Core Java & Java8:

=================

**1)HashMap internal implementation. if you are working with Java8 what is the newly added feature in that?**

In Java 8, while the `HashMap` implementation itself didn't change significantly, Java introduced the concept of "Red-Black Trees" for handling hash collisions in `HashMap`.

Prior to Java 8, when multiple elements hashed to the same bucket in a `HashMap`, a linked list was used to handle these collisions. However, in Java 8, if a bucket contains too many elements (specifically, if a bucket's size exceeds a certain threshold), instead of a linked list, a balanced tree structure (Red-Black Tree) is used for better performance.

This change was made to improve the worst-case time complexity of certain operations from O(n) to O(log n) when dealing with hash collisions. This helps maintain better performance for large HashMap’s where hash collisions are more likely to occur.

This optimization aims to reduce the linear traversal time of a linked list in case of hash collisions, especially when the number of elements in a bucket grows beyond a certain threshold, offering more predictable performance characteristics for HashMap operations.

The decision to transform a bucket from a linked list to a Red-Black Tree is based on the number of elements in the bucket and is part of Java's effort to improve the performance of `HashMap` in scenarios where hash collisions are prevalent

**What is Red-Black Tree?**

A Red-Black Tree is a self-balancing binary search tree with the following properties:

1. Binary Search Tree Property: Similar to other binary search trees, each node has at most two children, and for any given node:

- All nodes in its left subtree have values less than its own value.

- All nodes in its right subtree have values greater than its own value.

2. Color Property: Each node in a Red-Black Tree is colored either red or black

The Red-Black Tree must satisfy the following additional properties, ensuring it remains balanced:

3. Root Property: The root node is always black.

4. Red Property: Red nodes cannot have red children. In other words, no two adjacent (connected by an edge) red nodes can exist in the tree.

5. Black-Height Property: The number of black nodes on any path from a node to its descendant leaves must be the same for all paths. This property ensures that the longest path from the root to any leaf node is no more than twice the length of the shortest path.

These properties ensure that a Red-Black Tree remains approximately balanced, preventing it from becoming highly skewed and guaranteeing that the tree's height remains logarithmic relative to the number of nodes. This balance is crucial for maintaining efficient lookup, insertion, and deletion operations

### Operations in a Red-Black Tree:

- Insertion: When a new node is added, the tree performs a standard binary search tree insertion. After insertion, the tree might violate the Red-Black properties, so it re-balances itself by performing rotations and recoloring nodes to ensure that all properties hold.

- Deletion: Similar to insertion, deletion might violate the Red-Black properties. The tree performs standard binary search tree deletion and then adjusts the tree's structure and colors to maintain the Red-Black properties

### Advantages of Red-Black Trees:

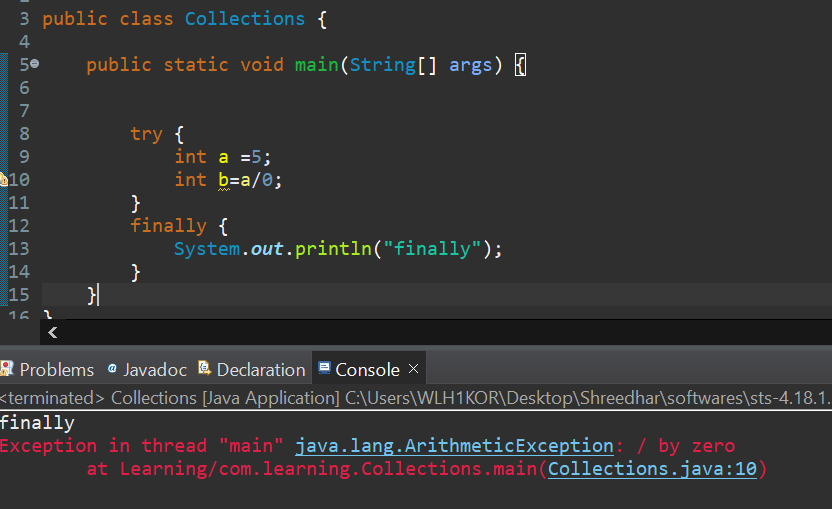
- Balanced Structure: Red-Black Trees maintain balance, ensuring that operations have a worst-case time complexity of O(log n).

- Efficient Operations: Lookup, insertion, and deletion operations remain efficient due to the balanced structure of the tree.

Red-Black Trees are widely used in various data structures and libraries due to their ability to provide a balance between efficient performance and maintaining a reasonably balanced tree structure.

**4)Can a try block can be followed a finally block?if so who will handle the exception that occured inside try block?**

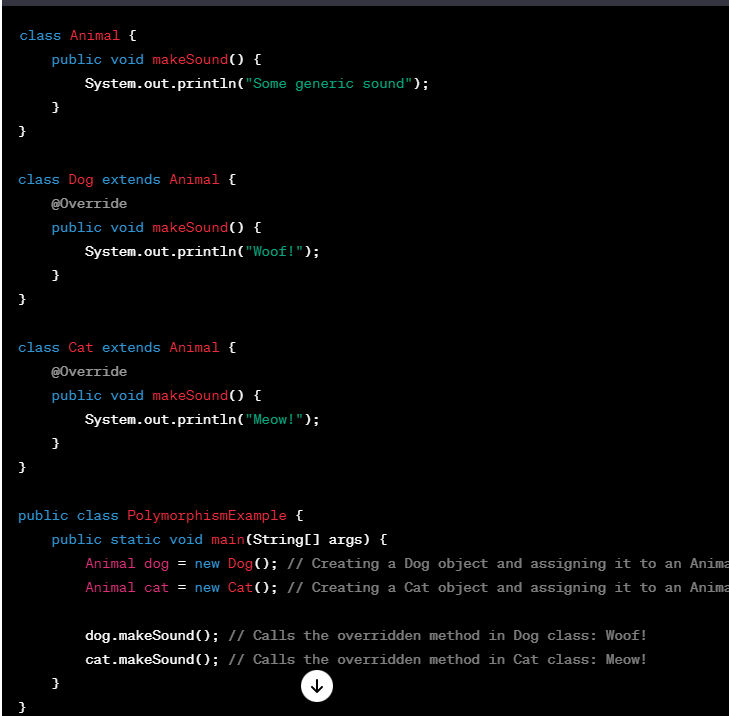
Yes, If the exception is not caught by any catch block within the try block, it will propagate out of the try block and might be caught by an enclosing try-catch block or result in program termination if not caught anywhere.



**3)what is dynamic polymorphism. how to achieve it explain with an example?**

Dynamic polymorphism in object-oriented programming refers to the ability of a language to invoke overridden methods at runtime. It allows a subclass to provide a specific implementation of a method that is already defined in its superclass, and the appropriate method to be executed is determined during runtime based on the actual type of the object.

In Java, dynamic polymorphism is achieved through method overriding, where a subclass provides a specific implementation of a method that is already present in its superclass.



**5)ArrayList vs LikedList?which is more efficient in memory usage?how?**

`ArrayList` and `LinkedList` are both implementations of the `List` interface in Java, but they use different underlying data structures, which impacts their efficiency in terms of memory usage and performance.

- ArrayList uses an array to store elements. It provides dynamic resizing, allowing it to grow dynamically as elements are added. When the internal array reaches its capacity, it's reallocated with a larger size, which can lead to occasional resizing overhead. However, it provides fast access to elements by index (`get` operation) since it directly uses array indices.

- LinkedList uses a doubly linked list structure where each element (node) holds a reference to the previous and next elements in the list. It doesn't require resizing like ArrayList does, but it uses more memory per element due to the overhead of maintaining these references. While insertion and deletion in LinkedList can be faster for certain operations (especially when dealing with a large number of insertions/deletions in the middle of the list), accessing elements by index (`get` operation) is slower compared to ArrayList because it needs to traverse the list from the beginning or end to reach a specific index.

Regarding memory efficiency:

- ArrayList typically consumes less memory per element since it primarily stores the actual elements in the array without the overhead of maintaining node references.

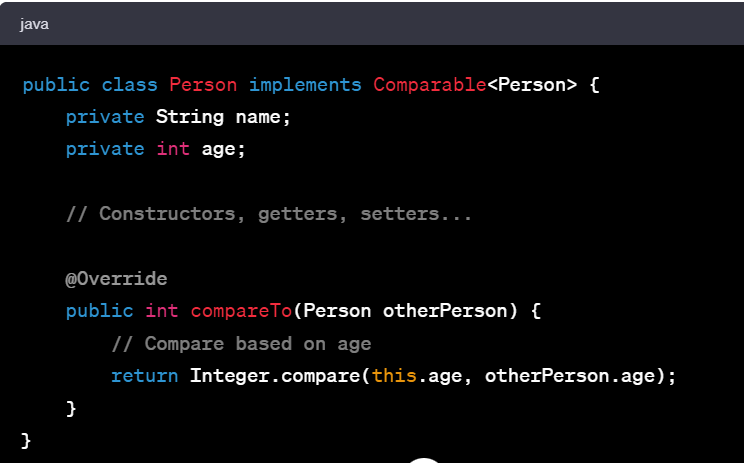
- LinkedList tends to consume more memory per element due to the overhead of maintaining references for each node (previous and next pointers).

In summary, if memory efficiency is a concern and you primarily require fast random access by index, `ArrayList` might be a better choice. However, if you need frequent insertions or deletions in the middle of the list and memory usage is not a critical concern, `LinkedList` might be more suitable.

**6)Comparable vs Comparator differences? Explain with real time example?**

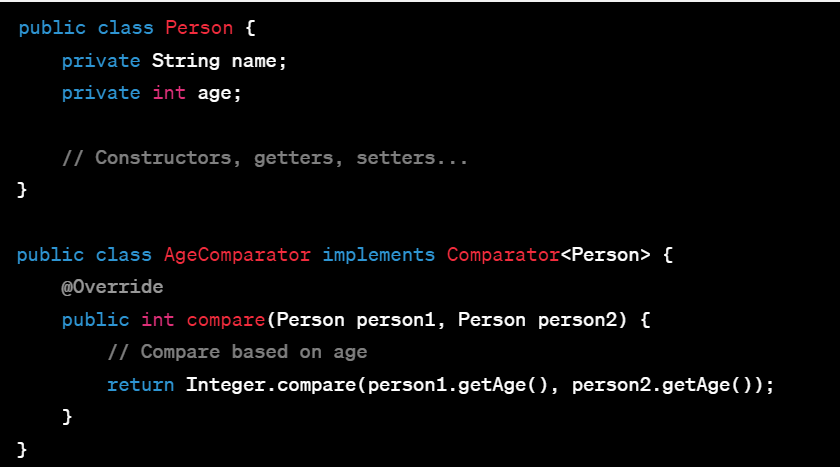
Absolutely! Both `Comparable` and `Comparator` are interfaces in Java used for sorting objects, but they differ in their implementation and usage.

- Comparable: This interface is implemented by a class to define a natural ordering for its objects. It means that the class itself contains the logic of how its instances should be compared to each other. The `Comparable` interface has a `compareTo` method that compares the current object with another object, returning a negative integer, zero, or a positive integer, indicating whether the current object is less than, equal to, or greater than the other object, respectively.



In this example, the `Person` class implements `Comparable` and defines how `Person` objects should be compared based on their age.

- Comparator: This interface is used to define multiple comparison rules for classes whose source code cannot be modified (external sorting logic). It provides flexibility by allowing the definition of multiple ways to compare objects. A `Comparator` object can be created separately from the class whose objects are being compared.



Here, `AgeComparator` is a separate class implementing `Comparator` that defines how `Person` objects should be compared based on age.

Real-time example:

Imagine a scenario where you have a list of `Person` objects and you want to sort them. If `Person` implements `Comparable`, you can simply use `Collections.sort()` to sort the list based on the natural ordering defined in the `compareTo` method.

If, however, you have different use cases where sometimes you want to sort by age, sometimes by name, and sometimes by some other criteria, you can create different `Comparator` implementations (e.g., `AgeComparator`, `NameComparator`) to sort the list differently without modifying the `Person` class itself.

So, in essence:

- `Comparable` is for natural ordering and is implemented by the class itself.

- `Comparator` is for external ordering and allows multiple comparison strategies for classes whose source code cannot be modified.

**7)Why wait(),notify(),notifyAll() methods are in Object class?**

The `wait()`, `notify()`, and `notifyAll()` methods are in the `Object` class in Java because they are fundamental mechanisms for managing threads and synchronization, and they operate on the monitor associated with an object.

Here's the rationale behind their placement:

1. wait(): This method is used for causing the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object. When a thread calls `wait()` on an object, it releases the lock on the object and goes into a waiting state until another thread notifies it or a specified amount of time elapses.

2. notify(): This method wakes up a single thread that is waiting on the monitor of the object. If multiple threads are waiting on the object, only one of them will be chosen to be awakened.

3. notifyAll(): This method wakes up all the threads that are waiting on the monitor of the object. It notifies all waiting threads and gives them an opportunity to proceed.

These methods are related to thread synchronization and inter-thread communication. Placing them in the `Object` class allows all objects in Java to act as potential monitors for synchronization purposes. Every Java object has an associated monitor, and these methods are used to interact with that monitor.

Since these methods deal with the concept of locking and waiting on a shared resource (the monitor associated with an object), it makes sense to define them in the root class `Object` so that all objects in Java can utilize these synchronization mechanisms without the need for a specialized or dedicated class/interface.

**8)what is metaspace in java8?**

In Java 8 and later versions, the "Metaspace" was introduced to replace the older "Permanent Generation" (PermGen) memory space. Metaspace is part of the native memory and it stores metadata about the classes used in a Java application, such as class names, method names, field names, annotations, and other related information.

Unlike the Permanent Generation, which had a fixed size and could cause OutOfMemoryErrors if classes or metadata were loaded excessively, Metaspace is dynamic and can expand or shrink based on the application's need for metadata.

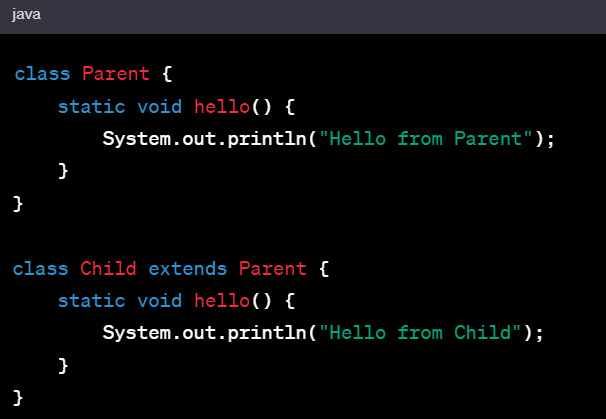
Metaspace memory is not limited by the `-XX:MaxMetaspaceSize` parameter, and it's automatically managed by the JVM. However, if your application is dynamically generating classes or using reflection extensively, it's still essential to monitor Metaspace usage to prevent potential memory issues.

Metaspace has improved memory management and reduces the likelihood of encountering memory-related errors due to class metadata.

**9)can we override static method in java? if so what is method hiding in java?**

In Java, you can't technically override a static method. When you define a method in a subclass with the same signature as a static method in the superclass, it's not considered overriding. Instead, it's called "method hiding" or "hiding static methods."

Here's an example to illustrate method hiding:



If you were to call `Parent.hello()` and `Child.hello()`, they would behave differently. However, this behavior isn't polymorphic; it's determined by the reference type at compile time.

Parent.hello(); // Output: Hello from Parent

Child.hello(); // Output: Hello from Child

The static method `hello()` in the `Child` class hides the method in the `Parent` class rather than overriding it. The method resolution is determined by the reference type, not the actual object instance.

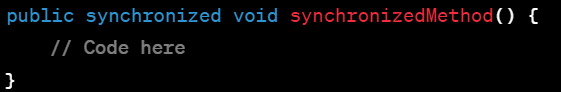
Remember, hiding static methods doesn't follow polymorphism or dynamic dispatch like instance methods. It's a compile-time decision based on the reference type.

**10)synchronization: which levels we can apply this?**

In Java, synchronization can be applied at different levels to control concurrent access to shared resources and ensure thread safety.

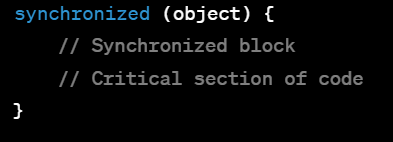
1. Method Level:

You can use the `synchronized` keyword to synchronize entire methods. When a method is synchronized, only one thread can execute that method on an object at a time. For example:



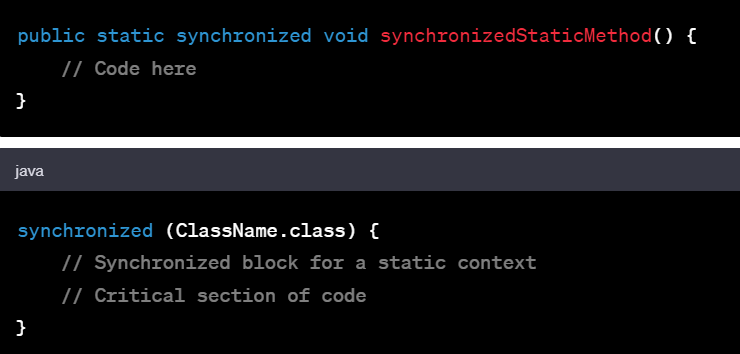
2. Block Level:

Synchronization can be applied to specific blocks of code using the `synchronized` block. This allows more fine-grained control over synchronization and can help improve performance by reducing the critical section.



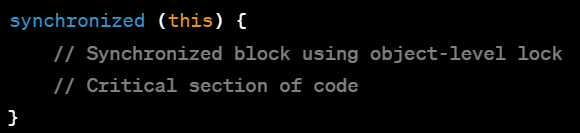
3. Static Methods and Blocks:

Static methods or blocks can also be synchronized, either by using the `synchronized` keyword on the method or using a synchronized block with a class-level lock.



4. Object Level:

Each object in Java has an intrinsic lock associated with it. You can use this lock to synchronize critical sections of code pertaining to specific objects.



5. Reentrant Locks:

Java's `java.util.concurrent.locks.ReentrantLock` allows more flexible control over locks, providing functionalities like fairness, try-lock, and lock interruption, unlike intrinsic locks.

These different levels of synchronization offer varying degrees of control and granularity over concurrent access to shared resources, allowing you to choose the appropriate level of synchronization based on your application's requirements.

**11)differences between ConcurrentHashMap and synchronizedmap**

`ConcurrentHashMap` and `Collections.synchronizedMap()` both provide thread-safe implementations for a `Map` in Java. However, they have significant differences in their implementations and performance characteristics:

1. ConcurrentHashMap:

- Concurrency Level: ConcurrentHashMap is designed for high-concurrency scenarios. It allows multiple threads to read and write concurrently without blocking each other (with some limitations).

- Segmented Structure: Internally, ConcurrentHashMap is divided into segments, and each segment can be locked independently. This allows multiple threads to operate on different segments simultaneously, reducing contention.

- Iterating: Iterators provided by ConcurrentHashMap are weakly consistent, meaning they reflect the state of the map at the time of their creation and may not show the most recent changes made during iteration.

- Performance: Provides better performance in highly concurrent environments due to its segmented structure and fine-grained locking mechanism.

- Scalability: It scales well under high contention scenarios because different threads can access different segments concurrently.

2. SynchronizedMap (Collections.synchronizedMap()):

- Synchronization: It wraps a standard `ap` implementation (like `HashMap`, `TreeMap`) with synchronized methods, ensuring that only one thread can access the map at a time.

- Locking: Utilizes a single lock for the entire map, which means that concurrent read operations are possible but not concurrent write operations.

- Iterating: Iterators provided by synchronized maps are fail-fast and will throw a `ConcurrentModificationException` if the map is modified while iterating.

- Performance: Can suffer performance-wise in high-concurrency scenarios due to its coarse-grained locking mechanism, where all operations share the same lock.

- Scalability: Might face contention issues under high load due to the single lock for the entire map.

In summary, ConcurrentHashMap is more suitable for high-concurrency scenarios where both read and write operations are frequent and concurrent, while synchronizedMap provides basic thread safety by wrapping a standard map but might have performance drawbacks in highly concurrent environments due to its single lock.

**12)How can you avoid ConcurrentModificationException? what are fail-fast and fail-safe Collections available in java?**

ConcurrentHashMap and CopyOnWriteArrayList are the two fail-safe collections.rest all are fail-fast.

**13)How ConcurrentHashMap and CopyOnWriteArrayList avoid ConcurrentModificationException?**

Both `ConcurrentHashMap` and `CopyOnWriteArrayList` provide mechanisms to avoid `ConcurrentModificationException` during iteration, but they use different approaches:

1. ConcurrentHashMap:

- Iterators provided by `ConcurrentHashMap` are weakly consistent. They don't throw `ConcurrentModificationException` even if the map is modified during iteration.

- The `ConcurrentHashMap` achieves this by using internal structures that allow the iterators to work on the snapshot of the map at the time of their creation. They don't rely on a single global lock but use different mechanisms to provide thread safety.

- Modifications made after the iterator's creation might or might not be visible to the iterator, but the iterator doesn't throw exceptions due to concurrent modifications.

2. CopyOnWriteArrayList:

- `CopyOnWriteArrayList` maintains a separate copy of the underlying array whenever modifications are made (add, set, remove operations). It uses an "immutable snapshot" approach.

- When you perform a modification operation on a `CopyOnWriteArrayList`, it creates a new copy of the internal array and applies the modification to this new copy. Iterators on the original array continue to work on the old copy they were created with.

- Iterators obtained from a `CopyOnWriteArrayList` will not throw `ConcurrentModificationException` because they work on a snapshot of the array that doesn't change during iteration.

Both classes utilize different strategies to handle concurrent modifications. `ConcurrentHashMap` allows modifications and provides a weakly consistent view during iteration, whereas `CopyOnWriteArrayList` ensures that iterators work on an immutable snapshot of the list, preventing concurrent modifications from affecting the ongoing iteration.

**13)Garbage Collection process**

It is process in which releasing of memory will be excuted which is held by unreachable objects

**14)How many class loaders are available in java.what are those.**

In Java, there are primarily three built-in class loaders provided by the Java ClassLoader hierarchy:

1. Bootstrap Class Loader (Primordial Class Loader):

- It's the parent of all other class loaders in Java.

- Responsible for loading core Java classes located in the `<JAVA\_HOME>/lib` directory.

- Implemented in native code and is not represented by a specific Java class.

2. Extension Class Loader (Platform Class Loader):

- Child of the Bootstrap Class Loader.

- Loads classes from the Java extension directories, usually `<JAVA\_HOME>/lib/ext`.

- It loads classes required by the JVM extensions.

3. System Class Loader (Application Class Loader):

- Child of the Extension Class Loader.

- Loads classes from the classpath specified by the `java.class.path` system property.

- Responsible for loading application-specific classes.

Additionally, developers can create their custom class loaders by extending the `java.lang.ClassLoader` class. Custom class loaders are often used in scenarios where specific loading behavior or isolation is needed, such as loading classes from network locations, databases, or custom resources.

**15)A general java program execution. what exactly compiler will do in a program execution?**

1. Compilation:

- Source Code: You start with a Java source code file, typically saved with a `.java` extension.

- Compilation Process: The Java compiler (`javac`) translates the human-readable source code into bytecode, a platform-independent intermediate representation of the code.

- Bytecode: The resulting bytecode is saved in `.class` files. These contain instructions that the Java Virtual Machine (JVM) can understand and execute.

2. Class Loading:

- Loading: During program execution, the JVM's Class Loader loads the required `.class` files into memory.

- Linking (Verification, Preparation, Resolution): After loading, the JVM performs linking tasks like verification (ensuring bytecode adheres to Java language rules), preparation (allocating memory for static fields and initializing them with default values), and resolution (replacing symbolic references with direct references).

3. Execution:

- JIT Compilation: The JVM may employ Just-In-Time (JIT) compilation to further optimize the bytecode. This involves translating parts of the bytecode into native machine code at runtime for improved performance.

- Interpreter: Initially, the JVM may interpret the bytecode to execute the instructions. As the program runs and certain parts are identified as hotspots, the JIT compiler optimizes and generates native machine code for these portions, replacing the interpreted code.

4. Runtime Operations:

- Memory Allocation: The JVM manages memory dynamically, allocating memory for objects in the heap and other data structures.

- Garbage Collection: As the program executes, the JVM performs garbage collection to reclaim memory occupied by objects that are no longer reachable, thereby ensuring efficient memory usage

5. Execution Termination:

- Program Completion: The program continues to execute until it completes its tasks or encounters an exception or error that causes termination.

- Cleanup: Upon completion or termination, resources are released, and the JVM shuts down.

In essence, the compiler's primary role is to translate human-readable Java source code into platform-independent bytecode. The JVM then manages the execution of this bytecode, handling various tasks such as class loading, linking, bytecode interpretation, JIT compilation, memory management, and more to execute the Java program.

**15)What is native machine code?**

Native machine code refers to the low-level instructions that a specific computer's processor (CPU) directly executes. It's the set of instructions that the CPU understands and executes without requiring any further translation or interpretation.

**16)What is Externalization in java?which methods you will use to achieve it?**

In Java, externalization refers to a mechanism that allows a class to explicitly control how its instances are serialized and deserialized. Unlike serialization (handled by `Serializable` interface), which provides automatic serialization and deserialization, externalization (`Externalizable` interface) gives the class more control over the serialization process.

To achieve externalization in Java, you need to implement the `Externalizable` interface and define specific methods:

1. Externalizable Interface:

- This interface doesn't have any methods but serves as a marker interface.

- To use externalization, your class needs to implement this interface.

2. Define `writeExternal` and `readExternal` Methods:

- `writeExternal(ObjectOutput out)`: This method is invoked during serialization. You define what data from the object needs to be written to the output stream (`ObjectOutput`).

- `readExternal(ObjectInput in)`: This method is invoked during deserialization. It reads the data written during serialization and reconstructs the object.



In this example, `MyClass` implements the `Externalizable` interface and provides implementations for `writeExternal` and `readExternal` methods. These methods explicitly define how the object's state is written to and read from the stream during serialization and deserialization.

Remember, when using externalization, you have to manually handle the serialization and deserialization process for the object's state. This provides more control but requires explicit coding for handling the serialization and deserialization operations.

**17)what is Serialization**

Serialization in Java refers to the process of converting an object's state into a format that can be easily stored, transmitted, or reconstructed at a later time or in a different environment. It allows objects to be converted into a sequence of bytes that can be written to a file, sent over a network, or stored in a database.

Key points about Serialization:

1. Object State Conversion: Serialization involves converting the state of an object (its fields and values) into a stream of bytes.

2. Serializable Interface: To enable serialization, a class must implement the `java.io.Serializable` marker interface. This interface doesn't have any methods; it acts as a marker to indicate that the objects of this class can be serialized.

3. ObjectOutputStream and ObjectInputStream: In Java, serialization is achieved using classes like `ObjectOutputStream` and `ObjectInputStream`. The `ObjectOutputStream` writes the object's state to an output stream, while the `ObjectInputStream` reads the serialized data from an input stream and reconstructs the object.

