

4. Custom ClassLoaders:

- Apart from the built-in ClassLoaders, developers can also create custom ClassLoaders to load classes in specific ways or from non-standard sources.

- Custom ClassLoaders can be useful in scenarios like loading classes from databases, network locations, encrypted files, etc.

- Developers can subclass the `ClassLoader` class or extend one of the existing ClassLoaders and override the `findClass()` method to provide custom class-loading behavior.

The ClassLoader hierarchy in Java forms a parent-child relationship, where each ClassLoader (except the Bootstrap ClassLoader) has a parent ClassLoader. When a class needs to be loaded, the ClassLoader first checks its parent ClassLoader for the class. If the parent cannot find the class, the ClassLoader attempts to load the class itself.

This delegation model allows ClassLoaders to form a chain, where child ClassLoaders delegate class-loading requests to their parent ClassLoader recursively until the class is found or the Bootstrap ClassLoader is reached.

Understanding ClassLoaders is essential for certain Java technologies like application servers, where multiple modules or applications may need to load classes independently, and custom ClassLoaders are used to enforce class isolation and versioning.

## Object class in java and methods present in that class

In Java, the `Object` class is the root class of all classes in the Java class hierarchy. It is an implicitly defined superclass for all Java classes, and any class that does not explicitly extend another class inherits from the `Object` class. As a result, the `Object` class is present in the inheritance hierarchy of every Java class, directly or indirectly.

The `Object` class provides a set of commonly used methods that are available to all Java objects. These methods are inherited by all classes in Java, regardless of whether they are explicitly defined in the class or not. Some of the important methods present in the `Object` class are:

1. `equals(Object obj)`: Determines if the current object is equal to the specified object `obj`. By default, this method checks for reference equality (i.e., if the objects are the same instance). It is recommended to override this method in user-defined classes to provide custom equality checks based on the object's state.

2. `hashCode()`: Returns an integer hash code value for the object. This method is used in conjunction with the `equals()` method, and objects that are considered equal should have the same hash code.

3. `toString()`: Returns a string representation of the object. The default implementation returns the fully qualified name of the class concatenated with the object's hash code. It is often overridden in user-defined classes to provide a meaningful string representation.

4. `getClass()`: Returns the runtime class of the object. It returns an instance of the `Class` class that represents the object's actual class.

5. `notify()`, `notifyAll()`, `wait()`: These methods are used for inter-thread communication and synchronization. They are used in multi-threading scenarios for communication between threads using shared objects.

6. `clone()`: Creates and returns a copy of the object. This method performs a shallow copy, and for a deep copy, the user must override this method accordingly.

7. `finalize()`: Called by the garbage collector before reclaiming the object's memory. It is used to perform cleanup operations on the object before it is garbage collected.

The `Object` class provides default implementations for these methods, but they can be overridden in user-defined classes to provide customized behavior based on the specific requirements of the class.

It is important to note that when overriding any of these methods, it is necessary to follow their contracts and conventions to ensure proper functioning in the context of the entire Java object model.

## Serialization and Deserialization

Serialization and deserialization are processes used in Java to convert objects into a byte stream (serialization) and restore objects from a byte stream (deserialization). This allows objects to be saved to a file, transmitted over a network, or stored in a database. Let's understand these concepts:

1. Serialization:

- Serialization is the process of converting an object into a byte stream, which can be stored or transmitted.

- To make an object serializable, the class must implement the Serializable interface. This interface acts as a marker, indicating that the object can be serialized.

- During serialization, the object's state (values of its variables) is converted into a sequence of bytes.

- The ObjectOutputStream class is used to write the object to a file or stream.

- Example:

```java

class MyClass implements Serializable {

// Class members and methods

}

// Serialization

MyClass obj = new MyClass();

FileOutputStream fileOut = new FileOutputStream("object.ser");

ObjectOutputStream out = new ObjectOutputStream(fileOut);

out.writeObject(obj);

out.close();

fileOut.close();

```

2. Deserialization:

- Deserialization is the process of reconstructing an object from a byte stream.

- To perform deserialization, the class must have the same serialVersionUID as when the object was serialized.

- During deserialization, the byte stream is converted back into an object with its original state.

- The ObjectInputStream class is used to read the object from the file or stream.

- Example:

```java

// Deserialization

FileInputStream fileIn = new FileInputStream("object.ser");

ObjectInputStream in = new ObjectInputStream(fileIn);

MyClass obj = (MyClass) in.readObject();

in.close();

fileIn.close();

```

Important points to note:

- Not all objects can be serialized. Classes that cannot be serialized include those that contain non-serializable fields or classes explicitly marked as transient.

- Static and transient fields are not serialized.

- When a serializable object is deserialized, the constructor of the serialized class is not called. Instead, the object is created directly from the serialized byte stream.

Serialization and deserialization provide a convenient way to store and transmit objects in Java. They are widely used in various scenarios such as distributed systems, caching, and object persistence.

## Callable and Runnable interface in java

Both `Callable` and `Runnable` are interfaces in Java that are used to define tasks that can be executed concurrently. They are typically used in multi-threading scenarios to perform tasks in separate threads. However, they have some differences in their capabilities and how they work:

1. Runnable Interface:

- `Runnable` is a functional interface introduced in Java 1.0.

- It has a single abstract method `run()`, which does not take any arguments and does not return a value. This method represents the task that the `Runnable` object can execute.

- When a class implements the `Runnable` interface, it defines the task to be executed, and the task is executed by calling the `run()` method.

Example of using the `Runnable` interface:

```java

public class MyRunnableTask implements Runnable {

@Override

public void run() {

// Task logic here

System.out.println("Runnable Task is running in thread: " + Thread.currentThread().getName());

}

}

public class Main {

public static void main(String[] args) {

MyRunnableTask myRunnable = new MyRunnableTask();

Thread thread = new Thread(myRunnable);

thread.start();

}

}

```

2. Callable Interface:

- `Callable` is a generic functional interface introduced in Java 5 as part of the java.util.concurrent package.

- It has a single abstract method `call()`, which returns a result of type `V` and can throw an `Exception`.

- The `call()` method represents the task that the `Callable` object can execute, and it is similar to the `run()` method in the `Runnable` interface. The difference is that `call()` can return a result, whereas `run()` cannot.

Example of using the `Callable` interface with `ExecutorService`:

```java

import java.util.concurrent.Callable;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

public class MyCallableTask implements Callable<String> {

@Override

public String call() throws Exception {

// Task logic here

return "Callable Task is running in thread: " + Thread.currentThread().getName();

}

}

public class Main {

public static void main(String[] args) {

MyCallableTask myCallable = new MyCallableTask();

ExecutorService executorService = Executors.newSingleThreadExecutor();

Future<String> future = executorService.submit(myCallable);

try {

String result = future.get();

System.out.println(result);

} catch (Exception e) {

e.printStackTrace();

} finally {

executorService.shutdown();

}

}

}

```

In this example, we use `ExecutorService` to submit the `Callable` task for execution. The `call()` method returns a result, and we can obtain the result using the `Future` object returned by the `submit()` method.

In summary, both `Runnable` and `Callable` are used for concurrent programming, but `Callable` provides a way to get a result from the task, whereas `Runnable` does not return any value. Use `Runnable` when you need simple concurrent execution, and use `Callable` when you need a result from the task and handle potential exceptions.

## What is Exception and Hirerachy of Exception.

In Java, an exception is an event that occurs during the execution of a program and disrupts the normal flow of the program's instructions. When an exception occurs, it is thrown, and the program must handle it to prevent abnormal termination or unexpected behavior.

Exceptions in Java are objects that represent various types of errors and exceptional conditions that can occur during program execution. Java provides a rich hierarchy of exception classes to cover different scenarios, and this hierarchy is based on the `Throwable` class.

The hierarchy of exceptions in Java can be broadly divided into two categories:

1. Checked Exceptions:

- Checked exceptions are exceptions that the compiler requires the programmer to handle explicitly by either using a `try-catch` block or declaring them with a `throws` clause in the method signature.

- All checked exceptions are subclasses of `java.lang.Exception`.

- Examples of checked exceptions include `IOException`, `SQLException`, and `InterruptedException`.

2. Unchecked Exceptions (Runtime Exceptions):

- Unchecked exceptions, also known as runtime exceptions, are exceptions that the compiler does not require the programmer to handle explicitly.

- They are subclasses of `java.lang.RuntimeException`.

- Examples of unchecked exceptions include `NullPointerException`, `ArrayIndexOutOfBoundsException`, and `ArithmeticException`.

The `Throwable` class is at the top of the exception hierarchy and has two main subclasses:

1. `Error`:

- `Error` represents serious problems that are generally beyond the application's control and should not be caught or handled by the application code.

- Examples of errors include `OutOfMemoryError`, `StackOverflowError`, and `NoClassDefFoundError`.

2. `Exception`:

- `Exception` is the superclass of all checked exceptions and can be used to represent a wide range of exceptional conditions that may be handled by the application code.

- It has several subclasses, including `IOException`, `SQLException`, and user-defined checked exceptions.

When an exception is thrown, the Java runtime searches for an appropriate exception handler to handle it. If it does not find any appropriate handler in the current method, it goes up the call stack until it finds a suitable `try-catch` block or the program terminates if no handler is found.

Proper exception handling is essential in Java to ensure that programs handle exceptional conditions gracefully and avoid unexpected crashes or undesired behavior. By using the appropriate `try-catch` blocks or declaring exceptions with `throws`, developers can control the flow of the program when exceptional conditions occur.

## Custom Exception and How we can create custom Exceptions

In Java, you can create custom exceptions by defining your own exception class that extends one of the existing exception classes provided by Java, typically `Exception` or one of its subclasses. By creating custom exceptions, you can handle specific exceptional scenarios in your application more effectively.

To create a custom exception, follow these steps:

1. Create a new Java class that extends an existing exception class (e.g., `Exception` or one of its subclasses).

2. Define constructors for your custom exception class. You can provide multiple constructors to support different ways of initializing the exception.

3. Optionally, you can add additional fields and methods to the custom exception class to provide more context and information about the exception.

4. Ensure that your custom exception class follows the best practices and naming conventions for exception classes, including meaningful names and clear documentation.

Here's an example of how to create a custom exception class:

```java

// Custom exception class extending Exception

public class CustomException extends Exception {

// Constructor with a message

public CustomException(String message) {

super(message);

}

// Constructor with a message and a cause

public CustomException(String message, Throwable cause) {

super(message, cause);

}

}

```

Now, you can use this custom exception in your code as needed:

```java

public class MyClass {

public void performTask(int value) throws CustomException {

if (value < 0) {

// Throw the custom exception with a specific message

throw new CustomException("Value cannot be negative.");

}

// Perform the task with the value

}

public static void main(String[] args) {

MyClass obj = new MyClass();

try {

obj.performTask(-5);

} catch (CustomException e) {

// Handle the custom exception

System.out.println("Caught CustomException: " + e.getMessage());

}

}

}

```

In this example, we have created a custom exception `CustomException`. In the `performTask()` method, we check if the input value is negative, and if so, we throw the custom exception with a specific message. In the `main()` method, we catch the `CustomException` and handle it by printing the error message.

Custom exceptions allow you to create meaningful and domain-specific exceptions that make the exception handling process more organized and informative. They also provide a clear separation between different types of exceptional scenarios in your application.

## Exception Handling while overriding methods

When overriding methods in Java, you need to consider the exception handling rules defined by the Java language. These rules ensure that the overridden method's exception handling is compatible with the method it is overriding. Here are the key points to keep in mind when handling exceptions while overriding methods:

1. Checked Exceptions:

- When overriding a method, the overridden method in the superclass may declare checked exceptions in its method signature (using `throws` clause).

- The overriding method in the subclass can either:

a. Not throw any checked exceptions, even if the overridden method does. This is permissible because the subclass method can handle the checked exception internally or convert it to a runtime exception.

b. Throw the same checked exceptions as the overridden method or a subclass of the declared checked exceptions.

c. Throw no exceptions at all (i.e., no `throws` clause in the method signature) if the overridden method does not declare any checked exceptions.

Example:

```java

class Parent {

void doSomething() throws IOException {

// Some code that may throw IOException

}

}

class Child extends Parent {

@Override

void doSomething() {

// Child method may not throw IOException or can throw a runtime exception

}

}

```

2. Unchecked Exceptions (Runtime Exceptions):

- When overriding a method, the subclass method can throw any unchecked exceptions (runtime exceptions) without any restriction.

- The overriding method can also choose not to throw any runtime exceptions if the overridden method does throw runtime exceptions.

Example:

```java

class Parent {

void doSomething() {

// Some code that may throw a runtime exception

}

}

class Child extends Parent {

@Override

void doSomething() {

// Child method can throw a runtime exception or no exception at all

}

}

```

In summary, when overriding methods, the overriding method can handle checked exceptions differently from the overridden method. However, the overriding method cannot throw checked exceptions that are not compatible with the exceptions declared by the overridden method. For unchecked exceptions (runtime exceptions), the overriding method has more flexibility and can throw any runtime exceptions or choose not to throw any exception at all.

By following these rules, you can ensure that the exception handling in the subclass is compatible with the superclass, allowing for smooth and predictable exception handling in your application.

## Yeild vs Join Methods

`yield()` and `join()` are methods used in multi-threading in Java to control the execution flow of threads. They serve different purposes and have distinct effects on thread execution:

1. `yield()` method:

- The `yield()` method is a static method defined in the `Thread` class.

- When a thread calls `yield()`, it voluntarily gives up its current time slice to allow other threads of the same priority to run.

- It is a hint to the scheduler that the current thread is willing to yield its use of the CPU temporarily.

- The actual behavior of `yield()` depends on the operating system and the thread scheduler.

Example of using `yield()`:

```java

public class MyThread extends Thread {

@Override

public void run() {

for (int i = 1; i <= 5; i++) {

System.out.println(Thread.currentThread().getName() + ": " + i);

Thread.yield(); // Give up the CPU time to other threads

}

}

}

public class Main {

public static void main(String[] args) {

MyThread thread1 = new MyThread();

MyThread thread2 = new MyThread();

thread1.start();

thread2.start();

}

}

```

In this example, two threads are created, and each thread calls `yield()` inside its `run()` method. As a result, the threads may take turns executing their loops rather than running continuously.

2. `join()` method:

- The `join()` method is an instance method defined in the `Thread` class.

- When a thread calls `join()` on another thread, it waits for that thread to complete its execution before continuing with its own execution.

- The calling thread will be blocked until the target thread finishes.

- The `join()` method is useful when you want to ensure that specific threads complete their tasks before the main thread or any other dependent thread continues its execution.

Example of using `join()`:

```java

public class MyThread extends Thread {

@Override

public void run() {

for (int i = 1; i <= 5; i++) {

System.out.println(Thread.currentThread().getName() + ": " + i);

}

}

}

public class Main {

public static void main(String[] args) {

MyThread thread1 = new MyThread();

MyThread thread2 = new MyThread();

thread1.start();

try {

thread1.join(); // Wait for thread1 to complete

} catch (InterruptedException e) {

e.printStackTrace();

}

thread2.start();

}

}

```

In this example, the main thread waits for `thread1` to complete using `join()`. Once `thread1` completes, `thread2` starts its execution.

In summary, `yield()` is used to give a hint to the scheduler for temporarily relinquishing the CPU, allowing other threads to run. On the other hand, `join()` is used to wait for another thread to complete its execution before continuing with the current thread's execution.

## Reentrant lock

\*\*ReentrantLock in Java\*\*

- `ReentrantLock` is a type of lock in Java that allows a thread to acquire the same lock multiple times.

- This property is called "reentrant" or "recursive" locking, enabling a thread to enter a lock block without being blocked by itself.

- The `ReentrantLock` class provides more flexibility and features compared to the traditional `synchronized` keyword.

- Methods:

1. `lock()`: Acquires the lock. Blocks if the lock is not available.

2. `unlock()`: Releases the lock. Must be called from the same thread that acquired the lock.

3. `tryLock()`: Attempts to acquire the lock. Returns `true` if successful, `false` otherwise.

4. `tryLock(long time, TimeUnit unit)`: Attempts to acquire the lock within a specified time.

- Example:

```java

import java.util.concurrent.locks.ReentrantLock;

public class MyThread extends Thread {

private static ReentrantLock lock = new ReentrantLock();

@Override

public void run() {

lock.lock();

try {

// Critical section protected by the lock

System.out.println(Thread.currentThread().getName() + " is inside the critical section.");

// Some other operations...

} finally {

lock.unlock();

}

}

}

```

- The ability to reacquire the lock can be useful but requires careful programming to avoid deadlocks or synchronization issues.

- `ReentrantLock` also supports fairness, allowing threads to acquire the lock in the order they requested it, reducing thread starvation.

## Constructor Chaining and Types of Constrcutor

Constructor chaining in Java refers to the process of calling one constructor from another constructor within the same class or between a subclass and its superclass. It allows you to reuse code and set up common behavior across different constructors. Constructor chaining is achieved using the `this()` keyword to call another constructor or the `super()` keyword to call a constructor from the superclass.

Types of Constructors in Java:

1. Default Constructor:

- A default constructor is automatically provided by Java if no constructor is explicitly defined in a class.

- It has no parameters and does not perform any specific initialization.

- For example:

```java

public class MyClass {

// Default constructor is automatically provided if not defined explicitly

}

```

2. Parameterized Constructor:

- A parameterized constructor is a constructor that takes one or more parameters.

- It allows you to initialize the object with specific values at the time of object creation.

- For example:

```java

public class Student {

private String name;

private int age;

public Student(String name, int age) {

this.name = name;

this.age = age;

}

}

```

3. Copy Constructor:

- A copy constructor is a constructor that creates a new object by copying the state of another object of the same class.

- It is useful when you want to create a new object with the same values as an existing object.

- For example:

```java

public class Person {

private String name;

private int age;

public Person(Person other) {

this.name = other.name;

this.age = other.age;

}

}

```

4. Chained Constructor (Overloaded Constructor):

- A chained constructor is a constructor that calls another constructor of the same class using `this()`.

- It allows you to reuse the constructor's code and provide different initialization options with different parameters.

- For example:

```java

public class Rectangle {

private int width;

private int height;

public Rectangle(int sideLength) {

this(sideLength, sideLength);

}

public Rectangle(int width, int height) {

this.width = width;

this.height = height;

}

}

```

5. Superclass Constructor (Super Constructor):

- A superclass constructor is a constructor that is called from a subclass constructor using `super()`.

- It allows you to initialize the superclass's state before the subclass's state is initialized.

- For example:

```java

public class Animal {

protected String species;

public Animal(String species) {

this.species = species;

}

}

public class Dog extends Animal {

private String name;

public Dog(String species, String name) {

super(species); // Call the superclass constructor

this.name = name;

}

}

```

By utilizing constructor chaining and defining different types of constructors, you can create flexible and efficient object initialization in Java classes based on various requirements.

## can we override static methods in java

No, you cannot override static methods in Java. In Java, static methods belong to the class rather than the instances of the class. They are associated with the class itself, and not with any specific object of that class.

When a subclass defines a static method with the same signature (method name, return type, and parameter types) as a static method in the superclass, it does not override the static method; instead, it hides it.

Here's an example to illustrate this behavior:

```java

class Parent {

static void staticMethod() {

System.out.println("Static method in Parent class.");

}

}

class Child extends Parent {

static void staticMethod() {

System.out.println("Static method in Child class.");

}

}

public class Main {

public static void main(String[] args) {

Parent.staticMethod(); // Output: Static method in Parent class.

Child.staticMethod(); // Output: Static method in Child class.

}

}

```

As you can see, when you call `staticMethod` on the `Child` class, it does not override the method from the `Parent` class but hides it. The choice of which static method to call is determined at compile-time based on the reference type.

It's important to remember that static methods should typically be accessed using the class name (e.g., `Parent.staticMethod()` or `Child.staticMethod()`) and not through instances of the class (`parentInstance.staticMethod()` or `childInstance.staticMethod()`). Attempting to access a static method through an instance will produce a warning and is discouraged.

## Can we create a constructor of Abstract class in JAVA

Yes, you can create constructors in an abstract class in Java. An abstract class can have constructors just like any other regular class. However, there are some important points to consider when defining constructors in an abstract class:

1. Abstract classes cannot be instantiated directly:

- Since abstract classes cannot be instantiated directly (i.e., you cannot create objects of an abstract class), the purpose of constructors in an abstract class is primarily to initialize the instance variables or perform some common setup that subclasses can use through constructor chaining.

2. Constructor chaining in abstract classes:

- Abstract classes can have constructors that can be used to initialize their own instance variables and invoke constructors of their superclass (if any).

- This allows constructor chaining from the subclass through the abstract class constructor to the superclass constructor.

Example of an abstract class with a constructor:

```java

abstract class Shape {

protected String name;

public Shape(String name) {

this.name = name;

}

// Abstract method (no implementation)

public abstract double getArea();

}

class Circle extends Shape {

private double radius;

public Circle(String name, double radius) {

super(name); // Call the constructor of the abstract class

this.radius = radius;

}

@Override

public double getArea() {

return Math.PI \* radius \* radius;

}

}

public class Main {

public static void main(String[] args) {

Circle circle = new Circle("Circle 1", 5.0);

System.out.println(circle.name); // Output: Circle 1

System.out.println(circle.getArea()); // Output: 78.53981633974483

}

}

```

In this example, the `Shape` class is an abstract class with a constructor that initializes the `name` instance variable. The `Circle` class extends the `Shape` class and invokes the constructor of the abstract class using `super(name)` to set the `name` of the circle. The `Circle` class also provides its implementation of the `getArea()` method.

Remember that even though you can define constructors in an abstract class, you cannot directly create instances of the abstract class using the `new` keyword. You can only create objects of the concrete subclasses that extend the abstract class.

## Overloading and overriding

Overloading and overriding are two important concepts in Java that deal with methods in classes and inheritance.

1. Method Overloading:

- Method overloading is a feature in Java that allows a class to define multiple methods with the same name but with different parameter lists.

- The methods must have different numbers or types of parameters, or both, to be considered overloaded.

- The return type of the method is not considered when overloading methods.

- Method overloading is resolved at compile-time based on the method signature.

- It provides the ability to use the same method name for different operations, making the code more readable and expressive.

Example of method overloading:

```java

public class Calculator {

public int add(int a, int b) {

return a + b;

}

public double add(double a, double b) {

return a + b;

}

public int add(int a, int b, int c) {

return a + b + c;

}

}

```

2. Method Overriding:

- Method overriding is a feature in Java that allows a subclass to provide a specific implementation of a method that is already defined in its superclass.

- The method signature (name, return type, and parameter list) of the overriding method in the subclass must exactly match the method in the superclass.

- The purpose of method overriding is to provide a specific implementation of the method in the subclass, which replaces the implementation provided by the superclass.

- Method overriding is resolved at runtime based on the actual object type (dynamic binding) and is a fundamental concept in achieving runtime polymorphism.

Example of method overriding:

```java

class Animal {

void makeSound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

@Override

void makeSound() {

System.out.println("Dog barks");

}

}

```

In this example, the `Dog` class overrides the `makeSound()` method of the `Animal` class with its specific implementation.

In summary, method overloading allows a class to define multiple methods with the same name but different parameter lists, while method overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass. Both concepts contribute to the versatility and flexibility of Java programming, promoting code reuse and polymorphism.

## classnot found vs NoClassDefFoundError

`ClassNotFoundException` and `NoClassDefFoundError` are both Java runtime exceptions related to class loading and classpath issues. However, they occur in different scenarios and have different causes:

1. `ClassNotFoundException`:

- `ClassNotFoundException` is thrown when the Java Virtual Machine (JVM) tries to load a class at runtime using the `Class.forName()` method, and the class with the specified name cannot be found on the classpath.

- It can also occur when you try to load a class using the `ClassLoader` and the class is not present on the classpath.

- This exception typically occurs when a class that your code depends on is missing or unavailable at runtime.

2. `NoClassDefFoundError`:

- `NoClassDefFoundError` is thrown when the JVM tries to load the definition of a class at runtime, and it had previously successfully loaded the class definition, but now the definition is no longer available.

- It is typically caused when a class was present at compile-time but is missing at runtime, or when there is a conflict between different versions of a class on the classpath.

- The most common cause of this error is when there is a mismatch between the classpath used at compile-time and the classpath used at runtime.

To summarize, `ClassNotFoundException` is thrown when the JVM cannot find a class at runtime, whereas `NoClassDefFoundError` is thrown when the JVM finds the class at compile-time, but the class definition is not available at runtime.

In both cases, it is important to check the classpath settings and ensure that all required classes are available during both compile-time and runtime.

## checked vs unchecked exception

In Java, exceptions are categorized into two main types: checked exceptions and unchecked exceptions (also known as runtime exceptions).

1. \*\*Checked Exceptions\*\*:

- Checked exceptions are exceptions that the Java compiler mandates to be handled explicitly by the programmer. This means that when you call a method that can throw a checked exception, you must either handle the exception using a `try-catch` block or declare the exception in the method signature using the `throws` keyword.

- Examples of checked exceptions in Java include `IOException`, `SQLException`, and `InterruptedException`.

Example of handling a checked exception:

```java

import java.io.FileInputStream;

import java.io.FileNotFoundException;

public class Main {

public static void main(String[] args) {

try {

FileInputStream fileInputStream = new FileInputStream("file.txt");

} catch (FileNotFoundException e) {

System.out.println("File not found: " + e.getMessage());

}

}

}

```

2. \*\*Unchecked Exceptions (Runtime Exceptions)\*\*:

- Unchecked exceptions, also known as runtime exceptions, do not require explicit handling by the programmer. They are typically caused by programming errors, such as division by zero, accessing an array index out of bounds, or attempting to call a method on a `null` reference.