4. Transient Fields: Fields in a class that are marked as `transient` are excluded from serialization. This is useful for excluding certain fields (e.g., caching variables or non-serializable resources) from being serialized.

5. Versioning and Compatibility: Versioning becomes important when dealing with serialized objects, especially when the object's structure or fields change over time. Special handling is required to maintain backward and forward compatibility when reading serialized objects.



Serialization is used in various scenarios, such as storing object state persistently, sending objects across a network, or implementing caching mechanisms. It allows objects to be converted into a portable format that can be reconstructed back into objects when needed.

**17)What is transient in java?**

In Java, the `transient` keyword is used as a modifier for class variables to indicate that they should not be serialized when the object they belong to is serialized using Java's built-in serialization mechanism.

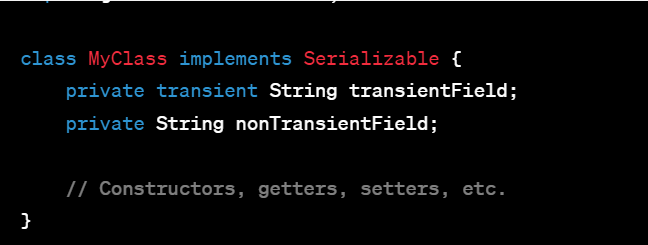
When an object is serialized, all of its non-transient fields are converted into a stream of bytes that represent the object's state. However, marking a field as `transient` tells the Java serialization mechanism to exclude that specific field from the serialization process.

Common reasons to use `transient`:

1. Security: Certain fields might contain sensitive information that should not be persisted or transmitted, such as passwords or encryption keys. Marking these fields as `transient` ensures they aren't serialized, helping maintain security.

2. Derived or Calculated Fields: Fields that can be recalculated or derived from other data in the object need not be stored during serialization. Marking such fields as `transient` avoids unnecessary serialization and saves space.

3. Non-Serializable Objects: Fields referencing non-serializable objects (objects that don't implement `Serializable`) need to be marked as `transient` to avoid serialization errors. For instance, fields referencing network connections or threads.



In this example, `transientField` is marked as `transient`, so when an object of `MyClass` is serialized, the value of `transientField` will not be included in the serialized stream. However, `nonTransientField` will be serialized as usual because it's not marked as `transient`.

Remember, when an object is deserialized, transient fields will be initialized to their default values (null for objects, 0 for numeric types) since their values were not serialized. It's essential to handle such fields appropriately after deserialization if their values are crucial for the object's functionality.

**18)Java 8 features.what is lambda expression?what is Functional interface in java8?**

In Java 8, lambda expressions and functional interfaces were introduced as key features to facilitate functional programming concepts within the language.

### Lambda Expression:

A lambda expression is a concise way to represent an anonymous function—a function without a name that can be passed around as a parameter to other methods or stored in variables. It enables treating functionality as a method argument or code as data. Lambda expressions provide a way to write more readable and maintainable code by reducing boilerplate code.

Syntax of a lambda expression:

(parameter\_list) -> { body }

// Example of a lambda expression

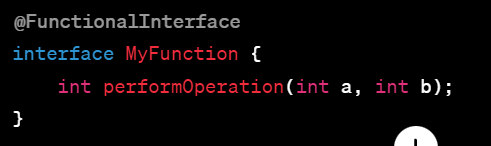
Function<Integer, Integer> square = (x) -> x \* x;

int result = square.apply(5); // result will be 25

### Functional Interface:

A functional interface in Java is an interface that contains only a single abstract method. Functional interfaces serve as the basis for using lambda expressions because they provide a target type for lambda expressions and method references.

The `@FunctionalInterface` annotation (though optional) is used to ensure that an interface qualifies as a functional interface. It helps prevent accidental addition of multiple abstract methods, which would break the single abstract method contract required for lambda expressions.

```

Functional interfaces can also have default methods (methods with implementation) and static methods without violating the single abstract method rule.

Lambda expressions are often used in combination with functional interfaces, allowing concise representation of behavior to be passed around in code. They enable a more functional style of programming in Java, making it easier to write cleaner, more modular, and expressive code.

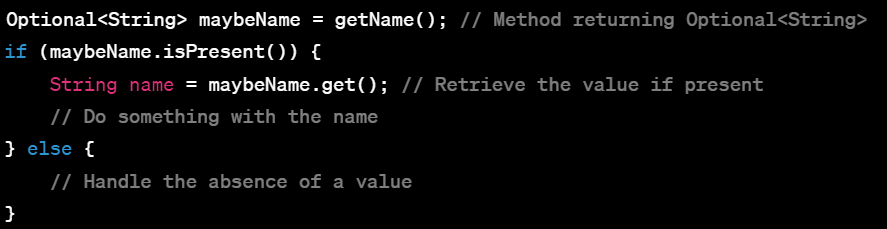
**19)What are Optional class usages?**

The `Optional` class in Java is introduced in Java 8 as a container object used to represent a value that may or may not be present. It helps to avoid null checks and to handle scenarios where a value might be absent, reducing the risk of null pointer exceptions.

### Usages of the `Optional` class:

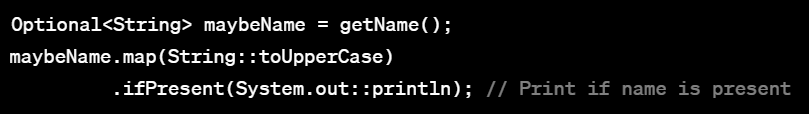
1. Avoiding Null Pointer Exceptions:

Instead of directly returning null, methods can return an `Optional` object, indicating that a value might be absent. This encourages developers to explicitly handle the absence of a value rather than relying on null checks.



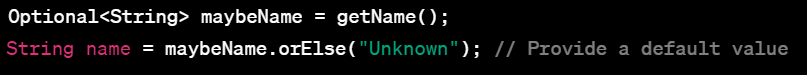
2. Chaining Operations:

`Optional` provides methods to chain operations safely, such as `map`, `flatMap`, `filter`, etc., allowing you to perform transformations or checks on the value if it is present.



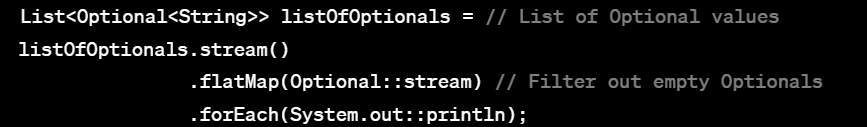
3. Default Values:

Using `orElse` or `orElseGet`, you can provide default values when the optional value is absent.



4. Handling with Streams:

`Optional` integrates well with Java streams, allowing you to process values within streams that may or may not be present.



Using `Optional` encourages a more explicit and safer approach to handling potentially absent values, making the code more readable and robust by forcing developers to handle the absence of a value explicitly rather than relying on unchecked null references. However, it's essential to use `Optional` judiciously, especially when it comes to API design, to ensure it brings clarity and doesn't introduce unnecessary complexity.

**20)Differences between map() and flatmap() in java8**

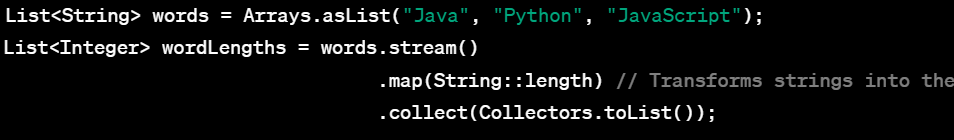
In Java 8, both `map()` and `flatMap()` are methods commonly used in functional programming, particularly when working with streams or with certain types like Optional or Stream.

### `map()` Method:

- Usage: The `map()` method is used to transform each element of a stream or a similar data structure by applying a function to it.

- Functionality: It takes a Function as an argument, applies this function to each element in the stream, and returns a new stream containing the transformed elements.

- Output Type: The output of `map()` is a one-to-one mapping. If applied to a Stream of elements, it generates a Stream of transformed elements of the same type.

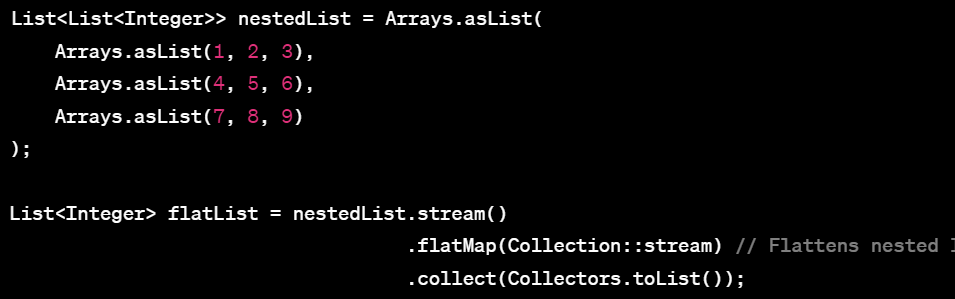


### `flatMap()` Method:

- Usage: The `flatMap()` method is used to flatten nested streams or collections into a single stream.

- Functionality: It takes a function that returns a stream or collection as an argument, applies this function to each element in the stream, and then flattens the resulting streams or collections into a single stream.

- Output Type: The output of `flatMap()` can be a one-to-many or many-to-many mapping. It "flattens" the resulting streams or collections into a single stream.



### Key Differences:

1. Transformation:

- `map()` applies a function to each element, transforming it into another element of the same type.

- `flatMap()` applies a function that returns a stream or collection for each element, and flattens the resulting streams or collections into a single stream.

2. Output Type:

- `map()` produces a one-to-one mapping from input elements to output elements.

- `flatMap()` can produce a one-to-many or many-to-many mapping, flattening nested streams or collections into a single stream.

Understanding the differences between `map()` and `flatMap()` is crucial for choosing the appropriate method based on the desired output and structure of the data being processed.

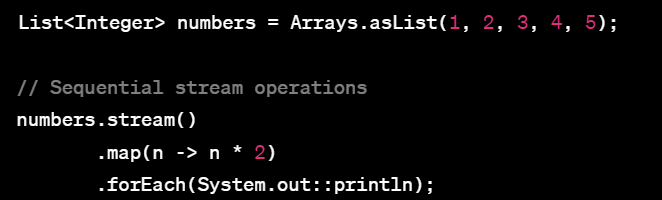
**21)what are parallel stream and sequential streams in java8?**

In Java 8, streams provide a powerful way to process collections of data in a functional style. Both parallel streams and sequential streams are mechanisms provided by the Stream API for performing operations on elements of a stream.

### Sequential Stream:

- Default Behavior: When you create a stream using the `stream()` method on a collection or use stream operations directly, it creates a sequential stream by default.

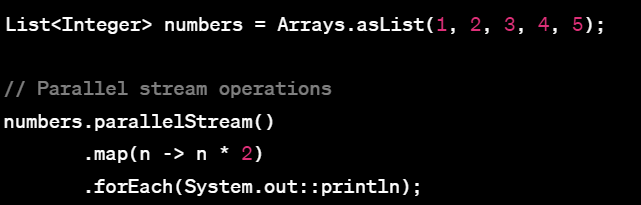
- Processing Order: Operations on a sequential stream are performed sequentially, one after the other, on a single thread.



### Parallel Stream:

- Creation: You can create a parallel stream by calling the `parallelStream()` method on a collection or converting an existing sequential stream to a parallel stream using `parallel()` method.

- Processing Order: Operations on a parallel stream are executed concurrently using multiple threads. The Stream API internally handles the parallelization of operations.



### Key Differences:

1. Execution:

- Sequential streams process elements sequentially on a single thread.

- Parallel streams process elements concurrently using multiple threads.

2. Performance:

- Parallel streams can potentially offer improved performance, especially when processing large amounts of data, by leveraging multiple cores of the CPU. However, this depends on various factors like the size of the data, nature of operations, and available computational resources. It's not always guaranteed that using parallel streams will be faster.

3. Thread Safety:

- Operations on sequential streams are inherently thread-safe because they are executed sequentially on a single thread.

- Operations on parallel streams might require additional care to ensure thread safety, especially when shared mutable state is involved, to prevent concurrent modification issues.

The choice between using sequential or parallel streams depends on factors such as the size of the data, the nature of the operations, and the hardware resources available. While parallel streams can offer performance benefits in certain scenarios, they might introduce overhead and synchronization costs, so it's essential to evaluate and test their performance for specific use cases.

**22)what are intermediate operations and terminal operations in java8?**

In Java 8 streams, operations can be categorized into two main types based on their behavior and when they are executed: intermediate operations and terminal operations.

### Intermediate Operations:

- Behavior: Intermediate operations are operations that are performed on a stream and return a new stream.

- Characteristics:

- They are typically chained together to form a pipeline of operations.

- Intermediate operations are lazy, meaning they do not start processing elements until a terminal operation is invoked.

- Examples of Intermediate Operations:

- `filter(Predicate)`: Filters elements based on a condition.

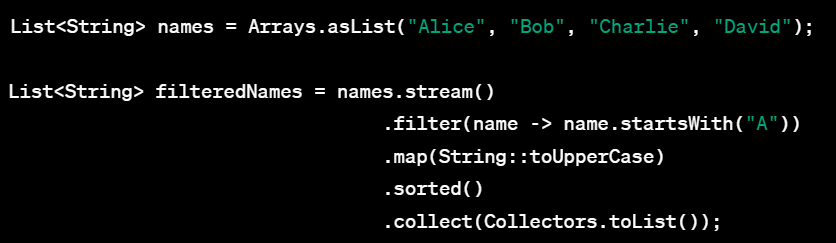
- `map(Function)`: Transforms elements using a given function.

- `sorted()`: Sorts the elements of the stream.

- `distinct()`: Removes duplicate elements from the stream.

- `limit(long)`: Limits the number of elements in the stream.

- `skip(long)`: Skips a specified number of elements in the stream.



### Terminal Operations:

- Behavior: Terminal operations are operations that trigger the processing of elements and produce a result or a side effect.

- Characteristics:

- They are responsible for closing the stream and triggering the evaluation of the entire stream pipeline.

- After a terminal operation is applied, the stream is consumed and cannot be reused.

- Examples of Terminal Operations:

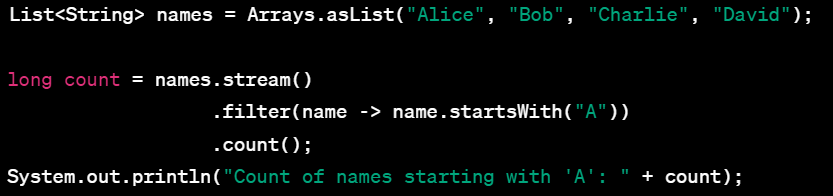
- `forEach(Consumer)`: Performs an action for each element in the stream.

- `collect(Collector)`: Accumulates elements into a collection or a summary result.

- `count()`: Returns the count of elements in the stream.

- `reduce()`: Performs a reduction on the elements.

- `anyMatch(Predicate)`, `allMatch(Predicate)`, `noneMatch(Predicate)`: Check if any, all, or none of the elements match a given condition.



Understanding the distinction between intermediate and terminal operations is crucial when working with streams in Java. Intermediate operations allow you to build a stream pipeline by transforming, filtering, or manipulating elements, while terminal operations are required to trigger the actual processing of the stream and produce a result.

**23)what are the differences between future() and completableFuture in java8?**

In Java 8, both `Future` and `CompletableFuture` are abstractions used for asynchronous programming and handling results that will be available in the future. However, they differ in several aspects, especially in terms of capabilities and flexibility.

### Future:

- Basic Asynchronous Result Holder: `Future` represents a result that may not be available yet but will be available in the future.

- Limited Capabilities: With `Future`, you can perform basic operations like checking if the computation is done, canceling the computation, and getting the result (blocking if necessary) or handling exceptions.

- Blocking Operations: It provides `get()` method for fetching the result, but this method is blocking, meaning it will wait until the result is available or the operation is completed.

- Limited Composition: It has limited support for combining multiple asynchronous computations or handling errors and timeouts.

- Introduced in Java 5: `Future` was introduced in Java 5 as a way to represent the result of an asynchronous computation.

### CompletableFuture:

- Enhanced and Extensible Future: `CompletableFuture` is an extension of `Future` and provides more powerful capabilities for asynchronous programming.

- Combination and Chaining: `CompletableFuture` supports composition operations like `thenApply()`, `thenCombine()`, `thenCompose()`, etc., which allow chaining of asynchronous operations and combining results without blocking.

- Non-Blocking Methods: It offers non-blocking methods for combining and handling results. For example, `thenApply()` and `thenCompose()` allow you to apply functions to the result asynchronously.

- Exception Handling: Provides better exception handling and timeout mechanisms using `exceptionally()` and `completeExceptionally()`, allowing more control over error handling.

- Introduced in Java 8: `CompletableFuture` was introduced in Java 8 as part of the enhanced concurrency features.

### Differences Summary:

- `CompletableFuture` is an enhancement of `Future`, providing more capabilities for asynchronous programming, composition, and non-blocking operations.

- `Future` offers basic asynchronous result handling with limited composition and blocking operations.

- `CompletableFuture` allows better composition of asynchronous tasks, non-blocking operations, and enhanced error handling compared to `Future`.

When working with asynchronous computations in Java, especially in modern applications requiring complex asynchronous workflows, `CompletableFuture` provides more flexibility and power in handling asynchronous tasks and composing multiple computations compared to the basic capabilities offered by `Future`.

**24)Is @FunctionalInterface annotation mandatory for a functional interface?**

No, the `@FunctionalInterface` annotation is not mandatory for an interface to be considered a functional interface in Java. The presence of the annotation is optional; it's used for explicit declaration and documentation purposes.

### Characteristics of a Functional Interface:

1. Single Abstract Method (SAM): A functional interface is an interface that contains exactly one abstract method.

2. Implicit Contract: The single abstract method establishes an implicit contract, representing a functional contract for lambdas or method references.

3. Usage with Lambdas: Functional interfaces are used in conjunction with lambda expressions or method references to provide implementations for their single abstract method.

### Purpose of `@FunctionalInterface`:

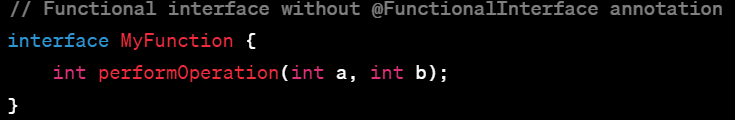
- Documentation: The `@FunctionalInterface` annotation is used to indicate that an interface is intended to be a functional interface.

- Compile-Time Check: It helps in detecting accidental addition of multiple abstract methods, causing a compilation error if the interface does not meet the criteria of having a single abstract method.

- Clarity and Intent: Adding the annotation explicitly states the design intent of the interface, making it clear to developers that this interface is intended for use with lambdas or functional programming.

### Scenario Without `@FunctionalInterface`:

If an interface meets the criteria of having a single abstract method but does not have the `@FunctionalInterface` annotation, it's still considered a functional interface as long as it satisfies the single abstract method requirement.



In the absence of the annotation, the `MyFunction` interface is still a functional interface because it contains only one abstract method (`performOperation()`).

While the annotation isn't mandatory, using it explicitly helps in documenting the intent of the interface and allows the compiler to perform a compile-time check to ensure the interface follows the functional interface contract, providing additional clarity and safety in your codebase.

**25)why default and static methods are introduced in functional interface?what is the use of them?**

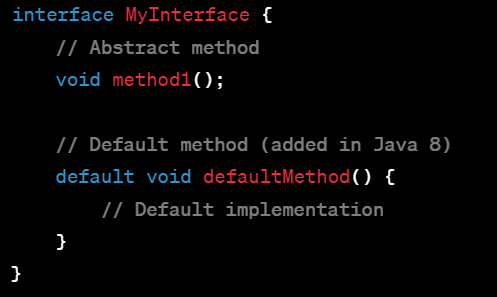
Default and static methods were introduced in Java interfaces, including functional interfaces, to provide additional flexibility and backward compatibility without breaking existing codebases when new methods are added to interfaces.

### Default Methods in Interfaces:

- Backward Compatibility: Default methods allow interfaces to evolve over time without breaking the implementing classes. When a new method is added to an interface, existing classes that implement the interface do not need to provide an implementation for the new method.

- Additional Methods: They enable the addition of new methods with default implementations in interfaces without forcing all implementing classes to redefine these methods.

- Facilitates Interface Evolution: Default methods support the evolution of interfaces by allowing the addition of new methods without breaking existing code.

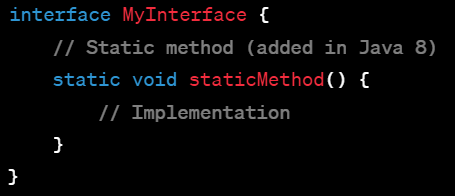


### Static Methods in Interfaces:

- Utility Methods: Static methods in interfaces allow the definition of utility methods that are related to the interface but do not depend on any specific implementation.

- Organizational Purposes: They help in grouping utility methods that are closely related to the interface itself.

- Cannot be Overridden: Static methods in interfaces cannot be overridden by implementing classes. They can only be accessed using the interface name itself.



### Use Cases:

- Interface Evolution: Default methods allow interfaces to evolve by adding new methods without breaking existing implementations.

- Utility Methods: Static methods in interfaces are useful for providing utility methods that are related to the interface but are not tied to a specific implementation.

- Convenience and Organization: Both default and static methods contribute to organizing related methods within interfaces and offer a convenient way to provide method implementations or utility functions.

Default and static methods in interfaces are especially beneficial in scenarios where interfaces need to evolve while maintaining backward compatibility and where common utility methods related to the interface are required. They offer a way to provide implementations or utility functionalities directly within interfaces without relying on abstract implementations in implementing classes.

**26)what is the use of Wrapper classes in java**  
Wrapper classes in Java are used to convert primitive data types into objects. They encapsulate primitive data types and provide utility methods and additional functionalities that are not available with primitives.

**27)what is the difference between StringBuilder and StringBuffer?**

`StringBuilder` and `StringBuffer` are both classes in Java used for manipulating strings, but they differ primarily in terms of their synchronization behavior and performance.

### StringBuilder:

- Non-Synchronized: `StringBuilder` is not thread-safe, meaning it's not synchronized for use in concurrent (multi-threaded) environments.

- Performance: It offers better performance compared to `StringBuffer` because it's not synchronized. However, in a single-threaded environment, the difference might not be noticeable for typical use cases.

- Introduced in Java 5: `StringBuilder` was introduced in Java 5 to provide a more efficient way of manipulating strings without synchronization overhead.

- Usage: It's suitable for use in single-threaded scenarios or situations where synchronization is not a concern.

### StringBuffer:

- Synchronized: `StringBuffer` is thread-safe because its methods are synchronized, ensuring safe use in concurrent environments where multiple threads may access and modify it simultaneously.

- Performance Impact: Due to synchronization, `StringBuffer` can have a performance impact compared to `StringBuilder` when used in a single-threaded environment because of the synchronization overhead.

- Legacy Class: `StringBuffer` existed in Java before `StringBuilder`. It was the primary choice for string manipulation when thread safety was a requirement.

- Usage: It's suitable for scenarios where thread safety is essential, especially in multi-threaded applications.

### Key Differences:

- Synchronization: `StringBuilder` is not synchronized, while `StringBuffer` is synchronized for thread safety.

- Performance: `StringBuilder` generally performs better due to lack of synchronization overhead, especially in single-threaded scenarios.

- Thread Safety: `StringBuilder` is not inherently thread-safe, while `StringBuffer` ensures thread safety by synchronizing its methods.

### Which to Choose?

- If thread safety is not a concern and you're working in a single-threaded environment, use `StringBuilder` for better performance.

- If you need thread safety or you're working in a multi-threaded environment, opt for `StringBuffer`.

In most cases, especially in modern single-threaded applications, `StringBuilder` is preferred due to its better performance. However, in situations where thread safety is required, `StringBuffer` remains a suitable choice.

**29)explain about String constantpool? Heap vs Stack memory differences**

Sure, here's an explanation of the String constant pool and the differences between heap and stack memory:

### String Constant Pool:

- What is it? The String constant pool is a special area in the heap memory where the JVM stores string literals.

- Storage of String Literals: When you create a string using a literal (e.g., `String str = "Hello";`), the string is stored in the constant pool.

- Avoids Duplication: The constant pool avoids duplicate strings by reusing existing string literals. If you create another string with the same value, it refers to the same string in the constant pool.

- Optimization: It helps in memory optimization by conserving memory and reducing the number of String objects created.

### Heap Memory vs. Stack Memory:

- Heap Memory:

- Purpose: Heap memory is used for dynamic memory allocation.

- Storage: Objects and instances of classes are stored in the heap memory.

- Access: It's accessible to all parts of the application.

- Allocation: Memory allocation and deallocation in the heap are handled by the garbage collector.

- Size: It's larger in size compared to the stack.

- Stack Memory:

- Purpose: Stack memory is used for static memory allocation.

- Storage: Local variables, method calls, and references to objects in heap memory are stored in the stack.

- Access: Access is limited to the scope of the method or block where the variables are declared.

- Allocation: Memory allocation and deallocation in the stack are automatic and handled as method calls are made and completed (LIFO).

- Size: It's comparatively smaller in size than heap memory.

### Differences Summary:

- String Constant Pool: It's a specific part of the heap memory that stores string literals.

- Heap Memory: Used for dynamic memory allocation, storing objects, accessible globally.

- Stack Memory: Used for static memory allocation, storing local variables and method calls, limited to method/block scope.

The String constant pool, heap memory, and stack memory serve different purposes in managing memory in Java. Understanding their roles helps in efficient memory management and utilization within Java applications.

**30)String with new keyword and without new keyword when you assign some value?**

In Java, when you create a String object with the `new` keyword, you explicitly create a new instance of the String object in the heap memory. For example:

String str1 = new String("Hello");

Here, `new String("Hello")` creates a new String object in the heap memory.

On the other hand, when you create a String without the `new` keyword using a string literal (e.g., `"Hello"`), it's stored in the String constant pool. For example:

String str2 = "Hello";

Here, `"Hello"` is a string literal that's stored in the String constant pool. When you assign it to `str2`, Java checks if an equivalent string exists in the constant pool. If it does, it reuses that string. If not, it creates a new string in the pool.

### Key Differences:

- Using `new` Keyword: Always creates a new String object in the heap memory, regardless of whether an equivalent string exists in the constant pool.

- Without `new` Keyword (String Literal): Checks the String constant pool for an equivalent string and reuses it if available. If not available, it creates a new string in the pool.

String literals benefit from being immutable and automatically interned in the String constant pool for memory efficiency and optimization. Using the `new` keyword to create strings should generally be avoided unless there's a specific need for separate instances. String literals provide better memory utilization due to their reuse in the constant pool.

**31)method overloading vs method overriding**

Method Overloading and Method Overriding are two fundamental concepts in Java related to methods in classes. They serve different purposes:

### Method Overloading:

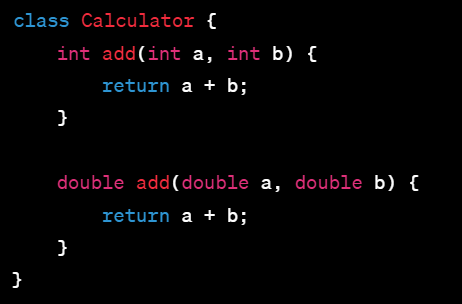
- Definition: Method overloading occurs when multiple methods in the same class have the same name but different parameters (number, type, or order).

- Signature Difference: Overloaded methods must have different method signatures (parameters).

- In the Same Class: Overloading occurs within the same class.

- Compile-Time Polymorphism: The decision of which method to call is made by the compiler based on the method signature.

- Return Type: Overloaded methods can have the same or different return types.



### Method Overriding:

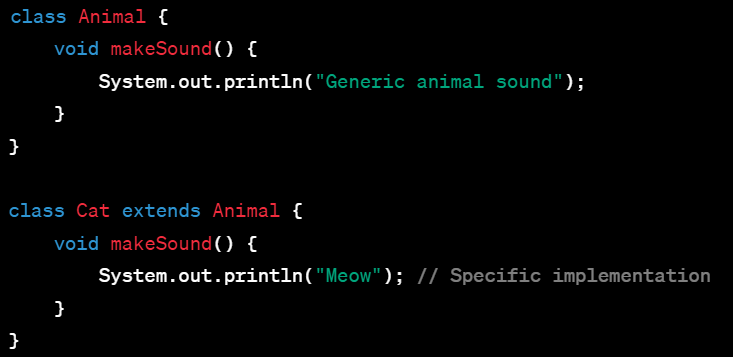
- Definition: Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass with the same signature (method name, parameters, and return type).

- Inheritance Required: Overriding occurs in a subclass inheriting from a superclass.

- Run-Time Polymorphism: The decision of which method to call is made at runtime based on the actual object type.

- Purpose: It's used to provide a specialized implementation of a method in the subclass, changing the behavior of the inherited method.

- Restrictions: The method in the subclass must have the same method signature (name, parameters, and return type) as the method in the superclass.



### Key Differences:

- Purpose: Overloading is used to define multiple methods with the same name but different parameters in the same class. Overriding is used to provide a specific implementation of a method in a subclass.

- Relationship: Overloading occurs in the same class, while overriding occurs in a superclass-subclass relationship.

- Decision Time: Overloaded method resolution is determined at compile time, while overridden method selection is determined at runtime.

Both overloading and overriding are essential concepts in Java, enabling polymorphism and code reusability but serving different purposes within the context of method definitions in classes.

**32)How can you manage Garbage Collection in java?**

In Java, the garbage collection process is handled automatically by the JVM (Java Virtual Machine). However, developers can influence garbage collection behavior and manage memory more efficiently by using certain techniques and best practices:

### 1. Nullify Unused References:

- Set objects to `null` when they are no longer needed. This allows the garbage collector to reclaim memory occupied by these unreferenced objects during its next cycle.

### 2. Use `finalize()` Sparingly:

- Avoid relying on the `finalize()` method to release resources. It's not guaranteed to be called promptly by the garbage collector and can lead to resource leaks.

### 3. Memory Profiling and Analysis:

- Employ tools like VisualVM, Java Mission Control, or Profilers to identify memory leaks or inefficient memory usage in your application. Analyze memory consumption and object creation.

### 4. Manual Invocation of `System.gc()`:

- Though generally not recommended, you can suggest garbage collection by calling `System.gc()`. However, the decision to perform garbage collection remains with the JVM.

### 5. Optimize Object Creation:

- Reuse objects whenever possible instead of creating new ones frequently, especially for objects that are resource-intensive or frequently used.

### 6. Use Weak References, Soft References, and Phantom References:

- These types of references allow more control over how and when objects are reclaimed by the garbage collector, but they require careful handling and understanding of their behavior.

### 7. Adjusting JVM Settings:

- Fine-tune JVM parameters related to heap size, garbage collection algorithms, and memory management based on the specific requirements and behavior of your application.

### 8. Manage External Resources Properly:

- Ensure proper handling of external resources like file handles, database connections, and network connections by closing them explicitly when they are no longer needed.

### 9. Minimize Object Finalization:

- Minimize the usage of finalizers or `finalize()` methods. Instead, explicitly release resources using try-with-resources or finally blocks.

While developers can influence garbage collection behavior to some extent, it's essential to strike a balance between manual memory management and letting the JVM handle it automatically. Often, focusing on writing efficient code and following best practices for memory usage is more effective than manually managing garbage collection.

**Finalize() is deprecated in java 9**

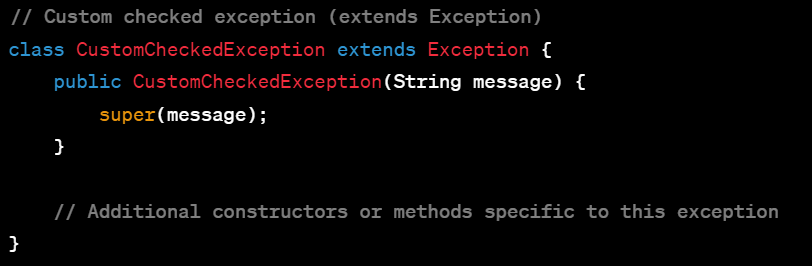
**33)How HashSet works internally?**

Internally uses HashMap, working is same as HashMap

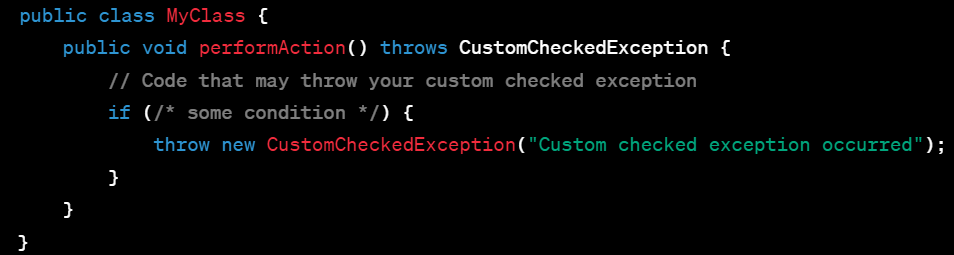
**35)How can you implement Custom checked exceptions and unchecked exception?**

To implement custom checked and unchecked exceptions in Java, you can create your own exception classes that extend either `java.lang.Exception` for checked exceptions or `java.lang.RuntimeException` for unchecked exceptions. Here's how you can create both types:

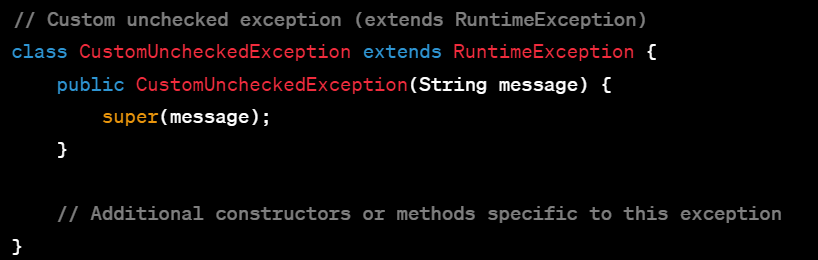
### Custom Checked Exception:



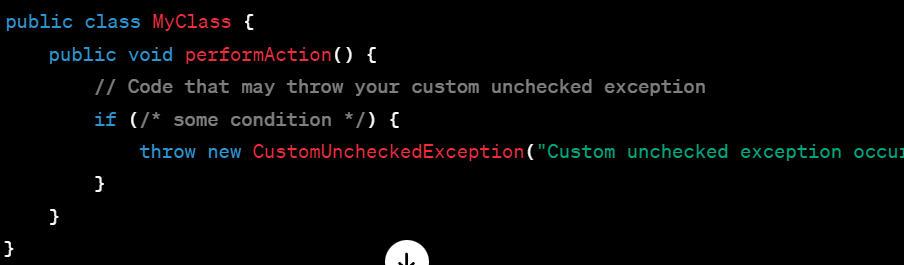
Usage of the custom checked exception:



### Custom Unchecked Exception:



}



### Key Points:

- Checked Exception (`Exception`): Must be declared in the method signature or handled using try-catch blocks.

- Unchecked Exception (`RuntimeException`): Doesn't need to be declared in method signatures and can propagate up the call stack without being caught explicitly.

Custom exceptions allow you to create more meaningful and specific exceptions tailored to your application's needs. Always strive to provide informative messages and handle exceptions appropriately based on your application's logic and requirements.

**36)differnce between throw and throws keyword?**

The `throw` and `throws` keywords in Java are related to exception handling, but they serve different purposes:

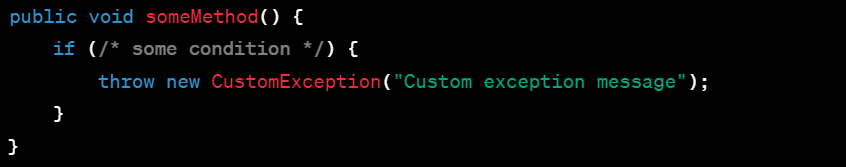
### throw Keyword:

- Usage: The `throw` keyword is used to explicitly throw an exception within a method or block of code.

- Purpose: It is used when a specific condition or situation arises that requires the program to signal an exceptional circumstance.

- Syntax: `throw` followed by an instance of an exception or throwable object.

- Example:



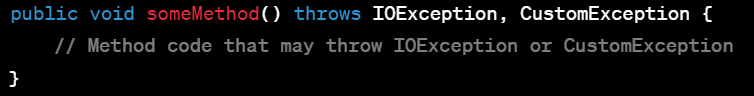
### throws Keyword:

- Usage: The `throws` keyword is used in the method signature to declare that a method might throw certain types of exceptions.

- Purpose: It specifies the exceptions that can be thrown by the method but doesn't handle the exceptions itself; instead, it passes the responsibility to the calling method or caller.

- Syntax: After the method name in the method signature, followed by the list of exception classes separated by commas.

- Example:



### Key Differences:

- `throw` is used to explicitly throw an exception within a method or block of code, indicating an exceptional situation.

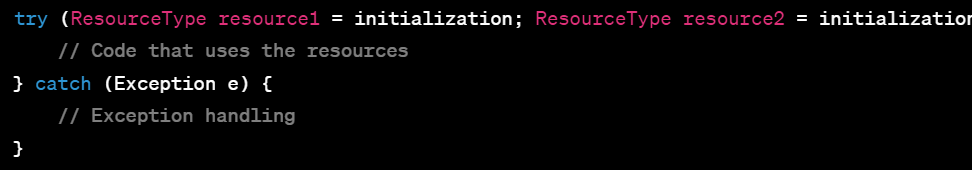
- `throws` is used in the method signature to declare the types of exceptions that a method might throw, transferring the responsibility of handling exceptions to the calling method or caller.

While `throw` is used to initiate an exception, `throws` is used to declare the potential exceptions that a method might propagate to its caller, indicating the need for handling those exceptions at some level in the code hierarchy.

**37)try with resource concepts**

The "try-with-resources" statement in Java is a feature introduced in Java 7 that simplifies the handling of resources that need to be explicitly closed after usage, such as files, database connections, or network connections. It ensures that these resources are closed properly, even if an exception occurs during their usage.

The basic syntax of try-with-resources is as follows:



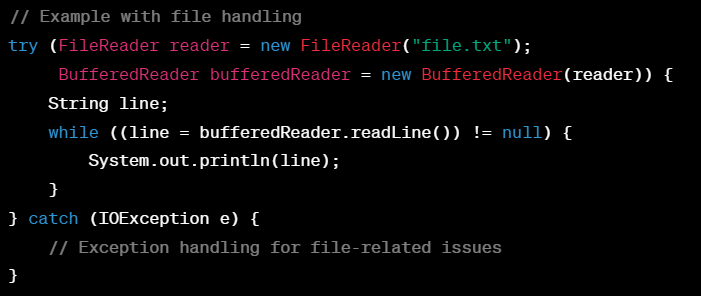
### How it Works:

- The resources declared in the try-with-resources statement are initialized within the parentheses after the `try` keyword.

- The resources must implement the `AutoCloseable` or `Closeable` interface. These interfaces provide the `close()` method for releasing the resources.

- The try-with-resources statement automatically closes the declared resources at the end of the block, whether the code within the block completes normally or an exception occurs.

- The resources are closed in the reverse order of their declaration.



In the above example:

- `FileReader` and `BufferedReader` are initialized within the try-with-resources block.

- Both `FileReader` and `BufferedReader` implement `AutoCloseable`, allowing them to be used in the try-with-resources statement.

- After the execution of the block (either normally or due to an exception), both resources will be closed automatically in the reverse order of their declaration.

Using try-with-resources helps in writing concise and clean code for handling resources that need explicit closure, reduces the chances of resource leaks, and improves code readability by handling resource cleanup automatically.

**38)ClassNotFoundException and NoClassDefinitionFound Error?**

`ClassNotFoundException` and `NoClassDefFoundError` are both related to class loading and occur when the Java Virtual Machine (JVM) encounters issues while trying to load a class. However, they represent different scenarios:

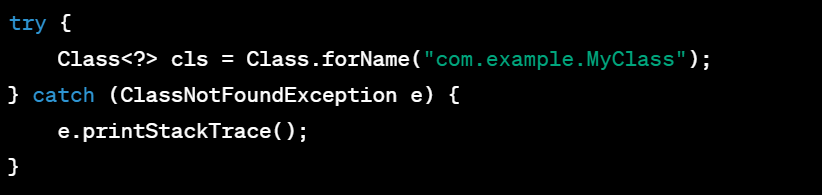
### ClassNotFoundException:

- Scenario: Occurs when the JVM tries to load a class dynamically using `Class.forName()` or `Classloader.loadClass()`, and the specified class with the given name is not found in the classpath.

- Cause: Typically happens at runtime when the application tries to load a class by its name dynamically, but the class is not present in the classpath or is not available during runtime.

- Handling: To resolve `ClassNotFoundException`, ensure that the required class is present in the classpath or bundled with the application's runtime dependencies.

### Example of ClassNotFoundException:



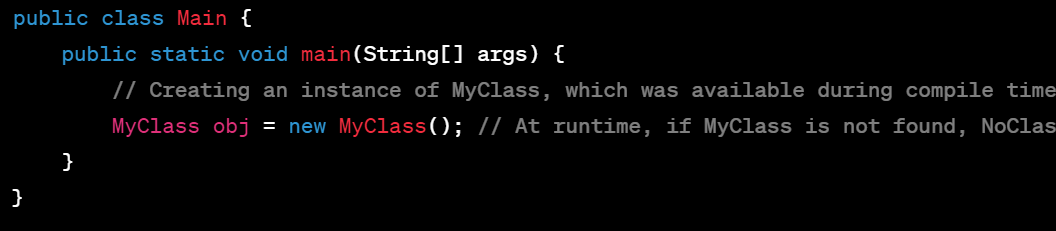
### NoClassDefFoundError:

- Scenario: Occurs when the JVM previously successfully loaded a class during the compilation and while resolving its dependencies, but at runtime, it cannot find the definition of the class.

- Cause: Typically happens when the class was available during compile time but becomes unavailable during runtime due to a missing dependency or a change in the classpath.

- Handling: To resolve `NoClassDefFoundError`, check the classpath, ensure all dependencies are available during runtime, and there are no conflicts or changes causing the class to become inaccessible.

### Example of NoClassDefFoundError:



### Key Difference:

- `ClassNotFoundException` occurs when the JVM tries to load a class dynamically and cannot find the class with the specified name in the classpath.

- `NoClassDefFoundError` occurs when a class that was available during compilation is no longer available during runtime due to missing dependencies or classpath issues.

Both errors indicate issues related to class loading, but their causes and contexts differ based on when and why the class loading failure occurs during the application's execution.

**39)Have you worked on threads? What are thread life cycle status?**

In Java, a thread goes through various states in its lifecycle, which represent different stages of its existence and execution. The thread states are as follows:

### 1. New:

- Description: When a thread is created but not yet started with the `start()` method.

- Methods: The `start()` method transitions the thread from the `New` state to the `Runnable` state.