- Unlike checked exceptions, you are not required to declare unchecked exceptions in the method signature or handle them using `try-catch` blocks.

- Examples of unchecked exceptions in Java include `ArithmeticException`, `NullPointerException`, and `ArrayIndexOutOfBoundsException`.

Example of an unchecked exception:

```java

public class Main {

public static void main(String[] args) {

int x = 10;

int y = 0;

int result = x / y; // This will throw an ArithmeticException at runtime

}

}

```

In summary, checked exceptions are typically used for exceptional situations that are expected to be recoverable, while unchecked exceptions are used for programming errors and other situations that may not be easily recoverable. Handling checked exceptions explicitly enforces a higher level of error handling and robustness in the code, whereas unchecked exceptions provide more flexibility and convenience but require careful programming to avoid potential runtime issues.

## for Loop types in Collection

In Java, there are different types of loops that you can use to iterate over collections. The most common loop types for iterating through collections are:

1. Enhanced for Loop (Foreach Loop):

- The enhanced for loop, also known as the foreach loop, is a simple and concise way to iterate over elements in a collection.

- It is available in Java 5 and later versions and is specifically designed for collections and arrays.

- It automatically handles the iteration and simplifies the code by removing the need for index variables.

- The syntax for the enhanced for loop is as follows:

```java

for (ElementType element : collection) {

// Code to be executed for each element

}

```

Example:

```java

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

for (String name : names) {

System.out.println(name);

}

```

2. Iterator:

- The Iterator is a more flexible and powerful way to iterate over elements in a collection.

- It is available in Java 1.2 and provides methods like `hasNext()` and `next()` to traverse through the collection.

- It allows you to safely remove elements from the collection during iteration using the `remove()` method.

- The Iterator is ideal when you need to modify the collection while iterating or when you want more control over the iteration process.

- Example:

```java

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

Iterator<String> iterator = names.iterator();

while (iterator.hasNext()) {

String name = iterator.next();

System.out.println(name);

}

```

3. ListIterator:

- The ListIterator is a specialized Iterator that is available only for List implementations (e.g., ArrayList, LinkedList).

- It provides bidirectional traversal (both forward and backward) and additional methods like `hasPrevious()` and `previous()`.

- ListIterator is useful when you need to traverse a List in both directions and modify elements in the List.

- Example:

```java

List<String> names = new ArrayList<>(Arrays.asList("Alice", "Bob", "Charlie"));

ListIterator<String> listIterator = names.listIterator();

while (listIterator.hasNext()) {

String name = listIterator.next();

System.out.println(name);

}

```

These loop types allow you to efficiently iterate over elements in collections and perform various operations on the elements as needed. Choose the appropriate loop type based on your specific requirements and the functionality you need to implement.

## Garbage collector internal working and how many types of Garbage collector present in Java?

\*\*Garbage Collection in Java - Short Notes:\*\*

Garbage Collection (GC) is a crucial part of the Java Virtual Machine (JVM) responsible for reclaiming memory occupied by unreferenced objects.

\*\*Internal Working:\*\*

1. Mark Phase: Identifies reachable objects by starting from root objects and traversing the object graph through references.

2. Sweep Phase: Reclaims memory occupied by unreachable objects (garbage).

3. Compaction (optional): Rearranges live objects in the heap to reduce memory fragmentation.

\*\*Types of Garbage Collectors:\*\*

1. Serial: Default in older JVMs, uses a single thread.

2. Parallel: Default in Java 8+, uses multiple threads for higher throughput.

3. G1 (Garbage-First): For large heap sizes, divides heap into regions.

4. CMS (Concurrent Mark-Sweep): For low-latency applications, reduces pause times.

5. ZGC: For low-latency with large heaps, aims to keep pauses short.

\*\*Choice of GC:\*\* Depends on heap size, latency requirements, and hardware resources. Can be configured using JVM flags.

Java's Garbage Collection mechanism ensures efficient memory management and prevents memory leaks by automatically reclaiming unused memory. Different GC algorithms cater to diverse application needs, offering trade-offs between throughput and latency.

## Array List Internal Working

\*\*Internal Working of ArrayList in Java - Short Note:\*\*

- `ArrayList` is internally implemented as a dynamic array, represented by an object array.

- When an `ArrayList` is created, it starts with a default initial capacity (e.g., 10).

- As elements are added, the `ArrayList` keeps track of the number of elements.

- When the number of elements exceeds the current capacity, the `ArrayList` automatically resizes its underlying array to accommodate more elements.

- The resizing process typically involves creating a new, larger array and copying all elements from the old array to the new one.

- Resizing the array allows `ArrayList` to efficiently grow or shrink in size as needed without requiring manual memory management.

- Random access to elements by index takes constant time (O(1)) because the elements are stored in contiguous memory locations.

- Inserting or removing elements at the end of the `ArrayList` (using `add()` or `remove()` without specifying an index) is efficient (O(1) average time complexity).

- However, inserting or removing elements at a specific index may require shifting elements in the array, which takes linear time (O(n)) where n is the number of elements in the list.

\*\*Example:\*\*

```java

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<Integer> numbers = new ArrayList<>();

numbers.add(10);

numbers.add(20);

numbers.add(30);

}

}

```

In this example, we create an `ArrayList` of integers and add elements to it. As elements are added, the `ArrayList` internally manages the resizing of the underlying array to accommodate more elements efficiently.

## SOLID Design Principle in JAVA

SOLID is an acronym that represents five design principles used to create maintainable, flexible, and robust software systems, aiming to achieve object-oriented design goals such as modularity, extensibility, and reusability.

1. \*\*Single Responsibility Principle (SRP):\*\*

- A class should have only one reason to change, meaning it should have a single responsibility.

- According to SRP, a class should encapsulate only one functionality or behavior, and if there are multiple reasons to change a class, it should be divided into smaller classes, each with a single responsibility.

2. \*\*Open/Closed Principle (OCP):\*\*

- A class should be open for extension but closed for modification.

- OCP suggests that the behavior of a class can be extended without modifying its existing code, often achieved through inheritance, interfaces, or abstract classes.

3. \*\*Liskov Substitution Principle (LSP):\*\*

- Objects of a superclass should be replaceable with objects of its subclass without affecting the correctness of the program.

- In other words, subclasses should be substitutable for their base class, and they should adhere to the contracts (interfaces) defined by the base class.

4. \*\*Interface Segregation Principle (ISP):\*\*

- A class should not be forced to implement interfaces that are not related to it.

- ISP advises focusing on specific behaviors for classes, which means using smaller, specialized interfaces instead of large, general-purpose interfaces.

5. \*\*Dependency Inversion Principle (DIP):\*\*

- High-level modules should not depend on low-level modules. Both should depend on abstractions.

- DIP promotes loose coupling between classes, where abstractions (interfaces or abstract classes) are used instead of concrete implementations. This makes high-level modules independent of specific low-level implementations, enhancing flexibility and ease of making changes.

Applying SOLID principles in Java helps create a well-structured and flexible codebase, which is easier to understand, maintain, and extend. These principles enable the development of software that is more resilient to changes and better separates concerns, resulting in a modular and maintainable system.

## Generics with and without coding differences

When coding in Java, there are significant differences between using generics and not using generics. Let's explore the key distinctions:

1. Without Generics:

- In older versions of Java (pre-Java 5), collections like ArrayList, HashMap, etc., did not support generics.

- Without generics, you can store any type of object in the collection, leading to potential type safety issues at runtime.

- You need to manually cast the objects retrieved from the collection to the desired type, which can result in ClassCastException if the casting is incorrect.

- Example without generics:

```java

ArrayList list = new ArrayList();

list.add("Hello");

String str = (String) list.get(0); // Casting required

```

2. With Generics:

- Java introduced generics in Java 5 to provide compile-time type safety and eliminate the need for explicit casting.

- Generics allow you to specify the type of objects that can be stored in a collection or used in a class/interface.

- Generics ensure that the correct type is used at compile-time and can detect type mismatches before runtime.

- Example with generics:

```java

ArrayList<String> list = new ArrayList<String>();

list.add("Hello");

String str = list.get(0); // No casting required

```

Key benefits of using generics:

- Type safety: Generics ensure that you use the correct types and catch type mismatches at compile-time, reducing the risk of runtime errors.

- Readability: Generics make code more readable by explicitly specifying the types being used.

- Compile-time checks: The compiler can perform static type checks and provide better error messages when using generics.

In summary, using generics in Java provides compile-time type safety, improves code readability, and eliminates the need for explicit casting. It is recommended to use generics whenever possible to ensure type safety and produce more robust and maintainable code.

## Integer value Equality a==b

In Java, the equality operator (`==`) is used to compare the values of primitive data types and object references. When comparing two primitive data types, such as `int`, `double`, `char`, etc., the `==` operator checks if the values are equal.

For primitive data types, `a == b` will be true if the values of `a` and `b` are the same, and false otherwise.

Example:

```java

int a = 5;

int b = 5;

boolean result = a == b; // result will be true, as the values of a and b are both 5

```

However, when comparing object references, the `==` operator checks if the references point to the same memory location (i.e., the same object), not if the objects themselves have the same content.

Example:

```java

String str1 = "Hello";

String str2 = "Hello";

String str3 = new String("Hello");

boolean result1 = str1 == str2; // result1 will be true, as both str1 and str2 point to the same "Hello" object in the String pool

boolean result2 = str1 == str3; // result2 will be false, as str3 is a new object with a different memory location

```

In the above example, `str1` and `str2` point to the same string literal in the String pool, so `str1 == str2` is true. However, `str3` is a new object created using the `new` keyword, so `str1 == str3` is false since they refer to different objects.

To compare the content or values of objects, you should use the `equals()` method, which is overridden in many classes to provide content-based comparison.

Example:

```java

boolean result3 = str1.equals(str3); // result3 will be true, as str1 and str3 have the same content "Hello"

```

In summary, when using the equality operator `==`, it is essential to understand whether you are comparing primitive data types or object references. For object comparison, use the `equals()` method to check for content equality, unless you specifically need to check if two references point to the same object in memory.

## String s1 = new String("abc"); String s2 = new String("abc"); what happen if we write this in java

In Java, when you create two `String` objects using the `new` keyword, as in the following code:

```java

String s1 = new String("abc");

String s2 = new String("abc");

```

Two separate `String` objects will be created in memory. Even though the content of both strings is the same ("abc"), they will be stored as two distinct objects with different memory addresses. This is because the `new` keyword explicitly creates a new object, regardless of whether an existing string with the same content already exists in the String pool (the pool where literal strings are stored).

To illustrate this, let's consider the following code:

```java

String s1 = new String("abc");

String s2 = new String("abc");

System.out.println(s1 == s2); // Output: false

System.out.println(s1.equals(s2)); // Output: true

```

The first print statement (`s1 == s2`) compares the references of `s1` and `s2`, and it will return `false` because `s1` and `s2` refer to different objects in memory.

The second print statement (`s1.equals(s2)`) compares the content of `s1` and `s2`, and it will return `true` because both strings have the same content ("abc").

In general, when dealing with `String` objects, you should use the `equals` method to compare their contents, not the `==` operator, as the `equals` method is designed to compare the actual content of the strings. Additionally, in most cases, it is recommended to use string literals directly (e.g., `"abc"`) instead of using the `new` keyword to create `String` objects, as string literals are automatically interned (pooled) by Java and can be more memory-efficient.

## Enum in JAVA

\*\*Enum in Java - Short Note with Example:\*\*

In Java, an enum (short for enumeration) is a special data type used to define a set of named constants representing fixed values. Enumerations provide a way to create a group of related constants, making the code more readable, type-safe, and maintainable.

\*\*Example:\*\*

Let's consider an example of defining an enum to represent the days of the week:

```java

public class EnumExample {

// Enum for days of the week

public enum Day {

SUNDAY,

MONDAY,

TUESDAY,

WEDNESDAY,

THURSDAY,

FRIDAY,

SATURDAY

}

public static void main(String[] args) {

// Using the enum constants

Day today = Day.MONDAY;

System.out.println("Today is: " + today);

// Switch statement with enum

switch (today) {

case MONDAY:

System.out.println("It's Monday!");

break;

case FRIDAY:

System.out.println("It's Friday!");

break;

default:

System.out.println("It's neither Monday nor Friday.");

}

}

}

```

Output:

```

Today is: MONDAY

It's Monday!

```

In this example, we have defined an enum named `Day`, representing the days of the week. Enum constants are declared using uppercase letters by convention. The `Day` enum contains seven constants: `SUNDAY`, `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, and `SATURDAY`.

In the `main` method, we demonstrate how to use enum constants. We assign `Day.MONDAY` to the variable `today` and print its value. We also use a `switch` statement with the `today` enum to print a different message based on the day of the week.

Enums provide several benefits:

1. Type safety: Enums are type-safe, which means you cannot assign a value that is not part of the enum constants.

2. Readability: Enums improve code readability by giving meaningful names to constants.

3. Easy comparison: You can compare enum constants using `==`, making the code clearer and less error-prone.

4. Enum methods: Enums can have methods and fields, just like regular classes, allowing you to add behavior to enum constants.

Enums are commonly used to represent fixed sets of constants, such as days of the week, months, status codes, and more, making code more expressive and maintainable.

## Access modifiers available in Java ? Please give me the order starting with the most restrictive.

In Java, there are four access modifiers that control the visibility and accessibility of classes, fields, methods, and constructors. They are listed below in order from the most restrictive to the least restrictive:

1. `private`: The `private` access modifier restricts access to only within the same class where the member (field or method) is declared. It is the most restrictive access level.

2. `default` (package-private): If no access modifier is specified, the member has package-private (default) access. It allows access within the same package but not from outside the package.

3. `protected`: The `protected` access modifier allows access within the same package and also from subclasses (in any package), even if they are in different packages.

4. `public`: The `public` access modifier has the least restrictions and allows access from any class or package, including subclasses outside the package.

Here's the order of access modifiers from most restrictive to least restrictive:

1. `private`

2. `default` (package-private)

3. `protected`

4. `public`

It's important to use the appropriate access modifier based on the desired visibility and the design requirements of your classes and members. Using more restrictive access levels can help encapsulate the internal details of a class and prevent unintended access from other parts of the code, enhancing code maintainability and security.

## int arr[] = {​​1,2,3}​​ find first index using stream

To find the first index of a specific element in an array using Java streams, you can use the `IntStream.range()` method to create a stream of indices and then use the `filter()` method to filter the indices based on the element value. Once you find the first matching index, you can use the `findFirst()` method to get the optional result.

Here's how you can do it:

```java

import java.util.OptionalInt;

import java.util.stream.IntStream;

public class Main {

public static void main(String[] args) {

int arr[] = {1, 2, 3};

int elementToFind = 2;

OptionalInt firstIndex = IntStream.range(0, arr.length)

.filter(i -> arr[i] == elementToFind)

.findFirst();

if (firstIndex.isPresent()) {

System.out.println("First index of " + elementToFind + " is: " + firstIndex.getAsInt());

} else {

System.out.println("Element not found in the array.");

}

}

}

```

In this example, we have an array `arr` containing elements [1, 2, 3]. We want to find the first index of the element with the value 2. The `IntStream.range(0, arr.length)` creates a stream of indices from 0 to `arr.length - 1`. Then, the `filter()` method filters the indices based on the condition `arr[i] == elementToFind`, where `i` is the current index. The `findFirst()` method returns the first matching index as an `OptionalInt`. If the element is found, the first index is printed; otherwise, a message is displayed indicating that the element was not found in the array.

## Program to Non repeatable first character of the given string

public static char findFirstNonRepeatingCharacter(String str) {

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

for (char c : str.toCharArray()) {

charFrequencies.put(c, charFrequencies.getOrDefault(c, 0) + 1);

}

for (char c : str.toCharArray()) {

if (charFrequencies.get(c) == 1) {

return c;

}

}

return '\0';

}

JAVA 8:

import java.util.LinkedHashMap;

import java.util.Map;

import java.util.Optional;

import java.util.stream.Collectors;

public class FirstNonRepeatedCharacter {

public static void main(String[] args) {

String input = "aaabcccdeeeffdb";

char firstNonRepeated = findFirstNonRepeatedCharacter(input);

System.out.println("First non-repeated character: " + firstNonRepeated);

}

public static char findFirstNonRepeatedCharacter(String input) {

Map<Character, Long> charCounts = input.chars()

.mapToObj(c -> (char) c)

.collect(Collectors.groupingBy(

character -> character,

LinkedHashMap::new,

Collectors.counting()

));

Optional<Character> firstNonRepeated = charCounts.entrySet().stream()

.filter(entry -> entry.getValue() == 1)

.map(Map.Entry::getKey)

.findFirst();

return firstNonRepeated.orElse('\0'); // Return null character if no non-repeated character found

}

}

## Program to even numbers.

public class EvenNumberPrinter {

public static void printEvenNumbers(int start, int end) {

if (start % 2 != 0) {

start++; // Ensure the starting number is even

}

for (int i = start; i <= end; i += 2) {

System.out.print(i + " ");

}

}

public static void main(String[] args) {

int start = 10;

int end = 30;

System.out.println("Even numbers between " + start + " and " + end + ":");

printEvenNumbers(start, end);

}

}

## Program to Sort 2 different arrays.

import java.util.Arrays;

public class SortArrays {

public static void main(String[] args) {

// Sample arrays

int[] numbersArray = {9, 4, 7, 2, 1, 5};

String[] namesArray = {"Alice", "Bob", "Charlie", "Eve", "David"};

// Sort the arrays

Arrays.sort(numbersArray);

Arrays.sort(namesArray);

// Print the sorted arrays

System.out.println("Sorted numbersArray: " + Arrays.toString(numbersArray));

System.out.println("Sorted namesArray: " + Arrays.toString(namesArray));

}

}

## Program to Star symbols in triangle pattern.

public static void printStarTriangle(int rows) {

for (int i = 1; i <= rows; i++) {

// Print spaces before stars

for (int j = 1; j <= rows - i; j++) {

System.out.print(" ");

}

// Print stars

for (int k = 1; k <= 2 \* i - 1; k++) {

System.out.print("\*");

}

// Move to the next line

System.out.println();

}

}

## Program to Remove trailing spaces of the given number.

public class TrailingSpacesRemover {

public static String removeTrailingSpaces(String number) {

return number.trim();

}

public static void main(String[] args) {

String number = " 12345 ";

System.out.println("Original number: '" + number + "'");

String result = removeTrailingSpaces(number);

System.out.println("Number without trailing spaces: '" + result + "'");

}

}

## Filter array list and Hashmap using Streams in JAVA 8

In Java 8, you can use streams to filter elements from an ArrayList and HashMap based on certain conditions. Here are examples for both cases:

\*\*Filtering ArrayList using Streams:\*\*

Suppose we have an ArrayList of integers, and we want to filter out the even numbers from the list.

```java

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Collectors;

public class Main {

public static void main(String[] args) {

List<Integer> numbers = new ArrayList<>();

numbers.add(1);

numbers.add(2);

numbers.add(3);

numbers.add(4);

numbers.add(5);

// Filter even numbers using streams

List<Integer> evenNumbers = numbers.stream()

.filter(n -> n % 2 == 0)

.collect(Collectors.toList());

System.out.println("Even Numbers: " + evenNumbers);

}

}

```

Output:

```

Even Numbers: [2, 4]

```

\*\*Filtering HashMap using Streams:\*\*

Suppose we have a HashMap with names as keys and ages as values, and we want to filter out the people with age greater than 30.

```java

import java.util.HashMap;

import java.util.Map;

import java.util.stream.Collectors;

public class Main {

public static void main(String[] args) {

Map<String, Integer> ageMap = new HashMap<>();

ageMap.put("John", 25);

ageMap.put("Alice", 35);

ageMap.put("Bob", 28);

ageMap.put("Eve", 40);

// Filter people with age greater than 30 using streams

Map<String, Integer> filteredAgeMap = ageMap.entrySet().stream()

.filter(entry -> entry.getValue() > 30)

.collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));

System.out.println("People with Age > 30: " + filteredAgeMap);

}

}

```

Output:

```

People with Age > 30: {Alice=35, Eve=40}

```

In both examples, we used the `filter` method on the stream to apply the desired condition. The filtered elements were then collected into a new list (for the ArrayList example) or a new map (for the HashMap example) using the `collect` method with appropriate collectors.

# Spring:

## Scopes available in spring

Certainly! Here are examples of each of the common bean scopes in Spring:

1. Singleton (default scope):

```java

@Component

public class SingletonBean {

// ...

}

```

When you define a bean without specifying a scope, it defaults to the singleton scope. In this example, the `SingletonBean` class is annotated with `@Component`, making it a singleton bean by default. The same instance of `SingletonBean` will be returned for each request for this bean.

2. Prototype:

```java

@Component

@Scope("prototype")

public class PrototypeBean {

// ...

}

```

To define a prototype bean explicitly, you can use the `@Scope("prototype")` annotation. In this example, the `PrototypeBean` class is annotated with `@Component` and `@Scope("prototype")`. Each time this bean is requested, a new instance of `PrototypeBean` will be created.

3. Request:

```java

@Component

@Scope(value = "request", proxyMode = ScopedProxyMode.TARGET\_CLASS)

public class RequestBean {

// ...

}

```

For request-scoped beans, you need to enable the appropriate web configuration and include the `proxyMode` attribute to ensure proper proxying. In this example, the `RequestBean` class is annotated with `@Component` and `@Scope("request")`, with `proxyMode` set to `ScopedProxyMode.TARGET\_CLASS`. A new instance of `RequestBean` will be created for each HTTP request.

4. Session:

```java

@Component

@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET\_CLASS)

public class SessionBean {

// ...

}

```

Similar to the request scope, you need to enable the necessary web configuration and include the `proxyMode` attribute for session-scoped beans. In this example, the `SessionBean` class is annotated with `@Component` and `@Scope("session")`, with `proxyMode` set to `ScopedProxyMode.TARGET\_CLASS`. One instance of `SessionBean` will be created per user session.

5. Application:

```java

@Component

@Scope(value = "application", proxyMode = ScopedProxyMode.TARGET\_CLASS)

public class ApplicationBean {

// ...

}

```

For application-scoped beans, you also need the appropriate web configuration and the `proxyMode` attribute to enable proper proxying. In this example, the `ApplicationBean` class is annotated with `@Component` and `@Scope("application")`, with `proxyMode` set to `ScopedProxyMode.TARGET\_CLASS`. A single instance of `ApplicationBean` will be created per ServletContext.

6. WebSocket:

```java

@Component

@Scope(value = "websocket", proxyMode = ScopedProxyMode.TARGET\_CLASS)

public class WebSocketBean {

// ...

}

```

WebSocket-scoped beans require the necessary web configuration and the `proxyMode` attribute for proper proxying. In this example, the `WebSocketBean` class is annotated with `@Component` and `@Scope("websocket")`, with `proxyMode` set to `ScopedProxyMode.TARGET\_CLASS`. An instance of `WebSocketBean` will be created per WebSocket connection.

These examples demonstrate the usage of different bean scopes in Spring, including singleton, prototype, request, session, application, and WebSocket scopes. The specific scope is determined by the `@Scope` annotation and additional configuration, such as enabling web-related features and specifying proxy modes when necessary.

## Define scope in Spring

In Spring, you can specify the scope of a bean using the `@Scope` annotation. The `@Scope` annotation is applied to the bean class, and it allows you to define the scope of the bean as one of the standard scopes provided by Spring or a custom scope defined in your application.

Here's how you can specify scope in Spring with an example:

1. Using Standard Scopes:

Spring provides several standard scopes that you can use out-of-the-box. You can specify the scope by providing the scope name as a value to the `@Scope` annotation.

Example:

```java

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Component;

@Component

@Scope("singleton") // Default scope, same as not specifying @Scope

public class MySingletonBean {

// Bean properties and methods

}

```

In this example, the `MySingletonBean` is declared with the `@Scope("singleton")` annotation, which specifies the default scope, i.e., the singleton scope. This means that only one instance of `MySingletonBean` will be created and reused throughout the application context.

Similarly, you can use `"prototype"`, `"request"`, `"session"`, `"global session"`, or `"application"` as scope values for the corresponding standard scopes.

2. Using Custom Scopes:

If you have specific requirements that are not met by the standard scopes, you can define custom scopes in your application. Custom scopes are typically used when you need more control over the lifecycle of your beans.

Example:

```java

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Component;

@Component

@Scope("myCustomScope")

public class MyCustomScopedBean {

// Bean properties and methods

}

```

In this example, `MyCustomScopedBean` is declared with the `@Scope("myCustomScope")` annotation, which specifies a custom scope named `"myCustomScope"`. You will need to define the custom scope and its corresponding `Scope` implementation in your application configuration.

To define a custom scope, you can implement the `org.springframework.beans.factory.config.Scope` interface, which requires you to implement the `get`, `remove`, and `registerDestructionCallback` methods to manage the lifecycle of the custom-scoped beans.

Overall, specifying the scope of a bean in Spring allows you to control how Spring manages the instances of your beans, making it an essential feature for managing the lifecycle and behavior of components in a Spring application.

## Stereo Types Annotations

In Spring, stereotype annotations are special annotations used to define the roles of classes within a Spring application context. These annotations help to configure and manage beans more effectively by providing additional metadata to the Spring container. By using stereotype annotations, you can avoid manual configuration and make your code more concise and readable.