### 2. Runnable:

- Description: The thread is ready to run and waiting for its turn to be picked for execution by the CPU.

- Methods: When the thread scheduler selects it, the thread moves to the `Running` state.

### 3. Running:

- Description: The thread is executing its task or code logic.

- Methods: The thread can transition back to the `Runnable` state after its time slice (quantum) is completed or voluntarily yields the CPU.

### 4. Waiting:

- Description: The thread is waiting indefinitely for another thread's signal or notification to resume execution.

- Methods: It can transition back to `Runnable` after receiving a notification or signal.

### 5. Timed Waiting:

- Description: The thread is waiting for a specified amount of time.

- Methods: It can transition back to `Runnable` after the specified time elapses or upon receiving a notification.

### 6. Blocked:

- Description: The thread is waiting for a monitor lock to enter or re-enter a synchronized block/method.

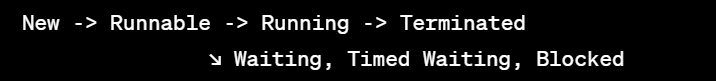
- Methods: It transitions back to `Runnable` when the lock becomes available.

### 7. Terminated:

- Description: The thread has completed its task and has either exited normally or due to an unhandled exception.

- Methods: Once a thread reaches this state, it cannot transition to any other state.

### Diagram of Thread State Transitions:



Understanding the thread lifecycle helps in managing threads efficiently and handling synchronization, waiting, and termination appropriately within multi-threaded applications.

**40)difference between sleep() and wait() methods?**

The `sleep()` and `wait()` methods in Java are used to pause the execution of a thread, but they differ in their purposes, behavior, and where they can be used:

### sleep():

- Belongs to: `Thread` class (`static` method).

- Purpose: Pauses the execution of the current thread for a specified amount of time.

- Usage: Used for introducing a delay or pausing execution in a single-threaded environment.

- Invocation: `Thread.sleep(long millis)` or `Thread.sleep(long millis, int nanos)`.

- Interrupt Handling: When a thread is sleeping and another thread interrupts it, it throws `InterruptedException`.

- Lock Release: Does not release any locks held by the thread while sleeping.

### wait():

- Belongs to: `Object` class.

- Purpose: Pauses the current thread and releases the lock it holds on the object it was called on.

- Usage: Typically used in a multi-threaded environment for synchronization between threads and to coordinate their execution.

- Invocation: `wait()` or `wait(long millis)` with an optional timeout.

- Interrupt Handling: When a thread is waiting and another thread interrupts it, it throws `InterruptedException`.

- Lock Release: Releases the lock on the object it was called on, allowing other threads to acquire the lock and enter synchronized blocks or methods.

### Key Differences:

1. Class Association: `sleep()` is a static method of the `Thread` class, while `wait()` is a method of the `Object` class.

2. Purpose: `sleep()` is used to introduce a delay or pause execution, while `wait()` is used for synchronization and coordination between threads.

3. Lock Release: `sleep()` does not release any locks held by the thread, whereas `wait()` releases the lock on the object, allowing other threads to enter synchronized blocks or methods on that object.

### Example Usage:



In summary, `sleep()` is used for introducing delays or pausing execution, while `wait()` is used for synchronization and coordination between threads by releasing the lock on the object it's called on.

**41)what are deadlocks.what is a deamonthread?thread priority in Threads in java**

### Deadlocks:

- Definition: Deadlock occurs in a multi-threaded environment when two or more threads are blocked forever, each waiting for the other to release a resource or lock.

- Cause: It typically arises when multiple threads acquire locks on resources in a way that they depend on each other's release of resources, resulting in a circular waiting scenario.

- Prevention: Avoiding circular dependencies on locks, using timeouts, or employing a strategy like ordering locks to prevent deadlocks.

### Daemon Threads:

- Definition: Daemon threads are service threads that run in the background and provide services to non-daemon threads. They are considered low priority and are used for tasks that don't require completion before the program terminates.

- Characteristics: Daemon threads are terminated automatically when all non-daemon threads finish execution or when the program exits.

- Example: Garbage collection is performed by a daemon thread in Java.

### Thread Priority:

- Definition: Thread priority is an integer value that determines the scheduling preference of a thread.

- Range: Priorities range from `MIN\_PRIORITY` (1) to `MAX\_PRIORITY` (10) defined in the `Thread` class.

- Influence on Scheduling: Higher priority threads are scheduled more often than lower priority threads, but the exact behavior can depend on the underlying operating system's thread scheduler.

- Note: Relying solely on thread priorities for critical behavior in an application isn't recommended due to platform-dependent behavior and unpredictability.

Understanding these concepts helps in managing threads effectively, preventing deadlocks, utilizing daemon threads for background tasks, and considering thread priorities for scheduling, though thread priorities might not behave consistently across different operating systems.

**42)What are some in-built Functional Interfaces in java8?**

Predicate (which implements test () method internally when you call filter() method on stream()),

Function (apply ()-- method),Consumer(accept()),Supplier(get()).

**43)How can you create a custom Hashmap in java? Is it mandatory to override hashcode() and equals() methods in Custom class?**

Yes mandatory.but please read about equals

Creating a custom HashMap in Java involves implementing a data structure that mimics the basic functionality of a HashMap. While you can create your own version for learning purposes, in practical scenarios, using the built-in `java.util.HashMap` is recommended due to its efficiency and well-tested implementation.

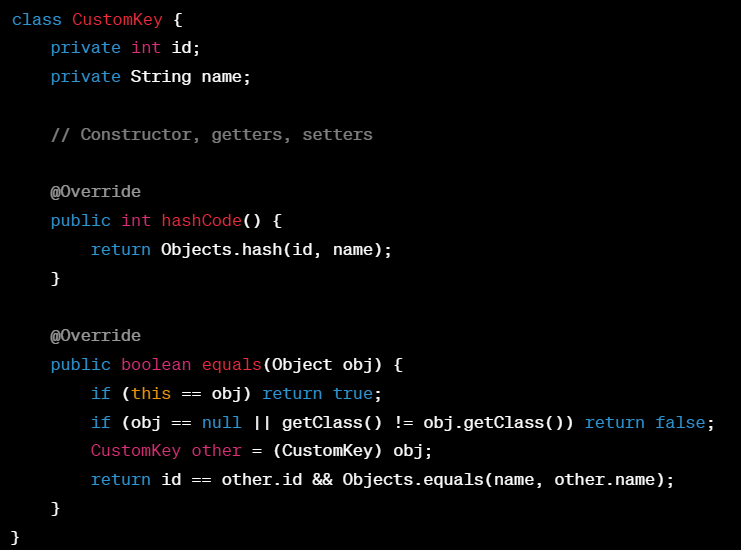
Regarding the second part of your question:

### Override `hashCode()` and `equals()` in Custom Class?

- Mandatory for HashMap Usage: If you intend to use instances of a custom class as keys in a HashMap or any other hash-based collection (`HashSet`, `Hashtable`, etc.), it's crucial to override both `hashCode()` and `equals()` methods.

- Reason: These methods are used by HashMap to determine the hash bucket where an object will be stored and to check for equality among keys.

### Example:



- `hashCode()` should return the same value for objects that are considered equal according to `equals()`.

- `equals()` should compare the contents of the objects to determine equality.

Not implementing these methods properly can lead to unexpected behavior in hash-based collections, where keys might not be found as expected or might lead to incorrect retrieval of values associated with those keys.

So, while creating a custom HashMap, it's advisable to ensure that the objects used as keys override `hashCode()` and `equals()` as per the contract defined in the `Object` class.

**44)Explain Stream API utilities methods?**

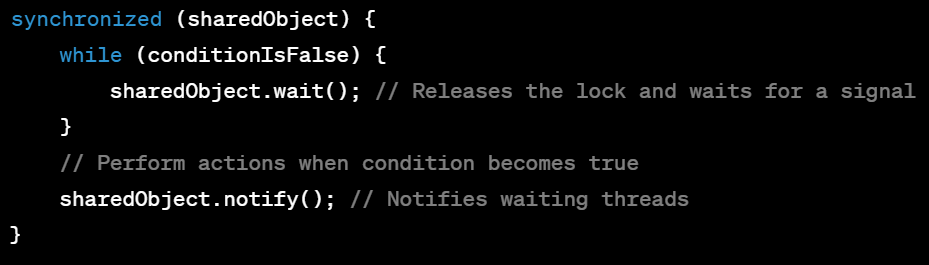
stream(),map(),flatmap(),filter(),distinct(),sorted(),limit(),reduce(),mapToInt(),mapToDouble(),peek(),skip(),forEach(),collect(),count(),anyMatch(),allMatch(),findFirst(),findAny() etc.

**45)What is inter-thread communication in java?**

Inter-thread communication in Java refers to the mechanism by which threads synchronize and communicate with each other for coordinating their activities or sharing data in a multi-threaded environment. This communication is essential for threads to work collaboratively and efficiently.

### Techniques for Inter-thread Communication:

1. Shared Objects/Monitors: Threads can communicate by sharing objects (resources) and using synchronized blocks or methods to control access to these shared resources. `wait()`, `notify()`, and `notifyAll()` methods are used within synchronized blocks for communication between threads sharing the same monitor.



2. Signaling Mechanism: Threads can use signaling mechanisms (`wait()`, `notify()`, `notifyAll()`) provided by the `Object` class to indicate when certain conditions are met or when resources are available for consumption by other threads.

3. Blocking Queues: Classes like `java.util.concurrent.BlockingQueue` provide a higher-level abstraction for inter-thread communication, allowing threads to wait for data to be available or space to become available in a shared queue.

### Key Methods for Inter-thread Communication:

- `wait()`: Causes the current thread to wait until another thread invokes `notify()` or `notifyAll()` for the same object.

- `notify()`: Wakes up a single thread that is waiting on the object's monitor.

- `notifyAll()`: Wakes up all threads that are waiting on the object's monitor.

### Example Scenario:

- Producer-Consumer Problem: Multiple producer threads produce data, which is consumed by multiple consumer threads. Threads coordinate and synchronize access to a shared buffer or queue using synchronization and inter-thread communication techniques.

Inter-thread communication enables threads to synchronize their activities, share resources/data, and coordinate their execution to achieve tasks cooperatively within a multi-threaded environment, ensuring proper synchronization and avoiding race conditions or inconsistencies.

**48)A try block can be followed by finally block?is catch block mandatory to follow try always?**

Yes. Not mandatory if it is followed by a finally and if try is with try-with-resouce.

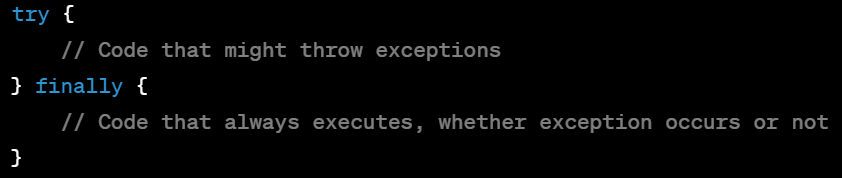
without these two catch is needed to handle exceptions

Yes, a `try` block in Java can be followed by a `finally` block. The `finally` block, if present, executes whether an exception is thrown or not, allowing you to perform cleanup or resource releasing operations that should be executed regardless of whether an exception occurred or not within the `try` block.

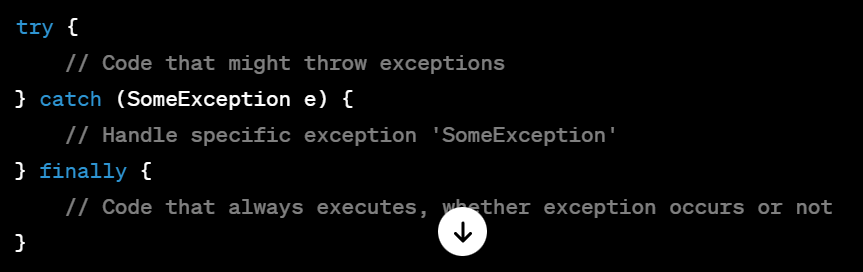
Regarding the `catch` block:

- `catch` Block Usage: The `catch` block is not mandatory after a `try` block if a `finally` block is present. However, if you want to handle exceptions that may occur within the `try` block, you should include one or more `catch` blocks after the `try` block to catch and handle specific exceptions or perform actions based on different types of exceptions.

### Syntax with try-finally:



### Syntax with try-catch-finally:



### Key Points:

- `finally` block is often used for resource cleanup or releasing operations (e.g., closing files, releasing connections).

- Including `catch` blocks is optional after a `try` block if you only want to execute the `finally` block for cleanup purposes without handling specific exceptions.

- When both `catch` and `finally` blocks are present, the `catch` block is used to handle exceptions, and the `finally` block is used for cleanup, executing after the `catch` block.

Including `finally` ensures that critical cleanup operations occur regardless of whether an exception was caught or not, which is essential for maintaining program integrity in various scenarios.

**50)Exception Handling Scenario when try block with multiple catch blocks:**

When we handling try with multiple catch blocks, if try block is throwing an ArithmaticException, if first catch block is handling that ArithmaticException and second catch is Exception – then first one only will be executed. If it is reversed order compile time error will be occurred.

Proper Order:

============

try {

int I = 1/0;

// Some code that may throw an ArithmeticException

} catch (ArithmeticException ex) {

// Code to handle ArithmeticException

} catch (Exception ex) {

// Code to handle any other exception

}

Compile time error order:

========================

try {

int x = 1/0;

or

int x = null;

int result = x;

// Some code that may throw an ArithmeticException

} catch (Exception ex) {

// Code to handle any other exception

} catch (ArithmeticException ex) {

// Code to handle ArithmeticException

}

**51)Can we add StringBuilder object to TreeSet like below?**

yes

TreeSet<StringBuilder> myset = new TreeSet<>();

myset.add(new StringBuilder("vensee"));

**52)How can you create Threads?what happens if you call start() method two times in a single call?**

extending Tread and implementating Runnable interface by ovverriding run() method. IllegalThreadStateException occured.

**53)What is volatile keyword?what is transient keyword in java? what are atomic variables?**

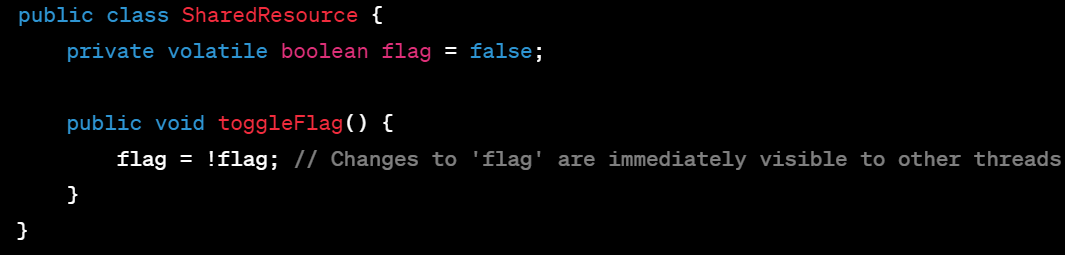
Sure, let's break down each of these concepts:

### 1. `volatile` Keyword:

- Purpose: The `volatile` keyword in Java is used to declare a variable as volatile, ensuring that its value is always read from and written to main memory, not from the thread's local cache.

- Thread Visibility: It guarantees visibility of changes made by one thread to other threads, preventing thread-local caching of variables.

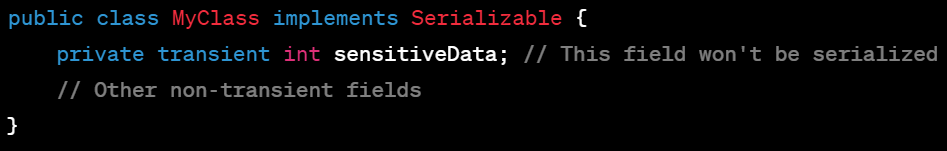
- Usage: Often used in multi-threaded scenarios where a variable is accessed by multiple threads without explicit synchronization.

- 

### 2. `transient` Keyword:

- Purpose: The `transient` keyword in Java is used to indicate that a field should not be serialized when the object is converted into a byte stream (e.g., during object serialization).

- Usage: Fields marked as `transient` are excluded from the serialization process.

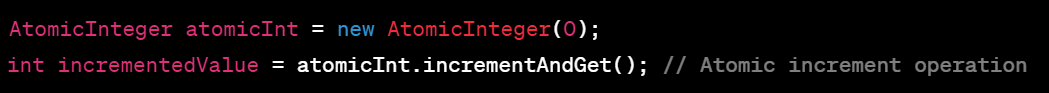


### 3. Atomic Variables:

- Definition: Atomic variables provide operations that can be performed atomically without interference from other threads.

- Usage: Classes like `AtomicInteger`, `AtomicBoolean`, `AtomicLong`, etc., in `java.util.concurrent.atomic` package provide atomic operations like `get`, `set`, `compareAndSet`, etc.

- Purpose: They are used for thread-safe operations on single variables without the need for explicit synchronization.



### Key Points:

- `volatile` ensures visibility of variable changes across threads.

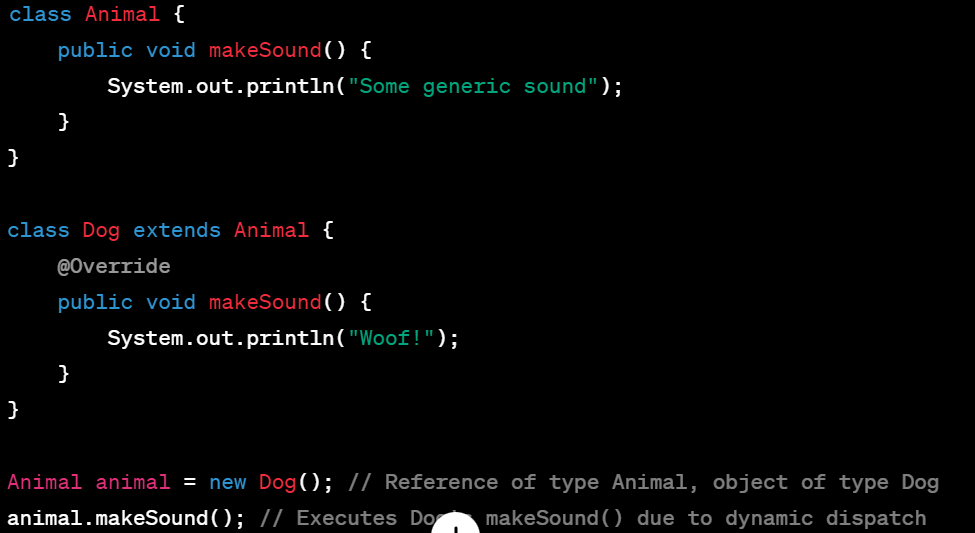
- `transient` prevents fields from being serialized.

- Atomic variables offer thread-safe operations without explicit synchronization, useful in scenarios where single variables are shared across multiple threads.

Understanding these keywords and constructs helps in developing thread-safe and efficient multi-threaded applications while handling serialization and ensuring the proper visibility of shared variables across threads.

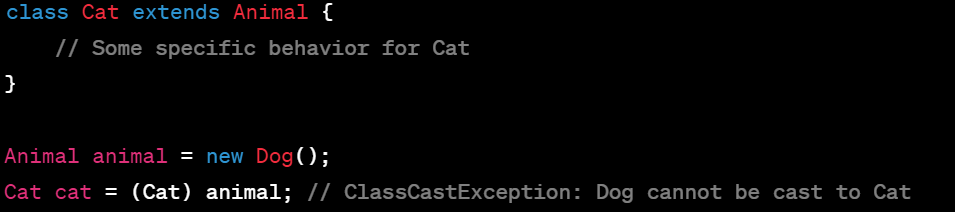
**54)What is dynamic dispatch in java?when can we expect ClassCastException?**

Dynamic dispatch in Java refers to the process where the appropriate method to be executed for an overridden method call is determined at runtime, based on the actual object type rather than the reference type. This mechanism enables polymorphic behavior, allowing a subclass object to be treated as an instance of its superclass.



### Expecting `ClassCastException`:

- A `ClassCastException` occurs when there is an attempt to cast an object to a type that is not compatible with its actual runtime type.



In the above example, `animal` refers to a `Dog` object, and casting it to `Cat` leads to a `ClassCastException` because a `Dog` object cannot be treated as a `Cat`.

`ClassCastException` typically occurs during runtime when incorrect casting is attempted, such as casting an object to a type that's incompatible with its actual runtime type. To avoid this exception, ensure that the object being cast is compatible with the target type, or use `instanceof` to check compatibility before casting.

**55)what is copyConstructor in java?**

A copy constructor in Java is a constructor that creates a new object by copying the values of another object of the same class. It's used to create a new instance with the same state as an existing object, often facilitating the creation of deep copies.

### Characteristics of a Copy Constructor:

- Parameter: It takes an object of the same class as its parameter.

- Creates New Object: It creates a new instance of the class by copying the values from the provided object.

- Customizable: Allows custom logic for copying object state, especially when deep copying is required.



### Key Points:

- The copy constructor allows the creation of a new object with the same state as an existing object.

- It's often used to perform deep copying of an object's state, especially when the class contains references to mutable objects.

- Customizing the copy constructor allows for specific behavior during object copying, enabling manipulation of state or references if needed.

Implementing a copy constructor provides flexibility and control when creating new objects based on the state of existing objects, ensuring that the new instance has the same data without sharing references to the original object's mutable data.

**56)How is composition is better than inheritance?aggregation vs composition**

Composition and inheritance are two fundamental concepts in object-oriented programming, each with its strengths and purposes. Here's a comparison and the advantages of composition over inheritance:

### Composition vs. Inheritance:

### Inheritance:

- Definition: Inheritance establishes an "is-a" relationship between classes, where a subclass inherits properties and behavior from a superclass.

- Advantages:

- Facilitates code reuse by inheriting methods and fields from a superclass.

- Supports polymorphism, allowing subclasses to be used interchangeably with their superclass.

- Challenges:

- Can lead to a rigid and inflexible class hierarchy.

- Increases coupling between classes, making the codebase less maintainable.

- May violate the principle of "favor composition over inheritance" in cases where inheritance isn't the most suitable relationship.

### Composition:

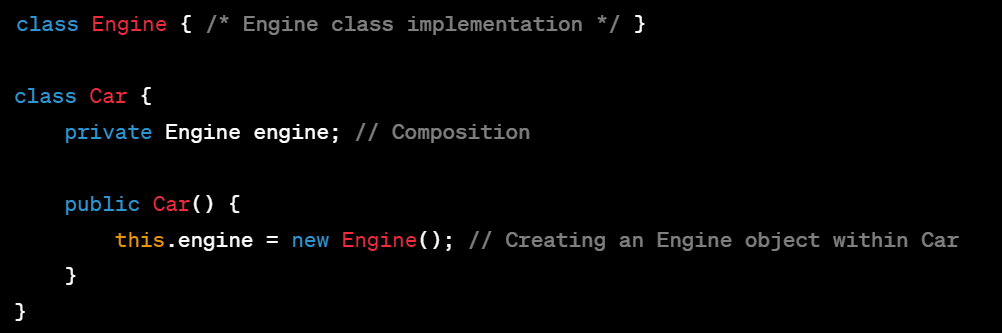
- Definition: Composition establishes a "has-a" relationship between classes, where one class contains an instance of another class as a member field.

- Advantages:

- Promotes code reuse by using objects as building blocks, allowing flexibility in combining different functionalities.

- Enables better encapsulation, as the containing class has control over its composed objects.

- Facilitates easier modification, as changes to composed objects don't affect the containing class or other components.



### Aggregation vs. Composition:

- Composition: Strong form of association where the composed object has a lifecycle controlled by the container. The composed object is part of the container and cannot exist without it.

- Aggregation: Weaker form of association where objects are related but can exist independently. The aggregated object has its own lifecycle and doesn't depend on the container object.

### Advantages of Composition over Inheritance:

1. Flexibility: Composition allows for more flexible and modular design by assembling functionalities rather than building complex class hierarchies.

2. Encapsulation: Encourages better encapsulation and reduced coupling between classes.

3. Easier Modification: Changes to composed objects don't affect the containing class, making the code easier to maintain and modify.

4. Code Reusability: Facilitates code reuse by using smaller, more specialized components that can be combined in different ways.

In summary, while inheritance is powerful for certain scenarios, composition often provides more flexibility, maintainability, and better design possibilities by favoring object composition over class inheritance, especially when designing complex systems or building reusable components.

**57)What is marker interface?name few in java?can we create custom marker interface?**

A marker interface in Java is an interface that doesn't declare any methods or fields but serves as a "marker" or "tag" to convey some information to the compiler or runtime about the class that implements it. It simply marks the implementing classes as having some specific characteristics or capabilities.

### Examples of Marker Interfaces in Java:

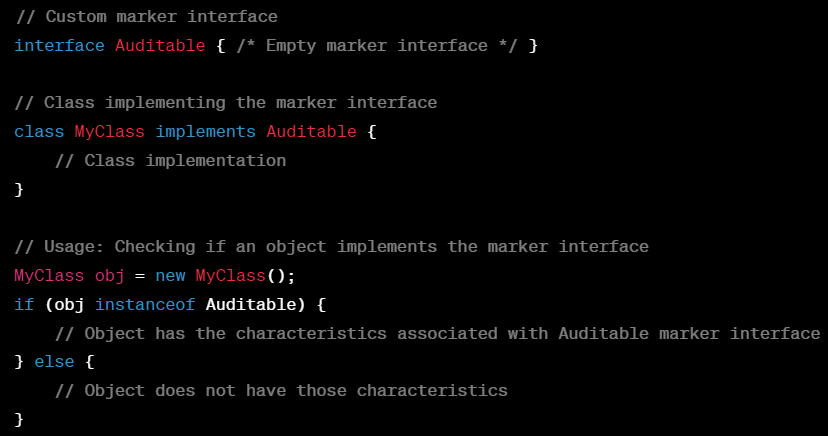
1. `Serializable`: Indicates that a class is serializable and instances of that class can be serialized.

2. `Cloneable`: Indicates that a class allows creating a copy of its instances using `clone()` method.

3. `RandomAccess`: Indicates that a list supports fast random access.

### Creating Custom Marker Interfaces:

Yes, you can create your own custom marker interfaces by defining an interface without any methods. Implementing classes then indicate that they possess certain traits or capabilities associated with that marker interface.



### Advantages of Marker Interfaces:

- Provides a way to categorize or classify classes without adding any additional methods or fields.

- Allows frameworks or APIs to check for specific capabilities or traits in objects.

### Considerations:

- Marker interfaces don't define any behavior; they just serve as a flag for the compiler or runtime.

- With the introduction of annotations in Java, marker interfaces are less commonly used in modern Java development.

While marker interfaces were widely used before annotations became prevalent, they still offer a straightforward way to indicate certain capabilities or traits associated with implementing classes in a Java application.

**58)When you will get InvalidClassException in java?What is the importance of SerialVersionUID?\**

`InvalidClassException` in Java occurs during deserialization when the class definition of a serialized object differs from its definition at the time of serialization. This exception is thrown to indicate that the version of the serialized class is incompatible with the locally available class definition.

### Causes of InvalidClassException:

1. Mismatched SerialVersionUID: The most common cause is a mismatch in the `SerialVersionUID`.

2. Class Structure Changes: Changes in the class structure (like adding/removing fields or methods) after serialization.

3. Different Class Versions: Deserialization of an object from a class version that's not compatible with the current class definition.

### Importance of `SerialVersionUID`:

- `SerialVersionUID` is a unique identifier associated with a serializable class that serves as a version control mechanism during serialization and deserialization.

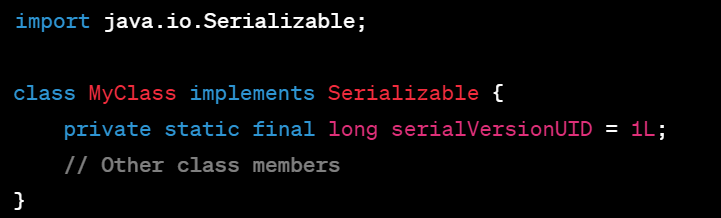
- It's a private static final field declared explicitly within a serializable class.

- If not specified explicitly, Java computes the `SerialVersionUID` based on the class structure, which might change with modifications, potentially leading to compatibility issues.

### Significance of `SerialVersionUID`:

1. Version Control: Ensures that serialized objects are compatible with the class definition during deserialization.

2. Prevents Incompatibility: Helps avoid `InvalidClassException` by allowing the runtime to check whether the serialized and local class definitions match.



### Key Points:

- `InvalidClassException` occurs during deserialization due to incompatible class versions.

- `SerialVersionUID` helps maintain compatibility between serialized and local class versions by acting as a version control identifier.

- It's good practice to declare and manage `SerialVersionUID` explicitly to avoid compatibility issues when serializing and deserializing objects, especially when dealing with persistence or communication between different versions of an application.

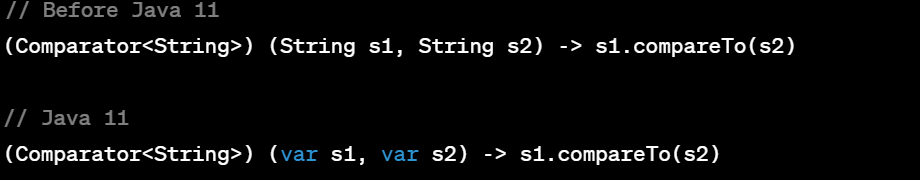
**59)what are some important features added into Java11?**

Java 11 introduced several noteworthy features and improvements. Some of the key additions include:

### 1. Local-Variable Syntax for Lambda Parameters:

- Feature: Allows `var` to be used in lambda expressions.

- Benefit: Increases readability and conciseness in lambda expressions.



### 2. HTTP Client (Standard):

- Feature: Introduction of a standardized HTTP client (HTTP/2 support) in the `java.net.http` package.

- Benefit: Modernizes the existing HTTP API, offering a more flexible and reactive approach for HTTP communication.

### 3. Launch Single-File Source-Code Programs:

- Feature: Ability to run single-file source-code programs without explicitly compiling them.

- Benefit: Simplifies execution of small Java programs by running directly from source files using `java` command.

### 4. Epsilon Garbage Collector:

- Feature: A no-op garbage collector for performance testing and scenarios where no memory reclamation is needed.

- Benefit: Allows performance testing without overhead from a garbage collector.

### 5. Nest-Based Access Control:

- Feature: Introduces the concept of nested classes as a runtime entity.

- Benefit: Enhances access control by allowing private access between enclosing and nested classes.

### 6. Flight Recorder (JFR) and JDK Mission Control (JMC) Open-Sourced:

- Feature: Open-sourcing of Java Flight Recorder (JFR) and JDK Mission Control (JMC).

- Benefit: Enables better monitoring, diagnostics, and profiling of Java applications for a wider range of users.

### 7. `String` Methods:

- Additional methods like `isBlank()`, `lines()`, `repeat(int count)`, `strip()`, `stripLeading()`, and `stripTrailing()` were added to the `String` class, enhancing string manipulation and handling.

These features and improvements in Java 11 focused on enhancing developer productivity, improving performance, and refining existing functionalities in the Java ecosystem.

**60)What is the difference between string literal and string object.how their memroy mechanism will be?which is useful why?**

when you assign some values to a string without new keyword - its a literal-it will be stored in string constant pool and no need to allocate memory in heap

if you create an instance for string with a new keyword its a string object- it will be directly created in heap memory with seprate address location.

**2)All Oops concepts**

**34)what is the order of elements when add them into each Collection type and Map interface subclasses?**

SpringMVC /Springboot:

======================

**1)what are the features/uses of springboot over Spring MVC?**

Spring Boot and Spring MVC are both parts of the larger Spring Framework, but they serve different purposes and offer distinct advantages:

### Spring MVC:

- Purpose: Spring MVC is a web framework built on the Spring Framework, providing components to develop web applications following the MVC (Model-View-Controller) architectural pattern.

- Features:

- Offers robust support for building web applications by providing controllers, views, and models.

- Handles HTTP requests, manages sessions, and supports various view technologies.

- Requires configuration for setting up the application, dependencies, and server deployment.

### Spring Boot:

- Purpose: Spring Boot is a higher-level framework that simplifies and accelerates the process of building and deploying Spring-based applications.

- Features/Advantages over Spring MVC:

1. Auto-configuration: Reduces the need for manual configuration by providing sensible defaults. It configures the application based on dependencies in the classpath, reducing boilerplate code.

2. Embedded Servers: Comes with embedded servlet containers like Tomcat, Jetty, or Undertow, allowing applications to be run as standalone JARs without external server setup.

3. Opinionated Defaults: Provides opinionated defaults for configuration, making it easier to get started with Spring applications.

4. Production-Ready Features: Includes production-ready features like health checks, metrics, security, and externalized configuration.

5. Dependency Management: Simplifies dependency management and reduces version conflicts with Spring Boot Starter POMs.

6. Actuator: Provides monitoring and management endpoints for applications.

7. CLI (Command Line Interface): Offers a CLI tool to bootstrap and develop Spring-based applications quickly.

### Uses of Spring Boot Over Spring MVC:

- Rapid Development: Spring Boot significantly reduces development time by providing defaults and minimizing configuration overhead.

- Simplified Deployment: Applications can be easily packaged as standalone JAR files with embedded servers, simplifying deployment and scaling.

- Reduced Configuration: Avoids manual configuration by employing sensible defaults, allowing developers to focus on application logic rather than setup and configuration.

While Spring MVC is excellent for building web applications, Spring Boot's main advantage lies in its ability to streamline and accelerate the development and deployment processes by providing opinionated defaults and reducing the need for manual setup and configuration. It's a powerful tool for rapidly building production-ready applications.

**2)what exactly is Enabling AutoConfiguration in spring boot?what is the difference between @Controller vs @RestController?**

Enabling AutoConfiguration in Spring Boot refers to the mechanism by which the framework automatically configures the application based on its classpath and dependencies. This feature analyzes the environment, including dependencies and configurations present in the project, and automatically configures the Spring application, reducing the need for explicit configuration.

### Enabling AutoConfiguration in Spring Boot:

- Purpose: Simplifies the setup process by providing sensible defaults and reducing the need for manual configuration.

- Mechanism: Spring Boot scans the classpath, detects libraries, and applies configurations and beans based on conventions, annotations, and properties found in the project.

- Advantages: Reduces boilerplate code, accelerates development, and ensures that the application is set up with best practices.

### Difference between `@Controller` and `@RestController` in Spring MVC:

- `@Controller`:

- Used to define a class as a controller in Spring MVC.

- Primarily used to create web applications following the MVC pattern.

- Methods annotated with `@RequestMapping` or its variants handle HTTP requests and return views.

- `@RestController`:

- Introduced in Spring 4.0 to simplify the creation of RESTful web services.

- Combines `@Controller` and `@ResponseBody`.

- Methods annotated with `@RequestMapping` or its variants return the response directly in the requested format (JSON/XML/etc.), typically used for RESTful APIs.

### Key Differences:

- `@Controller` is typically used for creating web applications following the MVC pattern, while `@RestController` is used specifically for building RESTful APIs.

- Methods in `@Controller` return views, while methods in `@RestController` return the response directly without any view rendering, usually in JSON or XML format.

- `@RestController` is a specialized version of `@Controller` meant specifically for building RESTful services, offering a shorthand for combining controller and response body annotations.

Choosing between `@Controller` and `@RestController` depends on the application's use case. If you're building a traditional web application with views, use `@Controller`. For building RESTful services/APIs where data is returned directly, `@RestController` is more appropriate.

**3)Explain @SpringbootApplication?what it will have internally**

Refer my springbboot notes book

**4)what is devtools plugin in springboot?**

Spring Boot DevTools is a set of development tools provided by Spring Boot that aims to enhance the development experience by providing features that help in iterative development, automatic application restarts, and improved debugging capabilities.

### Features and Functionalities of Spring Boot DevTools:

1. Automatic Restart:

- DevTools enables automatic application restarts whenever a classpath change is detected.

- Speeds up the development process by eliminating the need to manually restart the application after code changes.

2. Live Reload:

- Allows the live reloading of static resources (HTML, CSS, JavaScript) in the browser without requiring a full-page refresh.

- Changes made to resources are automatically reflected in the browser during development.

3. Remote Application Restart:

- Supports remote debugging and application restarts.

- Developers can restart a remotely running application via a JMX bean or using IDE plugins.

4. Global Settings Configuration:

- Configurable settings to disable specific DevTools features if needed.

- Helps in customizing the behavior of DevTools based on project requirements.

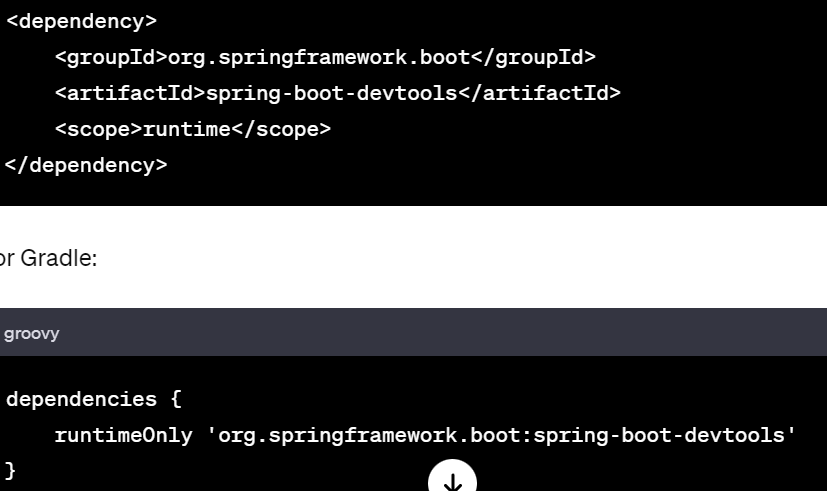
5. Integration with Build Tools:

- Seamless integration with build tools like Maven or Gradle.

- Developers can easily enable or disable DevTools for their projects.

### How to Enable DevTools in a Spring Boot Application:

To include DevTools in a Spring Boot project, add the dependency to the project's build configuration (e.g., `pom.xml` for Maven or `build.gradle` for Gradle):



### Usage:

Once DevTools is included as a dependency, it automatically works in the background, providing the mentioned functionalities during the development phase without requiring additional configuration.

Spring Boot DevTools significantly improves the development experience by reducing the turnaround time for code changes, providing live reload functionality, and supporting quick application restarts during iterative development cycles.

**5)How can you maintain different profiles in spring boot?**

In Spring Boot, profiles are used to define different configurations for different environments or scenarios, allowing developers to maintain settings specific to development, testing, production, etc. Here's how you can manage profiles in Spring Boot:

### 1. Profile-specific Properties Files:

- Create separate properties or YAML files for each profile.

- Name these files following the pattern `application-{profile}.properties` or `application-{profile}.yml`.

### Example:

- `application-dev.properties`: Configuration for the development profile.

- `application-prod.properties`: Configuration for the production profile.

- `application-test.properties`: Configuration for the testing profile.

### 2. YAML Configuration:

- For YAML-based configuration, you can have a single `application.yml` with profile-specific sections.

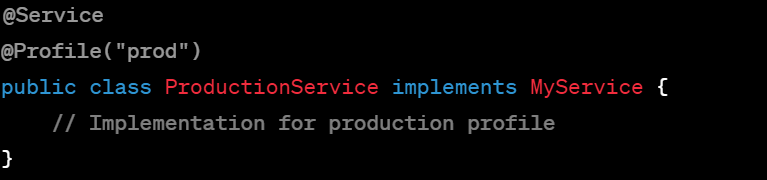
### Example:

```

### 3. `@Profile` Annotation:

- Use `@Profile` annotation to specify that a bean should be created only for specific profiles.

### Example:



### 4. Programmatic Activation:

- Activate profiles programmatically by setting the `spring.profiles.active` property.

### Example:

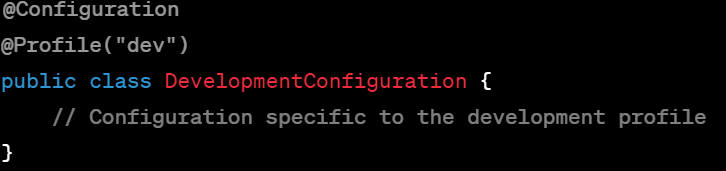
- Command line: `java -jar -Dspring.profiles.active=prod myApplication.jar`

- In `application.properties`: `spring.profiles.active=prod`

### 5. `@Configuration` Classes:

- Use `@Configuration` classes to define beans specific to different profiles.

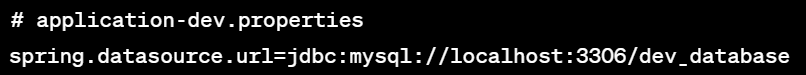
### Example:



### 6. Environment-specific Configuration:

- Override properties based on the environment, e.g., `application-{profile}.properties` files specific to environments like `application-dev.properties`, `application-prod.properties`.

### Example (Properties):



Spring Boot provides flexible ways to manage profiles, allowing developers to maintain different configurations for various environments or scenarios. By leveraging profiles, you can easily switch between settings for development, testing, production, etc., ensuring your application behaves differently based on the active profile.

**6)How springboot is light weight framework?**

Spring Boot is considered a lightweight framework due to several design principles and features that contribute to its lightweight nature:

### 1. Opinionated Defaults and Auto-Configuration:

- Sensible Defaults: Spring Boot provides opinionated defaults, reducing the need for extensive manual configuration.

- Auto-Configuration: It automatically configures the application based on the classpath, reducing the setup overhead.

### 2. Reduced Configuration Overhead:

- Minimal Configuration: Developers can bootstrap a Spring Boot application with minimal configuration, as it provides sensible defaults and eliminates boilerplate code.

- Standalone Applications: Embedded servers and self-contained JARs reduce the need for external server setups, making applications more self-contained.

### 3. Dependency Management:

- Starter POMs: Spring Boot Starter POMs manage dependencies, reducing version conflicts and ensuring coherent dependencies.

- Reduced Jar File Size: Uses only necessary dependencies, resulting in smaller and more optimized JAR files.

### 4. Embedded Servers:

- Embedded Containers: Spring Boot includes embedded servers like Tomcat, Jetty, or Undertow, eliminating the need for separate server installations.

- Simplifies Deployment: Applications can be packaged as standalone JARs, making deployment simpler and more lightweight.

### 5. Simplified Development:

- Rapid Development: Automatic restarts, live reloading, and streamlined development tools (DevTools) speed up development cycles.

- Developer-Friendly Features: Built-in tools for metrics, monitoring, and management make the development process more efficient.

### 6. Modular and Focused Approach:

- Modularity: Spring Boot offers a modular and focused approach, allowing developers to use only the necessary components and features for their applications.

- Integration with Other Spring Projects: Integrates seamlessly with other Spring projects, leveraging their functionalities as needed.

### 7. Emphasis on Convention over Configuration:

- Convention Over Configuration: Spring Boot follows the principle of convention over configuration, reducing the need for explicit configuration by providing sensible defaults.

### 8. Spring Ecosystem:

- Leverages Spring Ecosystem: Built on the foundation of the Spring Framework, Spring Boot leverages the power and features of the Spring ecosystem while simplifying its usage.

By combining opinionated defaults, auto-configuration, reduced setup overhead, embedded servers, and simplified development tools, Spring Boot minimizes the effort required for setup and development, making it a lightweight and developer-friendly framework for building robust and efficient applications.

**7)Is there any difference between @Component and @Service annotations? Can we use @Component instead of @Service?**

Yes, both `@Component` and `@Service` annotations are used to declare Spring beans, and `@Service` is actually a specialized form of `@Component`. They can be used interchangeably in most scenarios, but they do have different semantic meanings.

### `@Component` Annotation:

- `@Component` is a generic stereotype annotation that is used to indicate a Spring-managed component.

- It is a generic marker for any Spring-managed component class.

### `@Service` Annotation:

- `@Service` is a specialization of `@Component` and is intended to be used specifically for declaring services in a business logic layer.

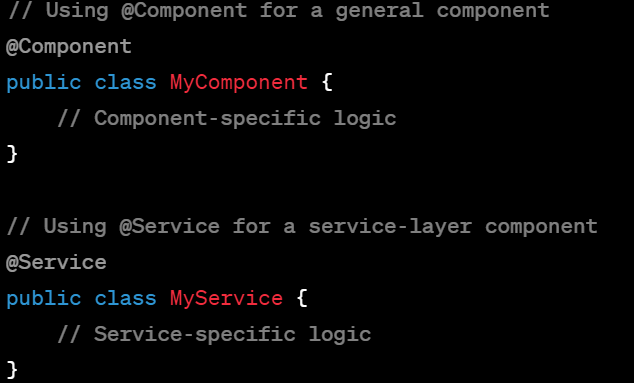
- It's used to annotate service-layer classes and represents the service layer in the application

### Difference and Usage:

- Conceptually, `@Service` is a more specific form of `@Component` and is used to clarify the role of a class in the codebase.

- `@Service` carries additional semantics indicating that it's specifically used for services or business logic, providing better readability and understanding of the code's intent.

- While `@Service` is intended for service-layer classes, `@Component` can be used for any general Spring-managed component.



In practice, both `@Component` and `@Service` can be used to declare beans, and Spring treats them similarly in terms of bean creation and management. However, using `@Service` provides better clarity and semantic meaning when declaring service-layer components, making the code more expressive and understandable.

**8)What will you do if some external server need to be integrated with Springboot?**

Integrating an external server or service with a Spring Boot application involves several steps depending on the nature of the integration, whether it's a RESTful service, a database, a message broker, or any other external system. Here's a general approach:

### 1. Identify the Integration Points:

- Determine the type of external server or service to integrate (REST API, database, messaging service, etc.).

- Understand the requirements, endpoints, authentication mechanisms, and data formats needed for integration.

### 2. Choose Integration Mechanism:

- RESTful Service: Use Spring's `RestTemplate` or `WebClient` for HTTP-based integrations.

- Database Integration: Use Spring Data JPA, JDBC templates, or other data access mechanisms.

- Messaging Services: Use Spring Integration, JMS, or other messaging frameworks.

### 3. Configure Dependencies:

- Add necessary dependencies to the project's build configuration (e.g., Maven `pom.xml` or Gradle `build.gradle`) for the chosen integration mechanism.

### 4. Implement Integration Logic:

- RESTful Service Integration:

- Use `RestTemplate` or `WebClient` to perform HTTP requests and handle responses.

- Configure headers, parameters, and authentication details if required.

- Database Integration:

- Configure database properties in `application.properties` or `application.yml`.

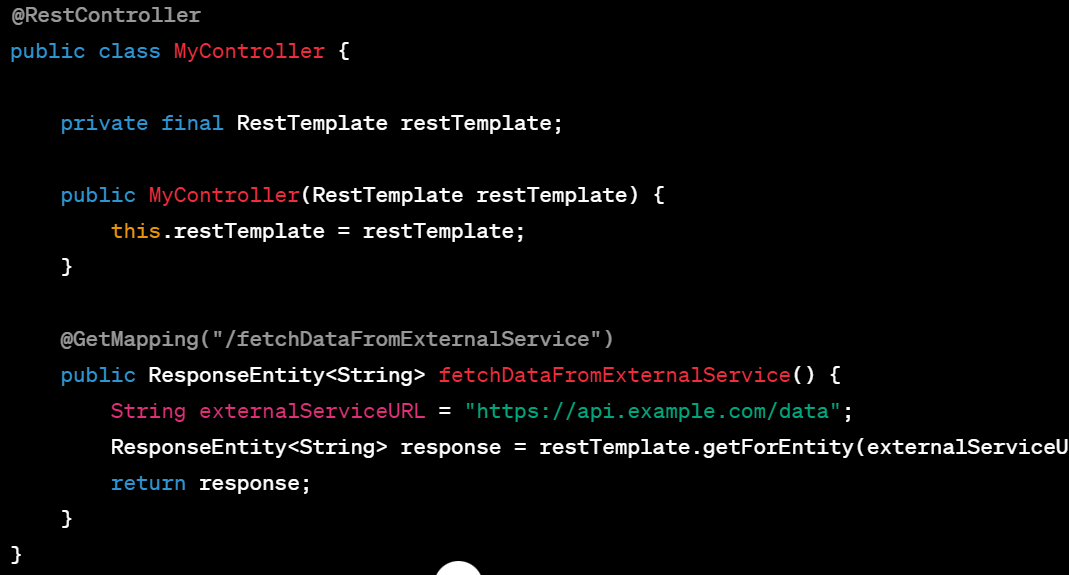
- Use Spring Data JPA or JDBC templates to interact with the database.

- Messaging Services:

- Configure messaging-specific components like `JmsTemplate`, `RabbitTemplate`, or others.

- Implement logic to send/receive messages using the configured components.

### Example of RESTful Service Integration:



### 5. Handle Error Scenarios:

- Implement error handling and resilience mechanisms (retry logic, fallbacks) for handling failures and intermittent issues with the external server.

### 6. Test and Debug:

- Test the integration thoroughly in different scenarios, including success cases and error scenarios.

- Use logging and debugging tools to troubleshoot any issues during integration.

### 7. Secure Integration:

- Implement security measures like authentication, authorization, and encryption where necessary to secure communication with the external server.

By following these steps, you can effectively integrate an external server or service with a Spring Boot application, leveraging Spring's extensive ecosystem and providing robust, scalable, and maintainable integrations.

**9)Suppose if ten users called/accessed a Controller class method at a time, how many instances spring will create? Explain in-detail**

**Ans : Should read about thread pool mechanism which is created by container and assigns each request to separate thread.**

In Spring, by default, controllers are singleton beans. This means that Spring will create a single instance of a controller and share that instance among all requests. When ten users simultaneously access a controller method, Spring will use the same controller instance to handle all those requests.

### Singleton Scope in Spring:

- Singleton Scope (Default): In Spring, singleton scope means that only one instance of a bean is created per Spring container (per application context).

- Controller as Singleton: By default, controllers (`@Controller`, `@RestController`) are singleton beans in Spring.

### Implications of Singleton Controllers:

1. Shared Instance: All users' requests are handled by the same controller instance.

2. Thread-Safety: Controller methods should be thread-safe since they're shared among multiple threads. Ensure that instance variables or shared resources within the controller are handled safely to prevent race conditions.

### Handling Simultaneous Requests:

- Concurrency: Spring Framework, by itself, doesn't manage concurrency at the controller level.

- Servlet Containers: Servlet containers (like Tomcat, Jetty) manage threads for incoming requests. Each request typically runs in its own thread.

### Thread-Safety Considerations:

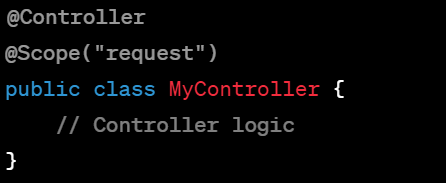
- Controllers should avoid mutable shared state between requests to maintain thread safety.

- Use local variables or request-scoped beans (`@RequestScope`) to store request-specific data.

### Configuring Scope:

- If a different scope is required for a controller (e.g., a new instance per request), you can annotate the controller with `@Scope("request")`. This creates a new instance for each incoming request.

### Example (Custom Scope for Controller):



### Summary:

By default, Spring creates a single instance (singleton) of a controller, and this instance is shared among multiple users making concurrent requests. It's crucial to ensure that controller methods and shared resources within the controller are thread-safe when handling simultaneous requests to prevent any unexpected behavior due to concurrency. If a different behavior is desired, Spring's scope annotations can be used to modify the bean's scope, such as `@Scope("request")` for a new instance per request.

**10)How to integrate Springdata JPA with springboot?what are main interfaces available in Springdata JPA**

Integrating Spring Data JPA with Spring Boot is straightforward and involves several steps:

### Steps to Integrate Spring Data JPA with Spring Boot:

1. Add Dependencies:

- Add the necessary dependencies for Spring Data JPA and your chosen database driver to your `pom.xml` (for Maven) or `build.gradle` (for Gradle).

2. Configure Data Source:

- Configure the database connection properties in `application.properties` or `application.yml`, specifying the database URL, username, password, etc.

3. Create Entity Classes:

- Define entity classes annotated with `@Entity` that represent tables in your database.

- Annotate fields with JPA annotations (`@Id`, `@Column`, etc.) to define the mapping between entities and database tables.

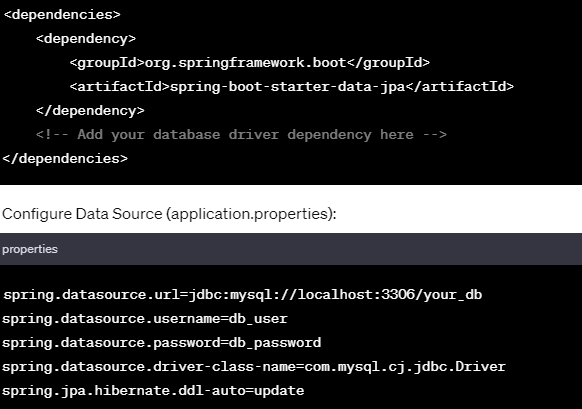
4. Create Repository Interfaces:

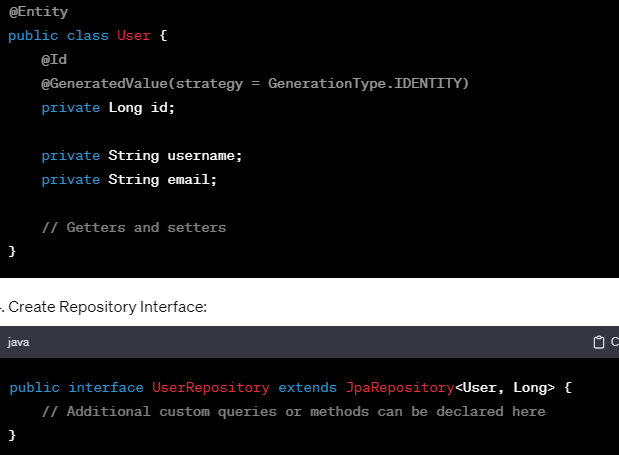
- Define repository interfaces that extend Spring Data's `JpaRepository` or its derivatives. These interfaces provide CRUD operations for your entities without needing to write implementation code.

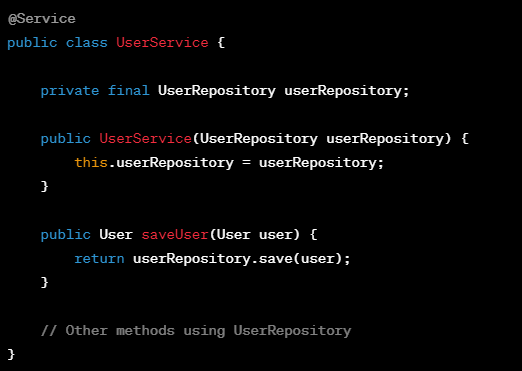
5. Use Repository Methods:

- Use methods provided by repository interfaces to perform database operations like saving, querying, updating, and deleting entities.

### Example:







#### 1. Add Dependencies (Maven):

#### 2. Configure Data Source (application.properties):

#### 3. Create Entity Class:

#### 4. Create Repository Interface:

#### 5. Usage in Service or Controller:

### Main Interfaces Available in Spring Data JPA:

- `JpaRepository`: Provides CRUD (Create, Read, Update, Delete) operations and common querying methods.

- `PagingAndSortingRepository`: Extends `JpaRepository` and adds pagination and sorting capabilities.

- `CrudRepository`: Basic CRUD operations without additional querying methods.

- `QueryByExampleExecutor`: Supports query-by-example operations.

These interfaces provide convenient and powerful abstractions for interacting with the database, offering a wide range of functionalities to perform database operations efficiently without writing boilerplate code.

**11)How can you provide security to your API's that you have implemented?**

Securing APIs in a Spring Boot application can be achieved using Spring Security, a powerful and customizable authentication and authorization framework. Here's an overview of steps to secure your APIs:

### 1. Add Spring Security Dependency:

- Include the Spring Security dependency in your `pom.xml` (Maven) or `build.gradle` (Gradle).

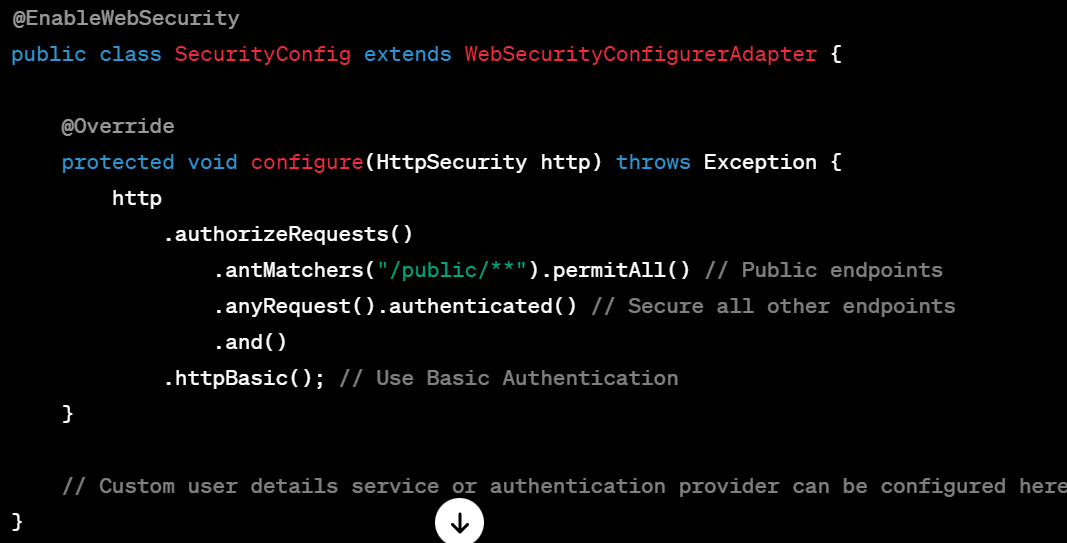
### 2. Configure Security:

- Create a security configuration class extending `WebSecurityConfigurerAdapter`.

- Override `configure(HttpSecurity http)` to define security rules for endpoints and resources.

- Configure authentication mechanisms like Basic Auth, OAuth, JWT, etc.

### Example Configuration for Basic Authentication:



### 3. Define User Roles and Permissions:

- Define roles and permissions for users using annotations like `@Secured`, `@PreAuthorize`, or `@RolesAllowed` on controller methods.

### Example of Role-Based Access Control (RBAC):



### 4. Implement Authentication Mechanisms:

- Configure authentication mechanisms like JWT, OAuth, or integrate with an external identity provider.

### 5. Use HTTPS/SSL:

- Always use HTTPS to encrypt data transferred between the client and server, ensuring secure communication.

### 6. Test Security Configurations:

- Test your security configurations thoroughly to ensure endpoints are properly secured and access is restricted as intended.

### 7. Customize Security as Needed:

- Spring Security offers extensive customization options. Customize as per your application's specific security requirements.

### Additional Tips:

- CORS Configuration: Use CORS (Cross-Origin Resource Sharing) to control access to resources from different domains.

- Input Validation: Validate and sanitize user input to prevent common security threats like SQL injection, XSS attacks, etc.

By configuring Spring Security in your Spring Boot application, you can control access to your APIs, implement authentication and authorization mechanisms, and ensure that your endpoints are secure according to your application's requirements.

**12)How can expose and consume REST APIs in springboot?**

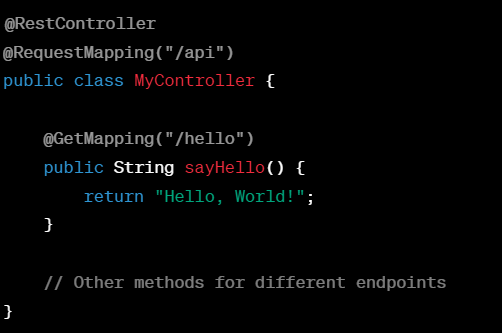
In Spring Boot, exposing and consuming REST APIs is straightforward and involves creating controllers to expose APIs and using various techniques to consume those APIs. Here are the steps to expose and consume REST APIs:

### Expose REST APIs:

#### 1. Create Controller:

- Create a controller class annotated with `@RestController` or `@Controller` and map methods to HTTP endpoints using `@RequestMapping`, `@GetMapping`, `@PostMapping`, etc.

#### Example:

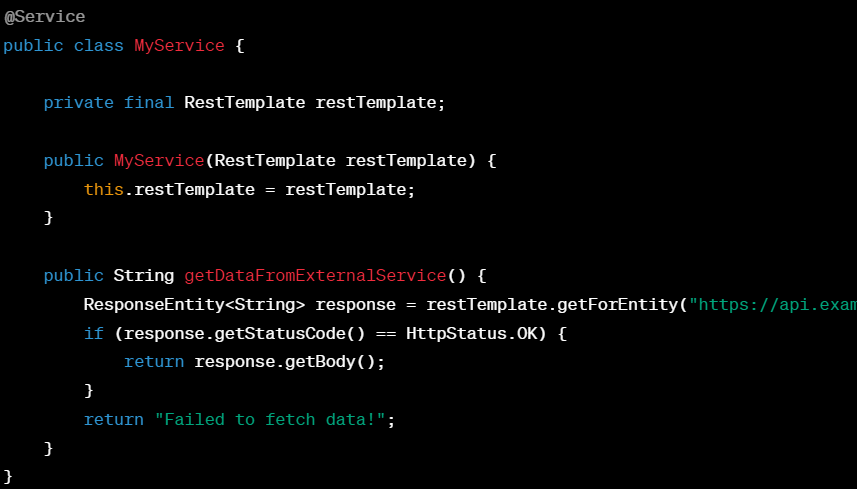


### Consume REST APIs:

#### 1. Using RestTemplate:

- Use `RestTemplate` (synchronous) or `WebClient` (asynchronous) to consume REST APIs from other services or endpoints.

#### Example with RestTemplate:



#### 2. Using Feign Client:

- Use Feign, a declarative REST client provided by Spring Cloud, to create a client interface that maps to the REST API.

#### Exampe with Feign Client:



### Test REST APIs:

- Use tools like Postman, cURL, or browser extensions to test the exposed APIs by sending HTTP requests (GET, POST, PUT, DELETE) and verifying the responses.

### Summary:

- Expose APIs: Create controllers using annotations like `@RestController` and map methods to endpoints.

- Consume APIs: Use `RestTemplate`, `WebClient`, or Feign Client to consume APIs from other services.

- Test APIs: Validate the exposed APIs by testing them with appropriate tools or clients.

By following these steps, you can easily expose your own REST APIs using Spring Boot and consume APIs from other services or endpoints within your application.

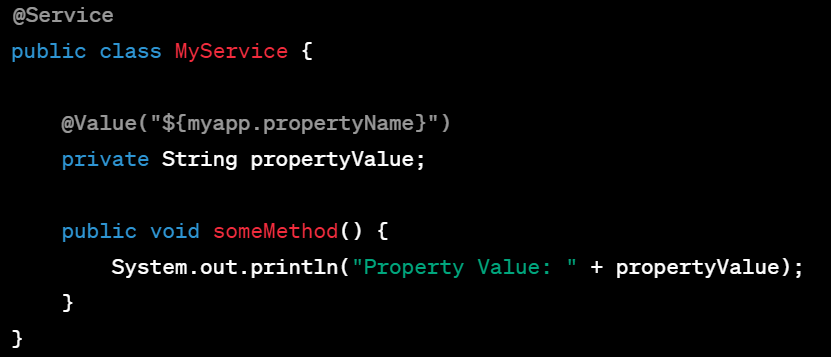
**14)How can you read properties file values in springboot classes?**

In Spring Boot, reading property values from configuration files like `application.properties` or `application.yml` can be done using the `@Value` annotation or by using the `Environment` or `@ConfigurationProperties` mechanism.

### Using @Value Annotation:

#### 1. Read Single Property:

- Use `@Value` annotation to inject a single property value into a class field or method parameter.



### Using Environment:

#### 2. Inject Environment:

- Inject `Environment` and use its `getProperty()` method to retrieve property values.

#### Example:

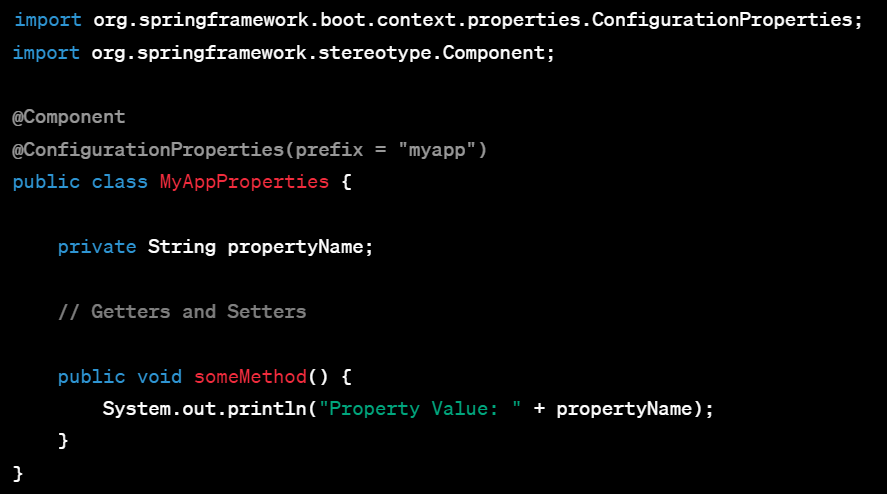


### Using @ConfigurationProperties:

#### 3. Define Configuration Class:

- Create a configuration class annotated with `@ConfigurationProperties` to bind properties to a POJO.