Here are the common stereotype annotations in Spring:

1. `@Component`: This is a generic stereotype annotation for any Spring-managed component. If a class is annotated with `@Component`, Spring automatically detects and registers it as a bean in the application context.

2. `@Service`: This is a specialization of `@Component` and is used to annotate service classes. It indicates that the class provides a service in the application.

3. `@Repository`: This is a specialization of `@Component` and is used to annotate repository or DAO classes. It indicates that the class is responsible for data access and storage.

4. `@Controller`: This is a specialization of `@Component` and is used to annotate controller classes in a Spring MVC application. It indicates that the class handles web requests and serves as the entry point for request processing.

5. `@RestController`: This is a specialization of `@Controller` and is used to annotate classes that combine `@Controller` and `@ResponseBody`. It is commonly used in RESTful web services to directly return data as JSON or XML.

6. `@Configuration`: This annotation is used to indicate that a class defines Spring bean configuration. It is used along with the `@Bean` annotation to define beans and their dependencies programmatically.

7. `@Bean`: This annotation is used inside `@Configuration` classes to declare a Spring bean. The method annotated with `@Bean` produces a bean instance and makes it available to the Spring container.

Using stereotype annotations, you can define your beans and let Spring automatically scan and register them in the application context, reducing the need for explicit XML configuration.

Example:

```java

@Service

public class ProductService {

// Service implementation...

}

@Repository

public class ProductRepository {

// Data access implementation...

}

@RestController

public class ProductController {

@Autowired

private ProductService productService;

// Request handling methods...

}

```

In the above example, the `ProductService`, `ProductRepository`, and `ProductController` classes are annotated with stereotype annotations, allowing them to be automatically registered as Spring beans in the application context.

Remember to enable component scanning in your Spring configuration to allow Spring to detect and manage classes with stereotype annotations. You can do this by using `@ComponentScan` or `context:component-scan` in XML configuration.

## Spring mvc architecture

Spring MVC is a web framework based on the Model-View-Controller (MVC) architectural pattern. It provides a structured approach to building web applications by separating the concerns of data handling, user interface, and application flow. The architecture of Spring MVC consists of the following components:

1. Model:

- The Model represents the data and business logic of the application. It typically consists of JavaBeans or POJOs (Plain Old Java Objects) that hold the data and the logic to process and manipulate that data.

- The Model is responsible for interacting with the database, processing user input, and performing business operations.

- In Spring MVC, the Model is typically represented by the `@ModelAttribute` annotated classes or `@Entity` classes in case of using Spring Data JPA.

2. View:

- The View is responsible for rendering the user interface and presenting the data to the user. It is the presentation layer of the application.

- In Spring MVC, views can be created using technologies like JSP (JavaServer Pages), Thymeleaf, FreeMarker, or other template engines. These views are responsible for displaying the data to the user in a readable format.

3. Controller:

- The Controller acts as an intermediary between the Model and the View. It receives user requests, processes them, invokes appropriate actions in the Model, and selects the appropriate view to display the results.

- In Spring MVC, Controllers are implemented as classes annotated with `@Controller`. Methods inside the controller class are annotated with `@RequestMapping` or other specialized annotations to map specific URLs and HTTP methods to the corresponding controller methods.

4. DispatcherServlet:

- The DispatcherServlet is the central front controller in Spring MVC. It receives all incoming requests and then delegates the requests to the appropriate controller for handling.

- The DispatcherServlet uses HandlerMappings to determine which controller should handle the request and HandlerAdapters to invoke the corresponding controller method.

- Once the controller method has processed the request and generated the appropriate Model and View objects, the DispatcherServlet uses a ViewResolver to find the corresponding view and render the response.

5. HandlerMapping:

- HandlerMapping is responsible for mapping incoming requests to the appropriate Controller and handler method.

- Spring provides various built-in HandlerMappings like `RequestMappingHandlerMapping`, `BeanNameUrlHandlerMapping`, etc.

6. HandlerAdapter:

- HandlerAdapter is responsible for invoking the appropriate controller method and handling the interaction between the DispatcherServlet and the Controller.

- Different types of controllers may require different HandlerAdapters. Spring provides several built-in HandlerAdapters to support various controller types, such as `RequestMappingHandlerAdapter`, `SimpleControllerHandlerAdapter`, etc.

7. ViewResolver:

- ViewResolver is responsible for resolving the logical view name returned by the controller to the actual view implementation that should be rendered and returned to the user.

- Spring provides various built-in ViewResolvers like `InternalResourceViewResolver` for JSP views, `ThymeleafViewResolver` for Thymeleaf views, etc.

The interaction among these components is handled by the Spring container, which manages the lifecycle of beans and controls the flow of the application. The separation of concerns in the Spring MVC architecture makes the development and maintenance of web applications more manageable and modular. It allows developers to focus on specific aspects of the application without affecting other components, leading to a more organized and scalable codebase.

## What is spring cloud

Spring Cloud is a set of tools and frameworks provided by the Spring community to simplify the development and deployment of distributed and microservices-based applications in a cloud environment. It builds upon the core features of the Spring Framework and provides additional capabilities to address the challenges associated with developing cloud-native applications.

Spring Cloud offers various components and modules that help developers with common tasks in cloud-based applications, such as service discovery, configuration management, load balancing, circuit breaking, distributed tracing, and more. It leverages well-known technologies and standards, making it easier to integrate with existing cloud platforms and services.

Some key components and features of Spring Cloud include:

1. Service Discovery: Spring Cloud provides tools like Netflix Eureka or Spring Cloud Consul for service registration and discovery. This allows microservices to locate and communicate with each other dynamically without hardcoding IP addresses or hostnames.

2. API Gateway: Spring Cloud Gateway offers a lightweight, dynamic routing and filtering mechanism to handle requests coming into a microservices-based system. It acts as an entry point to the application and can perform various tasks like load balancing, rate limiting, and authentication.

3. Distributed Configuration: Spring Cloud Config allows you to externalize the configuration of microservices, making it easier to manage configuration properties across different environments and services.

4. Circuit Breaker: Spring Cloud Circuit Breaker provides integration with circuit breaker libraries like Netflix Hystrix or Resilience4j. It helps in handling failures and isolating failing components to prevent cascading failures in a distributed system.

5. Distributed Tracing: Spring Cloud Sleuth offers distributed tracing capabilities by generating and propagating trace and span IDs across microservices, making it easier to debug and monitor requests across the system.

6. Cloud Messaging: Spring Cloud Stream provides abstractions for messaging middleware like Apache Kafka, RabbitMQ, or Apache Kafka Streams, allowing easy integration with message-driven microservices.

7. OAuth2 and JWT: Spring Cloud Security offers integration with OAuth2 and JSON Web Tokens (JWT) for securing microservices and managing authentication and authorization.

8. Kubernetes Support: Spring Cloud Kubernetes provides integration with Kubernetes, enabling seamless interaction with the Kubernetes environment for service discovery and configuration.

Spring Cloud provides these features and more, helping developers to build cloud-native applications with ease and adhere to best practices in cloud-based architectures. It promotes the principles of microservices, modularity, scalability, and resilience, making it a popular choice for building modern cloud applications.

## bean creation programmatically and conditionally in spring

In Spring, you can enable bean creation programmatically and conditionally using the `@Conditional` annotation along with `@Configuration` classes or `@Bean` methods. The `@Conditional` annotation allows you to specify a condition that must be satisfied for the bean to be created and registered in the Spring application context.

Here's how you can enable bean creation programmatically and conditionally:

1. Using `@Conditional` with `@Configuration` class:

```java

@Configuration

public class AppConfig {

@Bean

@Conditional(MyCondition.class)

public MyBean myBean() {

return new MyBean();

}

}

```

In this example, the `MyBean` bean will be created and registered in the Spring context only if the `MyCondition` class returns `true`. You can create your own custom condition by implementing the `org.springframework.context.annotation.Condition` interface and overriding the `matches()` method.

```java

public class MyCondition implements Condition {

@Override

public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {

// Add your condition logic here

return true; // Return true to enable bean creation, false otherwise

}

}

```

2. Using `@Conditional` with `@Bean` method:

```java

@Configuration

public class AppConfig {

@Bean

@Conditional(MyCondition.class)

public MyBean myBean() {

return new MyBean();

}

}

```

The result is the same as in the previous example. The `MyBean` bean will be created if the `MyCondition` class returns `true`.

You can use this conditional bean creation approach to customize the Spring application context based on specific conditions, such as the presence of certain properties, environment variables, classpath settings, or any other custom logic you need. By leveraging the `@Conditional` annotation, you can have more fine-grained control over which beans are created and registered in the application context, leading to a more flexible and dynamic configuration based on runtime conditions.

## Application Context and Bean Factory in spring

In the Spring Framework, the `ApplicationContext` and `BeanFactory` interfaces are both used for managing and accessing beans in a Spring application. However, they differ in terms of their features, capabilities, and the level of functionality they provide.

\*\*ApplicationContext\*\*:

- The `ApplicationContext` interface is a subinterface of the `BeanFactory` interface. It provides an advanced and enhanced container for managing beans and their dependencies.

- It extends the functionality of the `BeanFactory` by providing additional features such as internationalization, event publication, and application-specific context management.

- The `ApplicationContext` is typically preferred over the `BeanFactory` in most applications, as it provides a more comprehensive and feature-rich container.

\*\*BeanFactory\*\*:

- The `BeanFactory` interface is the basic and most fundamental container for managing beans in a Spring application.

- It provides the fundamental functionality of managing bean instantiation, configuration, and dependency resolution.

- The `BeanFactory` acts as the core container and is responsible for creating, configuring, and managing beans based on the bean definitions provided.

- It is a lightweight container and focuses primarily on the basic bean management functions.

In summary, the `BeanFactory` interface provides the basic functionality of managing beans, while the `ApplicationContext` interface extends it with additional features and functionality. The `ApplicationContext` is recommended for most applications due to its enhanced capabilities and additional features such as event handling and internationalization support.

When using Spring, you will typically work with the `ApplicationContext` rather than directly with the `BeanFactory` interface. Spring provides different implementations of the `ApplicationContext` interface, such as `ClassPathXmlApplicationContext` for XML-based configuration, `AnnotationConfigApplicationContext` for annotation-based configuration, and others.

Example usage of `ApplicationContext`:

```java

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

MyBean bean = context.getBean(MyBean.class);

// Use the bean...

```

Example usage of `BeanFactory`:

```java

BeanFactory factory = new XmlBeanFactory(new ClassPathResource("beans.xml"));

MyBean bean = factory.getBean(MyBean.class);

// Use the bean...

```

In both cases, you retrieve beans from the container by calling `getBean()` and providing the bean class or name as an argument.

I hope this clarifies the difference between `ApplicationContext` and `BeanFactory` in the Spring Framework. Let me know if you have any further questions!

## Spring singleton works internally

In the Spring Framework, the singleton scope is one of the most commonly used bean scopes. When a bean is defined with the singleton scope, only one instance of that bean is created and managed within the container.

Spring manages singleton beans using a cache, ensuring that subsequent requests for the same bean return the same shared instance from the cache. This reduces object creation overhead and promotes consistency in the application.

It's important to note that singleton beans are not thread-safe by default, so proper synchronization mechanisms should be used when accessing singleton beans concurrently.

## Request handler in spring

In Spring, there are several types of request handlers that you can use to handle incoming HTTP requests and process them accordingly. These request handlers are used in Spring MVC to define the endpoints of your web application and handle different types of requests. Here are the common types of request handlers in Spring:

1. `@Controller`:

- The `@Controller` annotation is used to mark a class as a request handler in Spring MVC.

- Methods inside a `@Controller` class are annotated with various mapping annotations like `@RequestMapping`, `@GetMapping`, `@PostMapping`, etc., to specify the URL paths and HTTP methods they handle.

- Controllers are used to handle different types of requests and serve as the entry points for processing those requests.

2. `@RestController`:

- The `@RestController` annotation is a specialized version of `@Controller`, designed specifically for building RESTful web services.

- Methods inside a `@RestController` class are annotated with mapping annotations like `@GetMapping`, `@PostMapping`, etc., to handle specific URL paths and HTTP methods for RESTful API endpoints.

- Unlike `@Controller`, the `@RestController` annotation automatically includes the `@ResponseBody` annotation on all the handler methods, making it easy to return data directly as JSON or XML.

3. `HttpRequestHandler`:

- The `HttpRequestHandler` interface is a low-level approach to handling requests in Spring MVC.

- To use this approach, you implement the `HttpRequestHandler` interface and override the `handleRequest()` method, which receives the incoming request and writes the response directly to the `HttpServletResponse` object.

- This type of handler provides more control over the request and response processing, but it requires more manual handling of the response generation.

4. `SimpleControllerHandlerAdapter`:

- The `SimpleControllerHandlerAdapter` is an adapter that allows you to use simple, non-annotated controllers in Spring MVC.

- Instead of using `@Controller`, you create a simple controller class that implements the `Controller` interface, which has a single method called `handleRequest()`.

- The `SimpleControllerHandlerAdapter` is automatically registered by Spring and maps requests to the `handleRequest()` method of your simple controllers.

5. `WebRequestHandler`:

- The `WebRequestHandler` interface is similar to `HttpRequestHandler`, but it provides access to additional features of the web request and response through the `WebRequest` and `WebResponse` objects.

- Implementing this interface gives you more fine-grained control over the web request and response processing.

The most commonly used request handler is `@Controller`, which allows you to create flexible and robust controllers to handle various types of requests and web application endpoints. However, you can choose the appropriate type of request handler based on the specific requirements and complexity of your application.

## Multiple datasources in spring

In Spring, you can configure multiple data sources by defining separate `DataSource` beans and using the `@Qualifier` annotation to specify which data source to use in different parts of your application. Here's a short guide with an example:

Step 1: Define Data Source Properties

Define the properties for each data source in your configuration file (e.g., `application.properties` or `application.yml`).

```properties

# Data Source 1

datasource1.url=jdbc:mysql://localhost:3306/db1

datasource1.username=user1

datasource1.password=password1

# Data Source 2

datasource2.url=jdbc:mysql://localhost:3306/db2

datasource2.username=user2

datasource2.password=password2

```

Step 2: Create Data Source Configurations

Create separate configuration classes for each data source to define the `DataSource` beans.

```java

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.boot.jdbc.DataSourceBuilder;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import javax.sql.DataSource;

@Configuration

public class DataSourceConfig1 {

@Bean(name = "dataSource1")

@ConfigurationProperties(prefix = "datasource1")

public DataSource dataSource1() {

return DataSourceBuilder.create().build();

}

}

@Configuration

public class DataSourceConfig2 {

@Bean(name = "dataSource2")

@ConfigurationProperties(prefix = "datasource2")

public DataSource dataSource2() {

return DataSourceBuilder.create().build();

}

}

```

Step 3: Use Data Sources in Your Application

Use the defined data sources in your application by autowiring them with the `@Qualifier` annotation.

```java

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.stereotype.Service;

import javax.sql.DataSource;

@Service

public class MyService {

private final DataSource dataSource1;

private final DataSource dataSource2;

@Autowired

public MyService(@Qualifier("dataSource1") DataSource dataSource1,

@Qualifier("dataSource2") DataSource dataSource2) {

this.dataSource1 = dataSource1;

this.dataSource2 = dataSource2;

}

// Use dataSource1 and dataSource2 in your service methods

}

```

With this configuration, you can interact with multiple databases or data sources by using the appropriate `DataSource` in different parts of your Spring application. The `@Qualifier` annotation allows you to specify which data source to inject, and you can also use `@Primary` to indicate a default data source when needed.

## Spring AOP

Spring AOP (Aspect-Oriented Programming) is a powerful feature of the Spring Framework that allows you to modularize cross-cutting concerns in your application. Cross-cutting concerns are functionalities that are applicable to multiple parts of your application, such as logging, security, caching, transaction management, etc. Spring AOP allows you to separate these concerns from the core business logic and apply them uniformly across different parts of your application.

Key Concepts in Spring AOP:

1. Aspect: An aspect is a modular unit of cross-cutting concern. It encapsulates the behavior that is to be applied to multiple parts of the application. Aspects can be thought of as reusable modules or interceptors.

2. Join Point: A join point is a specific point in the application's execution where an aspect can be applied. In Spring AOP, join points are method invocations, which are the primary targets of interception.

3. Advice: Advice is the actual behavior that an aspect provides. It defines what should happen at a particular join point. Common types of advice include "before" advice (executed before a method invocation), "after" advice (executed after a method invocation, regardless of its outcome), "around" advice (wraps a method invocation, providing complete control over the method's behavior), etc.

4. Pointcut: A pointcut is an expression that defines which join points should be matched and intercepted by an aspect. It is used to specify where the advice should be applied in the codebase.

5. Weaving: Weaving is the process of applying aspects to the target objects to create the final proxy objects. Spring AOP uses proxy-based weaving, where it creates proxy objects that encapsulate the target objects, intercepting method invocations and applying the appropriate advice.

Spring AOP uses a combination of XML-based configuration or Java-based configuration (using annotations) to define aspects, advice, pointcuts, and weaving. You can choose the appropriate method based on your preference and application structure.

Example of Spring AOP:

Let's consider an example of logging aspect using Spring AOP:

1. Create an Aspect for logging:

```java

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.springframework.stereotype.Component;

@Aspect

@Component

public class LoggingAspect {

@Before("execution(\* com.example.service.\*.\*(..))")

public void beforeServiceMethodExecution() {

System.out.println("Before executing a service method...");

}

}

```

2. Apply the Aspect to a Service:

```java

import org.springframework.stereotype.Service;

@Service

public class MyService {

public void doSomething() {

System.out.println("Doing something...");

}

}

```

When `doSomething()` method of `MyService` is invoked, the logging aspect defined in `LoggingAspect` will be executed before the method execution, and the message "Before executing a service method..." will be printed.

Spring AOP is a powerful tool that helps keep the core business logic clean and separate from cross-cutting concerns, promoting better modularity and maintainability in your application.

## Connect spring with hibernate

Connecting Spring with Hibernate involves integrating the Spring Framework with the Hibernate ORM (Object-Relational Mapping) library to manage the persistence layer of the application. Hibernate simplifies database interactions by mapping Java objects to database tables.

Here's a step-by-step guide to connecting Spring with Hibernate:

Step 1: Set Up Hibernate Configuration

Create a Hibernate configuration file, typically named `hibernate.cfg.xml`, to configure Hibernate settings like database connection details, dialect, and entity mapping. Place this file in the classpath.

```xml

<!-- hibernate.cfg.xml -->

<hibernate-configuration>

<session-factory>

<!-- Database connection settings -->

<property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>

<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/mydb</property>

<property name="hibernate.connection.username">username</property>

<property name="hibernate.connection.password">password</property>

<!-- Hibernate dialect for MySQL -->

<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

<!-- Enable Hibernate's automatic session context management -->

<property name="hibernate.current\_session\_context\_class">thread</property>

<!-- Mapping of entity classes -->

<mapping class="com.example.model.EntityClass1"/>

<mapping class="com.example.model.EntityClass2"/>

<!-- Add more entity classes here -->

</session-factory>

</hibernate-configuration>

```

Step 2: Create Entity Classes

Define your entity classes that represent the database tables. Annotate the classes with Hibernate annotations, such as `@Entity`, `@Table`, `@Column`, etc.

```java

// EntityClass1.java

@Entity

@Table(name = "table\_name1")

public class EntityClass1 {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(name = "column\_name1")

private String name;

// Getters and setters

}

```

Step 3: Set Up Spring Configuration

Create a Spring configuration file, typically named `applicationContext.xml` or use Java-based configuration using `@Configuration` and `@ComponentScan`.

```xml

<!-- applicationContext.xml -->

<context:component-scan base-package="com.example"/>

<!-- Add any additional Spring beans here if needed -->

```

Step 4: Integrate Spring with Hibernate

Configure Spring to manage Hibernate's `SessionFactory` and provide a Hibernate `TransactionManager` using the `LocalSessionFactoryBean` and `HibernateTransactionManager`.

```xml

<!-- applicationContext.xml -->

<bean id="sessionFactory" class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">

<property name="configLocation" value="classpath:hibernate.cfg.xml"/>

</bean>

<bean id="transactionManager" class="org.springframework.orm.hibernate5.HibernateTransactionManager">

<property name="sessionFactory" ref="sessionFactory"/>

</bean>

```

Step 5: Use Hibernate with Spring

Now, you can inject the `SessionFactory` or `HibernateTemplate` into your Spring components, such as services, and use Hibernate for database operations.

```java

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.orm.hibernate5.HibernateTemplate;

import org.springframework.stereotype.Service;

@Service

public class MyService {

private final HibernateTemplate hibernateTemplate;

@Autowired

public MyService(HibernateTemplate hibernateTemplate) {

this.hibernateTemplate = hibernateTemplate;

}

public void saveEntity(EntityClass1 entity) {

hibernateTemplate.save(entity);

}

// Other methods for data access and manipulation

}

```

By following these steps, you have successfully integrated Spring with Hibernate, allowing you to leverage Spring's features while managing the persistence layer using Hibernate.

## @Autowired and @Inject in spring? Can we replace @autowired with @inject

In Spring, both `@Autowired` and `@Inject` are used for dependency injection, and they serve a similar purpose:

1. `@Autowired`: This Spring-specific annotation automatically wires dependencies between beans.

Example:

```java

public class MyService {

private MyRepository myRepository;

@Autowired

public MyService(MyRepository myRepository) {

this.myRepository = myRepository;

}

}

```

2. `@Inject`: This annotation is part of the Java Dependency Injection (JSR-330) specification, and it is supported by Spring as well.

Example:

```java

public class MyService {

private MyRepository myRepository;

@Inject

public MyService(MyRepository myRepository) {

this.myRepository = myRepository;

}

}

```

You can use either `@Autowired` or `@Inject` in your Spring application. However, `@Autowired` provides additional features, such as optional dependencies and qualifiers, which make it more powerful and commonly used in Spring applications. To use `@Inject`, ensure you have the necessary dependencies for JSR-330 in your project.

## Get bean in core spring

To get a bean in core Spring, you need to create an application context and retrieve the bean from the context. Below is an example of how to do this:

1. Create a simple Spring bean:

```java

public class MyBean {

private String message;

public void setMessage(String message) {

this.message = message;

}

public String getMessage() {

return message;

}

}

```

2. Create the Spring configuration XML file (e.g., `applicationContext.xml`):

```xml

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<!-- Define the MyBean bean -->

<bean id="myBean" class="com.example.MyBean">

<property name="message" value="Hello, this is a Spring bean!" />

</bean>

</beans>

```

3. Create the main Java application class:

```java

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

// Create the application context and load the Spring configuration

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

// Get the MyBean bean from the context

MyBean myBean = (MyBean) context.getBean("myBean");

// Use the bean

String message = myBean.getMessage();

System.out.println(message);

}

}

```

4. Run the `MainApp` class, and you should see the output: "Hello, this is a Spring bean!"

In this example, we create a simple Spring bean named `MyBean`, configure it in the `applicationContext.xml` file, and retrieve it using the `getBean()` method of the `ApplicationContext` class. Once we have the bean, we can use it as needed.

Please make sure to have the required Spring dependencies in your classpath to run the example successfully. In this example, you need the `spring-context` and `spring-beans` libraries. If you're using Maven or Gradle, include the necessary dependencies in your build file.

## Lifecycle of Bean in spring

In Spring, the lifecycle of a bean refers to the stages that a bean goes through, from its instantiation to its destruction. This understanding is crucial for efficient resource management and executing necessary operations at different stages of a bean's existence. The bean lifecycle in Spring consists of the following phases:

1. Instantiation: The bean is created using the constructor or factory method. The container locates the bean's definition and initializes the object.

2. Dependency Injection: Spring injects any dependencies required by the bean, typically provided through properties, constructor arguments, or method parameters.

3. Initialization: The bean is initialized by invoking its `init` methods (if present). These methods can be annotated with `@PostConstruct` or implement the `InitializingBean` interface.

4. Use: The bean is now available for use and can perform its intended functionality.

5. Destruction: When the application context is closed or the bean is no longer needed, the container calls the bean's `destroy` methods (if present). These methods can be annotated with `@PreDestroy` or implement the `DisposableBean` interface.

6. Garbage Collection: Finally, the bean becomes eligible for garbage collection by the JVM when there are no more references to it.

Example:

```java

import javax.annotation.PostConstruct;

import javax.annotation.PreDestroy;

public class MyBean {

@PostConstruct

public void init() {

System.out.println("Bean initialized.");

}

public void performTask() {

System.out.println("Task performed.");

}

@PreDestroy

public void destroy() {

System.out.println("Bean destroyed.");

}

}

```

In this example, the `MyBean` class has `init` and `destroy` methods annotated with `@PostConstruct` and `@PreDestroy`, respectively. These methods will be automatically called by the Spring container during the bean's initialization and destruction phases.

By understanding the bean lifecycle, you can utilize the provided hooks (such as `@PostConstruct` and `@PreDestroy`) to execute setup and cleanup tasks, ensuring that your beans are efficiently managed and utilized in a Spring application.

## @Primary annotation in Spring

In Spring, the `@Primary` annotation is used to indicate a primary bean when there are multiple beans of the same type in the application context. It is typically used with `@Autowired` or `@Inject` to specify a preferred bean for autowiring when there is ambiguity.