#### Example:

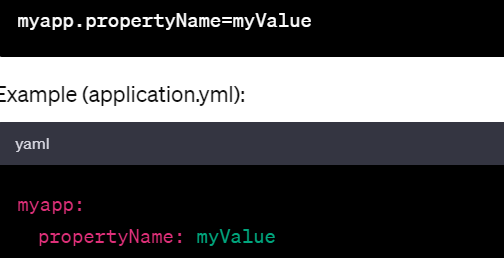
;

### Reading from application.properties or application.yml:

- Properties should be defined in `application.properties` or `application.yml`.

- Use the `prefix` in `@ConfigurationProperties` to specify the property hierarchy.

#### Example (application.properties):



### Summary:

- Use `@Value` for simple property injection.

- Use `Environment` to retrieve properties dynamically.

- Use `@ConfigurationProperties` to bind properties to a dedicated configuration class.

Choose the approach based on the complexity of property management and the need for strong typing or configuration class structures. These methods offer flexibility in accessing properties from configuration files in Spring Boot applications.

**15)What is DI and IOC in spring?**

Dependency Injection (DI) and Inversion of Control (IoC) are core concepts in the Spring Framework that facilitate loose coupling between components and enhance modularity, maintainability, and testability of applications.

### Inversion of Control (IoC):

- IoC refers to the design principle where the control of object creation and management is inverted or delegated to an external container or framework.

- In traditional programming, objects are responsible for creating and managing their dependencies.

- In IoC, the responsibility of creating, managing, and injecting dependencies is shifted to an external entity, usually a container or framework like Spring.

- IoC reduces the coupling between classes, allowing for better modularization and easier testing.

### Dependency Injection (DI):

- DI is a specific implementation of IoC that refers to the process of injecting the dependencies of a class from an external source.

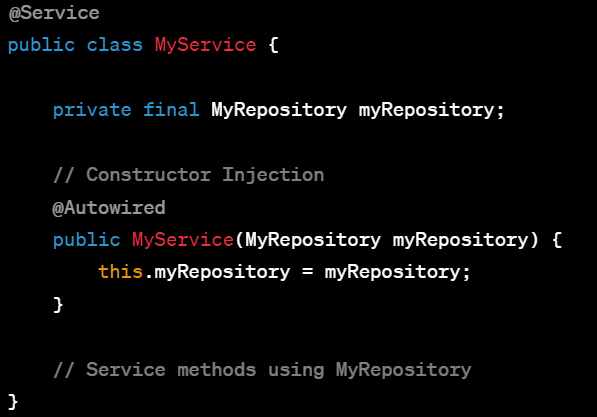
- Dependencies are typically provided to a class through constructor injection, setter injection, or method injection.

- The dependencies are resolved and provided by the IoC container, removing the need for classes to instantiate their own dependencies.

- DI helps in achieving loose coupling, making code more maintainable, flexible, and easily testable.

### Example of Dependency Injection in Spring:

Consider a service class that depends on a repository for data access:



In this example:

- The `MyService` class receives an instance of `MyRepository` through constructor injection.

- Spring's IoC container automatically provides the `MyRepository` instance to `MyService` when it's created.

### Benefits of DI and IoC:

- Loose Coupling: Reduces dependencies between classes, making the code more maintainable and adaptable to changes.

- Testability: Easier to test components in isolation by replacing dependencies with mocks or stubs.

- Modularity: Promotes modular design and improves code reusability.

- Flexibility: Facilitates the management of object lifecycle and configuration externally.

Spring Framework heavily relies on DI and IoC to manage and wire components together, enabling developers to build modular, maintainable, and easily testable applications.

**16)What is bean life cycle states in spring mvc?how many types of containers are available for us?**

In Spring MVC, the bean lifecycle refers to the various stages an object goes through from its instantiation to its destruction within the Spring container. The bean lifecycle consists of several states:

### Bean Lifecycle States in Spring MVC:

1. Instantiation:

- At this stage, the bean instance is created, typically using a constructor.

2. Initialization:

- After instantiation, Spring performs initialization tasks:

- `@PostConstruct` annotated methods or `InitializingBean` interface's `afterPropertiesSet()` method is called.

- Bean properties are set.

- Custom initialization logic is executed.

3. In Use:

- The bean is in a usable state and can be accessed and used by other beans or components.

4. Destruction/Disposal:

- When the application context is closed or when explicitly configured, beans go through disposal:

- `@PreDestroy` annotated methods or `DisposableBean` interface's `destroy()` method is called.

- Custom cleanup logic is executed.

### Types of Containers in Spring:

1. BeanFactory:

- The basic container interface providing bean instantiation, configuration, and management features.

- Lazy loading of beans (beans are created only when requested).

2. ApplicationContext:

- Extends the `BeanFactory` functionality.

- Provides additional features like event propagation, internationalization, message sources, etc.

- Eagerly loads singleton beans during startup.

- Various implementations like `ClassPathXmlApplicationContext`, `AnnotationConfigApplicationContext`, etc.

3. WebApplicationContext (part of ApplicationContext):

- Specialized for web applications.

- Inherits features of `ApplicationContext`.

- Provides additional features like access to ServletContext, automatic registration of beans, etc.

- Typically used in Spring MVC applications.

### Summary:

- Bean lifecycle in Spring MVC involves instantiation, initialization, being in use, and destruction stages.

- The available containers in Spring include `BeanFactory`, `ApplicationContext`, and `WebApplicationContext`.

- Each container provides various features and functionalities catering to different application requirements.

Understanding the bean lifecycle helps in managing resources effectively, performing initialization/cleanup tasks, and integrating with Spring's features seamlessly. The choice of container depends on the specific needs and features required by the application.

**17)what will happen when you are accessing a properties file values and if property key or value not present or entire variable is not presented in the file?**

When accessing properties from a properties file in Spring or Java, the behavior depends on how the properties are accessed and the configuration setup.

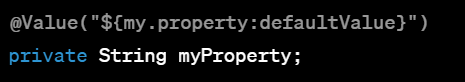
### Behavior when a Property Key is Not Present:

1. Using @Value Annotation:

- If accessing properties using `@Value` annotation and the key is not present:

- If a default value is provided, it will be used.

- If no default value is provided, it might throw a `NoSuchBeanDefinitionException` or use a default value defined in the annotation.



2. Using Environment or PropertySourcesPlaceholderConfigurer:

- When accessing properties using `Environment` or `PropertySourcesPlaceholderConfigurer` and the key is not present:

- It might return `null` or throw a `NoSuchBeanDefinitionException`, depending on how it's configured.

### Behavior when the Entire Variable is Not Present in the File:

- If the entire variable (key-value pair) is missing or commented out:

- Spring won't find and bind the property to the corresponding field or value.

- When trying to access this property, it will either return `null` or throw a `NoSuchBeanDefinitionException`, depending on the access method.

**18)How can you handle custom exceptions in your spring boot applications?**

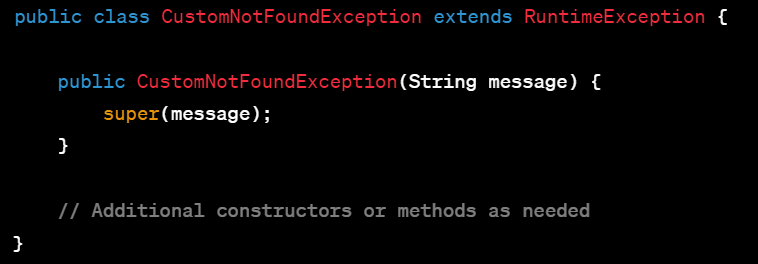
Handling custom exceptions in a Spring Boot application involves creating custom exception classes and implementing exception handling mechanisms to provide meaningful responses to different types of errors. Here are the steps:

### 1. Create Custom Exception Classes:

- Create custom exception classes that extend from `RuntimeException` or its subclasses.

- Add necessary constructors and methods to customize the exception behavior.

#### Example:



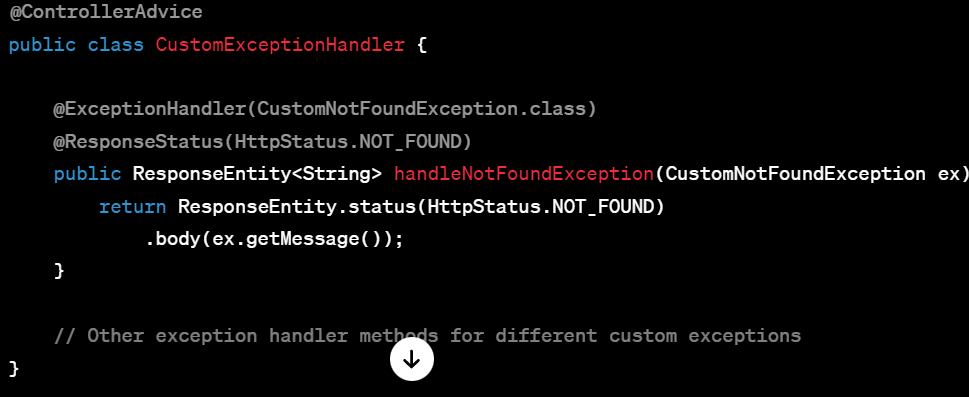
### 2. Implement Exception Handling:

#### a. Using @ExceptionHandler:

- Use `@ExceptionHandler` in a `@ControllerAdvice` or `@RestControllerAdvice` class to handle specific exceptions globally.

- Define methods annotated with `@ExceptionHandler` to handle specific exceptions and return appropriate responses.

#### Example:

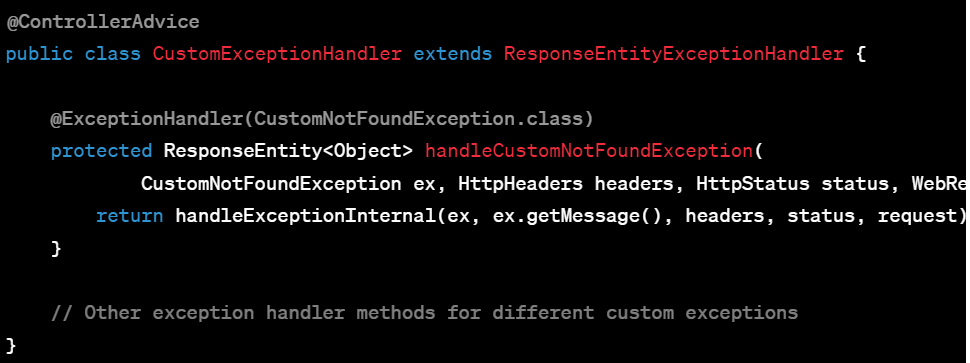


#### b. Using ResponseEntityExceptionHandler:

- Extend `ResponseEntityExceptionHandler` to create a custom exception handler for a broader set of exceptions.

- Override methods to handle specific exception types and customize error responses.

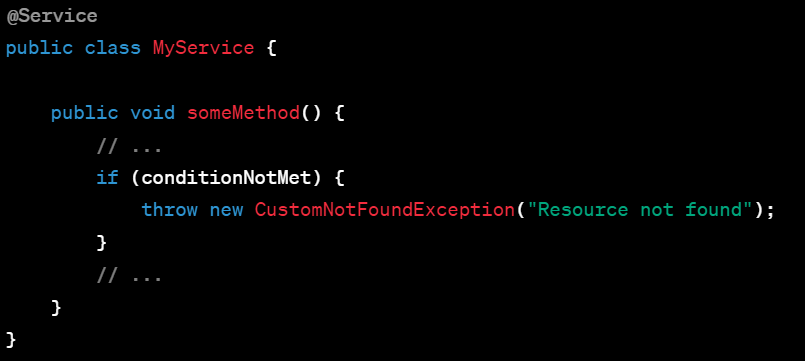
#### Example:



### 3. Throwing Custom Exceptions:

- Throw custom exceptions from your service or controller layer when handling specific error conditions.

#### Example:



### Summary:

- Create custom exception classes to represent specific error conditions.

- Implement exception handling using `@ExceptionHandler` in a `@ControllerAdvice` or extend `ResponseEntityExceptionHandler`.

- Throwing custom exceptions from your code when specific error conditions occur.

By following these steps, you can effectively handle custom exceptions in your Spring Boot application and provide appropriate error responses to different types of exceptions or error scenarios.

**19)How are you handling logs in your springboot application after deploying into different environments?**

Handling logs in a Spring Boot application across different environments involves configuring logging frameworks and adapting log levels, destinations, and formats based on the environment. Here's how you can manage logs in different environments:

### 1. Use Spring Boot's Default Logging Configuration:

- Spring Boot uses SLF4J (Simple Logging Facade for Java) as a facade for various logging frameworks like Logback, Log4j2, or JDK logging.

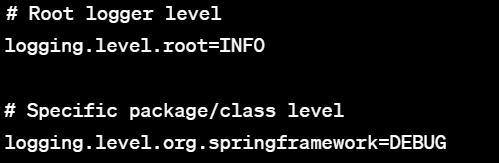
- It provides sensible default configurations for logging in the `application.properties` or `application.yml` file.

### 2. Configure Logging Levels:

- Adjust logging levels for different environments (e.g., `dev`, `test`, `prod`) to control the amount of logging information.

- Set log levels for specific packages or classes to control the verbosity of logs.

#### Example (application.properties):

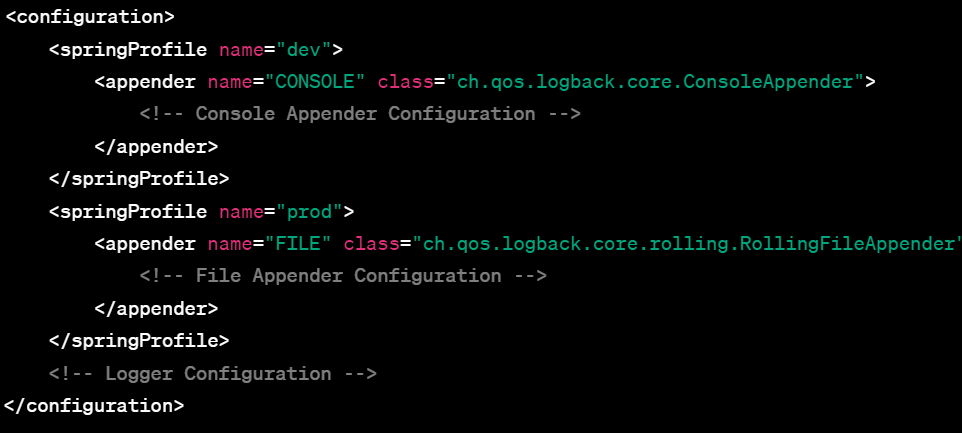


### 3. Modify Logging Destinations:

- Configure different log destinations (console, files, databases) for different environments to control where logs are stored.

- Use appenders or handlers provided by the chosen logging framework.

#### Example (Logback Configuration - `logback-spring.xml`):



### 4. Externalize Log Configuration:

- Externalize logging configurations into separate files (e.g., `logback-spring.xml`, `log4j2.xml`) for each environment.

- Use Spring Profiles to load specific logging configurations based on the active profile.

#### Example (Logback Configuration for Different Environments):

- `logback-dev.xml` for development environment.

- `logback-prod.xml` for production environment.

### 5. Dynamically Change Logging at Runtime:

- Implement dynamic log level changes using Actuator endpoints or custom endpoints to modify log levels without restarting the application.

#### Example:

- Use Spring Boot Actuator's `/actuator/loggers` endpoint to dynamically modify log levels.

**20)How you create spring boot application? Please explain some primary dependencies in pom.xml**

Spring Initializer,STS,Console and parent,web,test

**21)What are idempotent HTTP methods in REST?**

Idempotent HTTP methods in RESTful APIs are operations that can be safely repeated multiple times without causing different outcomes beyond the initial application. These methods ensure that performing the same operation multiple times will result in the same state as if it had been performed only once.

### Idempotent HTTP Methods:

1. GET:

- The `GET` method is inherently idempotent as it retrieves resources without modifying the server state. Multiple GET requests for the same resource should return the same response.

2. PUT:

- `PUT` replaces or creates a resource at a specific URL.

- When you send the same `PUT` request multiple times with the same payload, it should result in the same resource state.

3. DELETE:

- `DELETE` removes a resource at a specific URL.

- Executing multiple `DELETE` requests for the same resource doesn't change the state beyond the initial deletion.

### Non-Idempotent Methods:

1. POST:

- `POST` is not inherently idempotent as it creates new resources on the server.

- Sending the same `POST` request multiple times might result in multiple resources being created.

2. PATCH:

- `PATCH` applies partial modifications to a resource.

- Repeated `PATCH` requests might alter the resource differently each time if different patch operations are applied.

### Why Idempotent Methods Are Important:

- Safety: Idempotent methods ensure that repeating requests doesn't lead to unexpected or unintended changes in the server state.

- Reliability: They make it safer to retry requests that may have failed previously due to network issues or other reasons.

- Caching: Idempotent methods are cacheable by nature, as subsequent requests yield the same response.

In RESTful design, leveraging idempotent methods helps in creating robust, predictable, and reliable APIs, contributing to better scalability and reliability of the system.

**22)what exactly it means @RestController?**

In Spring MVC, `@RestController` is a specialized version of the `@Controller` annotation that is typically used in RESTful web service implementations. It's a convenience annotation that combines `@Controller` and `@ResponseBody` annotations, indicating that the return value of the methods should be directly written into the HTTP response body as JSON or XML.

### Key Points about `@RestController`:

1. Combines `@Controller` and `@ResponseBody`:

- `@Controller` is used to mark classes as Spring MVC controllers.

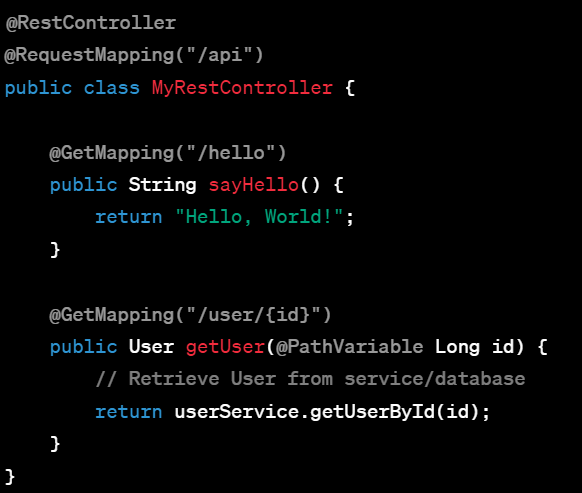
- `@ResponseBody` indicates that the return value of methods should be written directly to the response body.

2. Return Values as Response Body:

- Methods inside a `@RestController` return domain objects or DTOs directly, and Spring automatically serializes these objects to JSON/XML for HTTP responses.

3. Commonly Used in RESTful Services:

- Suited for creating RESTful web services where responses are typically in JSON or XML format.



- In the above example, `@RestController` at the class level indicates that this controller deals primarily with RESTful services.

- The `sayHello()` method returns a string directly, which will be sent as the response body.

- The `getUser()` method returns a `User` object, and Spring automatically serializes it to JSON/XML and sends it as the response body.

**23)differences between getForEntity() vs getForObject() vs exchange() on resttemplate?**

In Spring's `RestTemplate`, `getForEntity()`, `getForObject()`, and `exchange()` are methods used to perform HTTP GET requests, but they differ in their functionalities and the kind of response they provide:

### 1. `getForEntity()`:

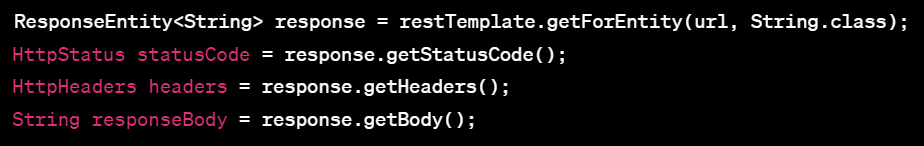
- Functionality:

- Retrieves the HTTP response as an entire `ResponseEntity` containing the response body, headers, and status.

- Provides access to the response headers and status code along with the response body.

- Usage:

- Typically used when you need detailed information from the HTTP response, including headers and status.



### 2. `getForObject()`:

- Functionality:

- Retrieves the HTTP response body directly, mapping it to the specified Java object.

- Does not provide access to response headers or status code directly.

- Usage:

- Ideal when you only need the response body mapped to a specific Java object and not interested in headers or status code.



### 3. `exchange()`:

- Functionality:

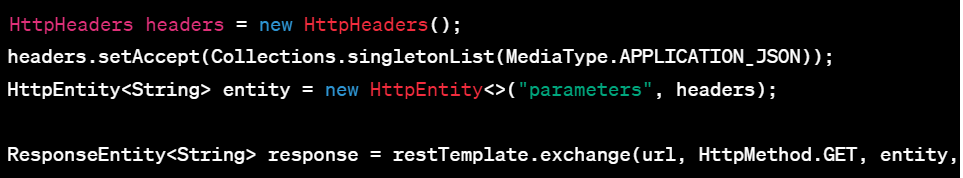
- Provides the most flexibility among the three methods by allowing complete control over the HTTP request and response.

- Returns a `ResponseEntity` like `getForEntity()` but requires specifying the HTTP method explicitly.

- Usage:

- Useful when you need fine-grained control over request parameters, headers, and when working with HTTP methods other than GET (e.g., PUT, POST, DELETE).

- Example:



- `getForEntity()` returns the complete `ResponseEntity` with status, headers, and body.

- `getForObject()` directly maps the response body to a Java object, without providing access to headers or status.

- `exchange()` offers full control over request parameters and headers, providing a `ResponseEntity` and allowing customization of the HTTP method. Use it when you need detailed control over the request and response.

**24)differences between RestTemplate and FeignClient and Webclient?**

`RestTemplate`, `Feign`, and `WebClient` are different ways to consume RESTful services in Spring, each with its own characteristics and functionalities:

### 1. RestTemplate:

- Synchronous: Traditionally used for synchronous communication with RESTful services.

- Imperative Style: Based on imperative programming, making blocking calls.

- Configurable: Provides straightforward configuration and customization options.

- Mature and Stable: Long-standing part of Spring framework.

- Thread Safety: Generally not considered thread-safe, but instances can be reused within a single thread.

### 2. Feign (Declarative HTTP Client):

- Declarative Style: Allows declaring RESTful service interfaces using annotations.

- Integration with Eureka: Integrates easily with Netflix Eureka for service discovery.

- Interface-Based: Generates client code from interfaces, reducing boilerplate code.

- Netflix OSS Component: Part of the Netflix OSS suite.

- Synchronous by Default: Supports asynchronous calls but defaults to synchronous behavior.

### 3. WebClient:

- Reactive and Non-Blocking: Built for reactive programming and supports non-blocking I/O.

- Fluent API: Provides a fluent and functional style of writing HTTP requests.

- Asynchronous: Supports reactive programming paradigms, allowing asynchronous calls.

- Thread Safety: Designed to be fully thread-safe and can be shared among multiple threads.

- Functional Endpoints: Enables building HTTP requests using functional endpoints.

### Differences:

- Synchronous vs. Asynchronous: RestTemplate and Feign are typically used for synchronous communication, while WebClient is designed for reactive and asynchronous communication.

- Imperative vs. Declarative: RestTemplate is imperative, Feign allows a more declarative style, and WebClient offers a fluent and functional API.

- Thread Safety: RestTemplate instances are generally not thread-safe, while WebClient is designed to be thread-safe.

- Programming Style: Feign provides a more abstract way of defining REST clients through interfaces, while RestTemplate and WebClient require more explicit configuration and method calls.

### Choosing Between Them:

- Use `RestTemplate` if you need synchronous communication and prefer the imperative programming style.

- Choose `Feign` if you prefer a declarative approach for REST clients and plan to use Netflix Eureka for service discovery.

- Opt for `WebClient` if you are building reactive applications or need non-blocking I/O and prefer a functional and fluent API.

The choice between them often depends on the project's requirements, architectural considerations, and whether you're working with traditional blocking applications or reactive, non-blocking systems.

**25)what is the difference between Mono vs Flux?**

`Mono` and `Flux` are two fundamental types in Project Reactor, a reactive programming library used in Java for building reactive applications. They represent reactive streams for handling asynchronous data sequences, but they differ in their cardinality and use cases:

### Mono:

- Cardinality: Represents a stream emitting at most one element.

- Use Case: Ideal for handling zero or one element responses, like fetching a single user profile or a single database record.

- Methods: Provides operations like `map`, `flatMap`, `filter`, etc., similar to the ones in standard Java `Stream`.

- Termination: Emits a single element, an error, or no elements.

### Flux:

- Cardinality: Represents a stream emitting zero or more elements.

- Use Case: Suited for handling multiple elements, like processing a list of items or continuous data streams.

- Methods: Offers the same operations as `Mono` but also includes batch processing methods like `buffer`, `window`, etc.

- Termination: Emits multiple elements, an error, or no elements.

### Differences:

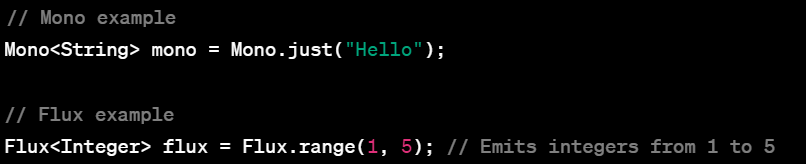
- Cardinality: Mono represents a stream of zero or one element, while Flux represents zero or more elements.

- Use Cases: Mono is used for handling single-value asynchronous operations, whereas Flux is used for multiple-value asynchronous operations.

- Termination: Mono terminates with one or zero elements, while Flux can terminate with multiple elements.

- Methods: Both offer similar operations, but Flux includes additional batch processing methods for multiple elements.

### Example:



### Summary:

- `Mono` and `Flux` are both reactive stream types in Project Reactor.

- `Mono` represents zero or one element, suitable for single-value asynchronous operations.

- `Flux` represents zero or more elements, suitable for multiple-value asynchronous operations or continuous data streams.

- Both offer similar reactive operations, but `Flux` includes additional methods for batch processing multiple elements.

**26)what is Asynchronous/reactive way of programming?**

Asynchronous or reactive programming is an approach to handling computations and I/O operations in a non-blocking, event-driven manner, where tasks are executed independently of the main program flow. It allows systems to handle multiple tasks concurrently without blocking threads, improving performance and scalability. Here are key aspects:

### Key Characteristics:

1. Non-Blocking Operations:

- Asynchronous programming enables tasks to be executed without waiting for the completion of preceding tasks.

- I/O operations, such as network requests or file reading, do not block the main execution thread.

2. Event-Driven Approach:

- Operations are triggered by events or signals, and callbacks are used to handle the results or events.

- Utilizes event loops or reactive streams to manage and respond to events asynchronously.

3. Concurrency and Parallelism:

- Allows handling multiple tasks concurrently without dedicating a thread per task, optimizing resource utilization.

- Enables parallel execution of independent tasks, improving performance in multi-core environments.

4. Reactive Programming Paradigm:

- Emphasizes the use of streams, observables, or publishers to represent sequences of data or events over time.

- Provides operators and transformations to handle and manipulate asynchronous data streams.

### Benefits:

- Scalability: Non-blocking operations allow efficient utilization of system resources, enhancing scalability.