Example:

```java

@Component

@Primary

public class EmailService implements MessageService {

public void sendMessage(String message) {

// Logic for sending an email

}

}

@Component

public class SmsService implements MessageService {

public void sendMessage(String message) {

// Logic for sending an SMS

}

}

@Component

public class NotificationService {

private final MessageService messageService;

@Autowired

public NotificationService(MessageService messageService) {

this.messageService = messageService;

}

public void sendNotification(String message) {

messageService.sendMessage(message);

}

}

```

In this example, we have two implementations of the `MessageService` interface: `EmailService` and `SmsService`. Both are marked as `@Component` and eligible for autowiring.

By adding `@Primary` to the `EmailService`, we indicate that it is the primary bean for autowiring when a `MessageService` is requested. So, in the `NotificationService` constructor, when Spring resolves the `MessageService` dependency, it will inject the `EmailService` bean as the primary choice due to the `@Primary` annotation.

Similarly, if another class, such as `MyService`, has a dependency on `MessageService` and uses `@Autowired`, the `EmailService` bean will be injected by default due to its `@Primary` annotation.

Using `@Primary`, you explicitly specify a primary bean for autowiring, resolving ambiguity when multiple candidates of the same type are available. This ensures that the intended bean is selected as the default choice for autowiring.

## @Qualifier annotation in Spring

In Spring MVC, the `@Qualifier` annotation is used to disambiguate between multiple beans of the same type in the application context. When you have multiple implementations of an interface or multiple beans of the same type, `@Qualifier` helps specify which bean should be autowired or injected.

Example:

```java

@Component

@Qualifier("implementation1")

public class Implementation1 implements MyInterface {

// Implementation details

}

@Component

@Qualifier("implementation2")

public class Implementation2 implements MyInterface {

// Implementation details

}

```

In this example, two implementations of the `MyInterface` interface, `Implementation1` and `Implementation2`, are defined. The `@Qualifier` annotation is used to assign unique identifiers to each implementation.

```java

@Component

public class MyService {

@Autowired

@Qualifier("implementation1")

private MyInterface myImplementation;

// Rest of the class

}

```

In the `MyService` class, the `@Qualifier("implementation1")` annotation specifies that the bean with the qualifier "implementation1" should be injected into the `myImplementation` field.

By using `@Qualifier`, you can specify which specific bean instance should be wired, resolving any ambiguity when multiple beans of the same type exist. This ensures that the correct bean is injected based on the provided qualifier value.

`@Qualifier` can be used with constructor injection, field injection, or setter injection. Ensure that the qualifier value specified in `@Qualifier` matches the qualifier value defined in the corresponding bean definition.

## @Component and @Bean

In Spring Framework, `@Bean` and `@Component` are annotations used to define beans and manage dependencies, but they differ in their usage and purpose.

1. \*\*@Bean\*\*: The `@Bean` annotation is used at the method level to explicitly declare a bean. It is typically used within a configuration class or a class annotated with `@Configuration`. When you annotate a method with `@Bean`, it tells Spring that the method should be used to instantiate and configure the bean. The return value of the method represents the bean instance.

Example:

```java

@Configuration

public class MyConfiguration {

@Bean

public MyBean myBean() {

return new MyBean();

}

}

```

In this example, the `myBean()` method is annotated with `@Bean`, indicating that it should be used to create a bean of type `MyBean`. The method body contains the logic to instantiate and configure the bean, and the returned object becomes the managed bean in the Spring application context.

2. \*\*@Component\*\*: The `@Component` annotation is a generic annotation used to mark a class as a Spring-managed component. It is a broad annotation that can be used to annotate any class you want to manage as a Spring bean. Classes annotated with `@Component` are automatically detected and registered as beans in the Spring application context.

Example:

```java

@Component

public class MyComponent {

// Class implementation

}

```

In this example, `MyComponent` is annotated with `@Component`, indicating that it is a Spring component. It will be automatically scanned and registered as a bean in the Spring application context.

The key difference between `@Bean` and `@Component` is in their usage and granularity. `@Bean` is used at the method level to explicitly define individual beans, while `@Component` is used at the class level to mark a class as a Spring component that should be managed as a bean.

Typically, `@Bean` is used when you need fine-grained control over the bean creation process, such as when creating beans with custom configuration or initializing external resources. On the other hand, `@Component` is used for general bean registration and is suitable for most scenarios where you want Spring to automatically detect and manage your beans.

It's also worth mentioning that `@Component` is a stereotype annotation, meaning it serves as a general-purpose marker for any Spring-managed component, while `@Bean` is used specifically to define individual beans and their configuration.

I hope this clarifies the difference between `@Bean` and `@Component` in Spring. Let me know if you have any further questions!

## IOC and DI

Inversion of Control (IOC) and Dependency Injection (DI) are core principles in the Spring Framework, promoting loose coupling and better separation of concerns in your application.

1. Inversion of Control (IOC):

- IOC is a design principle where control over the flow of your application is shifted from your code to a container (e.g., Spring container).

- The container manages the configuration and lifecycle of objects (beans), and your application uses these beans without worrying about their creation or management.

2. Dependency Injection (DI):

- DI is a specific implementation of IOC, achieved by injecting dependencies from the outside rather than creating them within a class.

- Dependencies are provided to a class at runtime, promoting loose coupling and testability.

Example of DI in Spring:

```java

// Interface representing a UserRepository

public interface UserRepository {

List<User> getAllUsers();

void saveUser(User user);

// Other methods...

}

// Implementation of UserRepository

public class UserRepositoryImpl implements UserRepository {

// Implementation of the UserRepository methods...

}

// UserService that depends on UserRepository

public class UserService {

private UserRepository userRepository;

// Constructor-based Dependency Injection

public UserService(UserRepository userRepository) {

this.userRepository = userRepository;

}

// Business logic using UserRepository

public List<User> getAllUsers() {

return userRepository.getAllUsers();

}

public void saveUser(User user) {

userRepository.saveUser(user);

}

// Other methods...

}

```

In the Spring configuration, you wire the beans together:

```java

@Configuration

public class AppConfig {

@Bean

public UserRepository userRepository() {

return new UserRepositoryImpl();

}

@Bean

public UserService userService() {

return new UserService(userRepository());

}

}

```

With DI, Spring automatically creates instances of `UserRepositoryImpl` and `UserService`, injecting the `UserRepository` dependency into the `UserService`.

By embracing IOC and DI in Spring, you achieve a more maintainable and flexible application architecture, as the Spring container manages object creation and dependency management, allowing you to focus on writing business logic and reducing coupling between components.

## Lifecycle of Maven

The lifecycle of Maven refers to the series of phases and goals that are executed during the build process of a Maven project. Maven follows a predefined and standardized build lifecycle, and each phase represents a specific step in the build process. The Maven lifecycle consists of three built-in lifecycles: default, clean, and site. Each lifecycle is made up of various phases, and each phase consists of a set of goals that are executed in a specific order. Here are the main phases of the default Maven lifecycle:

1. validate: Validates the project is correct and all necessary information is available.

2. compile: Compiles the source code of the project.

3. test: Runs unit tests for the compiled code.

4. package: Packages the compiled code into a distributable format, such as JAR, WAR, or EAR.

5. verify: Runs any checks on the results of integration tests to ensure quality criteria are met.

6. install: Installs the package into the local Maven repository, making it available for other projects to use.

7. deploy: Copies the final package to a remote repository, making it available for other developers to use.

In addition to the default lifecycle, Maven provides two additional built-in lifecycles:

1. clean lifecycle: Consists of phases to clean the project's build artifacts.

2. site lifecycle: Consists of phases to create project documentation and reports.

Each phase is associated with one or more default goals. You can also bind custom goals to the lifecycle phases, giving you control over the build process and allowing you to customize it to meet your project's specific requirements.

To execute a specific phase or goal in the Maven lifecycle, you use the `mvn` command followed by the phase or goal name. For example:

- To compile the source code, you run: `mvn compile`

- To run tests, you run: `mvn test`

- To package the project, you run: `mvn package`

By following the Maven lifecycle, you ensure that each step of the build process is performed in the correct order and that the project is built consistently across different environments. Maven's lifecycle and plugin system provide a powerful and standardized way to manage the build and project management processes in your Java projects.

## Types of view resolver

In Spring MVC, a view resolver is responsible for resolving logical view names returned from controller methods into actual view implementations that will be used to render the response. Different types of view resolvers are available in Spring, allowing you to choose the most suitable one based on your application's requirements. Here are some commonly used view resolvers in Spring:

1. InternalResourceViewResolver:

- This view resolver is commonly used to resolve JSP views. It prepends a prefix and appends a suffix to the logical view name to create the path to the JSP file.

- For example, if the logical view name is "home," and the prefix is "/WEB-INF/views/" and the suffix is ".jsp," the resolver will look for the view file at "/WEB-INF/views/home.jsp."

2. UrlBasedViewResolver:

- UrlBasedViewResolver is a more general-purpose view resolver that allows you to use different view technologies like JSP, Thymeleaf, FreeMarker, etc.

- It uses patterns to match logical view names with view implementations. You can specify different view classes for different patterns.

3. ResourceBundleViewResolver:

- This view resolver is used when you have views defined in property files. It resolves the logical view name using a resource bundle.

- It's helpful when you want to internationalize your views and store view names in different property files based on different locales.

4. TilesViewResolver:

- TilesViewResolver is used in combination with Apache Tiles framework, which provides a templating mechanism for defining page layouts and assembling views.

- It allows you to reuse common layouts and components across different views, improving code reusability and maintainability.

5. ContentNegotiatingViewResolver:

- This view resolver is used for content negotiation, allowing the application to return different views based on the requested media types (e.g., JSON, XML, HTML).

- It helps to handle scenarios where the same data needs to be rendered in different formats.

6. BeanNameViewResolver:

- BeanNameViewResolver resolves views based on their bean names defined in the Spring application context.

- It can be used to create custom views as Spring beans and use them based on their bean names.

It's worth noting that the choice of view resolver depends on your application architecture, view technology, and requirements. The most commonly used view resolver is the InternalResourceViewResolver for resolving JSP views in Spring MVC applications. However, if you are using other view technologies like Thymeleaf, FreeMarker, or Velocity, you can use the corresponding view resolvers accordingly.

## Suffix and Prefix in view resolver bean

In Spring MVC, a view resolver bean is used to resolve logical view names returned by controller methods into the actual view templates that will be rendered to the user. The view resolver uses the "prefix" and "suffix" properties to construct the complete path to the view template.

Example using InternalResourceViewResolver:

```xml

<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/views/" />

<property name="suffix" value=".jsp" />

</bean>

```

In this example, the InternalResourceViewResolver is configured with a prefix of "/WEB-INF/views/" and a suffix of ".jsp". When a controller method returns the logical view name "home", the view resolver will construct the view path as "/WEB-INF/views/home.jsp" and render the corresponding JSP file.

By using the view resolver with appropriate prefixes and suffixes, you can organize your view templates in a standardized way and easily switch between different view technologies without changing the controller logic.

# Spring Boot:

## @SpringBootApplication annotation.

The `@SpringBootApplication` annotation in Spring Boot is a powerful meta-annotation that combines three essential annotations:

1. `@Configuration`: Indicates that the class is a Spring configuration class that defines and configures Spring beans.

2. `@EnableAutoConfiguration`: Enables Spring Boot's auto-configuration mechanism, which automatically configures various components based on the classpath and dependencies. This reduces the need for explicit configuration and provides sensible defaults.

3. `@ComponentScan`: Enables component scanning, allowing Spring to automatically discover and register beans in the specified package and its sub-packages.

By using `@SpringBootApplication`, you can quickly create a Spring Boot application with minimal configuration, taking advantage of Spring's powerful features and conventions.

Example:

```java

@SpringBootApplication

public class MySpringBootApplication {

public static void main(String[] args) {

SpringApplication.run(MySpringBootApplication.class, args);

}

}

```

In this example, `@SpringBootApplication` is applied to the main class of the Spring Boot application. It enables component scanning in the package and its sub-packages, performs auto-configuration, and starts the application context when the `main` method is executed.

With just one annotation, you can easily bootstrap a Spring Boot application and leverage the benefits of Spring's inversion of control, auto-configuration, and component scanning.

## Annotations in Spring Boot application

In a Spring Boot application, various annotations are used to configure and customize different aspects of the application. These annotations simplify the development process and enable the auto-configuration mechanism provided by Spring Boot. Here are some of the commonly used annotations in a Spring Boot application:

1. `@SpringBootApplication`: As mentioned earlier, this meta-annotation combines `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan`. It is used to annotate the main class of the Spring Boot application.

2. `@RestController`: This annotation is used to define a RESTful controller in the application. It combines `@Controller` and `@ResponseBody`, indicating that the return value of the methods should be serialized directly into the response body.

3. `@RequestMapping`: This annotation is used to map HTTP requests to specific controller methods. It defines the URL pattern that the method should handle and the HTTP method (GET, POST, PUT, DELETE, etc.) it should respond to.

4. `@Autowired`: This annotation is used for dependency injection. It allows Spring to automatically wire beans (components, services, etc.) into the fields, constructor, or setter methods of other beans.

5. `@Service`: This annotation is used to mark a class as a service component. It is typically used for business logic and service layer classes.

6. `@Repository`: This annotation is used to mark a class as a repository component. It is used for data access and working with databases.

7. `@Component`: This annotation is a generic stereotype annotation used to mark a class as a Spring component. It serves as a generic stereotype for any Spring-managed component.

8. `@Configuration`: This annotation is used to define a Spring configuration class. It is used along with `@Bean` to create and configure Spring beans.

9. `@Value`: This annotation is used to inject values from properties files or environment variables into Spring beans.

10. `@ConditionalOnProperty`: This annotation is used to conditionally enable or disable a bean or configuration based on the presence and value of specific properties.

11. `@EnableCaching`: This annotation is used to enable Spring's caching support. It allows you to cache the results of methods to improve performance.

12. `@EnableTransactionManagement`: This annotation enables Spring's declarative transaction management. It allows you to use `@Transactional` on methods to define transactional behavior.

These are just a few examples of the many annotations available in Spring Boot. Each annotation serves a specific purpose and contributes to the overall ease of development and configuration in a Spring Boot application. The extensive use of annotations in Spring Boot makes it easier to set up and manage complex applications with less boilerplate code.

## Spring and spring boot

Sure! Here are the key differences between Spring and Spring Boot in simple points in Hinglish:

1. \*\*Purpose\*\*:

- Spring: Spring ek comprehensive Java framework hai jo enterprise-level applications banane ke liye various modules aur features provide karta hai.

- Spring Boot: Spring Boot Spring framework ka ek extension hai jo Spring applications banane ki process ko simplify aur accelerate karta hai, focusing on rapid application development aur easy setup pe.

2. \*\*Configuration\*\*:

- Spring: Spring mein aapko application ko manually XML-based ya Java-based configuration ke through set up karna hota hai. Aapko various dependencies aur configurations ko khud manage karna padta hai.

- Spring Boot: Spring Boot opinionated defaults aur auto-configuration provide karta hai, manual configuration ki jarurat kam kar deta hai. Ye application ko sensible defaults ke saath set up karta hai, jisse minimal setup ke saath start karne mein aasani hoti hai.

3. \*\*Embedded Server\*\*:

- Spring: Spring mein ek embedded web server included nahi hota hai, isliye aapko application ko external web server jaise Tomcat ya Jetty mein deploy karna padta hai.

- Spring Boot: Spring Boot ek embedded web server (Tomcat, Jetty, ya Undertow) include karta hai, jisse standalone web applications banane mein madad milti hai, aur application ko bina external server ke run kar sakte hai.

4. \*\*Dependency Management\*\*:

- Spring: Spring mein dependencies ko manually manage karna padta hai, jo complex aur time-consuming ho sakta hai.

- Spring Boot: Spring Boot dependency management provide karta hai aur Starter POMs ke through dependencies ko manage karne ka process asaan banata hai. Ye dependencies ko automatically resolve aur configure karta hai.

5. \*\*Production-Ready Features\*\*:

- Spring: Spring production-ready features ko enforce nahi karta hai, aur aapko unhe manually implement karna padta hai ya third-party libraries ke through use karna padta hai.

- Spring Boot: Spring Boot production-ready applications ke liye best practices promote karta hai. Ye health checks, metrics, logging, aur externalized configurations jaise features ko out-of-the-box provide karta hai.

6. \*\*Ease of Use\*\*:

- Spring: Spring ko set up karne mein aur configuration karne mein thoda jyada manual effort chahiye, jo complex applications ke liye suitable ho sakta hai.

- Spring Boot: Spring Boot developer-friendly hai aur boilerplate code ko kam kar deta hai. Ye jaldi start karne aur chote applications ya microservices banane ke liye perfect hai.

## Actuator in Spring Boot

In Spring Boot, Actuator is a set of production-ready features that provide insight into the running application's health, metrics, monitoring, and management. It allows developers and system administrators to monitor and manage the application effectively in a production environment. Actuator exposes various endpoints that can be accessed via HTTP or JMX (Java Management Extensions).

Actuator provides several out-of-the-box features, including:

1. Health Checks: The `/actuator/health` endpoint provides information about the application's health status. It indicates whether the application is running correctly or if there are any problems.

2. Metrics: The `/actuator/metrics` endpoint provides various application-specific metrics, such as CPU usage, memory usage, request counts, and many others. These metrics can be used for performance monitoring and optimization.

3. Environment Information: The `/actuator/env` endpoint provides information about the application's environment, including system properties, environment variables, and configuration properties.

4. Logging: The `/actuator/loggers` endpoint allows you to view and modify the logging levels of different components in the application at runtime.

5. Thread Dump: The `/actuator/threaddump` endpoint captures and provides a thread dump, which can be useful for identifying threads that are stuck or causing performance issues.

6. Heap Dump: The `/actuator/heapdump` endpoint generates and downloads a heap dump, which is useful for analyzing memory-related issues.

7. Auditing: The `/actuator/auditevents` endpoint provides information about application auditing events.

8. Info: The `/actuator/info` endpoint can be used to provide custom application information.

To use Actuator in a Spring Boot application, you need to include the `spring-boot-starter-actuator` dependency in your project's build configuration (e.g., Maven or Gradle). Once included, Actuator endpoints are automatically enabled, and you can access them using HTTP or JMX.

For example, to access the health endpoint, you can make an HTTP GET request to `/actuator/health`. Similarly, to access the metrics endpoint, you can make an HTTP GET request to `/actuator/metrics`.

It's important to note that Actuator endpoints expose sensitive information about the application, so it is crucial to secure the Actuator endpoints appropriately in production environments. Spring Security can be used to secure the Actuator endpoints based on specific roles or permissions.

Actuator is a valuable tool for monitoring and managing Spring Boot applications, making it easier to diagnose and troubleshoot issues in production environments.

## Embedded server in Spring Boot

Sure! Here's a point-wise short note on embedded servers in Spring Boot:

- Embedded Server: An embedded server in Spring Boot refers to a web server integrated directly into the application.

- Standalone Executable: It allows the Spring Boot application to run as a standalone executable, eliminating the need for deploying to an external web server.

- Supported Servers: Spring Boot provides built-in support for embedded servers like **Tomcat, Jetty, and Undertow.**

- Simplified Deployment: With an embedded server, there's no need to separately deploy the application to an external web server.

- Self-Contained Executable: The application's executable JAR or WAR file includes the embedded server and all necessary dependencies, making it self-contained and easy to distribute.

- Reduced Overhead: Using an embedded server reduces memory and resource overhead by eliminating the need for running a separate web server process.

- Fast Startup: Embedded servers are optimized for fast startup times, benefiting applications in microservices or cloud-native environments.

- Easy Configuration: By including the appropriate starter dependency, the embedded server is automatically configured for the application.

- Execution: The Spring Boot application can be run as an executable JAR using the `java -jar` command, making deployment simple and portable.

Overall, embedded servers in Spring Boot streamline the deployment and distribution of web applications, providing a convenient and efficient way to run applications as self-contained executables.

## Servers along with tomcat in spring boot

In Spring Boot, besides Tomcat, there are other embedded servers that you can use to run your web applications without the need for deploying them to an external web server. Spring Boot provides built-in support for the following embedded servers:

1. Jetty:

- Jetty is a lightweight and high-performance web server and servlet container.

- To use Jetty as the embedded server in your Spring Boot application, include the `spring-boot-starter-jetty` dependency in your project's build configuration.

2. Undertow:

- Undertow is a flexible and high-performance web server designed for non-blocking and asynchronous processing.

- To use Undertow as the embedded server in your Spring Boot application, include the `spring-boot-starter-undertow` dependency in your project's build configuration.

3. Netty (Community-contributed):

- Netty is an asynchronous event-driven network application framework.

- Netty is a community-contributed embedded server and is not officially supported by Spring Boot.

- To use Netty as the embedded server in your Spring Boot application, include the `spring-boot-starter-netty` dependency in your project's build configuration.

You can choose the embedded server that best suits your application's requirements and performance needs. The choice of the embedded server can also depend on factors like the nature of the application, the expected workload, and specific features provided by each server.

Example of using Jetty as the embedded server in a Maven-based project:

```xml

<dependencies>

<!-- Other dependencies -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jetty</artifactId>

</dependency>

</dependencies>

```

With the appropriate embedded server starter included, Spring Boot will automatically use the specified server to run your application when you launch it as a standalone executable using the `java -jar` command.

Remember that Spring Boot's embedded server support simplifies the deployment and distribution of web applications and allows you to create self-contained executable JAR files that can be run anywhere with ease.

## Handle transaction in spring boot

Sure! Here's a short note on handling transactions in Spring Boot with an example:

\*\*Handling Transactions in Spring Boot:\*\*

1. \*\*Enable Transaction Management:\*\*

Enable transaction management in the Spring Boot application by adding the `@EnableTransactionManagement` annotation to the main configuration class.

```java

@SpringBootApplication

@EnableTransactionManagement

public class MySpringBootApplication {

public static void main(String[] args) {

SpringApplication.run(MySpringBootApplication.class, args);

}

}

```

2. \*\*Define Transactional Methods:\*\*

Use the `@Transactional` annotation on methods that need to be executed within a transactional context.

```java

@Service

public class MyService {

@Autowired

private MyRepository myRepository;

@Transactional

public void performTransactionalOperation() {

// Perform database operations (e.g., insert, update, delete)

}

// Other methods...

}

```

3. \*\*Configure Transaction Propagation (Optional):\*\*

Customize transaction behavior using attributes like propagation, isolation level, read-only, rollback rules, etc.

```java

@Transactional(propagation = Propagation.REQUIRED, isolation = Isolation.READ\_COMMITTED, readOnly = false)

public void performTransactionalOperation() {

// Perform database operations

}

```

In this example, the `performTransactionalOperation()` method is annotated with `@Transactional`, indicating that any database operations within this method should be executed within a transactional context. If an exception occurs during the method execution, the transaction will be rolled back, ensuring data consistency.

Spring Boot's `@Transactional` annotation simplifies the handling of transactions, allowing you to focus on your business logic without worrying about low-level transaction management.

## Deploy spring boot application on other servers what you will do

To deploy a Spring Boot application on other servers, you need to create an executable JAR or a WAR file and then deploy it to the desired server. Here's what you need to do in your code to ensure smooth deployment on other servers:

1. Create an Executable JAR or WAR:

In Spring Boot, you can create an executable JAR or WAR file that includes an embedded server, allowing the application to run as a standalone application. To create an executable JAR, you need to include the `spring-boot-maven-plugin` in your Maven build configuration.

Example `pom.xml` for creating an executable JAR:

```xml

<!-- Other dependencies and configurations -->

<build>

<plugins>

<!-- Include the spring-boot-maven-plugin -->

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

```

Then, you can build the JAR using the `mvn package` command, and it will be located in the `target` directory.

2. Choose the Appropriate Embedded Server:

When deploying on other servers, you don't need to explicitly start the embedded server provided by Spring Boot. Instead, you can use the server provided by the hosting environment. In your code, ensure that the embedded server is not started or conflicts with the server provided by the hosting environment.

3. Configure Externalized Properties:

Make sure that your Spring Boot application is configured to read its configuration from external properties files. This allows you to modify the application's configuration without rebuilding the JAR. Commonly, applications read properties from a file named `application.properties` or `application.yml`. By externalizing the configuration, you can easily adjust settings specific to the deployment environment without modifying the code.

4. Set the Appropriate Context Path (if required):

When deploying on an external server, you might need to specify a context path for your application. The context path is the part of the URL that comes after the server domain. In Spring Boot, you can set the context path using the `server.servlet.context-path` property in the external properties file.

Example `application.properties`:

```

server.servlet.context-path=/myapp

```

With these considerations in place, your Spring Boot application should be ready for deployment on other servers. You can copy the executable JAR or WAR file to the target server and run it using the appropriate Java command. Remember to ensure that the hosting environment meets the requirements of your application (Java version, database, etc.) and that you have configured any required database connections or other external resources correctly for the new environment.

## How to configure simple spring security in spring boot

To configure simple Spring Security in a Spring Boot application, you can follow these steps:

1. Add Spring Security Dependency:

In your `pom.xml` (if using Maven) or `build.gradle` (if using Gradle), add the Spring Security dependency:

Maven:

```xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

```

Gradle:

```gradle

implementation 'org.springframework.boot:spring-boot-starter-security'

```

2. Configure Security Rules:

By default, Spring Security will enable basic authentication and require users to authenticate using a username and password. You can customize the security rules by creating a security configuration class that extends `WebSecurityConfigurerAdapter`.

```java

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http

.authorizeRequests()

.antMatchers("/public/\*\*").permitAll() // Allow access to public resources

.antMatchers("/private/\*\*").authenticated() // Require authentication for private resources

.and()

.formLogin(); // Use form-based login

}

}

```

In this example, we've defined a basic security configuration. Requests to URLs starting with `/public/` are allowed for everyone, while requests to URLs starting with `/private/` require authentication (login).

3. Create User Accounts (Optional):

If you want to have predefined user accounts for testing purposes, you can configure them in the application.properties or application.yml file.

Example `application.properties`:

```

spring.security.user.name=user

spring.security.user.password=password

```

With these properties, Spring Security will automatically create a user with the provided username and password for testing purposes.

4. Run the Application:

Once you've configured Spring Security, run your Spring Boot application. When you access the secured endpoints (e.g., /private/\*\* in the example above), Spring Security will prompt you to log in with the credentials provided in the configuration.

These are the basic steps to set up simple Spring Security in a Spring Boot application. Spring Security provides many features for advanced security configurations, such as using custom authentication providers, handling roles and authorities, enabling CSRF protection, etc. You can refer to the official Spring Security documentation for more detailed configurations and customizations: https://docs.spring.io/spring-security/site/docs/current/reference/html5/

## Important feature of Spring Boot

Spring Boot is a powerful framework that simplifies the development of Java applications, especially web applications and microservices. Some of the important features of Spring Boot include:

1. Auto-configuration: Spring Boot provides auto-configuration, which automatically configures the application based on the classpath and the dependencies present in the project. This feature reduces the need for explicit configuration and boilerplate code.

2. Embedded servers: Spring Boot includes embedded web servers (Tomcat, Jetty, or Undertow), allowing applications to run as standalone executable JAR files without the need for deploying them to an external web server.

3. Starter dependencies: Spring Boot uses "starter" dependencies, which are pre-configured sets of dependencies that provide specific functionality, such as web, data, security, etc. Including a starter dependency automatically pulls in all the required libraries and configurations.

4. Spring Boot Actuator: Actuator provides production-ready features for monitoring and managing the application, such as health checks, metrics, logging, environment information, and more.

5. Externalized configuration: Spring Boot allows you to externalize the application's configuration using properties files (application.properties or application.yml), making it easier to customize the application for different environments.

6. Spring Data JPA: Spring Boot provides integration with Spring Data JPA, simplifying database access and allowing developers to interact with databases using a rich set of repository interfaces.

7. Spring Security: Spring Boot integrates with Spring Security, enabling developers to add security features to their applications, such as authentication, authorization, and CSRF protection.

8. Thymeleaf and Spring MVC: Spring Boot includes support for Thymeleaf, a modern server-side template engine, and Spring MVC, which simplifies building web applications.

9. Actuator endpoints and custom metrics: Spring Boot Actuator endpoints can be extended to provide custom metrics and management information based on specific application requirements.

10. DevTools: Spring Boot DevTools provides developer-focused tools, such as automatic application restarts, remote debugging, and live reloading of templates and static resources.

11. Cloud-native support: Spring Boot is designed with cloud-native development in mind, providing features for easy integration with cloud platforms and microservices architectures.

12. Spring Boot CLI: The Spring Boot Command Line Interface (CLI) allows developers to quickly create, run, and test Spring Boot applications using simple commands.

These features, among others, make Spring Boot a popular choice for building robust and efficient Java applications with minimal configuration and maximum productivity. It streamlines the development process and allows developers to focus on writing business logic rather than dealing with boilerplate code and infrastructure setup.

## Profiling in Spring Boot

Profiling in Spring Boot is the process of gathering information about the application's performance and resource usage in different environments, such as development, testing, and production. It helps developers identify performance bottlenecks, memory leaks, and other issues that may impact the application's behavior and responsiveness. Spring Boot provides built-in support for profiling through the use of different profiles and Actuator endpoints.

Here's how profiling works in Spring Boot:

1. Profiles:

Spring Boot allows you to define profiles for different environments, such as "dev," "test," and "prod." Each profile can have its own configuration properties, enabling you to customize the behavior of the application based on the current environment. By default, Spring Boot uses the "default" profile if no other profile is active.

You can define profile-specific properties in separate property files, such as `application-dev.properties` for the "dev" profile, and `application-prod.properties` for the "prod" profile. Spring Boot will automatically pick the appropriate properties based on the active profile.

To activate a specific profile, you can use the `spring.profiles.active` property in the `application.properties` or `application.yml` file:

```properties

# application.properties

spring.profiles.active=dev

```

Or, you can specify it as a command-line argument:

```

java -jar myapp.jar --spring.profiles.active=dev

```

## Spring AOP with Spring Boot

Spring AOP (Aspect-Oriented Programming) is a powerful feature in the Spring Framework that allows you to separate cross-cutting concerns from the core business logic of your application. With Spring Boot, you can easily integrate Spring AOP to apply aspects across your application.

Here's how to use Spring AOP with Spring Boot:

1. Add Dependencies:

To use Spring AOP, you need to include the `spring-boot-starter-aop` dependency in your `pom.xml` (if using Maven) or `build.gradle` (if using Gradle).

Maven:

```xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-aop</artifactId>

</dependency>

```

Gradle:

```gradle

implementation 'org.springframework.boot:spring-boot-starter-aop'

```

2. Create Aspect:

An aspect in Spring AOP is a cross-cutting concern, such as logging, security, or performance monitoring. To create an aspect, you define a Java class annotated with `@Aspect`, and you define advice methods within that class. The advice methods contain the logic that should be executed before, after, or around the join points (places where the aspect is applied).

```java

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.springframework.stereotype.Component;

@Aspect

@Component

public class LoggingAspect {

@Before("execution(\* com.example.myapp.service.\*.\*(..))")

public void beforeServiceMethodExecution() {

System.out.println("Before executing service method.");

}

}

```

In this example, we define a simple logging aspect that prints a message before any method execution in the `com.example.myapp.service` package.

3. Enable Aspect Auto-Configuration (Optional):

By default, Spring Boot automatically detects and configures aspects when they are on the classpath. However, in some cases, you may need to explicitly enable aspect auto-configuration, especially if you're using custom configuration.

```java

import org.springframework.context.annotation.EnableAspectJAutoProxy;

@SpringBootApplication

@EnableAspectJAutoProxy

public class MySpringBootApplication {

// Main class code

}

```

4. Run the Application:

Once you've created the aspect and enabled aspect auto-configuration (if required), you can run your Spring Boot application. The aspect will automatically be applied to the methods matching the pointcut expression specified in the advice methods.

Spring AOP supports various types of advice, including `@Before`, `@After`, `@Around`, `@AfterReturning`, and `@AfterThrowing`, which you can use to define different aspects of your application.

Using Spring AOP with Spring Boot allows you to keep your application's core business logic clean and focused while separating cross-cutting concerns into aspects. It promotes a modular and maintainable architecture and makes it easy to apply aspects consistently across your application.

## Mutilple datasources with spring boot

\*\*Configuring Multiple Data Sources in Spring Boot\*\*

In Spring Boot, you can configure multiple data sources to work with different databases within a single application. Here's a step-by-step guide with a simple example:

1. \*\*Dependencies\*\*: Add the required dependencies in your `pom.xml` (Maven) or `build.gradle` (Gradle) file. For this example, we'll use H2 and MySQL databases.

```xml

<!-- H2 Database -->

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

<!-- MySQL Connector -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<scope>runtime</scope>

</dependency>

```

2. \*\*Configuration Properties\*\*: Define properties for each data source in `application.properties`:

```properties

# H2 Data Source

spring.datasource.h2.url=jdbc:h2:mem:testdb

spring.datasource.h2.username=sa

spring.datasource.h2.password=

# MySQL Data Source

spring.datasource.mysql.url=jdbc:mysql://localhost:3306/mydb

spring.datasource.mysql.username=root

spring.datasource.mysql.password=rootpassword

```

3. \*\*Create Configuration Classes\*\*: Create configuration classes for each data source. Here, we create `DataSource` beans for both H2 and MySQL:

```java

@Configuration

@EnableConfigurationProperties

public class DataSourceConfig {

@Bean

@ConfigurationProperties("spring.datasource.h2")

public DataSource h2DataSource() {

return DataSourceBuilder.create().build();

}

@Bean

@ConfigurationProperties("spring.datasource.mysql")

public DataSource mysqlDataSource() {

return DataSourceBuilder.create().build();

}

}

```

4. \*\*EntityManagerFactory and TransactionManager\*\*: Create `EntityManagerFactory` and `TransactionManager` beans for each data source:

```java

@Configuration

@EnableJpaRepositories(basePackages = "com.example.h2.repository", entityManagerFactoryRef = "h2EntityManagerFactory", transactionManagerRef = "h2TransactionManager")

public class H2JpaConfig {

@Bean

public LocalContainerEntityManagerFactoryBean h2EntityManagerFactory(EntityManagerFactoryBuilder builder, @Qualifier("h2DataSource") DataSource dataSource) {

return builder

.dataSource(dataSource)

.packages("com.example.h2.model") // Package where H2 entities are located

.persistenceUnit("h2DataSource")

.build();

}

@Bean

public JpaTransactionManager h2TransactionManager(@Qualifier("h2EntityManagerFactory") EntityManagerFactory entityManagerFactory) {

return new JpaTransactionManager(entityManagerFactory);

}

}

@Configuration

@EnableJpaRepositories(basePackages = "com.example.mysql.repository", entityManagerFactoryRef = "mysqlEntityManagerFactory", transactionManagerRef = "mysqlTransactionManager")

public class MySqlJpaConfig {

@Bean

public LocalContainerEntityManagerFactoryBean mysqlEntityManagerFactory(EntityManagerFactoryBuilder builder, @Qualifier("mysqlDataSource") DataSource dataSource) {

return builder

.dataSource(dataSource)

.packages("com.example.mysql.model") // Package where MySQL entities are located

.persistenceUnit("mysqlDataSource")

.build();

}

@Bean

public JpaTransactionManager mysqlTransactionManager(@Qualifier("mysqlEntityManagerFactory") EntityManagerFactory entityManagerFactory) {

return new JpaTransactionManager(entityManagerFactory);

}

}

```

5. \*\*Use Multiple Data Sources\*\*: Now, you can use different data sources in your repositories or services by specifying the corresponding `EntityManagerFactory` and `TransactionManager`:

```java

@Repository

public interface H2UserRepository extends JpaRepository<H2User, Long> {

// H2-specific repository methods

}

@Repository

public interface MySqlUserRepository extends JpaRepository<MySqlUser, Long> {

// MySQL-specific repository methods

}

```

By following these steps, your Spring Boot application is configured to work with multiple data sources, allowing you to interact with different databases separately within the same application.

## Global Exception Handling in Spring Boot

\*\*Global Exception Handling in Spring Boot\*\*

Global exception handling in Spring Boot allows you to handle exceptions that occur during request processing across the entire application. Here's a simple example of how to implement global exception handling:

1. \*\*Create a Custom Exception Class (Optional)\*\*:

Define custom exception classes if you need to represent specific types of errors in your application.

```java

public class CustomNotFoundException extends RuntimeException {

public CustomNotFoundException(String message) {

super(message);

}

}

```

2. \*\*Create a Global Exception Handler\*\*:

Create a class annotated with `@ControllerAdvice` to handle exceptions globally.

```java

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(CustomNotFoundException.class)

public ResponseEntity<String> handleNotFoundException(CustomNotFoundException ex) {

return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT\_FOUND);

}

@ExceptionHandler(Exception.class)

public ResponseEntity<String> handleOtherExceptions(Exception ex) {

return new ResponseEntity<>("An unexpected error occurred.", HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

```

In this example, we have a custom exception handler class with two methods. One method handles `CustomNotFoundException`, and the other handles any other generic `Exception`. Each method returns a custom `ResponseEntity` with an appropriate status code and error message.

3. \*\*Enable Global Exception Handling\*\*:

Make sure to include the package containing the `GlobalExceptionHandler` class in component scanning.

```java

@SpringBootApplication

@ComponentScan(basePackages = "com.example")

public class MySpringBootApplication {

// Main class code

}

```

With this setup, whenever an exception of a specified type (e.g., `CustomNotFoundException`) is thrown during request processing, the corresponding handler method will intercept the exception and return a customized response to the client.

Global exception handling is an effective way to provide consistent error handling and responses across your application. It centralizes the error handling logic and helps create a more user-friendly experience for clients when exceptions occur. Remember to avoid exposing sensitive information in error messages and handle exceptions appropriately based on the nature of your application.

## Read properties from application.properties

\*\*Reading Properties in Spring Boot\*\*

In Spring Boot, you can read properties from the `application.properties` file using `@Value` annotation or the `Environment` object.

1. \*\*Using `@Value` Annotation\*\*:

The `@Value` annotation injects property values into Spring beans.

```java

@Component

public class MyComponent {

@Value("${myapp.property1}")

private String property1;

@Value("${myapp.property2}")

private int property2;

// Getter methods (optional)

}

```

In this example, the `property1` and `property2` fields are injected with the values of `myapp.property1` and `myapp.property2` from `application.properties`.

2. \*\*Using `Environment` Object\*\*:

The `Environment` object provides access to properties.

```java

@Component

public class MyComponent {

private final Environment environment;

@Autowired

public MyComponent(Environment environment) {

this.environment = environment;

}

public void readProperties() {

String property1 = environment.getProperty("myapp.property1");

int property2 = Integer.parseInt(environment.getProperty("myapp.property2"));

// Use the properties

}

}

```

In this example, we use the `Environment` object to read properties.

Ensure the `application.properties` file is in `src/main/resources`. Spring Boot will automatically load properties from it during startup.

By reading properties from `application.properties`, you can easily configure your Spring Boot application without changing the source code, making it more flexible and configurable.

## Isolation and Propagation level in @Transactional

\*\*Spring's `@Transactional` Isolation and Propagation Levels\*\*

In Spring's `@Transactional` annotation, you can control transaction behavior using isolation and propagation levels:

1. \*\*Isolation Levels\*\*:

- `DEFAULT`: Uses the default isolation level of the underlying database (usually `READ\_COMMITTED`).

- `READ\_UNCOMMITTED`: Allows reading uncommitted changes made by other transactions, may lead to data inconsistency.

- `READ\_COMMITTED`: Allows reading only committed changes made by other transactions, avoids dirty reads.

- `REPEATABLE\_READ`: Ensures consistent reads throughout the transaction, avoids dirty and non-repeatable reads.

- `SERIALIZABLE`: Ensures complete isolation from other transactions, but can be restrictive.

```java

@Transactional(isolation = Isolation.READ\_COMMITTED)

public void someTransactionalMethod() {

// Method logic here

}

```

2. \*\*Propagation Levels\*\*:

- `REQUIRED`: Joins an existing transaction or starts a new one.

- `SUPPORTS`: Joins an existing transaction or runs non-transactionally.

- `MANDATORY`: Requires an existing transaction, throws an exception if none exists.

- `REQUIRES\_NEW`: Always starts a new transaction, suspends the current one if any.

- `NOT\_SUPPORTED`: Runs non-transactionally, suspends the current transaction if any.

- `NEVER`: Runs non-transactionally, throws an exception if a transaction exists.

- `NESTED`: Creates a nested transaction within the existing one.

```java

@Transactional(propagation = Propagation.REQUIRED)

public void someTransactionalMethod() {

// Method logic here

}

```

By using these isolation and propagation levels, you can control how transactions behave in different scenarios, ensuring data consistency and integrity in multi-threaded and multi-method environments.

## @RESTController vs @Controller

\*\*@Controller vs. @RestController in Spring\*\*

1. \*\*@Controller\*\*:

- Used for defining a controller class in Spring MVC for traditional web applications.

- Methods return a logical view name or a `ModelAndView` object.

- Responsible for processing requests and preparing data for rendering views.

```java

@Controller

public class MyController {

@RequestMapping("/home")

public String home() {

return "home"; // Returns the logical view name

}

}

```

2. \*\*@RestController\*\*:

- Used for creating RESTful web services that return response directly in the response body as JSON, XML, etc.

- Combines `@Controller` and `@ResponseBody`.

- Methods return the response data serialized directly into the response body.

```java

@RestController

public class MyRestController {

@RequestMapping("/api/data")

public MyData getData() {

// Retrieve data from a service or database

MyData data = // ...

return data; // Returns the response directly serialized as JSON

}

}

```

Choose `@Controller` for traditional web applications with views, and use `@RestController` for creating APIs that return data directly in the response body as JSON or XML.

## Change server port in Spring Boot

\*\*Changing Server Port in Spring Boot\*\*

You can change the server port in your Spring Boot application using various methods:

1. \*\*application.properties\*\* or \*\*application.yml\*\*:

Set `server.port` property in `application.properties` or `application.yml`:

```properties

# application.properties

server.port=8081

```

2. \*\*Command-Line Arguments\*\*:

Provide a command-line argument when starting the application:

```

java -jar myapp.jar --server.port=8081

```

3. \*\*System Environment Variable\*\*:

Set a system environment variable named `SERVER\_PORT`:

```

SET SERVER\_PORT=8081

```

4. \*\*Java System Property\*\*:

Specify the server port as a Java system property:

```

java -jar -Dserver.port=8081 myapp.jar

```

5. \*\*Programmatically in Code\*\*:

Set the server port programmatically in your main Spring Boot application class:

```java

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class MySpringBootApplication {

public static void main(String[] args) {

// Set the desired port programmatically

int port = 8081;

SpringApplication app = new SpringApplication(MySpringBootApplication.class);

app.setDefaultProperties(Collections.singletonMap("server.port", String.valueOf(port)));

app.run(args);

}

}

```

Choose the approach that best fits your requirements to change the server port in your Spring Boot application. The last specified value will take precedence if you use multiple methods to set the server port.

## Multiple application.properties files in springboot

\*\*Managing Multiple Configuration Property Files in Spring Boot\*\*

In Spring Boot, you can use multiple configuration property files to manage different configurations for your application. Here's how to do it:

1. \*\*Create Property Files\*\*:

Create the main `application.properties` (or `application.yml`) file with common properties. Then, create separate property files for different environments or scenarios, naming them based on the profiles:

- `application.properties` (Main configuration file with common properties)

- `application-dev.properties` (Development environment properties)

- `application-test.properties` (Testing environment properties)

- `application-prod.properties` (Production environment properties)

2. \*\*Define Properties\*\*:

Define the properties in each property file as needed. Properties defined in the specific environment files will override properties with the same name in the main `application.properties` file.

3. \*\*Activate Profile\*\*:

Activate a specific profile by setting the `spring.profiles.active` property in the `application.properties` or `application.yml` file:

```properties

# application.properties

spring.profiles.active=dev

```

Alternatively, you can specify the active profile as a command-line argument:

```

java -jar myapp.jar --spring.profiles.active=dev

```

Spring Boot will automatically pick the appropriate properties file based on the active profile.

4. \*\*Property Usage\*\*:

Use the properties in your application as usual, and Spring Boot will load the appropriate values based on the active profile.

By using multiple configuration property files, you can easily manage different configurations for various environments, scenarios, or categories, making your Spring Boot application more flexible and adaptable.

Example:

Imagine you have the following property files:

\*\*application.properties\*\*:

```properties

app.name=MyApp

app.version=1.0

```

\*\*application-dev.properties\*\*:

```properties

app.name=MyApp - Development

app.debug=true

```

\*\*application-prod.properties\*\*:

```properties

app.name=MyApp - Production

app.debug=false

```

When the `dev` profile is active, the property `app.name` will be overridden with the value "MyApp - Development", and `app.debug` will be set to `true`. Similarly, when the `prod` profile is active, `app.name` will be "MyApp - Production", and `app.debug` will be set to `false`.

Using multiple property files allows you to customize your application's behavior based on different environments or scenarios without modifying the code, making it easier to maintain and deploy.

## Application Server vs Web Server:

Application Server aur Web Server dono hi server software hote hain, lekin unke use cases aur functionalities mein thoda difference hota hai.

\*\*Web Server\*\*:

- Web Server web pages ko handle karta hai aur clients (jaise web browsers) ko static content serve karta hai, jaise HTML files, images, CSS, JavaScript, etc.

- Ye server static content ki HTTP requests ko receive karta hai aur unhe client browsers tak deliver karta hai.

- Web Server generally web applications ke front-end ko handle karta hai, jahaan par sirf static content hota hai aur dynamic content generate nahi hota hai.

- Kuch popular Web Servers hain Apache, Nginx, IIS (Internet Information Services) jo web pages ko deliver karne ke liye use hote hain.

\*\*Application Server\*\*:

- Application Server dynamic content ko handle karta hai, jo web applications mein use hota hai, jaise Java Servlets, JavaServer Pages (JSPs), EJB (Enterprise JavaBeans), .NET components, PHP scripts, etc.

- Ye server application logic aur data processing ko handle karta hai, jisse dynamic content generate hota hai, jaise user input processing, database queries, aur complex business logic.

- Application Server multiple clients ko support karte hain aur unke requests ko process kar ke dynamic responses generate karte hain.

- Kuch popular Application Servers hain Apache Tomcat, WildFly (JBoss), IBM WebSphere, Microsoft ASP.NET, etc.

Simplified Way mein, Web Server static content (HTML, CSS, images) serve karta hai aur Application Server dynamic content (Java, .NET, PHP, etc.) generate kar ke clients ko provide karta hai. Dono alag-alag tarah ke applications ke liye use hote hain, lekin kai baar dono ko ek saath bhi use kiya jata hai jab complex web applications ko handle karna hota hai.

# Hibernate and JPA :

## JPA vs Hibernate

\*\*JPA (Java Persistence API):\*\*

- Specification provided by Java for managing relational data in Java applications.

- Defines standard interfaces and annotations for Object-Relational Mapping (ORM) in Java.

- Part of Java EE and Jakarta EE specifications, vendor-independent.

- Allows Java developers to interact with relational databases using object-oriented approaches.

\*\*Hibernate:\*\*

- Popular and widely-used open-source ORM framework.

- Implements the JPA specification and provides additional features.

- Simplifies database access, automatically maps Java entities to database tables, and handles CRUD operations.

- Offers powerful features like caching, lazy loading, and query optimization for improved performance.

In summary, JPA is a standard specification for ORM in Java, while Hibernate is an ORM framework that implements the JPA specification and adds extra capabilities. Both JPA and Hibernate provide tools for mapping Java objects to relational database tables and simplify database operations in Java applications.

## Hiberbate object state

In Hibernate, objects go through different states during their lifecycle as they are managed and persisted in the database. The primary states of a Hibernate object are as follows:

1. \*\*Transient State\*\*:

An object is in the transient state when it is just created using the `new` keyword but is not associated with any Hibernate session. In this state, the object is not associated with any database table row, and any changes made to the object will not be reflected in the database.

2. \*\*Persistent State\*\*:

When a transient object is associated with a Hibernate session by calling the `save`, `persist`, `update`, `saveOrUpdate`, or `merge` methods, it enters the persistent state. In this state, the object is associated with a unique identifier and its state is synchronized with the database. Any changes made to the object will be tracked and can be persisted to the database later when the session is flushed or the transaction is committed.

3. \*\*Detached State\*\*:

An object becomes detached when it was previously associated with a Hibernate session but the session is closed, the object is evicted from the session cache, or the session is cleared. In this state, the object is no longer managed by the Hibernate session, and any changes made to the object will not be automatically synchronized with the database.

4. \*\*Removed State\*\*:

When an object is explicitly removed from the database using the `delete` or `remove` method of the Hibernate session, it enters the removed state. In this state, the object is marked for deletion, and the actual deletion from the database occurs when the session is flushed or the transaction is committed.

It's important to understand these states when working with Hibernate to ensure that object state changes are correctly managed and that the appropriate methods are used to persist, update, or remove objects from the database.

The state transitions can be summarized as follows:

- Transient state -> Persistent state: `save`/`persist`/`update`/`saveOrUpdate`/`merge` methods.

- Persistent state -> Transient state: Remove object reference from the Hibernate session cache.

- Persistent state -> Detached state: Close the Hibernate session, evict object from the session cache, or clear the session.

- Detached state -> Persistent state: Reattach the detached object to a new Hibernate session using `merge` or `update`.

- Persistent state -> Removed state: `delete`/`remove` method.

- Removed state -> Transient state: After deletion, the object is no longer associated with the database.

Understanding these object states is crucial for managing the persistence and behavior of Hibernate entities effectively.

## JPA Native @Param vs @Query

In the context of JPS (Java Persistence API) or JPA (Java Persistence API), `@Param` and `@Query` are not directly related to each other. Instead, they are separate annotations used in different scenarios.

1. `@Param`:

`@Param` is not a standard JPA annotation; instead, it's typically associated with Spring Data JPA, which is a framework built on top of JPA to simplify database access in Spring applications. The `@Param` annotation is used when you want to bind method parameters to named query parameters in Spring Data JPA repositories.