- Responsiveness: Helps create more responsive applications by avoiding thread blocking and allowing systems to handle multiple tasks simultaneously.

- Performance: Enables better performance, especially in I/O-bound applications, by leveraging concurrency and parallelism.

- Resource Utilization: Optimizes resource utilization by reducing thread overhead and context switching.

### Examples:

- Reactive Libraries: RxJava, Project Reactor (Mono, Flux), Akka Streams.

- Asynchronous I/O Frameworks: Node.js, Netty, asyncio (Python), async/await in C#.

### Use Cases:

- Web Servers and APIs: Handling concurrent requests efficiently.

- Real-time Applications: Chat applications, streaming services.

- I/O-Intensive Operations: File I/O, database operations, network requests.

### Summary:

Asynchronous or reactive programming revolves around executing tasks concurrently, handling non-blocking operations, and leveraging event-driven paradigms. It enhances system responsiveness, scalability, and resource utilization, making it well-suited for modern, high-performance applications, especially those dealing with I/O operations and concurrency.

**27)what is the difference between http Post vs Put vs patch?**

The HTTP methods `POST`, `PUT`, and `PATCH` are used to perform different operations on resources in a RESTful API, and they have distinct characteristics and purposes:

### POST:

- Purpose: Used to create a new resource or submit data to a specified resource.

- Idempotent: Generally considered non-idempotent; invoking the same POST request multiple times can result in the creation of multiple resources.

- Usage: Commonly used for creating resources on the server when the server assigns the resource ID.

- Example: Creating a new user by sending user data in the request body to `/users`.

### PUT:

- Purpose: Used to update or replace a resource at a specific URL.

- Idempotent: Intended to be idempotent; invoking the same PUT request multiple times with the same data should result in the same state as a single operation.

- Usage: Entire resource needs to be provided in the request; used when the client is updating or replacing the entire resource.

- Example: Updating user details by sending updated user data to `/users/{id}`.

### PATCH:

- Purpose: Used to apply partial modifications to a resource.

- Idempotent: Conceptually intended to be idempotent; repeating the same PATCH request should result in the same state as a single operation.

- Usage: Allows clients to send only the specific changes to be applied to the resource, useful for partial updates.

- Example: Modifying only specific fields of a user's data (e.g., updating only the user's email address) by sending a PATCH request to `/users/{id}` with the modified field(s).

### Summary:

- POST is used to create a new resource or submit data.

- PUT is used to update or replace a resource entirely.

- PATCH is used to apply partial modifications to a resource without replacing it entirely.

- Idempotency differs: POST is generally non-idempotent, while PUT and PATCH are designed to be idempotent (though implementation details might affect idempotency).

Choosing between these methods depends on the specific use case and the intended operation on the resource—whether it's creating, updating, or partially modifying data within a RESTful API.

**28)differences between HTTP method status code 401 vs 403 and 200 vs 202?**

### 401 Unauthorized vs 403 Forbidden:

- 401 Unauthorized:

- Indicates that the request requires user authentication.

- The client must provide valid credentials (usually via the `Authorization` header) for the server to grant access.

- The server sends this status code to prompt the client to provide valid credentials.

- 403 Forbidden:

- Indicates that the server understood the request, but the client doesn't have permission to access the requested resource.

- The server refuses to authorize the client's request, even if authentication credentials are provided.

- It's a definitive refusal, indicating that the server knows the client identity but won't grant access.

### 200 OK vs 202 Accepted:

- 200 OK:

- Indicates that the request was successful.

- The server processed the request and returned the requested resource or performed the requested action successfully.

- It's a standard success status code for successful HTTP requests.

- 202 Accepted:

- Indicates that the request has been accepted for processing, but the processing has not been completed.

- The request might be processing asynchronously, or it may not have completed its execution yet.

- It doesn’t necessarily mean that the action requested has been completed or will be completed successfully.

### Summary:

- 401 vs 403: Both relate to access control; 401 means authentication is required, while 403 means access is forbidden even if authentication is provided.

- 200 vs 202: Both indicate successful requests, but 202 signifies that the request has been accepted for processing but may not have been executed or completed yet.

Choosing the appropriate status code depends on the context and the semantics that best fit the response to the client's request in compliance with HTTP standards.

**29)when to use @PathVariable vs @RequestBody vs @RequestParam vs @QueryParam?**

The choice between `@PathVariable`, `@RequestBody`, `@RequestParam`, and `@QueryParam` in Spring (or similar frameworks) depends on how you want to extract data from an incoming HTTP request:

### @PathVariable:

- Usage: Extracts values directly from the URI path.

- Example: `/users/{id}` - `{id}` is a path variable.

- When to Use:

- Used to extract values directly from the URI, commonly used for resource identification in RESTful APIs.

- Suitable for accessing specific parts of the URL like IDs or resource identifiers.

### @RequestBody:

- Usage: Extracts the entire request body as an object.

- Example: Used with `POST`, `PUT`, or `PATCH` requests to receive JSON or XML data.

- When to Use:

- Used when the entire request body needs to be mapped to a Java object.

- Suitable for complex or structured data sent in the request body, such as JSON payloads.

### @RequestParam:

- Usage: Extracts query parameters or form data from the request URL.

- Example: `?name=value` in the URL; `name` is a query parameter.

- When to Use:

- Used for simple query parameters or form data sent via `GET` or `POST`.

- Suitable for accessing individual query parameters or form fields.

### @QueryParam (Not Standard in Spring MVC):

- Usage: Similar to `@RequestParam`, but typically used in JAX-RS (Java API for RESTful Web Services).

- Example: Extracts query parameters from the request URL.

- When to Use:

- Used in JAX-RS environments for accessing query parameters.

### Summary:

- Use `@PathVariable` for extracting values directly from the URI path.

- Use `@RequestBody` when you need to map the entire request body to a Java object, typically for complex data.

- Use `@RequestParam` for simple query parameters or form data.

- `@QueryParam` is specific to JAX-RS environments and is similar to `@RequestParam` but used in different contexts.

Choose the appropriate annotation based on the type of data being passed and where it's located within the HTTP request.

**30)@Priority annotation use in Springboot?@Qualifier annotation usage in Springboot**

### @Priority Annotation:

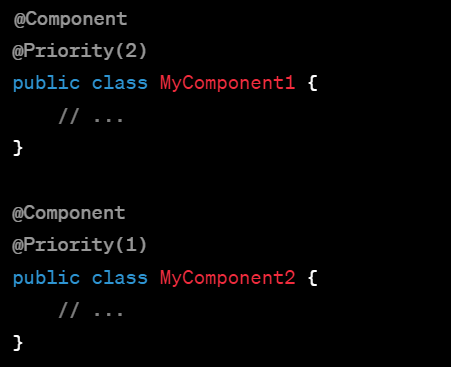
- Purpose: Defines the priority order for instances of a particular type when there are multiple candidates.

- Usage:

- Applied to beans or components to define their priority order when autowiring or resolving beans.

- Usually used in conjunction with `@Order` for defining a specific order among multiple instances of the same type.

### Example:



In this example, `MyComponent1` has a higher priority (lower number) compared to `MyComponent2`.

### @Qualifier Annotation:

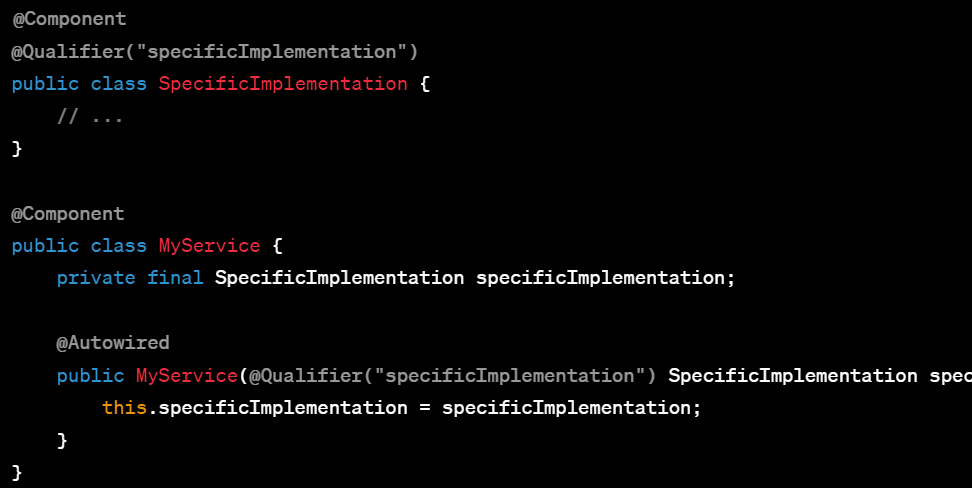
- Purpose: Specifies a qualifier for autowiring when multiple beans of the same type are present.

- Usage:

- Used to explicitly specify which bean to inject when there are multiple candidates of the same type.

- Applied together with `@Autowired` or `@Inject` to identify the bean to be injected.

### Example:



In this example, `@Qualifier("specificImplementation")` helps Spring identify which bean to inject into `MyService`.

### Summary:

- `@Priority` defines the priority order among multiple instances of the same type.

- `@Qualifier` specifies the specific bean to be injected when multiple candidates of the same type exist.

Both annotations help resolve ambiguity in autowiring scenarios when multiple beans of the same type are available for injection, but they address different aspects—priority ordering and bean selection based on qualifiers.

**31)what is Facade design pattern**

The Facade design pattern is a structural pattern that provides a simplified interface to a complex system, making it easier to use. It serves as a unified interface to a set of interfaces or subsystems, providing a higher-level interface that hides the complexities of the underlying system.

### Key Components:

1. Facade:

- Acts as an entry point to a complex system or a set of interfaces.

- Provides a simplified interface that shields the client from the complexities of the subsystem.

2. Subsystems:

- Consist of multiple classes or interfaces that represent various parts of the system.

- Contain the actual implementation details and functionalities

### Features and Benefits:

- Simplified Interface: Offers a simple and unified interface to interact with a complex system.

- Abstraction of Complexity: Hides the details and complexities of the subsystem from the client.

- Encapsulation: Provides a single entry point for client interactions, reducing dependencies and coupling.

- Promotes Maintainability: Facilitates easier maintenance and changes by isolating the changes within the facade.

### Example:

Consider a multimedia system comprising multiple components like audio player, video player, and projector. The Facade pattern can create a `MultimediaFacade` that provides methods like `playMovie()`, `playAudio()`, `stopMovie()`, etc. Internally, it delegates these calls to the appropriate subsystems (audio player, video player, projector) and shields the client from their complexities.

### When to Use:

- Complex Systems: Use when working with complex systems to provide a simpler interface for clients.

- Multiple Interfaces/Subsystems: When a system involves multiple interfaces or subsystems and needs to provide a unified interface.

### Summary:

The Facade pattern simplifies the usage of a complex system by providing a high-level interface that hides the complexities of the underlying subsystems. It encapsulates the subsystems' details, promoting easier usage, maintenance, and reduced coupling between clients and the system's internals.

**32)Factory, Builder and Session Design patterns?**

### Factory Design Pattern:

- Intent:

- Provides an interface for creating objects, but allows subclasses to alter the type of objects that will be created.

- Use Cases:

- When the creation of an object involves complex logic or conditions.

- When you want to decouple the object creation from the client code.

#### Types of Factory Patterns:

1. Simple Factory Pattern:

- A single factory method creates objects based on a given parameter or condition.

- It encapsulates object creation logic.

2. Factory Method Pattern:

- Defines an interface for creating an object but allows subclasses to alter the type of objects created.

- Uses inheritance to achieve polymorphic creation.

3. Abstract Factory Pattern:

- Creates families of related or dependent objects without specifying their concrete classes.

- Provides an interface for creating a set of related objects.

### Builder Design Pattern:

- Intent:

- Separates the construction of a complex object from its representation, allowing the same construction process to create various representations.

- Use Cases:

- When the object creation involves multiple steps or requires complex initialization.

- When you want to create an object with a large number of optional parameters.

#### Key Components:

1. Director:

- Controls the construction process by calling appropriate builder methods.

2. Builder:

- Defines an interface for constructing parts of a complex object.

- Provides methods for setting individual attributes.

3. Concrete Builders:

- Implement the builder interface to construct and assemble parts of the complex object.

### Session Design Pattern:

- Intent:

- Manages information about a particular user session in a web application.

- Use Cases:

- Maintaining user-specific data during a session.

- Managing user authentication and authorization.

#### Features:

1. User Session Data:

- Stores user-specific information across multiple interactions with the application.

2. Authentication and Authorization:

- Manages user login/logout, tracks session timeouts, and handles permissions.

3. Session Management:

- Handles session creation, storage, and deletion.

### Summary:

- Factory patterns deal with object creation:

- Simple Factory provides a centralized factory method.

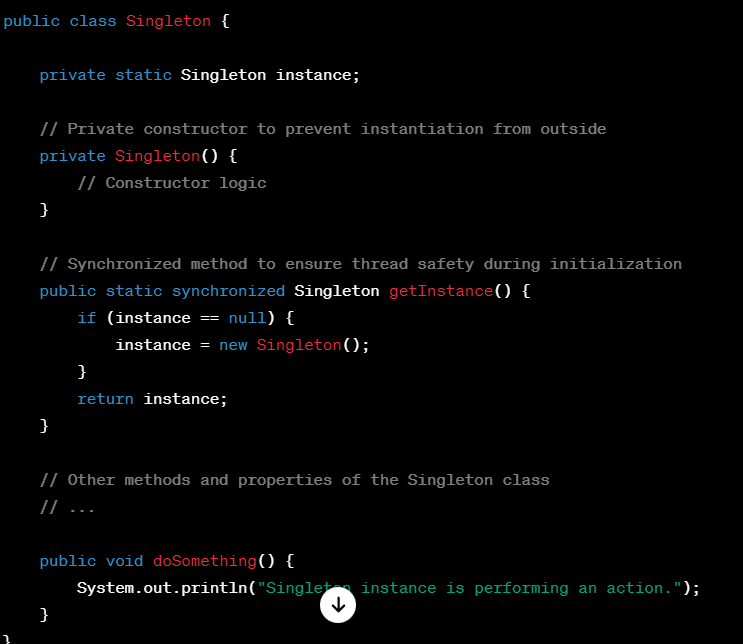
- Factory Method uses inheritance for object creation.

- Abstract Factory creates families of related objects.

- Builder separates the construction of complex objects from their representation.

- Session patterns manage user-specific data and behavior during application sessions, particularly in web applications.

**33)code to write Singleton design pattern with synchronization**



Explanation:

- The class has a private constructor to prevent direct instantiation from outside the class.

- `getInstance()` method is synchronized, ensuring that only one thread can enter at a time to initialize the Singleton instance.

- The instance is lazily created the first time `getInstance()` is called, ensuring it's created when needed.

- Subsequent calls to `getInstance()` return the already initialized instance.

- Note: While this ensures thread safety during initialization, it may incur a performance overhead due to locking.

It's important to note that in modern Java versions (starting from Java 5), you might consider using the `volatile` keyword with the `instance` variable to ensure proper visibility of changes across threads.



The `volatile` keyword ensures that changes made by one thread to the instance variable are visible to other threads immediately. This helps avoid issues with thread visibility and ensures the correctness of the Singleton pattern in a multi-threaded environment.

**35)what is convention over configuration in springboot?**

Convention over Configuration (CoC) is a software design paradigm that emphasizes the use of default conventions to simplify and streamline development by reducing the need for explicit configuration.

In the context of Spring Boot:

### Convention over Configuration in Spring Boot:

1. Default Configuration:

- Spring Boot uses sensible defaults to configure various aspects of the application.

- It minimizes the need for extensive configuration files by providing predefined settings and auto-configuration mechanisms.

2. Opinionated Defaults:

- Spring Boot is opinionated and provides default configurations based on best practices and common scenarios.

- It follows naming conventions and assumes certain defaults, reducing the need for developers to specify every detail explicitly.

3. Reduced Boilerplate Code:

- Spring Boot eliminates a significant amount of boilerplate configuration code by leveraging default settings.

- Developers only need to provide configurations for aspects that deviate from the standard conventions.

4. Auto-Configuration:

- Spring Boot uses auto-configuration to automatically configure the application based on the dependencies in the classpath.

- It detects the libraries and dependencies present and configures the application accordingly, reducing manual setup.

### Benefits of Convention over Configuration:

- Simplifies Development: Developers can focus more on business logic rather than spending time on extensive configuration setups.

- Faster Development: Default settings and auto-configuration speed up the development process.

- Consistency: Promotes consistency across projects and teams by following established conventions.

- Reduced Complexity: Minimizes the complexity of configuration files and settings.

### Example:

In Spring Boot, a simple web application can run with minimal configuration. If you follow naming conventions and place your Java classes in predefined locations (`@Controller`, `@Service`, `@Repository`, etc.), Spring Boot automatically identifies and configures them. It sets up default properties for database connections, embedded servers, logging, and more without needing explicit configuration.

Overall, Convention over Configuration in Spring Boot aims to streamline development by reducing the need for explicit configuration, thus making the development experience more efficient and straightforward.

**36)How are you handling exceptions in your spring boot project?what is the difference between @ControllerAdvice vs @ExceptionHandler?**

### Handling Exceptions in Spring Boot:

1. Using @ExceptionHandler:

- `@ExceptionHandler` is used within a controller to handle exceptions for specific controller methods.

- It allows you to define exception handling logic within the controller itself.

- You annotate a method with `@ExceptionHandler` and specify the exception type it should handle.

2. Using @ControllerAdvice:

- `@ControllerAdvice` is a global exception handler that applies across the application.

- It allows you to define global exception handling logic for multiple controllers.

- Annotate a class with `@ControllerAdvice` and include `@ExceptionHandler` methods to handle exceptions for multiple controllers or globally.

### Difference between @ControllerAdvice and @ExceptionHandler:

- @ExceptionHandler:

- Used at the method level within a controller.

- Handles exceptions only for the specific controller or controllers where it's defined.

- Allows fine-grained control over exception handling within individual controllers.

- @ControllerAdvice:

- Used at the class level to define a global exception handler.

- Handles exceptions globally across multiple controllers.

- Provides centralized exception handling and allows for consistent error handling across the application.

### Example:



In this example, `GlobalExceptionHandler` is a global exception handler using `@ControllerAdvice` to handle all exceptions of type `Exception` across the application. `MyController` has a specific exception handler using `@ExceptionHandler` to handle `NullPointerException` for that controller.

### Summary:

- `@ExceptionHandler` is used within controllers for specific exception handling.

- `@ControllerAdvice` is used to define global exception handling for multiple controllers or globally across the application.

**37)Differnce between @Qualifier vs @Primary annotations in spring boot?**

In Spring Boot, `@Qualifier` and `@Primary` are used for resolving ambiguities during dependency injection, but they serve different purposes:

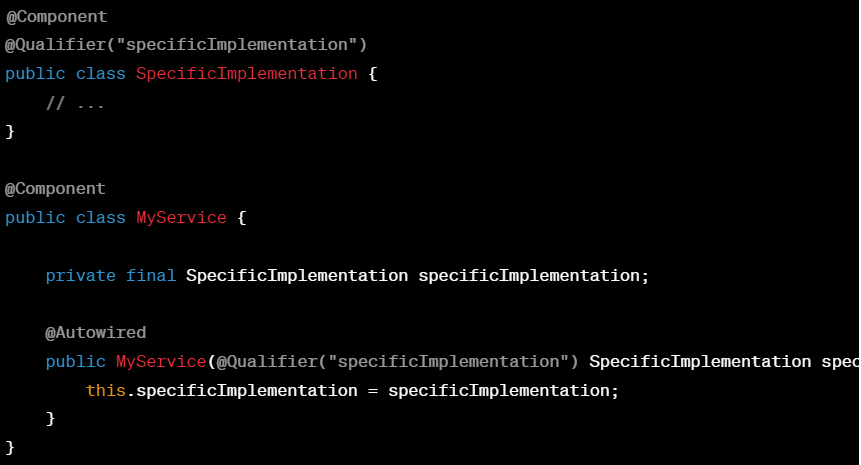
### @Qualifier Annotation:

- Purpose:

- Helps resolve ambiguity when multiple beans of the same type are available.

- Used to specify which bean should be injected when there are multiple candidates of the same type.

#### Example:



### @Primary Annotation:

- Purpose:

- Indicates the primary bean to be used when multiple beans of the same type are available.

- Used to give precedence to a specific bean when autowiring, especially in scenarios where no `@Qualifier` is provided.

#### Example:



### Differences:

- @Qualifier:

- Used to specify which particular bean to inject when multiple candidates of the same type exist.

- Provides explicit wiring by specifying the name of the bean to be injected.

- @Primary:

- Marks a bean as the primary candidate to be injected when there are multiple candidates of the same type.

- Implicitly used when no `@Qualifier` is provided and is considered the default choice.

### Usage Guidelines:

- Use `@Qualifier` when you want to specify a particular bean to be injected among multiple candidates.

- Use `@Primary` when you want to designate a default or primary bean to be injected when no specific `@Qualifier` is provided.

In scenarios where `@Qualifier` and `@Primary` are both used, the bean marked with `@Qualifier` takes precedence over the `@Primary` bean if the specific qualifier is specified during injection. Otherwise, the `@Primary` bean is used as the default choice.

**38)what is the use of @RefreshScope annotation in spring boot?**

The `@RefreshScope` annotation in Spring Boot is used in microservices architecture with Spring Cloud Config Server to enable dynamic reloading of configuration properties at runtime without requiring a restart of the entire application.

### Key Features:

1. Dynamic Property Refresh:

- Annotating a bean or a configuration class with `@RefreshScope` indicates that Spring should refresh the state of that bean when a refresh event occurs.

- Changes to the configuration in the Config Server trigger a refresh event, causing beans annotated with `@RefreshScope` to reload their properties.

2. Integration with Spring Cloud Config Server:

- Spring Cloud Config Server allows centralized management of application properties, which can be updated dynamically.

- When configuration properties change in the Config Server, the `@RefreshScope`-annotated beans update their state to reflect the changes.

### Usage:

1. Annotating Beans or Configuration Classes:

- Use `@RefreshScope` on Spring beans or configuration classes that you want to refresh dynamically.

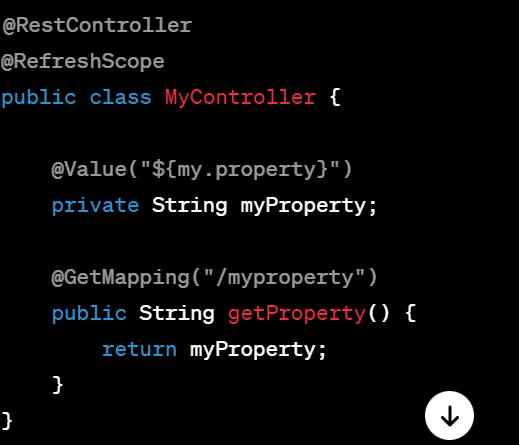
- When the properties related to these beans change in the Config Server, a refresh event is triggered.

2. Usage with Actuator and Refresh Endpoint:

- Combined with Spring Boot Actuator, you can expose a `/refresh` endpoint.

- Triggering a POST request to `/actuator/refresh` sends a refresh signal to all beans annotated with `@RefreshScope`, updating their properties.

### Example:



In this example, `@RefreshScope` is used on the `MyController` class to indicate that the `myProperty` field should be refreshed when there are changes to the related property in the Config Server.

### Summary:

- `@RefreshScope` enables dynamic property refreshing in Spring Boot applications integrated with Spring Cloud Config Server.

- Annotated beans or configurations update their state based on changes in the centralized configuration stored in the Config Server.

- Used in combination with Spring Boot Actuator to expose the `/refresh` endpoint for triggering property refreshes.

**39)What is default scope of a bean in spring? And what are other scopes available in SpringBoot?**

In Spring Framework, the default scope of a bean is singleton. This means that, by default, if no explicit scope is specified for a bean, Spring creates and maintains a single instance of that bean throughout the entire application context.

### Default Scope:

- Singleton:

- The default scope in Spring. It creates a single instance per Spring container (per ApplicationContext).

### Other Available Scopes in Spring Boot:

1. Singleton:

- Default scope. One instance per container.

2. Prototype:

- Creates a new bean instance each time it's requested.

- Not managed by the container after creation; clients are responsible for managing the bean's lifecycle.

3. Request:

- Creates a new instance for each HTTP request.

- Applicable in a web-aware ApplicationContext (e.g., in a web application).

4. Session:

- Creates a single instance for each HTTP session.

- Applicable in a web-aware ApplicationContext for maintaining state across multiple requests from the same user session.

5. Application (available in Spring Boot 2.4+):

- Creates a single instance per SpringApplication.

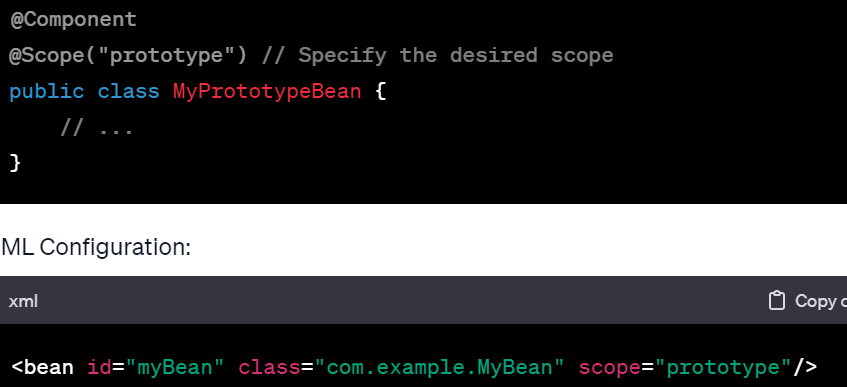
6. Websocket (available in Spring Boot 2.4+):

- Creates a single instance for each WebSocket session.

### Declaring Bean Scope:

You can specify the scope of a bean explicitly using the `@Scope` annotation or in XML configuration:

#### Using @Scope annotation:



### Choosing a Scope:

- Singleton is the default and most commonly used scope. It's suitable for stateless beans or beans that don't change state frequently.

- Prototype is useful for stateful beans or instances where you need a new object every time it's requested.

- Other scopes like Request, Session, Application, and Websocket are used specifically in web applications for managing bean instances across different HTTP requests or sessions.

Choosing the appropriate scope depends on the nature of the bean and its intended usage within the application.

**40)Under which tag or dependency you will add <Exclude> tag in pom.xml if you don't want to use in-built tomcat server?**

Ans:Under stater-web only

<artifactId>spring-boot-starter-web</artifactId>

<exclusions>

<exclusion>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-tomcat</artifactId>

</exclusion>

</exclusions>

**41)what is the difference between @RequestMapping and @PostMapping?**

In Spring MVC, `@RequestMapping` and `@PostMapping` are annotations used to map HTTP requests to specific handler methods in a controller. Here are the differences between them:

### @RequestMapping:

- Purpose:

- A versatile annotation used to map HTTP requests to handler methods.