For example, consider a Spring Data JPA repository:

```java

@Repository

public interface UserRepository extends JpaRepository<User, Long> {

@Query("SELECT u FROM User u WHERE u.username = :username AND u.age = :age")

User findByUsernameAndAge(@Param("username") String username, @Param("age") int age);

}

`

In this example, the `@Param` annotation is used to map the method parameters (`username` and `age`) to the named query parameters (`:username` and `:age`).

2. `@Query`:

`@Query` is a JPA annotation used to define custom queries directly in your JPA repository interfaces. It allows you to write JPQL (Java Persistence Query Language) or native SQL queries to fetch data from the database.

Here's an example of using `@Query` without `@Param`:

```java

@Repository

public interface ProductRepository extends JpaRepository<Product, Long> {

@Query("SELECT p FROM Product p WHERE p.category = :category")

List<Product> findByCategory(String category);

}

```

In this example, the `@Query` annotation defines a custom query that retrieves a list of products based on their category.

In summary, `@Param` is used to map method parameters to named query parameters in Spring Data JPA repositories, while `@Query` is used to define custom queries in JPA repositories, which can be used with or without `@Param` depending on the query's complexity and parameterization needs.

## Join table in JPA Repository

In JPA (Java Persistence API), you can perform joins between tables using JPQL (Java Persistence Query Language) or the Criteria API. JPQL is a SQL-like query language specific to JPA, and it allows you to write database-agnostic queries that work with your entity classes. Here's how you can perform joins in JPA using JPQL:

Assuming you have two entities `Author` and `Book`, where `Author` has a one-to-many relationship with `Book`:

1. JPQL Join Query:

You can perform a join between the `Author` and `Book` entities in JPQL using the `JOIN` keyword:

```java

import javax.persistence.EntityManager;

import javax.persistence.PersistenceContext;

import org.springframework.stereotype.Repository;

@Repository

public class YourRepository {

@PersistenceContext

private EntityManager entityManager;

public List<Author> findAuthorsWithBooks() {

String jpqlQuery = "SELECT DISTINCT a FROM Author a JOIN FETCH a.books";

TypedQuery<Author> query = entityManager.createQuery(jpqlQuery, Author.class);

return query.getResultList();

}

}

```

In this example, we are fetching all `Author` entities along with their associated `Book` entities using a `JOIN FETCH` clause. This ensures that the `Book` entities are eagerly fetched with the `Author` entities to avoid N+1 query issues.

2. Criteria API Join:

Alternatively, you can use the Criteria API to create a join between entities programmatically:

```java

import javax.persistence.EntityManager;

import javax.persistence.PersistenceContext;

import javax.persistence.criteria.CriteriaBuilder;

import javax.persistence.criteria.CriteriaQuery;

import javax.persistence.criteria.Join;

import javax.persistence.criteria.Root;

import org.springframework.stereotype.Repository;

@Repository

public class YourRepository {

@PersistenceContext

private EntityManager entityManager;

public List<Author> findAuthorsWithBooks() {

CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();

CriteriaQuery<Author> criteriaQuery = criteriaBuilder.createQuery(Author.class);

Root<Author> authorRoot = criteriaQuery.from(Author.class);

Join<Author, Book> bookJoin = authorRoot.join("books");

criteriaQuery.select(authorRoot).distinct(true);

return entityManager.createQuery(criteriaQuery).getResultList();

}

}

```

In this example, we use the Criteria API to create a join between the `Author` and `Book` entities using the `join` method of the `Root` interface.

Both approaches achieve the same result of fetching `Author` entities along with their associated `Book` entities. Choose the one that suits your preference and use case.

## N+1 problem in JPA

\*\*The N+1 Problem in JPA Repositories\*\*

The N+1 problem is a common performance issue in JPA repositories when fetching related entities in a collection. It occurs when an initial query retrieves a list of entities, and then, for each entity in the result, an additional query is executed to fetch related entities. This results in a large number of database queries, leading to increased database round trips and decreased application performance.

\*\*Example Scenario:\*\*

Let's consider entities `Department` and `Employee`, where each department can have multiple employees (one-to-many relationship).

\*\*Department Entity:\*\*

```java

@Entity

@Table(name = "departments")

public class Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@OneToMany(mappedBy = "department", fetch = FetchType.LAZY)

private List<Employee> employees = new ArrayList<>();

// Other fields, getters, setters, constructors, etc.

}

```

\*\*Employee Entity:\*\*

```java

@Entity

@Table(name = "employees")

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "department\_id")

private Department department;

// Other fields, getters, setters, constructors, etc.

}

```

\*\*Problem Description:\*\*

Let's say we want to fetch all departments along with their associated employees using the `DepartmentRepository`:

```java

@Repository

public interface DepartmentRepository extends JpaRepository<Department, Long> {

List<Department> findAll();

}

```

The `findAll()` method will retrieve all departments in one query. However, since the `employees` collection is lazily loaded (`FetchType.LAZY`), when accessing the `employees` of each department in the result, a separate query will be executed to fetch the employees for that specific department. This leads to the N+1 problem, where N is the number of departments, and there will be an additional query for each department, causing performance overhead.

\*\*Solution:\*\*

To avoid the N+1 problem, you can use JPQL with the `JOIN FETCH` or `LEFT JOIN FETCH` clause to eagerly fetch related entities in a single query. This way, all required data can be retrieved in one database query, reducing the number of round trips and improving performance.

```java

@Repository

public interface DepartmentRepository extends JpaRepository<Department, Long> {

@Query("SELECT DISTINCT d FROM Department d JOIN FETCH d.employees")

List<Department> findAllWithEmployees();

}

```

With this custom query, calling `departmentRepository.findAllWithEmployees()` will fetch all departments along with their associated employees in a single query, addressing the N+1 problem.

In summary, the N+1 problem occurs when related entities are fetched lazily, and additional queries are executed for each entity in a collection. Using eager fetching with JPQL (`JOIN FETCH` or `LEFT JOIN FETCH`) can help mitigate this issue and optimize the performance of JPA repositories.

## Mapping available in hibernate

In Hibernate, there are various types of mappings available to represent the relationships between Java entities (POJOs) and database tables. These mappings define how the objects' data should be persisted to the relational database. Here are the common mappings available in Hibernate:

1. \*\*@Entity\*\*:

The `@Entity` annotation is used to mark a Java class as an entity, representing a database table. Each instance of the entity class corresponds to a row in the table.

2. \*\*@Id\*\*:

The `@Id` annotation is used to specify the primary key attribute of an entity. It maps the primary key column of the database table.

3. \*\*@GeneratedValue\*\*:

The `@GeneratedValue` annotation is used in conjunction with `@Id` to specify how the primary key value is generated, such as `GenerationType.IDENTITY`, `GenerationType.SEQUENCE`, etc.

4. \*\*@Column\*\*:

The `@Column` annotation is used to map an entity attribute to a specific column in the database table. It allows you to specify various column properties, such as name, length, nullable, etc.

5. \*\*@ManyToOne\*\* and \*\*@OneToMany\*\*:

These annotations are used to represent a many-to-one and one-to-many relationship between entities, respectively.

6. \*\*@OneToOne\*\*:

The `@OneToOne` annotation is used to represent a one-to-one relationship between entities.

7. \*\*@ManyToMany\*\*:

The `@ManyToMany` annotation is used to represent a many-to-many relationship between entities. It typically requires a join table to map the association.

8. \*\*@JoinColumn\*\*:

The `@JoinColumn` annotation is used to specify the column used to join two related entities in a relationship.

9. \*\*@Embedded\*\* and \*\*@Embeddable\*\*:

These annotations are used to represent an embedded object (value type) within an entity.

10. \*\*@ElementCollection\*\*:

The `@ElementCollection` annotation is used to represent a collection of simple elements or embedded objects as part of an entity.

11. \*\*@Transient\*\*:

The `@Transient` annotation is used to mark an entity attribute that should not be persisted to the database.

These are some of the commonly used mappings available in Hibernate. Hibernate provides a wide range of annotations and XML configurations to support various mapping scenarios, allowing developers to effectively represent the complex relationships between Java objects and relational databases.

## get and load method in hibernate

\*\*Hibernate's `get` and `load` Methods\*\*

In Hibernate, both the `get` and `load` methods are used to retrieve entities from the database based on their primary key (ID). However, they have some differences in behavior:

1. `get` Method:

- The `get` method is used to fetch an entity from the database immediately.

- If the entity with the given ID exists in the database, the `get` method will execute the SQL query and return the actual entity object.

- If the entity is not found in the database, the `get` method will return `null`.

- Use `get` when you want to fetch an entity and need a definite answer (`null` if not found).

Example:

```java

Author author = session.get(Author.class, 1L);

```

2. `load` Method:

- The `load` method, on the other hand, returns a "proxy" object (a placeholder) without hitting the database immediately.

- The actual data is loaded from the database only when you access the proxy object's attributes.

- If the entity with the given ID is not found in the database, the `load` method will throw an `ObjectNotFoundException` when you access the proxy object.

- Use `load` when you're confident the entity exists and want to take advantage of lazy loading to optimize performance.

Example:

```java

Author author = session.load(Author.class, 1L);

// No database query executed yet

System.out.println(author.getName()); // Database query executed to fetch the author's name

```

In summary, use `get` when you need an immediate result (entity or `null`) and use `load` when you're confident the entity exists and want to lazily fetch its data to improve performance.

## save and persist method

\*\*Hibernate's `save` and `persist` Methods\*\*

In Hibernate, both the `save` and `persist` methods are used to save entity objects into the database, but they have different behaviors:

1. `save` Method:

- The `save` method saves the entity object into the database and immediately returns the generated identifier (primary key) after executing the insert SQL statement.

- If the entity object has a `@GeneratedValue` annotation or its ID is auto-generated by the database (e.g., using AUTO\_INCREMENT in MySQL), the `save` method will assign the generated ID to the entity object and return it.

- If the entity object has a pre-set ID (primary key), the `save` method will use that ID for insertion, and it might lead to issues if the ID already exists in the database.

- The `save` method is specific to Hibernate and is not part of the JPA specification. It provides more control over ID generation and allows saving an entity with a pre-set ID.

Example:

```java

Author author = new Author();

author.setName("John Doe");

Long id = (Long) session.save(author); // Insert the author and return the generated ID

```

2. `persist` Method:

- The `persist` method saves the entity object into the database, but it does not return anything (void method).

- It follows the JPA specification, and its behavior is defined by JPA.

- If the entity object has a `@GeneratedValue` annotation or its ID is auto-generated by the database, the `persist` method will not assign the generated ID to the entity object immediately. The ID will be set during the next flush or commit operation.

- If the entity object has a pre-set ID (primary key), the `persist` method will throw an exception if the ID already exists in the database. This ensures that the entity is not duplicated with an existing ID.

Example:

```java

Author author = new Author();

author.setName("Jane Doe");

session.persist(author); // Insert the author, but the ID will be set during the next flush/commit

```

In summary, use `save` when you need the generated ID immediately or when you want more control over ID generation. Use `persist` when you want to follow JPA's managed entity lifecycle and don't need the generated ID right away.

## Update vs Merge method

\*\*Hibernate's `update` and `merge` Methods\*\*

In Hibernate, both the `update` and `merge` methods are used to update entities in the database, but they have different behaviors and use cases:

1. `update` Method:

- The `update` method is used to reattach a detached entity to the current Hibernate session and synchronize its state with the database.

- If the entity object is already associated with the current Hibernate session (persistent), the `update` method has no effect because the changes made to the object are automatically persisted to the database during the session flush.

- If the entity object is detached (i.e., loaded from a previous Hibernate session or created outside of a session), the `update` method will attach the entity to the current session and schedule an update SQL statement during the next session flush. If the same ID already exists in the current session, the `update` method will throw an exception to avoid conflicts.

Example:

```java

Author author = session.get(Author.class, 1L); // Load an author object into the session

author.setName("Updated Name");

session.update(author); // Reattach and update the entity during the session flush

```

2. `merge` Method:

- The `merge` method is used to merge the state of a detached entity into the current Hibernate session or to persist a transient (new) entity with a pre-set ID, creating a new managed entity in the current session.

- If the entity object is already associated with the current Hibernate session (persistent), the `merge` method has no effect. It is typically used with detached entities to reattach them to the session, and any changes made to the detached entity are copied into a managed entity in the session. The returned object from the `merge` method is the managed entity, which should be used for further operations.

Example:

```java

Author detachedAuthor = createDetachedAuthor(); // Create a detached author object

detachedAuthor.setName("Updated Name");

Author managedAuthor = session.merge(detachedAuthor); // Merge the detached object into the session

```

In summary, use the `update` method to reattach a detached entity to the current session and update its state in the database. Use the `merge` method to merge the state of a detached entity into the current session or to persist a transient entity with a pre-set ID. Both methods serve different purposes, and the choice between them depends on your use case and whether you need to update an existing entity or create a new one from a detached object.

## Inheritance in hibernate

\*\*Hibernate Inheritance Mapping Strategies\*\*

In Hibernate, inheritance mapping strategies are used to represent the inheritance hierarchy among Java entities (POJOs) and map them to corresponding database tables. Hibernate supports three types of inheritance mapping strategies:

1. \*\*Single Table Inheritance\*\*:

- All subclasses (sub-entities) share the same database table, and a discriminator column is used to differentiate between them.

- This approach is suitable when subclasses have a lot of common attributes and only a few specific attributes.

- Example:

```java

@Entity

@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)

@DiscriminatorColumn(name = "vehicle\_type", discriminatorType = DiscriminatorType.STRING)

public class Vehicle {

// Common attributes

}

@Entity

@DiscriminatorValue("Car")

public class Car extends Vehicle {

// Car-specific attributes

}

@Entity

@DiscriminatorValue("Bike")

public class Bike extends Vehicle {

// Bike-specific attributes

}

```

2. \*\*Table Per Class\*\*:

- Each subclass has its own database table, containing both the attributes inherited from the superclass and the subclass-specific attributes.

- This approach is suitable when each subclass has a significant number of specific attributes, and you want to avoid having nullable columns in the database.

- Example:

```java

@Entity

@Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)

public class Vehicle {

// Common attributes

}

@Entity

public class Car extends Vehicle {

// Car-specific attributes

}

@Entity

public class Bike extends Vehicle {

// Bike-specific attributes

}

```

3. \*\*Joined Subclass\*\*:

- Each subclass has its own database table containing only the subclass-specific attributes.

- The common attributes are placed in the superclass table, and the tables are linked by a primary key-foreign key relationship between the superclass and subclass tables.

- This approach is suitable when each subclass has many specific attributes, and you want to avoid having nullable columns in the superclass table.

- Example:

```java

@Entity

@Inheritance(strategy = InheritanceType.JOINED)

public class Vehicle {

// Common attributes

}

@Entity

public class Car extends Vehicle {

// Car-specific attributes

}

@Entity

public class Bike extends Vehicle {

// Bike-specific attributes

}

```

The choice of inheritance strategy depends on the nature of the inheritance hierarchy, the size of the common and specific attributes in the subclasses, and the overall database design. Each strategy has its advantages and disadvantages, and it's essential to choose the one that best fits your application's requirements.

## Lazy or Eager loading in Hibernate

\*\*Lazy Loading and Eager Loading in Hibernate\*\*

Lazy loading and eager loading are two strategies used in Hibernate to control how related entities are fetched from the database when accessing associations between entities.

1. \*\*Lazy Loading\*\*:

- Lazy loading is the default behavior for associations in Hibernate.

- Related entities are not loaded from the database until they are explicitly accessed in your code.

- Actual data of the related entities is loaded from the database only when you try to access the properties or methods of the associated objects.

- Use lazy loading to improve performance by fetching only the necessary data from the database and avoiding unnecessary database queries.

```java

@Entity

public class Author {

@Id

private Long id;

private String name;

@OneToMany(mappedBy = "author", fetch = FetchType.LAZY)

private List<Book> books;

// Getters and Setters

}

```

2. \*\*Eager Loading\*\*:

- Eager loading fetches the related entities from the database immediately when the parent entity is fetched.

- All associated entities are fetched along with the parent entity in a single query, using join or additional queries depending on the configuration.

- Use eager loading for associations that are frequently accessed and always needed without triggering additional database queries each time.

```java

@Entity

public class Author {

@Id

private Long id;

private String name;

@OneToMany(mappedBy = "author", fetch = FetchType.EAGER)

private List<Book> books;

// Getters and Setters

}

```

Choosing between lazy loading and eager loading depends on your application's use case and performance requirements:

- Use lazy loading for associations that might not be accessed in every use case and when you want to optimize performance by reducing the number of database queries.

- Use eager loading for associations that are frequently accessed and need to be available without triggering additional database queries each time.

In general, it is a good practice to use lazy loading for most associations and consider using eager loading for specific use cases where you know the associated data will always be needed and it will not lead to performance issues. Keep in mind that you can override the default fetch type using the `fetch` attribute in the `@OneToMany`, `@ManyToOne`, `@OneToOne`, and other association annotations in Hibernate.

## Dirty read in hibernate and how can we overcome

\*\*Dirty Read in Hibernate\*\*

In Hibernate, a "dirty read" is a phenomenon that can occur in a concurrent environment when one transaction reads and uses data that has been modified by another transaction but not yet committed. This can lead to incorrect or inconsistent results.

To avoid dirty reads, you can use the Read Committed isolation level, which allows a transaction to read only committed data and avoids reading uncommitted data from other transactions. You can set the isolation level using the `@Transactional` annotation in Spring or the `hibernate.connection.isolation` property in the Hibernate configuration file.

Choose the appropriate isolation level based on your application's requirements and potential impact on data integrity and concurrency. Proper transaction management and database locking mechanisms can help ensure data consistency and prevent issues like dirty reads in a concurrent environment.

Certainly! Let's consider a simple example to demonstrate the dirty read phenomenon and how to avoid it using the Read Committed isolation level in Hibernate.

Suppose we have an entity class representing a bank account:

```java

@Entity

public class BankAccount {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String accountNumber;

private double balance;

// Getters, setters, constructors, etc.

}

```

Now, let's simulate two concurrent transactions trying to update the balance of the same bank account:

\*\*Transaction A\*\*:

```java

@Service

public class TransactionServiceA {

@Autowired

private BankAccountRepository bankAccountRepository;

@Transactional(isolation = Isolation.READ\_COMMITTED)

public void performTransactionA() {

BankAccount account = bankAccountRepository.findByAccountNumber("12345");

double currentBalance = account.getBalance();

double updatedBalance = currentBalance - 100.0;

account.setBalance(updatedBalance);

// Perform some other operations if needed

}

}

```

\*\*Transaction B\*\*:

```java

@Service

public class TransactionServiceB {

@Autowired

private BankAccountRepository bankAccountRepository;

@Transactional(isolation = Isolation.READ\_COMMITTED)

public void performTransactionB() {

BankAccount account = bankAccountRepository.findByAccountNumber("12345");

double currentBalance = account.getBalance();

double updatedBalance = currentBalance + 50.0;

account.setBalance(updatedBalance);

// Perform some other operations if needed

}

}

```

In this example, both Transaction A and Transaction B are running concurrently. Transaction A reads the balance of the bank account and deducts 100 units, while Transaction B reads the balance and adds 50 units.

\*\*Dirty Read Scenario\*\*:

1. Transaction A reads the balance as 500 units.

2. Before Transaction A commits, Transaction B reads the same balance as 500 units.

3. Transaction A updates the balance to 400 units and commits.

4. Transaction B updates the balance to 550 units and commits.

After both transactions have committed, the final balance should have been 450 units (500 - 100 + 50). However, due to dirty read, the final balance becomes 550 units.

\*\*Avoiding Dirty Read with Read Committed Isolation\*\*:

To avoid dirty read, we can use the Read Committed isolation level in both transactions. With Read Committed, each transaction reads only committed data, ensuring that Transaction B reads the updated balance after Transaction A commits.

```java

@Transactional(isolation = Isolation.READ\_COMMITTED)

public void performTransactionA() {

// ... Same code as before

}

@Transactional(isolation = Isolation.READ\_COMMITTED)

public void performTransactionB() {

// ... Same code as before

}

```

With Read Committed isolation, the final balance will be 450 units as expected, ensuring data consistency in the concurrent environment.

## Hibernate in Spring Boot

Sure! Here's a small note summarizing the steps to configure Hibernate in a Spring Boot application with an example:

\*\*Configuring Hibernate in Spring Boot:\*\*

1. \*\*Create a Spring Boot Project:\*\* Use Spring Initializr or your preferred IDE to create a new Spring Boot project with the necessary dependencies for JPA and Hibernate.

2. \*\*Configure Database Properties:\*\* In the application.properties or application.yml file, provide the database connection properties like URL, username, and password, along with Hibernate-specific properties. For example:

\*\*application.properties\*\*:

```properties

# Database connection properties

spring.datasource.url=jdbc:mysql://localhost:3306/mydatabase

spring.datasource.username=root

spring.datasource.password=yourpassword

# Hibernate properties

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

```

3. \*\*Create Entity Classes:\*\* Define entity classes with JPA annotations representing the database tables. For example:

```java

@Entity

public class Product {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private double price;

// Getters and setters, constructors, etc.

}

```

4. \*\*Create JPA Repositories:\*\* Create repository interfaces by extending JpaRepository to perform CRUD operations on entities. For example:

```java

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface ProductRepository extends JpaRepository<Product, Long> {

// Custom queries or methods can be defined here if needed

}

```

5. \*\*Enable JPA and Hibernate:\*\* Spring Boot's auto-configuration will automatically set up Hibernate based on the properties provided. Optionally, you can explicitly enable JPA repositories in your main application class using the `@EnableJpaRepositories` annotation. For example:

```java

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@SpringBootApplication

@EnableJpaRepositories(basePackages = "com.example.repository")

public class MySpringBootApplication {

public static void main(String[] args) {

SpringApplication.run(MySpringBootApplication.class, args);

}

}

```

With these simple steps and the provided example, you have configured Hibernate in your Spring Boot application. You can now use JPA repositories to interact with the database and perform CRUD operations on your entities seamlessly. Spring Boot's auto-configuration takes care of most of the setup, allowing you to focus on developing your application's logic efficiently. If needed, you can further customize Hibernate properties in the application configuration file to suit your specific requirements.

## commit and rollback in hibernate

Sure! Here's a small note summarizing the usage of transactions in Hibernate with examples:

\*\*Transactions in Hibernate:\*\*

In Hibernate, transactions are used to group database operations into a single unit of work, ensuring that either all operations are applied to the database as a whole (`commit`) or all changes are undone (`rollback`) in case of an error or exception.

\*\*1. Commit:\*\*

The `commit` method is used to apply changes made during a transaction to the database, making them permanent and visible to other transactions. It is typically called after a successful execution of a sequence of database operations.

```java

Session session = sessionFactory.openSession();

Transaction transaction = session.beginTransaction();

try {

// Perform database operations (inserts, updates, deletes)

session.save(entity1);

session.update(entity2);

session.delete(entity3);

transaction.commit(); // Save the changes to the database

} catch (Exception e) {

transaction.rollback(); // Roll back the transaction in case of an exception

e.printStackTrace();

} finally {

session.close(); // Close the session after the transaction is complete

}

```

\*\*2. Rollback:\*\*

The `rollback` method is used to undo all changes made during a transaction and restore the database to its state before the transaction began. It is called when an exception occurs during the transaction or when you explicitly want to discard the changes made.

```java

Session session = sessionFactory.openSession();

Transaction transaction = session.beginTransaction();

try {

// Perform database operations (inserts, updates, deletes)

session.save(entity1);

session.update(entity2);

session.delete(entity3);

if (someCondition) {

throw new SomeException("Something went wrong.");

}

transaction.commit(); // Save the changes to the database

} catch (Exception e) {

transaction.rollback(); // Roll back the transaction in case of an exception

e.printStackTrace();

} finally {

session.close(); // Close the session after the transaction is complete

}

```

\*\*Note:\*\*

- Always wrap your database operations within a transaction to ensure data integrity and consistency.

- Use `commit` to apply changes to the database, and `rollback` to discard changes in case of exceptions or when needed.

- Proper exception handling is crucial to ensure that the transaction is rolled back when errors occur.

- Remember to close the session after the transaction is complete to release resources.

By using transactions appropriately in Hibernate, you can maintain a consistent state of the data and avoid potential issues with concurrent access to the database. Transactions play a vital role in ensuring the correctness and reliability of your database operations.

## Second level cache how can we implement

Sure! Here's a small note summarizing the steps to implement the second-level cache using Ehcache in Hibernate:

\*\*Implementing Second-Level Cache with Ehcache in Hibernate:\*\*

1. \*\*Add Ehcache Dependency:\*\*

Include the Ehcache dependency in your project. For Maven, add the following to your `pom.xml`:

```xml

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-ehcache</artifactId>

<version><!-- Use the appropriate version of Hibernate --></version>

</dependency>

```

2. \*\*Enable Second-Level Cache in Hibernate Configuration:\*\*

In your Hibernate configuration (hibernate.cfg.xml or application.properties in Spring Boot), enable the second-level cache and specify the caching provider as "org.hibernate.cache.EhCacheProvider":

\*\*Hibernate XML Configuration (hibernate.cfg.xml):\*\*

```xml

<property name="hibernate.cache.use\_second\_level\_cache">true</property>

<property name="hibernate.cache.region.factory\_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</property>

```

\*\*Spring Boot application.properties:\*\*

```properties

spring.jpa.properties.hibernate.cache.use\_second\_level\_cache=true

spring.jpa.properties.hibernate.cache.region.factory\_class=org.hibernate.cache.ehcache.EhCacheRegionFactory

```

3. \*\*Annotate Entities for Caching:\*\*

To enable caching for specific entities, annotate them with `@Cacheable` and specify the cache region name using `@org.hibernate.annotations.Cache`.

```java

import javax.persistence.Cacheable;

import javax.persistence.Entity;

@Entity

@Cacheable

@org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.READ\_WRITE)

public class Book {

// Entity properties, getters, and setters

}

```

4. \*\*Configure Ehcache:\*\*

Create an Ehcache configuration file (ehcache.xml) in the classpath to configure the cache settings. Define the cache regions and their properties.

```xml

<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"

updateCheck="true" monitoring="autodetect" dynamicConfig="true">

<cache name="com.example.Book" maxEntriesLocalHeap="1000" eternal="false" timeToIdleSeconds="300"

timeToLiveSeconds="600" memoryStoreEvictionPolicy="LRU" />

</ehcache>

```

The `<cache>` element's `name` attribute should match the cache region name specified in the `@Cacheable` annotation on the entity.

With these configurations in place, Hibernate will use Ehcache as the second-level cache provider. The specified entities and query results will be cached, enhancing the performance of your application by reducing database access for frequently used data. The actual cache behavior and settings can be further customized by adjusting the cache configuration in the ehcache.xml file to suit your application's specific needs.

## @async and @lazy

Sure! Here's a small note with examples about the `@Async` and `@Lazy` annotations in Spring Framework:

1. \*\*@Async\*\*:

- The `@Async` annotation is used to execute a method asynchronously in a separate thread.

- It allows time-consuming tasks to run in the background, freeing up the main thread for other operations.

- To enable async support, use `@EnableAsync` in the configuration or `@SpringBootApplication` with `@EnableAsync` in the main class.

Example of using `@Async`:

```java

import org.springframework.scheduling.annotation.Async;

import org.springframework.stereotype.Service;

@Service

public class MyService {

@Async

public void asyncMethod() {

// Asynchronous code here

}

}

```

2. \*\*@Lazy\*\*:

- The `@Lazy` annotation is used to delay the creation of a bean until it is first requested.

- It helps improve application startup time and resource usage by creating beans only when needed.

- By default, beans are eagerly initialized during application context startup.

Example of using `@Lazy`:

```java

import org.springframework.context.annotation.Lazy;

import org.springframework.stereotype.Component;

@Component

@Lazy

public class MyLazyBean {

// Bean definition here

}

```

With these annotations, you can leverage asynchronous processing and optimize resource usage for your Spring applications. However, use them carefully and consider the implications of asynchronous execution and lazy bean initialization based on your specific use case.

## @Version

Sure! Here's a small note about the `@Version` annotation in JPA:

The `@Version` annotation in JPA is used for optimistic locking, a concurrency control mechanism to handle concurrent updates to the same entity. When an entity is marked with `@Version`, JPA automatically tracks changes to a version attribute.

Example:

```java

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Version;

@Entity

public class Product {

@Id

private Long id;

private String name;

private double price;

@Version

private int version; // Version attribute for optimistic locking

// Constructors, getters, and setters

}

```

During updates, if two transactions try to modify the same entity concurrently, JPA compares the version values before committing the changes. If the version values match, the changes are safely committed. If the version values don't match, it indicates that another transaction modified the entity, and JPA throws an `OptimisticLockException` to handle the conflict.

Optimistic locking with `@Version` provides better performance in multi-user environments while ensuring data consistency and avoiding overwrites. It is suitable for scenarios where conflicts are infrequent and locking the entire entity during updates is not necessary.

## Dialect in hibernate

In Hibernate, a dialect is a class that provides an implementation of database-specific SQL and database behavior. It acts as a bridge between Hibernate's generic SQL and the SQL specific to the database management system (DBMS) that Hibernate interacts with. The dialect is responsible for generating the appropriate SQL statements and handling the differences in SQL syntax, functions, and data types for different database systems.

Each database system has its own dialect in Hibernate. Some common examples of dialects are:

- MySQLDialect: For MySQL database

- OracleDialect: For Oracle database

- PostgreSQLDialect: For PostgreSQL database

- SQLServerDialect: For Microsoft SQL Server database

- H2Dialect: For H2 database

- and many more...

When you configure Hibernate for your application, you need to specify the appropriate dialect for the database you are using. Hibernate will use this dialect to generate and execute the SQL statements specific to that database.

For example, when using Hibernate with MySQL, you can configure the dialect in your Hibernate configuration file (hibernate.cfg.xml) or in your Spring Boot application.properties file:

Using Hibernate XML configuration (hibernate.cfg.xml):

```xml

<hibernate-configuration>

<session-factory>

<!-- Other configurations -->

<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

</session-factory>

</hibernate-configuration>

```

Using Spring Boot application.properties:

```properties

spring.datasource.url=jdbc:mysql://localhost:3306/mydatabase

spring.datasource.username=username

spring.datasource.password=password

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect

```

By specifying the appropriate dialect, Hibernate can generate the correct SQL statements and handle the database-specific behavior, ensuring that your application works seamlessly with the chosen database system.

# REST API:

## REST API

Sure! Here's a simple and concise explanation of REST:

\*\*REST (Representational State Transfer)\*\* is a set of rules and principles for designing web services that make it easy for computers to communicate with each other over the internet. It uses standard HTTP methods like GET, POST, PUT, and DELETE to interact with resources (data) identified by unique URLs.

\*\*Key Points:\*\*

- RESTful APIs follow a client-server architecture, where clients (like web browsers or mobile apps) request data from servers, and servers respond with the requested information.

- REST treats everything as a resource (e.g., user, product, image), and each resource has a unique URL (web address).

- Clients can use simple HTTP methods to perform actions on resources, like getting data (GET), creating new data (POST), updating data (PUT), and deleting data (DELETE).

- RESTful APIs are stateless, meaning that each request is independent and does not rely on any previous interactions.

- The data exchanged between client and server can be in different formats, such as JSON or XML.

In summary, REST is a flexible and widely-used approach for building web services that allow computers to communicate and exchange data in a standardized and simple way over the internet.

## Annotation in REST API:

REST API ko design aur implement karne ke liye Spring Framework mein kuch important annotations hote hain. Yeh annotations `spring-web` module mein provide kiye jate hain. Kuch pramukh REST API annotations hain:

1. \*\*@RestController\*\*: Ye annotation class ko RESTful controller mein transform karta hai. Isse Spring samajhta hai ki yeh class REST API endpoints ko expose karne ke liye hai.

2. \*\*@RequestMapping\*\*: Is annotation se hum URL paths ko map kar sakte hain. Class level par aur method level par dono jagah par use kiya jata hai.

3. \*\*@GetMapping, @PostMapping, @PutMapping, @DeleteMapping\*\*: Ye annotations specific HTTP methods (GET, POST, PUT, DELETE) ke liye mapping karte hain.

4. \*\*@PathVariable\*\*: Is annotation se hum URL se variables ko extract kar sakte hain. Jaise `/users/{id}` mein, `id` variable ko @PathVariable se extract kiya ja sakta hai.

5. \*\*@RequestParam\*\*: Is annotation se hum URL query parameters ko extract kar sakte hain. Jaise `/users?id=1` mein, `id` query parameter ko @RequestParam se extract kiya ja sakta hai.

6. \*\*@RequestBody\*\*: Is annotation se hum HTTP request body se data ko extract kar sakte hain. JSON data ya form data ko Object mein convert karne ke liye iska use hota hai.

7. \*\*@ResponseBody\*\*: Is annotation se hum method ki return value ko HTTP response body mein convert kar sakte hain. JSON data ya koi object ko response body mein convert karne ke liye iska use hota hai.

8. \*\*@ResponseStatus\*\*: Is annotation se hum method ka default HTTP response status set kar sakte hain.

In annotations ke madhyam se hum Spring mein RESTful API endpoints ko define kar sakte hain, jisse hum client applications ke saath data exchange kar sakte hain.

## Put and Post in Rest Web services

\*\*RESTful Web Services: POST and PUT\*\*

- \*\*POST\*\*: Used to submit data to create a new resource on the server. Non-idempotent, meaning multiple identical requests may have different outcomes. Server generates the URI for the new resource.

- \*\*PUT\*\*: Used to update or replace an existing resource on the server. Idempotent, making the same request multiple times has the same result. Server replaces the entire resource with the provided representation.

Choose POST for resource creation and PUT for resource updates in RESTful APIs. Each method serves a distinct purpose in handling data interactions with the server.

## What is JWT in web services

JWT stands for JSON Web Token. It is a compact, URL-safe, and self-contained token format used for securely transmitting information between parties, typically in web services and APIs. JWTs are widely used for authentication and authorization purposes in modern web applications and microservices architecture.

A JWT consists of three parts separated by dots:

1. \*\*Header\*\*: The header typically consists of two parts: the type of token (JWT) and the signing algorithm used, such as HMAC SHA256 or RSA.

2. \*\*Payload\*\*: The payload contains the claims. Claims are statements about an entity (typically the user) and additional data. There are three types of claims: reserved, public, and private claims.

3. \*\*Signature\*\*: To create the signature part, you have to take the encoded header, the encoded payload, a secret, and the algorithm specified in the header and sign that.

JWTs are often used for authentication by including user information and permissions in the payload. When a user logs in, the server creates a JWT and sends it back to the client. The client then includes the JWT in subsequent requests to access protected resources or APIs.

Advantages of using JWT:

- Compact and URL-safe format makes it easy to pass as a query parameter or in the request header.

- Stateless nature allows servers to validate tokens without the need for storing sessions or user information.

- Can be easily verified and trusted as the signature is used to verify the integrity of the token.

However, it's important to handle JWTs securely:

- Always use HTTPS to transmit JWTs to prevent man-in-the-middle attacks.

- Store the secret key securely on the server and never expose it to the client.

- Carefully consider the data stored in the payload, as JWTs are visible to the client and can be decoded.

JWTs are widely supported by various programming languages and frameworks, making them a popular choice for secure authentication and authorization in web services and APIs.

## How to authenticate web services using Token in Java

To authenticate web services using tokens in Java, you can use JSON Web Tokens (JWT) as the authentication mechanism. Here's a step-by-step guide on how to implement token-based authentication in Java:

1. \*\*Create a Java Web Application\*\*: Set up a Java web application using a web framework like Spring Boot or Java Servlets. For this example, we'll use Spring Boot.

2. \*\*Add Dependencies\*\*: Include the required dependencies for JWT in your project. For Spring Boot, you can add the following dependencies to your `pom.xml` file:

```xml

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-api</artifactId>

<version>0.11.2</version>

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-impl</artifactId>

<version>0.11.2</version>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-jackson</artifactId>

<version>0.11.2</version>

<scope>runtime</scope>

</dependency>

```

3. \*\*Create a JWT Utility Class\*\*: Create a utility class to handle JWT operations, such as generating and verifying tokens. You can use the `io.jsonwebtoken.Jwts` class provided by the JJWT library.

```java

import io.jsonwebtoken.Claims;

import io.jsonwebtoken.Jws;

import io.jsonwebtoken.Jwts;

import io.jsonwebtoken.SignatureAlgorithm;

import io.jsonwebtoken.security.Keys;

import java.security.Key;

import java.util.Date;

public class JwtUtil {

private static final String SECRET\_KEY = "your-secret-key"; // Replace with a secure secret key

public static String generateToken(String subject, long expirationMillis) {

Key key = Keys.hmacShaKeyFor(SECRET\_KEY.getBytes());

return Jwts.builder()

.setSubject(subject)

.setExpiration(new Date(System.currentTimeMillis() + expirationMillis))

.signWith(key, SignatureAlgorithm.HS256)

.compact();

}

public static String getSubjectFromToken(String token) {

Key key = Keys.hmacShaKeyFor(SECRET\_KEY.getBytes());

Jws<Claims> claims = Jwts.parserBuilder()

.setSigningKey(key)

.build()

.parseClaimsJws(token);

return claims.getBody().getSubject();

}

}

```

4. \*\*Authenticate User and Generate Token\*\*: When a user logs in or authenticates, generate a JWT token using the `generateToken` method and return it as part of the authentication response.

```java

// Authentication Controller

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class AuthController {

@PostMapping("/login")

public ResponseEntity<?> login(@RequestBody UserCredentials credentials) {

// Authenticate user (validate credentials) - This step depends on your authentication logic

if (isValidUser(credentials)) {

String token = JwtUtil.generateToken(credentials.getUsername(), 3600000); // Token valid for 1 hour

return ResponseEntity.ok(new AuthResponse(token));

} else {

return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Invalid credentials");

}

}

}

```

5. \*\*Secure Your Endpoints\*\*: For protected endpoints, you can use a filter or interceptor to validate the JWT token before allowing access to the resource.

```java

// Authorization Filter

import org.springframework.web.filter.OncePerRequestFilter;

import javax.servlet.FilterChain;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import java.io.IOException;

public class JwtAuthorizationFilter extends OncePerRequestFilter {

@Override

protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {

String authorizationHeader = request.getHeader("Authorization");

if (authorizationHeader != null && authorizationHeader.startsWith("Bearer ")) {

String token = authorizationHeader.substring(7); // Remove "Bearer " from the token string

try {

String subject = JwtUtil.getSubjectFromToken(token);

// Perform additional authorization checks if needed

// ...

filterChain.doFilter(request, response);

} catch (Exception e) {

response.setStatus(HttpServletResponse.SC\_FORBIDDEN);

}

} else {

response.setStatus(HttpServletResponse.SC\_UNAUTHORIZED);

}

}

}

```

6. \*\*Register the Filter\*\*: Register the JWT Authorization filter with your application to secure the endpoints.

```java

// Application Configuration

import org.springframework.boot.web.servlet.FilterRegistrationBean;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class AppConfig {

@Bean

public FilterRegistrationBean<JwtAuthorizationFilter> jwtAuthorizationFilter() {

FilterRegistrationBean<JwtAuthorizationFilter> registrationBean = new FilterRegistrationBean<>();

registrationBean.setFilter(new JwtAuthorizationFilter());

registrationBean.addUrlPatterns("/api/\*"); // Secure endpoints under "/api" path

return registrationBean;

}

}

```

Now, your web service is secured using token-based authentication. Users can obtain a JWT token upon successful login and use it to access protected resources in subsequent requests. The JWT token will be validated by the `JwtAuthorizationFilter` before allowing access to the protected endpoints.

# Database :

## Get First, second, third highest salary from employee table

\*\*Retrieving First, Second, and Third Highest Salaries without LIMIT:\*\*

To get the first, second, and third highest salaries from the `employee` table without using the `LIMIT` clause, you can utilize the `OFFSET` feature along with `ORDER BY` in standard SQL:

1. \*\*First Highest Salary\*\*:

```sql

**SELECT MAX(salary) AS first\_highest\_salary FROM employee;**

```

2. \*\*Second Highest Salary\*\*:

```sql

**SELECT MAX(salary) AS second\_highest\_salary FROM employee**

**WHERE salary < (SELECT MAX(salary) FROM employee);**

```

**SELECT salary FROM employees ORDER BY salary DESC LIMIT 1 OFFSET 1;**

3. \*\*Third Highest Salary\*\*:

```sql

SELECT MAX(salary) AS third\_highest\_salary

FROM employee

WHERE salary < (SELECT MAX(salary) FROM employee WHERE salary < (SELECT MAX(salary) FROM employee));

```

These queries avoid using the `LIMIT` clause by using nested subqueries to filter out the previously obtained maximum salary values. Note that if there are duplicate salaries, this method will still return distinct salary values for each position.

## Joins available in sql with example.

\*\*SQL Joins:\*\*

SQL joins are used to combine data from two or more tables based on related columns. Here are the main types of joins with examples:

\*\*1. INNER JOIN:\*\*

- Returns only the rows with matching values in both tables.

- Example:

```sql

SELECT emp\_id, emp\_name, department\_name

FROM employees

INNER JOIN departments ON employees.department\_id = departments.department\_id;

```

\*\*2. LEFT JOIN (LEFT OUTER JOIN):\*\*

- Returns all rows from the left table and matching rows from the right table. Returns NULL if no match in the right table.

- Example:

```sql

SELECT emp\_id, emp\_name, department\_name

FROM employees

LEFT JOIN departments ON employees.department\_id = departments.department\_id;

```

\*\*3. RIGHT JOIN (RIGHT OUTER JOIN):\*\*

- Returns all rows from the right table and matching rows from the left table. Returns NULL if no match in the left table.

- Example:

```sql

SELECT emp\_id, emp\_name, department\_name

FROM employees

RIGHT JOIN departments ON employees.department\_id = departments.department\_id;

```

\*\*4. FULL JOIN (FULL OUTER JOIN):\*\*

- Returns all rows from both tables. Returns NULL if no match in either table.

- Example:

```sql

SELECT emp\_id, emp\_name, department\_name

FROM employees

FULL JOIN departments ON employees.department\_id = departments.department\_id;

```

Self Join:

A self-join is a type of database table join where a table is joined with itself using aliases. It is useful when you need to establish relationships between rows within the same table. Here's a brief example to illustrate a self-join:

Consider a table called "Employees" with the following data:

```

Employees Table

----------------

ID | Name | ManagerID

--------------------------

1 | John | 3

2 | Alice | 3

3 | Michael | NULL

4 | Sarah | 1

5 | Tom | 2

```

To retrieve a list of employees along with their managers, we can use a self-join query:

```sql

SELECT e.Name AS EmployeeName, m.Name AS ManagerName

FROM Employees e

JOIN Employees m ON e.ManagerID = m.ID;

```

The query will give the result:

```

Result

------------------------------

EmployeeName | ManagerName

------------------------------

John | Michael

Alice | Michael

Sarah | John

Tom | Alice

```

In this example, we used the "Employees" table twice, once as "e" to represent employees and once as "m" to represent managers. The self-join connects the "ManagerID" of employees with the "ID" of their corresponding managers. This way, we get a list of employees with their respective managers.

Use the appropriate type of join depending on your data and the desired result set for combining data from multiple tables in SQL queries.

## Normalization in Database

\*\*Normalization in Database Design:\*\*

Normalization is a process in relational database design that aims to reduce data redundancy and improve data integrity. The main levels of normalization are:

1. \*\*First Normal Form (1NF)\*\*: Data is organized into tables with a unique identifier (primary key), and each column contains indivisible values.

2. \*\*Second Normal Form (2NF)\*\*: The table is in 1NF, and each non-key column is fully functionally dependent on the entire primary key.

3. \*\*Third Normal Form (3NF)\*\*: The table is in 2NF, and no non-key column is transitively dependent on the primary key.

Higher levels of normalization like BCNF and 4NF deal with more complex data dependencies. Normalization helps maintain data consistency and reduce anomalies, but striking a balance with denormalization may be necessary for optimized query performance.

## Types of indexes ? why do we use indexes in sql

\*\*SQL Indexes:\*\*

Indexes in SQL are data structures that enhance data retrieval performance in a database table. They come in various types, each serving specific use cases:

1. \*\*Clustered Index\*\*: Determines the physical order of data rows in a table. Useful for range scans and sorting operations.

Example:

```sql

CREATE CLUSTERED INDEX idx\_employee\_id ON employees (employee\_id);

```

2. \*\*Non-Clustered Index\*\*: Separate data structure containing sorted key values and pointers to data rows. Supports search queries.

Example:

```sql

CREATE NONCLUSTERED INDEX idx\_employee\_name ON employees (employee\_name);

```

3. \*\*Unique Index\*\*: Enforces uniqueness of values in indexed columns. Fast data retrieval based on unique keys.

Example:

```sql

CREATE UNIQUE INDEX idx\_employee\_email ON employees (email);

```

4. \*\*Composite Index\*\*: Created on multiple columns for queries involving multiple search or sorting columns.

Example:

```sql

CREATE INDEX idx\_employee\_department ON employees (department\_id, hire\_date);

```

5. \*\*Covering Index\*\*: Includes all columns needed to fulfill a query, avoiding data lookups.

Example:

```sql

CREATE INDEX idx\_employee\_covering ON employees (employee\_name) INCLUDE (salary);

```

Choose indexes wisely based on query patterns to strike a balance between performance improvement and data modification overhead.

## What is trigger ? trigger creation syntax and advantage ?

In SQL, a trigger is a database object associated with a table that automatically executes a specified action when a specific database event (e.g., INSERT, UPDATE, DELETE) occurs on the table. Triggers are used to enforce data integrity, maintain referential integrity, and automate certain tasks or business logic within the database.

\*\*Trigger Creation Syntax\*\*:

The syntax for creating a trigger in SQL varies slightly depending on the database management system (DBMS) you are using. Here is the general syntax for creating a trigger:

```sql

CREATE [OR REPLACE] TRIGGER trigger\_name

{BEFORE | AFTER | INSTEAD OF} {INSERT | UPDATE | DELETE} ON table\_name

[REFERENCING OLD AS old NEW AS new]

[FOR EACH ROW]

[WHEN (condition)]

BEGIN

-- Trigger logic goes here

END;

```

- `trigger\_name`: The name of the trigger.

- `BEFORE | AFTER | INSTEAD OF`: Specifies when the trigger should execute (before or after the specified event).

- `INSERT | UPDATE | DELETE`: Specifies the event that activates the trigger.

- `table\_name`: The name of the table to which the trigger is associated.

- `REFERENCING OLD AS old NEW AS new`: Used in row-level triggers to access the old and new values of the affected rows.

- `FOR EACH ROW`: Indicates that the trigger is a row-level trigger (executed for each affected row).

- `WHEN (condition)`: An optional condition that must be satisfied for the trigger to execute.

- `BEGIN` and `END`: The block where the trigger logic is defined.

\*\*Advantages of Triggers\*\*:

1. \*\*Data Integrity\*\*: Triggers are used to enforce data integrity rules, ensuring that data remains consistent and accurate across the database. For example, a trigger can prevent invalid data from being inserted into a table or enforce referential integrity between related tables.

2. \*\*Business Logic Enforcement\*\*: Triggers allow you to implement and enforce complex business rules and logic directly in the database. This ensures that critical business rules are always enforced, regardless of the application accessing the database.

3. \*\*Audit Trail\*\*: Triggers can be used to create an audit trail by logging changes made to specific tables. This can be valuable for compliance, debugging, and security purposes.

4. \*\*Automating Tasks\*\*: Triggers can automate various tasks and processes within the database, such as updating associated data in other tables, sending notifications, or performing calculations.

5. \*\*Complex Constraints\*\*: Triggers enable the implementation of complex constraints that cannot be easily achieved using standard table constraints.

6. \*\*Logging and Debugging\*\*: Triggers can be used to log events or errors for debugging and troubleshooting purposes.

Despite their advantages, triggers should be used judiciously, as they introduce additional complexity to the database and can impact performance, especially if poorly designed or used excessively. Careful consideration should be given to the use of triggers and their potential impact on data consistency, performance, and maintenance.

## How do you optimize complex SQL queries?

Optimizing complex SQL queries is essential to improve database performance and reduce query execution time. Here are some strategies to optimize complex SQL queries:

1. \*\*Use Indexes\*\*: Ensure that the columns used in the WHERE, JOIN, ORDER BY, and GROUP BY clauses are indexed. Indexes can significantly speed up data retrieval.

2. \*\*Avoid SELECT \*\*\*: Instead of selecting all columns from a table, specify only the required columns. This reduces the amount of data fetched and improves query performance.

3. \*\*Use EXPLAIN\*\*: Utilize the database's EXPLAIN or EXPLAIN ANALYZE command to analyze the query execution plan and identify potential bottlenecks.

4. \*\*Use Proper Joins\*\*: Choose the appropriate type of join (INNER JOIN, LEFT JOIN, etc.) based on your requirements. Ensure that join conditions are efficient and use indexed columns.

5. \*\*Avoid Subqueries\*\*: If possible, rewrite subqueries as JOINs or use common table expressions (CTEs) to simplify the query and improve performance.

6. \*\*Avoid Functions on Indexed Columns\*\*: Avoid using functions on indexed columns in the WHERE clause, as it may prevent the use of indexes.

7. \*\*Optimize WHERE Clause\*\*: Use indexed columns in the WHERE clause to filter rows efficiently. Use OR and IN clauses judiciously, as they can affect query performance.

8. \*\*Limit Result Set\*\*: If you don't need all the rows, use LIMIT (for MySQL) or TOP (for SQL Server) to limit the number of results returned.

9. \*\*Avoid DISTINCT\*\*: Use DISTINCT sparingly, as it can impact query performance. Consider using GROUP BY instead.

10. \*\*Avoid Cartesian Products\*\*: Be cautious when joining large tables, as it can lead to a Cartesian product and dramatically slow down the query.

11. \*\*Use UNION ALL\*\*: When combining multiple result sets, use UNION ALL instead of UNION if you don't need duplicate removal. UNION ALL is faster since it doesn't perform duplicate elimination.

12. \*\*Partitioning\*\*: For very large tables, consider partitioning the data to improve query performance for specific ranges.

13. \*\*Data Denormalization\*\*: In some cases, denormalizing data (introducing controlled redundancy) can improve query performance by reducing the number of joins.

14. \*\*Caching\*\*: Consider caching the results of frequently executed queries to reduce the load on the database.

15. \*\*Optimize Hardware and Configuration\*\*: Ensure that the database server has enough resources (CPU, memory, disk I/O) and is correctly configured for the workload.

Remember that the effectiveness of these optimization strategies depends on the specific database schema, data distribution, and workload. Analyze the query execution plan and profile the performance to identify the most significant bottlenecks and areas for improvement. It's essential to strike a balance between complex queries and query performance, and sometimes breaking down complex queries into smaller, optimized parts can lead to better overall performance.

## Query is delaying in fetching data from a tables ?

\*\*Identifying Slow Query Performance:\*\*

- \*\*Indexes\*\*: Check for appropriate indexes on query columns to avoid full table scans.

- \*\*Statistics\*\*: Ensure table and index statistics are up-to-date for better query optimization.

- \*\*Query Plan\*\*: Analyze the query execution plan for inefficient operations.

- \*\*Joins\*\*: Verify efficient join conditions and join order.

- \*\*Subqueries\*\*: Consider optimizing or rewriting subqueries as JOINs.

- \*\*WHERE Clause\*\*: Ensure selective and indexed WHERE clauses.

- \*\*Network Latency\*\*: Account for network delays if the database is remote.

- \*\*Hardware/Resources\*\*: Monitor server resources like CPU, memory, and disk usage.

- \*\*Concurrent Transactions\*\*: Check for long-running or blocking transactions.

- \*\*Table Fragmentation\*\*: Address table fragmentation through maintenance.

- \*\*Caching\*\*: Use caching to store frequently accessed data.

- \*\*Locking\*\*: Investigate if excessive locking is impacting performance.

- \*\*Query Complexity\*\*: Simplify complex queries when possible.

- \*\*Data Volume\*\*: Consider data volume and partitioning strategies.

- \*\*Isolation Level\*\*: Ensure appropriate isolation levels to avoid excessive locking.

- \*\*Data Denormalization\*\*: Evaluate denormalization for performance gain.

- \*\*SQL Server Configuration\*\*: Adjust server settings for better performance.

By investigating and addressing these factors, you can improve the performance of slow queries in the database. Keep in mind that performance tuning is an ongoing process as workload and data change over time.

## Query to find the details of employee with Maximum total salary

To find the details of the employee with the maximum total salary, you can use a SQL query with the `ORDER BY` and `LIMIT` clauses to sort the employees based on their total salary in descending order and then fetch the top record. Assuming you have two tables, one for employees (e.g., `employees`) and another for their salaries (e.g., `salaries`), you can use the following SQL query:

```sql

SELECT e.employee\_id, e.employee\_name, e.department, SUM(s.salary\_amount) AS total\_salary

FROM employees e

JOIN salaries s ON e.employee\_id = s.employee\_id

GROUP BY e.employee\_id, e.employee\_name, e.department

ORDER BY total\_salary DESC

LIMIT 1;

```

In this query:

1. We use the `JOIN` clause to combine the `employees` and `salaries` tables based on the `employee\_id` column.

2. The `SUM` function is used to calculate the total salary for each employee.

3. The `GROUP BY` clause groups the results by `employee\_id`, `employee\_name`, and `department`, ensuring that we get the total salary for each employee.

4. The `ORDER BY` clause sorts the results in descending order of the total salary.

5. The `LIMIT 1` clause restricts the output to only the top record, which will have the employee with the maximum total salary.

This query will give you the details of the employee with the highest total salary in the `employees` table, along with the total salary amount.

## What are constraints in sql

In SQL, constraints are rules and conditions applied to the data in database tables to maintain data integrity and enforce business rules. They help ensure that data in the tables follows specific rules and meets certain criteria. Constraints play a vital role in preventing invalid or inconsistent data from being inserted, updated, or deleted in the database.

Here are some common types of constraints in SQL:

1. \*\*Primary Key Constraint\*\*:

- Ensures that a column or a set of columns uniquely identifies each row in the table.

- Only one primary key constraint is allowed per table.

- The primary key column(s) must have unique and non-null values.

2. \*\*Unique Constraint\*\*:

- Ensures that the values in the specified column or set of columns are unique and cannot be duplicated within the table.

- Unlike a primary key, a unique constraint can have one or more null values.

3. \*\*Foreign Key Constraint\*\*:

- Establishes a relationship between two tables by linking the primary key of one table to the foreign key of another table.

- It ensures referential integrity and helps maintain data consistency across related tables.

4. \*\*Check Constraint\*\*:

- Allows you to specify a condition that must be true for each row in the table.

- The check constraint ensures that data adheres to a specific condition, such as a range of values or a pattern.

5. \*\*Not Null Constraint\*\*:

- Ensures that a specified column does not contain any null values.

- It enforces the requirement that the column must have a value for every row.

6. \*\*Default Constraint\*\*:

- Specifies a default value for a column that is used when an INSERT statement does not provide a value for that column.

- It allows you to set a predefined value for a column if the user does not explicitly provide one.

Constraints are an essential part of database design as they help maintain data integrity and consistency. By enforcing rules on the data, constraints prevent errors and ensure that the database remains accurate and reliable. Properly defining constraints is crucial in designing a robust and efficient database schema.

# Microservices:

## What are microservices

Microservices are a software architectural style that structures an application as a collection of small, independent, and loosely coupled services. Each service represents a specific business capability and can be developed, deployed, and scaled independently. Microservices architecture aims to break down a complex monolithic application into smaller, manageable components, each running as a separate service.

Key characteristics of microservices:

1. \*\*Service Independence\*\*: Each microservice is self-contained and operates independently of other services. They have their own codebase, database, and communication mechanism.

2. \*\*Decentralized Data Management\*\*: Each microservice manages its data, and there is no shared database among services. Communication between services is typically done through APIs.

3. \*\*Loose Coupling\*\*: Microservices are loosely coupled, meaning changes to one service do not directly impact other services. This allows for better maintainability and scalability.

4. \*\*Technology Diversity\*\*: Different microservices can be developed using different technologies, frameworks, and programming languages, allowing teams to use the best tool for their specific service.

5. \*\*Scaling\*\*: Each microservice can be scaled independently based on its individual resource needs, improving overall system performance and resource utilization.

6. \*\*Resilience and Fault Isolation\*\*: If one microservice fails, the rest of the system can continue to function without a complete outage.

7. \*\*Continuous Delivery\*\*: Microservices enable continuous delivery and deployment by allowing individual services to be updated and released independently.

8. \*\*Ease of Development\*\*: Smaller codebases and separation of concerns make it easier for developers to work on individual services, promoting faster development cycles.

Microservices architecture is well-suited for complex, large-scale applications, especially in scenarios where agility, scalability, and maintainability are essential. However, it also adds complexity in terms of service orchestration, communication, and monitoring, which need to be managed carefully.

It's important to consider factors such as team size, organization structure, and the complexity of the application before deciding to adopt microservices. While microservices offer numerous benefits, they also come with additional challenges that need to be addressed effectively for successful implementation.

## Advantages and disadvantages of microservice architecture

Microservice architecture offers various benefits, but it also comes with its own set of challenges. Here are some advantages and disadvantages of microservice architecture:

Advantages:

1. \*\*Scalability\*\*: Microservices allow individual services to be scaled independently based on their resource requirements. This enables better utilization of resources and cost optimization.

2. \*\*Modularity and Maintainability\*\*: Services are small and focused on specific business capabilities, making them easier to develop, test, and maintain. Changes to one service have limited impact on others, promoting continuous delivery.

3. \*\*Technology Diversity\*\*: Different services can be developed using different technologies, allowing teams to choose the best tools for their specific tasks.

4. \*\*Independent Deployment\*\*: Each service can be deployed independently, facilitating faster release cycles and reducing the risk of a single large deployment.

5. \*\*Fault Isolation\*\*: If one service fails, it doesn't bring down the entire system, as other services can continue to function independently.

6. \*\*Improved Team Productivity\*\*: Smaller, focused teams can work independently on individual services, leading to faster development and easier collaboration.

7. \*\*Flexibility and Agility\*\*: Microservices enable organizations to respond quickly to changing business requirements and adapt to new technologies.

Disadvantages:

1. \*\*Complexity\*\*: Microservices introduce a higher level of complexity compared to monolithic architecture. Managing service communication, versioning, and orchestration can become challenging.

2. \*\*Service Coordination\*\*: Proper service coordination and communication become crucial, as services need to interact with each other effectively.

3. \*\*Data Management\*\*: Managing data consistency across multiple services can be complex without a shared database. Maintaining data integrity becomes a challenge.

4. \*\*Latency and Overhead\*\*: Communication between services can introduce latency and increase network overhead, affecting overall system performance.

5. \*\*Distributed Systems Issues\*\*: Debugging and monitoring distributed systems can be more complex than monolithic applications.

6. \*\*Operational Complexity\*\*: Running and managing multiple services requires sophisticated infrastructure and monitoring tools.

7. \*\*Testing Challenges\*\*: Testing services in isolation and performing end-to-end testing can be challenging.

8. \*\*Overhead of API Development\*\*: Designing and maintaining APIs for each service requires careful planning and coordination.

The decision to adopt microservices should be based on the specific requirements and constraints of the project. While microservices offer various advantages, they are not a one-size-fits-all solution, and organizations need to weigh the benefits against the added complexity and challenges that come with microservices architecture.

## How you handle security between different microservices

Handling security between different microservices is a critical aspect of a microservices architecture. As each microservice operates independently and communicates over the network, it's essential to ensure that proper security measures are in place to protect sensitive data and prevent unauthorized access. Here are some common security practices for managing security between microservices:

1. \*\*Authentication and Authorization\*\*:

- Implement a robust authentication mechanism to verify the identity of users and services. Use techniques like JWT (JSON Web Tokens) or OAuth for authentication.

- Define fine-grained authorization rules to control access to different microservices and their endpoints. This prevents unauthorized access to sensitive resources.

2. \*\*Secure Communication\*\*:

- Use secure communication protocols such as HTTPS/TLS to encrypt data transmitted between microservices, preventing data interception and tampering.

- Implement mutual SSL (client and server certificates) for service-to-service authentication, ensuring that only trusted services can communicate with each other.

3. \*\*API Gateways\*\*:

- Use an API gateway as a single entry point for external requests. The API gateway can handle security-related concerns like authentication, rate limiting, and request validation before forwarding requests to individual microservices.

4. \*\*Centralized Identity and Access Management (IAM)\*\*:

- Consider using a centralized IAM system to manage user identities and access rights. This simplifies user authentication and authorization across all microservices.

5. \*\*OAuth and OpenID Connect\*\*:

- Implement OAuth and OpenID Connect for secure and standardized authentication and authorization between microservices and external clients.

6. \*\*Role-Based Access Control (RBAC)\*\*:

- Use RBAC to define roles and permissions for users and services, allowing for fine-grained access control.

7. \*\*Token-based Authentication\*\*:

- Use token-based authentication mechanisms (e.g., JWT) to exchange identity information between microservices securely.

8. \*\*Audit Logging\*\*:

- Implement audit logging to record important security-related events and monitor access to critical resources.

9. \*\*Security Testing\*\*:

- Regularly perform security assessments and penetration testing to identify vulnerabilities and weaknesses in the microservices.

10. \*\*Secure Configuration Management\*\*:

- Store sensitive configuration properties (e.g., passwords, API keys) securely, using environment variables or encrypted configuration files.

11. \*\*Container Security\*\*:

- If deploying microservices in containers, ensure container security best practices are followed to minimize attack surfaces and vulnerabilities.

It's important to have a well-defined security strategy and follow security best practices at every layer of the microservices architecture. Security considerations should be an integral part of the design, development, and deployment process for each microservice. Additionally, using standardized security protocols and tools can help ensure consistency and ease of implementation across different microservices.

## How to handle the transaction in microservices architecture

Handling transactions in a microservices architecture requires careful consideration due to the distributed nature of microservices. Transactions need to be managed in a way that ensures data consistency and integrity across multiple services. Here are some common approaches to handle transactions in a microservices architecture:

1. \*\*Distributed Transactions\*\*:

- In some cases, you may need to perform distributed transactions that involve multiple microservices and databases. This can be achieved using a distributed transaction management system, such as XA (eXtended Architecture) transactions, to coordinate the commit or rollback of multiple resources across services.

- However, distributed transactions can be complex and may lead to performance and scalability issues, so they should be used judiciously and only for critical cases.

2. \*\*Saga Pattern\*\*:

- The Saga pattern is a coordination pattern for managing distributed transactions in microservices. It breaks down a distributed transaction into a series of smaller, local transactions (sagas) that can be individually managed and rolled back if needed.

- Each saga represents a step in the overall transaction flow and can send messages to other services to request their participation in the saga. If a step fails, the corresponding compensation action can be triggered to undo the changes made in previous steps.

- Implementing sagas requires careful design and error handling to ensure proper coordination and compensation actions.

3. \*\*Event-Driven Architecture (EDA)\*\*:

- In an event-driven architecture, microservices communicate through events, which represent domain events or changes in the system state. Each microservice processes events asynchronously.

- Transactions can be managed within individual microservices, and any cross-service communication can be achieved through events. This decouples services, allowing them to handle their own data and transactions independently.

4. \*\*API Composition and Chained Requests\*\*:

- In some cases, you can design your APIs and microservices in a way that allows clients to make multiple requests and compose the necessary operations within a single transaction scope.

- The client can manage the transaction boundaries by making chained requests to different microservices and ensuring that all operations succeed before committing the transaction.

5. \*\*Compensating Actions and Idempotence\*\*:

- Ensure that your microservices are designed with compensating actions to handle potential failures during a transaction or saga.

- Make your services idempotent, meaning that the same operation can be performed multiple times without causing undesirable effects. This helps ensure that retries or compensation actions do not cause data inconsistencies.

Each approach has its trade-offs, and the choice of transaction management strategy depends on the specific requirements of your microservices architecture, the complexity of the interactions between services, and the level of data consistency and integrity needed for your application. Consider the impact on performance, complexity, and maintainability when deciding on the best approach for handling transactions in your microservices architecture.

## Centralized log monitoring in Microservices

Centralized log monitoring in a microservices architecture is crucial for gaining visibility into the entire system, tracking issues, and troubleshooting problems effectively. Centralized log monitoring allows you to aggregate logs from multiple microservices into a single location, making it easier to search, analyze, and monitor the system as a whole. Here are the key steps to implement centralized log monitoring in a microservices environment:

1. \*\*Log Aggregation\*\*: Configure each microservice to send its logs to a centralized logging system. The logs can be sent over the network using protocols like Syslog, Logstash, Fluentd, or directly to a log management service like Elasticsearch, Splunk, or Graylog.

2. \*\*Logging Libraries\*\*: Use logging libraries like Logback, Log4j2, or SLF4J in your microservices to format and send logs to the centralized logging system. These libraries allow you to customize log formats, log levels, and log destinations.

3. \*\*Log Forwarding\*\*: Consider using a log forwarding agent like Filebeat or Fluentd on each server or container to collect logs locally and forward them to the central logging system. This reduces the network overhead and ensures log availability even if the central logging system is temporarily unavailable.

4. \*\*Centralized Log Storage\*\*: Choose a centralized logging platform, such as Elasticsearch, Logstash, and Kibana (ELK stack), or other log management services like Splunk, Graylog, or Datadog. These platforms provide powerful search, analysis, and visualization capabilities.

5. \*\*Log Indexing and Searching\*\*: Ensure that the centralized logging system indexes the logs properly, enabling fast and efficient search queries across logs from different microservices.

6. \*\*Logging with Context\*\*: Include relevant contextual information in your logs, such as request IDs, service names, timestamps, and log levels. This information helps in tracing transactions and diagnosing issues across multiple microservices.

7. \*\*Real-time Monitoring and Alerts\*\*: Set up real-time monitoring and alerts for specific log events or patterns. This allows you to respond quickly to critical issues and anomalies in the system.

8. \*\*Log Rotation and Retention\*\*: Configure log rotation and retention policies to manage log storage efficiently and comply with data retention requirements.

9. \*\*Log Security\*\*: Ensure that log access is secure and restricted to authorized personnel only. Sensitive information should be appropriately masked or excluded from logs.

10. \*\*Distributed Tracing\*\*: Consider implementing distributed tracing in conjunction with centralized logging. Distributed tracing provides end-to-end visibility into the flow of requests across microservices and helps identify performance bottlenecks and latencies.

Centralized log monitoring simplifies the process of diagnosing and troubleshooting issues in a microservices architecture. It enables DevOps teams to have a unified view of the system's health and performance, making it easier to detect anomalies, perform root cause analysis, and optimize the overall system.

## What is hystrix, api gateway

Calling other microservices in a microservices architecture involves making HTTP requests or using messaging protocols to communicate with different services. Here are some common methods for calling other microservices:

1. \*\*RESTful API Calls\*\*:

- Most microservices communicate with each other using RESTful APIs over HTTP/HTTPS. Each microservice exposes a set of well-defined endpoints (URLs) that other services can call to request data or perform actions.

- When one microservice needs to call another, it constructs an HTTP request (GET, POST, PUT, DELETE, etc.) with the necessary parameters and headers and sends it to the target service's API endpoint.

- RESTful API calls are simple and widely used for inter-service communication in microservices architecture.

2. \*\*Service Discovery and Registry\*\*:

- In a dynamic microservices environment, where the locations and IP addresses of services may change frequently, service discovery and registry systems are used to locate other microservices.

- Popular service discovery tools include Eureka, Consul, and ZooKeeper. The calling microservice can use these tools to discover the address and port of the target service and then make an API call to the identified service.

3. \*\*Load Balancing\*\*:

- To ensure high availability and distribute the load across multiple instances of a microservice, load balancing is often used.

- When calling a microservice, a load balancer can distribute the request to one of the available instances of that microservice, providing fault tolerance and performance improvement.

4. \*\*Asynchronous Messaging\*\*:

- In some cases, microservices communicate asynchronously using messaging protocols like Apache Kafka, RabbitMQ, or ActiveMQ.

- Instead of calling a microservice directly, a message is sent to a message broker, and the target microservice consumes the message from the broker.

- Asynchronous messaging can help decouple services and provide more resilience to spikes in traffic.

5. \*\*gRPC\*\*:

- gRPC is a high-performance RPC (Remote Procedure Call) framework developed by Google. It allows you to define a service contract using Protocol Buffers and supports bidirectional streaming and binary protocols.

- gRPC can be a good choice when you require efficient and low-latency communication between microservices.

6. \*\*Circuit Breaker Pattern\*\*:

- When calling external microservices, it's essential to handle failures gracefully. The Circuit Breaker pattern can help with that.

- A circuit breaker monitors the calls to a microservice and can open the circuit (stop further calls) if the service is experiencing issues. It can fall back to a default behavior or return cached data if the service is unavailable.

It's important to design inter-service communication carefully, considering factors such as performance, security, fault tolerance, and scalability. Also, keep in mind that calling other microservices over the network introduces potential points of failure, so proper error handling and fallback mechanisms are crucial for building robust microservices.

## How to call other microservices

In a microservices architecture, it's common to have different services that communicate with each other through APIs. One way to make HTTP requests to other services' APIs from within a Java application is by using the `RestTemplate` class provided by the Spring Framework. The `RestTemplate` simplifies the process of making HTTP requests and handling responses.

Here's an example of how you can use the `RestTemplate` to call another REST API in a microservices architecture:

1. First, make sure you have the necessary dependencies in your project's `pom.xml` file if you're using Maven:

```xml

<dependencies>

<!-- Other dependencies -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

```

2. Create a Spring Boot application class:

```java

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.web.client.RestTemplate;

@SpringBootApplication

public class MyMicroserviceApplication {

public static void main(String[] args) {

SpringApplication.run(MyMicroserviceApplication.class, args);

}

}

```

3. Create a service or component where you'll use the `RestTemplate` to call another REST API:

```java

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.ResponseEntity;

import org.springframework.stereotype.Service;

import org.springframework.web.client.RestTemplate;

@Service

public class MyService {

private final RestTemplate restTemplate;

@Autowired

public MyService(RestTemplate restTemplate) {

this.restTemplate = restTemplate;

}

public void callOtherServiceApi() {

String apiUrl = "http://other-service-url/api/resource"; // Replace with the actual URL

ResponseEntity<String> response = restTemplate.getForEntity(apiUrl, String.class);

if (response.getStatusCode().is2xxSuccessful()) {

String responseBody = response.getBody();

System.out.println("Response from other service: " + responseBody);

} else {

System.err.println("Error response from other service: " + response.getStatusCodeValue());

}

}

}

```

In this example, the `RestTemplate` is injected into the `MyService` class using Spring's dependency injection. The `callOtherServiceApi` method makes a GET request to the specified API URL and handles the response accordingly.

Remember to configure the `RestTemplate` bean in your Spring Boot application's configuration. You can do this in the main application class:

```java

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.client.RestTemplate;

@Configuration

public class RestTemplateConfig {

@Bean

public RestTemplate restTemplate() {

return new RestTemplate();

}

}

```

Keep in mind that this is a basic example, and in a real-world scenario, you might need to handle more complex scenarios, such as handling different HTTP methods, request and response bodies, error handling, authentication, and so on.

## What is SAGA Design Pattern

The Saga Design Pattern is a distributed architectural pattern used to manage long-running and complex transactions in a microservices environment. It addresses the challenges of maintaining data consistency and integrity when a single business operation spans multiple microservices that operate independently and communicate asynchronously.

In a microservices architecture, a traditional ACID (Atomicity, Consistency, Isolation, Durability) transaction that spans multiple services is often not feasible due to the distributed nature of the system. The Saga pattern breaks down the distributed transaction into a sequence of smaller, local transactions (also known as saga steps), each managed by individual microservices. Each saga step updates the data and emits events to trigger subsequent steps or compensating actions if necessary.

Key concepts of the Saga Design Pattern:

1. \*\*Saga Orchestration\*\*: The saga is orchestrated by a central component called the Saga Orchestrator or Saga Coordinator. The orchestrator is responsible for coordinating the execution of saga steps and ensuring their correct order.

2. \*\*Saga Step\*\*: A saga step is an individual transactional operation performed by a microservice. Each step is idempotent, meaning it can be executed multiple times without causing inconsistencies.

3. \*\*Event-Driven Communication\*\*: Saga steps communicate with each other through events. When a saga step completes successfully, it emits an event to trigger the next step. If a step fails, it emits a compensating event to undo the changes made by the previous steps.

4. \*\*Compensating Action\*\*: A compensating action is a reverse operation that can undo the effects of a completed step. If a saga step fails, the orchestrator triggers compensating actions in reverse order to bring the system back to a consistent state.

5. \*\*Timeouts and Sagas Reversal\*\*: To handle failures and ensure eventual consistency, sagas include timeout mechanisms. If a saga step doesn't receive an expected event within a specified time, the saga can be reversed to undo the changes made by previous steps.

The Saga pattern is useful in scenarios where distributed transactions are complex, and traditional transaction management is not feasible or leads to performance bottlenecks. It allows for better fault tolerance, resiliency, and scalability in a microservices architecture.

It's important to note that implementing the Saga pattern can be challenging due to the need for event-driven communication, handling failures, and managing the complexity of long-running transactions. Proper design and careful error handling are essential to ensure the integrity and consistency of the data across microservices. Some distributed messaging systems, like Apache Kafka or RabbitMQ, are often used in conjunction with the Saga pattern to facilitate event-driven communication and reliable message delivery between microservices.

## Design Pattern in Microservices.

In a microservices architecture, various design patterns are employed to address specific challenges and promote best practices for developing scalable, maintainable, and loosely coupled microservices. Here are some common design patterns used in microservices:

1. \*\*Microservice Architecture\*\*: The foundational design pattern that defines the overall architecture as a collection of small, independent, and loosely coupled services that collaborate to build a larger application.

2. \*\*Service Registry and Discovery\*\*: A design pattern that uses a service registry (e.g., Eureka, Consul) to register and discover microservices dynamically. This allows services to find and communicate with each other without hardcoding the service endpoints.

3. \*\*API Gateway\*\*: The API Gateway pattern provides a single entry point for client requests and acts as a reverse proxy to route requests to appropriate microservices. It can handle tasks like authentication, rate limiting, caching, and load balancing.

4. \*\*Circuit Breaker\*\*: The Circuit Breaker pattern prevents cascading failures in a distributed system by monitoring the status of calls to a microservice. If the service fails, the circuit breaker opens, and subsequent calls are fast-failed or redirected to a fallback mechanism.

5. \*\*Bulkhead\*\*: The Bulkhead pattern isolates microservices into separate pools of resources (threads, connections, etc.), ensuring that the failure of one service does not impact the performance of others.

6. \*\*Saga\*\*: The Saga pattern manages long-running and complex distributed transactions by breaking them into smaller, manageable steps. It uses event-driven communication and compensating actions to ensure data consistency.

7. \*\*Event Sourcing\*\*: The Event Sourcing pattern stores all changes to an application's state as a sequence of events. It allows services to rebuild the current state and provides a history of all state changes.

8. \*\*CQRS (Command Query Responsibility Segregation)\*\*: CQRS separates the read and write operations into distinct models. It allows optimizing read and write operations independently and improves performance.

9. \*\*Domain-Driven Design (DDD)\*\*: DDD is a methodology that helps design microservices around the business domain. It focuses on creating a common language and understanding between domain experts and developers.

10. \*\*Saga Choreography\*\*: In contrast to Saga Orchestration, Saga Choreography relies on events and decentralized communication to coordinate transactions between services without a central coordinator.

11. \*\*Asynchronous Messaging\*\*: Using messaging protocols (e.g., Kafka, RabbitMQ) to enable asynchronous communication between microservices. This pattern decouples services and allows for greater scalability and fault tolerance.

12. \*\*Leader Election\*\*: Used for ensuring only one instance of a microservice performs a specific task, such as cron jobs or background processing.

13. \*\*Cross-Cutting Concerns\*\*: Design patterns for handling cross-cutting concerns like logging, monitoring, and security in a consistent and reusable manner across microservices.

Each design pattern addresses a specific aspect of microservices development and helps to overcome the challenges posed by the distributed nature of microservices architecture. Careful consideration of these patterns during the design phase can lead to a more robust and well-structured microservices system.

* Any Messageing Queue : KAFKA, Rabbit MQ, SQL

# JSP:

* Implicit Objects in JSP
* Directive in JSP
* How we will get session in servlet
* Expression Tags in JSP
* How to create implicit objects in JSP