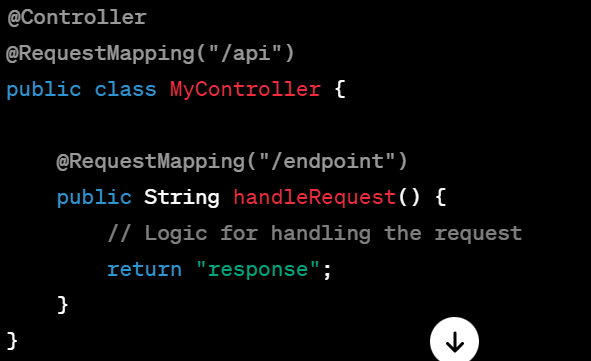
- It's a generic mapping annotation that can handle multiple HTTP request methods (GET, POST, PUT, DELETE, etc.) based on the configuration.

- Usage:

- Without specifying an HTTP method, it handles requests for all HTTP methods by default.

- It's more generic and can be used to map multiple HTTP methods to a single handler method.

#### Example:



### @PostMapping:

- Purpose:

- Specifically used to map HTTP POST requests to handler methods.

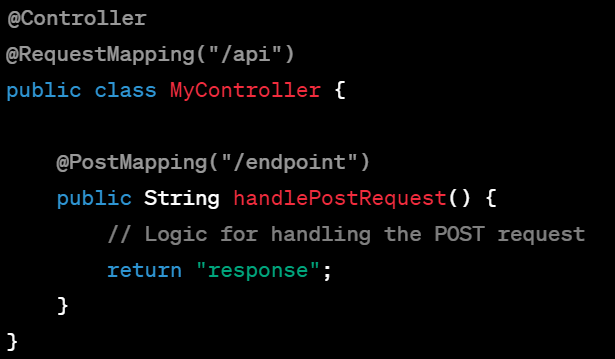
- Restricts the handler method to only handle POST requests.

- Usage:

- Specifically used when you want a method to handle only POST requests.

- Provides more explicit mapping, making it clearer that the method is intended to handle POST requests only.

#### Example:



### Differences:

- @RequestMapping is a generic mapping annotation that can handle multiple HTTP request methods, while @PostMapping is specifically used to handle only POST requests.

- @PostMapping provides clearer and more explicit mapping for methods that are intended to handle POST requests, improving code readability and understanding.

When choosing between them, use @RequestMapping when handling multiple HTTP methods or when handling both GET and POST requests in the same method. Use @PostMapping when the method is intended to handle only POST requests for better specificity and clarity in the code.

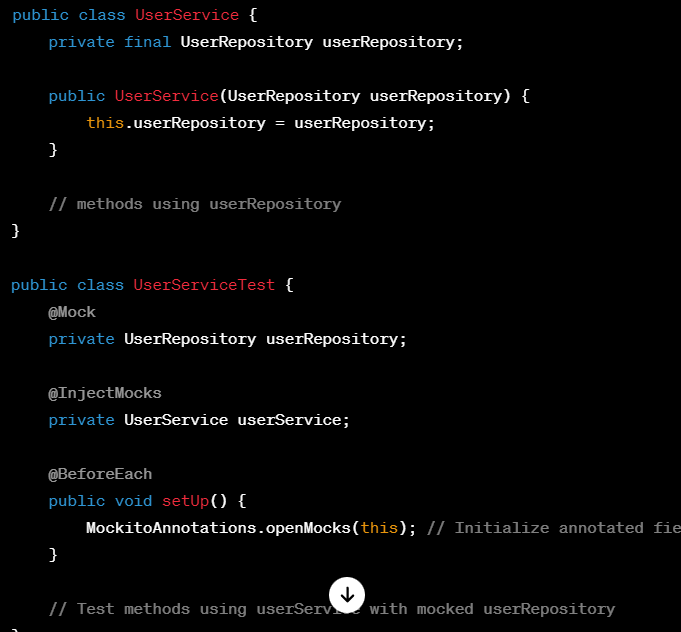
**43)difference between @Mock and @InjectMocks**

In the context of testing in Java with Mockito, `@Mock` and `@InjectMocks` are annotations used for setting up and managing dependencies within test cases.

- `@Mock`: This annotation is used to create a mock instance of a class or interface. Mock objects simulate the behavior of real objects but allow you to define their behavior for testing purposes. When you annotate a field with `@Mock`, Mockito creates a mock object of that type.

- `@InjectMocks`: This annotation is used to inject mocks into the fields of a tested object. When you annotate a field with `@InjectMocks`, Mockito attempts to inject the mock objects created with `@Mock` into this object.

For example, suppose you have a class `UserService` that depends on a `UserRepository`. You can use `@Mock` to create a mock `UserRepository` and `@InjectMocks` to inject that mock `UserRepository` into the `UserService` for testing:



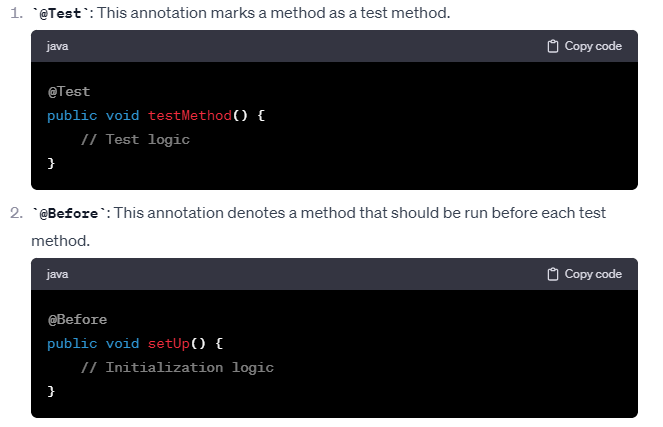
`@Mock` creates a mock `UserRepository`, while `@InjectMocks` injects this mock `UserRepository` into the `UserService` instance being tested.

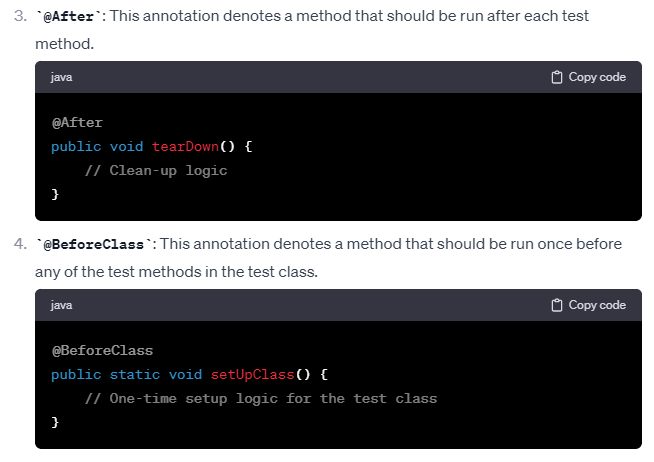
Keep in mind that while Mockito is a powerful tool for creating and managing mocks in tests, using mocks effectively requires an understanding of the behavior and interactions expected between different components in your code.

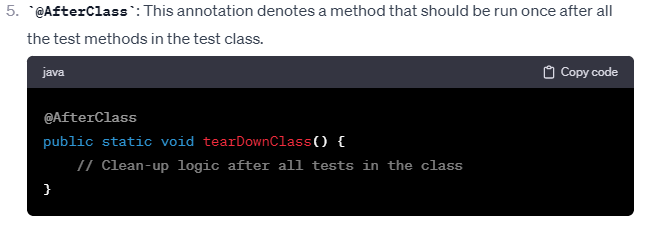
**44)what are some methods and annotations you have used from JUnit?**

Certainly! JUnit is a popular testing framework for Java that provides various annotations and methods to facilitate the creation and execution of test cases. Here are some commonly used annotations and methods from JUnit:

### Annotations:







### Assertion Methods:

JUnit provides various assertion methods to verify expected outcomes in test cases. Some commonly used assertion methods include:

- `assertEquals(expected, actual)`: Checks if two values are equal.

- `assertTrue(condition)` / `assertFalse(condition)`: Checks if a condition is true / false.

- `assertNull(object)` / `assertNotNull(object)`: Checks if an object is null / not null.

- `assertSame(expected, actual)` / `assertNotSame(expected, actual)`: Checks if two objects refer to the same / different object.

- `assertArrayEquals(expectedArray, resultArray)`: Checks if two arrays are equal.

### Test Suites:

JUnit also allows you to create test suites to run multiple test cases together.

- `@RunWith`: Annotation to specify the test runner class. For example, `@RunWith(JUnitPlatform.class)` for JUnit 5.

- `@Suite.SuiteClasses`: Annotate a class with `@Suite.SuiteClasses` to specify the test classes to include in the suite.

These are some fundamental annotations and methods in JUnit that help in creating and running test cases effectively. They provide a foundation for organizing tests, defining test conditions, and validating expected outcomes.

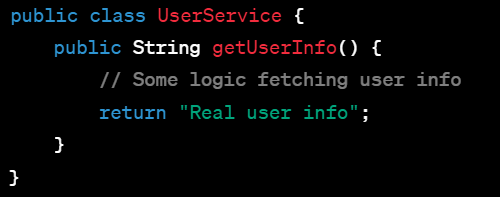
**45)what is @Spy annotation in JUnit?**

The `@Spy` annotation in JUnit (commonly used with Mockito) is used to create a partial mock of a real object. Unlike a mock object created with `@Mock`, a spy retains the real behavior of the object while allowing you to stub or verify specific methods.

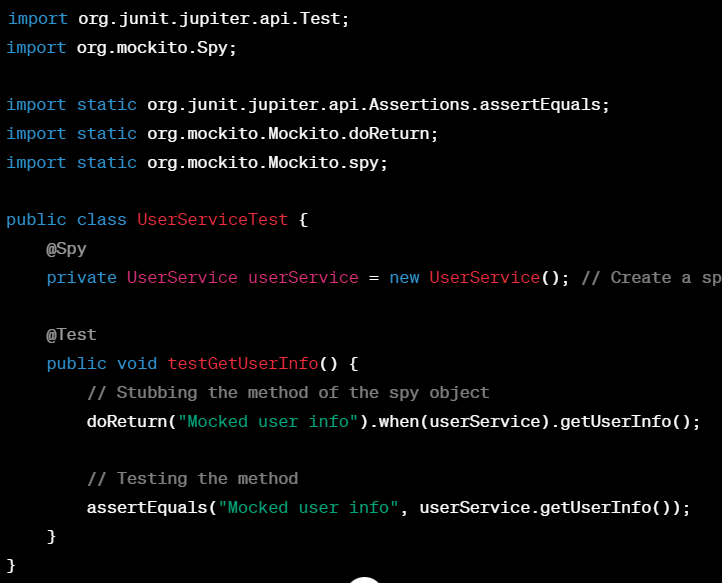
When you annotate an object with `@Spy`, Mockito creates a spy object that wraps the actual instance, and you can selectively mock or stub specific methods of this object while leaving the rest of the behavior unchanged.

Here's an example:

Suppose you have a `UserService` class:



Now, you want to test the `getUserInfo` method, but you also want to mock its behavior to return a custom value for testing purposes. You can use `@Spy` to partially mock this object while retaining its original behavior:



In this example, `@Spy` is used to create a spy of the `UserService` class. Then, using `doReturn(...).when(...)` syntax from Mockito, the behavior of the `getUserInfo` method is stubbed to return "Mocked user info" instead of the real implementation.

Remember, a spy retains the real behavior of the object by default, unless explicitly stubbed, allowing you to selectively mock or verify specific methods while maintaining the original functionality for the rest of the methods.

**46)how can you test private and void method using Junit?**

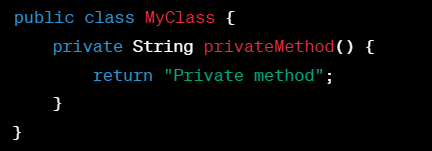
**using spy() method for private method and doNothing() for void methods**

Testing private methods directly using JUnit is generally discouraged because the primary focus of unit tests is on the public API of a class. Private methods are implementation details and are indirectly tested through the public methods that use them. However, there are a few ways you can approach testing private methods if necessary:

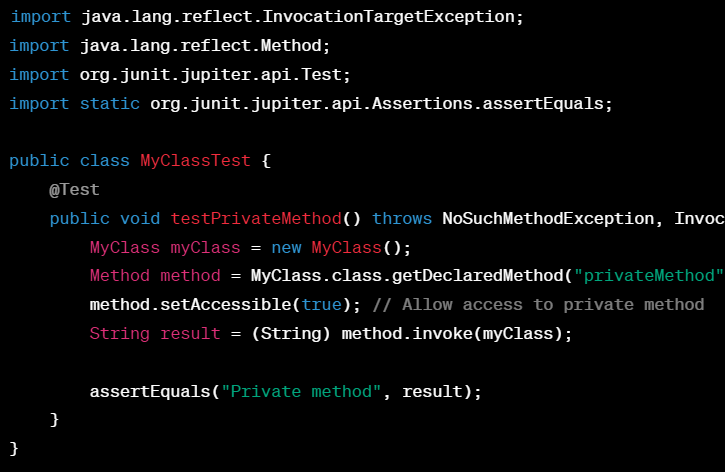
### Reflection

You can use Java's reflection to access and invoke private methods for testing. Here's an example:

Suppose you have a class `MyClass` with a private method `privateMethod()`:



In your test class:



This approach uses reflection to access and invoke the private method `privateMethod()` for testing purposes.

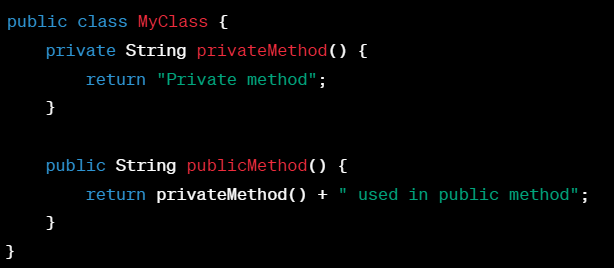
### Refactoring for Testability

Consider refactoring the code to improve testability. If a method is crucial for testing and it's currently private, you might consider making it package-private or protected, allowing it to be accessed by the test classes in the same package or subclass.

### Test Through Public Interface

The preferred approach is to test private method behavior indirectly through the public methods that use them. By testing the public methods that utilize the private methods, you indirectly validate the behavior of the private methods.

For example:



You'd test `publicMethod()` to ensure it behaves correctly, which in turn verifies the behavior of `privateMethod()` indirectly.

Remember, focusing on testing the class's public API ensures that tests remain resilient to changes in implementation details.

**48)For example if you want to update your existing project springboot (parent) version what steps you will follow?**

Updating the Spring Boot parent version involves a few steps to ensure a smooth transition and compatibility with your existing project. Here are the general steps you can follow:

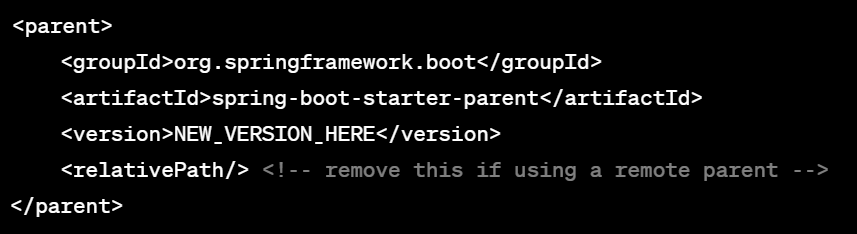
### 1. Identify the New Spring Boot Version

Identify the latest stable version of Spring Boot available. You can check the official Spring Boot website, release notes, or use Maven Central or Gradle's repository to find the latest version

### 2. Update the Parent Version in Your Project

#### Maven

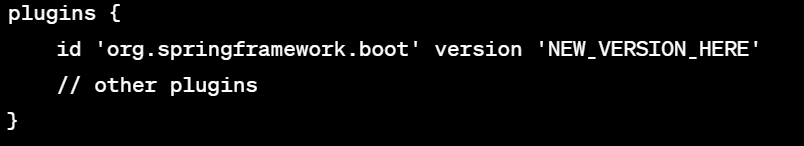
In your Maven `pom.xml`, update the `<parent>` section with the new Spring Boot version:



Replace `NEW\_VERSION\_HERE` with the latest Spring Boot version.

#### Gradle

If you're using Gradle with the Spring Boot plugin, update the `springBoot` plugin section in your `build.gradle`



Replace `NEW\_VERSION\_HERE` with the latest Spring Boot version.

### 3. Update Dependencies and Plugins

Check if any dependencies or plugins used in your project have specific versions tied to the Spring Boot version. Update these dependencies and plugins to versions compatible with the new Spring Boot version.

### 4. Run Tests and Perform Checks

After updating the Spring Boot version and related dependencies, run your tests and perform checks to ensure that your application functions as expected. Fix any compatibility issues or deprecated features encountered during this process.

### 5. Review Release Notes and Documentation

Review the release notes and documentation for the new Spring Boot version. Check for any breaking changes, deprecated features, or new features introduced that might affect your application.

### 6. Update Configuration (if needed)

Occasionally, new versions might introduce changes in default configurations or behavior. Update your application's configuration accordingly if any changes are needed to align with the new Spring Boot version.

### 7. Deploy and Validate

Once everything is updated and validated locally, deploy your application to a test environment or staging server. Perform thorough testing to ensure all functionality works as expected in a real-world environment.

### 8. Monitor and Address Issues

Monitor your application post-update to address any unexpected issues or performance problems that might arise due to the Spring Boot version update.

### Note:

Always consider taking backups or using version control to track changes when performing major updates like changing the Spring Boot parent version. This ensures you can rollback changes if necessary.

Lastly, keep in mind that major version updates might involve more complexities or breaking changes. It's crucial to test thoroughly and plan for potential issues during the update process.

**47)How to debug code in production environment when you find a bug or some issue?**

Debugging in a production environment requires a cautious approach to minimize disruption to live systems while investigating and resolving issues. Here are some steps and practices to consider:

### 1. Replicate and Identify the Issue

- Gather as much information as possible about the issue. Understand the steps or conditions that trigger the problem.

- Try to replicate the issue in a local development or staging environment if feasible. Having a reproducible scenario aids in diagnosing the problem without affecting the live system.

### 2. Logging and Monitoring

- Leverage logging frameworks (e.g., Log4j, SLF4J) to include detailed logs in your application. Ensure that logs capture relevant information, including timestamps, input/output data, error messages, and stack traces.

- Set up robust monitoring tools (e.g., New Relic, Prometheus, Grafana) to track system performance, errors, and other relevant metrics in real-time. Use alerts to notify when specific thresholds or errors occur.

### 3. Remote Debugging

- Some environments allow remote debugging capabilities. For Java applications, tools like Java Remote Debugging can attach a debugger to a running process, enabling you to inspect variables, step through code, and understand the state of the system.

- Be cautious when enabling remote debugging in production, as it may impact performance and security. Ensure it's only accessible to authorized personnel and disabled once the debugging session is complete.

### 4. Feature Flags and Rollbacks

- Implement feature flags or toggles in your codebase. This allows you to turn off specific features or functionalities temporarily in a live environment if they're causing issues.

- If the issue arises after a recent deployment or change, consider rolling back to the previous stable version until the problem is resolved. Automated deployment strategies like blue-green deployments or canary releases can help mitigate risks during deployments.

### 5. Data Collection and Analysis

- Collect relevant data or diagnostic information while the issue is occurring. This might include thread dumps, heap dumps, database queries, network traffic analysis, etc.

- Analyze error reports, crash dumps, or any other automated reports generated by the system to gain insights into the issue.

### 6. Impact Analysis and Risk Assessment

- Assess the impact of potential fixes or changes on the live environment. Consider the risk associated with applying changes in a production environment. Always have a rollback plan in case the fix introduces unforeseen issues.

### 7. Communication and Collaboration

- Communicate with stakeholders, including support teams, developers, and relevant personnel, to provide updates about the investigation and potential resolution steps.

- Collaborate with team members to review code, discuss potential fixes, and perform thorough testing before deploying any changes.

### 8. Test in Staging and Deployment

- Once a potential fix or solution is identified, test it rigorously in a staging environment that closely resembles the production setup.

- Deploy the fixed version to a limited subset of users (canary deployment) or during off-peak hours to minimize the impact before applying it broadly.

### Remember:

- Always prioritize the stability and integrity of the live system when debugging in a production environment.

- Document your actions, changes made, and outcomes for future reference and analysis.

- Perform post-mortem analysis after resolving the issue to understand its root cause and prevent similar incidents in the future.

In any case, proceed with caution, and consider the potential impact on users and the system while debugging in a production environment.

**13)How many modules are available in spring?why we use Spring AOP?(not that important for you)**

**34)Spring Security module in Spring boot**

**42)For unit testing which framework you are using?**

Micro Services:

==============

**1)What is microservices? why we need it? what are the advantages and disadvantages of them?**

Microservices is an architectural approach to building applications as a collection of small, independent, and loosely coupled services. Each service is designed to perform a specific business function and operates as a separate process, communicating with other services typically through APIs.

### Why do we need Microservices?

#### Scalability:

- Microservices allow for scaling individual components of an application independently based on demand. This enables efficient resource utilization and better handling of varying workloads.

#### Flexibility and Agility:

- They promote agility in development by enabling teams to work on different services independently, using different technologies or programming languages suitable for each service.

#### Fault Isolation and Resilience:

- With services running independently, a failure in one service doesn’t necessarily bring down the entire system. It promotes fault isolation, ensuring that failures are contained within specific services.

#### Continuous Deployment and DevOps:

- Microservices align well with DevOps practices and continuous deployment, allowing faster and more frequent releases of smaller, focused services.

### Advantages of Microservices:

1. Scalability: Services can be scaled independently to handle specific loads, optimizing resource utilization.

2. Flexibility and Technology Diversity: Different services can use different technologies or languages suited for their specific tasks.

3. Isolation and Fault Tolerance: Failures are contained within services, reducing the risk of system-wide failures.

4. Rapid Development: Independent teams can work on services concurrently, speeding up development cycles.

5. Enhanced Maintenance and Updates: Easier to maintain and update individual services without affecting the entire system.

### Disadvantages of Microservices:

1. Complexity: Managing a distributed system introduces complexity in deployment, monitoring, and inter-service communication.

2. Increased Overhead: The overhead of managing multiple services, inter-service communication, and data consistency can be higher.

3. Network Latency: Inter-service communication over the network might introduce latency compared to in-process calls.

4. Consistency Challenges: Ensuring consistency across services, especially in distributed transactions, can be challenging.

5. Operational Overhead: Requires additional efforts in setting up monitoring, logging, and managing deployments.

### Conclusion:

Microservices offer several benefits, especially in large and complex systems where flexibility, scalability, and rapid development are crucial. However, their adoption also comes with challenges, particularly in managing the increased complexity and ensuring effective communication and coordination between services. The decision to adopt microservices should consider the specific needs and constraints of the application and organization.

**2)what is API Gateway?which API gateway you are using in your application**

An API Gateway is a server or service that acts as an entry point into a system composed of microservices or backend services. It's a centralized point for managing, securing, and routing incoming client requests to the appropriate microservices.

### Functions of an API Gateway:

1. Routing: Directs client requests to the corresponding microservice based on the request's endpoint or parameters.

2. Load Balancing: Distributes incoming traffic across multiple instances of a service to optimize performance and reliability.

3. Authentication and Authorization: Handles authentication, verifies access tokens, and enforces authorization policies for incoming requests.

4. Security: Provides security mechanisms like encryption, rate limiting, and protection against common attacks (e.g., DDoS, SQL injection).

5. Monitoring and Logging: Collects metrics, logs, and monitoring data to track performance, errors, and usage patterns.

### Examples of API Gateways:

1. NGINX: Though not exclusively an API Gateway, NGINX is often used as a lightweight gateway for routing and load balancing purposes.

2. Amazon API Gateway: A fully managed service provided by AWS for building, deploying, and managing APIs at scale.

3. Kong: An open-source API Gateway built on top of NGINX and designed for microservices architectures.

4. Apigee (Google Cloud): A full-featured API management platform that includes an API Gateway among its capabilities.

5. Spring Cloud Gateway: An API Gateway built on top of Spring Framework by the Spring Cloud community.

As an AI, I don't have an application or use a specific API Gateway myself. However, organizations often choose API Gateways based on various factors like their existing infrastructure, cloud provider, scalability needs, security requirements, and compatibility with other services and tools in their ecosystem. The choice of an API Gateway often depends on the specific use case, budget, and technical preferences of the development team or organization.

**3)what is load balancing>how you have implemented in you application/project?**

Load balancing is a technique used in computing to distribute incoming network traffic across multiple servers or resources. Its primary goal is to optimize resource utilization, maximize throughput, minimize response time, and avoid overloading any single resource.

### How Load Balancing Works:

1. Traffic Distribution: When a request arrives at the load balancer, it decides which server or resource in a pool is best suited to handle the request. This decision is often based on predefined algorithms like round-robin, least connections, weighted distribution, etc.

2. High Availability: Load balancers ensure high availability by distributing traffic across multiple servers. If one server fails, the load balancer redirects traffic to healthy servers, minimizing downtime.

### Implementation in Applications/Projects:

Load balancing can be implemented at various levels within an application or infrastructure:

1. Application-Level Load Balancing:

- In a microservices architecture, an API Gateway or service mesh may handle load balancing between different microservices.

- Within an application, a load balancer might distribute requests across multiple instances of the application running on different servers or containers.

2. Network-Level Load Balancing:

- At the network level, hardware or software-based load balancers distribute traffic across multiple servers or data centers.

- Cloud providers often offer load balancing services (like AWS Elastic Load Balancing, Google Cloud Load Balancing) to balance traffic across instances or services deployed on their platforms.

3. DNS Load Balancing:

- DNS-based load balancing involves distributing traffic based on DNS responses that provide different IP addresses. For example, a DNS server might rotate through a list of IP addresses associated with a domain.

### In My Application/Project (Hypothetical Scenario):

Let's say I'm building a web application deployed on a cloud platform. To implement load balancing:

- I might use the cloud provider's load balancing service to distribute incoming HTTP/HTTPS traffic across multiple instances of my web application running on virtual machines or containers.

- The load balancer would handle routing requests to the least busy instance based on predefined algorithms (e.g., round-robin, least connections).

- Additionally, I might incorporate auto-scaling capabilities to dynamically add or remove instances based on traffic patterns, ensuring optimal performance during high traffic periods.

Load balancing is crucial for ensuring high availability, scalability, and efficient resource utilization in modern applications, especially those deployed across distributed environments or cloud infrastructures.

**5)how can you handle fault-tolerence situation in your application?which circuit breaker you have used in your application.**

In a microservices architecture, handling fault tolerance is crucial to ensure that the system remains robust and resilient in the face of failures. One common approach to handle faults is by using a Circuit Breaker pattern.

### Circuit Breaker Pattern:

The Circuit Breaker pattern is a design pattern used to prevent cascading failures in distributed systems. It monitors calls to remote services and, if a service is failing repeatedly, the circuit breaker trips, stopping further calls to that service. This prevents overloading the failing service and allows it to recover.

### Circuit Breaker Implementation:

In Java-based applications, a popular library used for implementing the Circuit Breaker pattern is Hystrix, which was part of Netflix OSS but is now in maintenance mode. It provides functionalities for fault tolerance, latency and fault injection, and isolation of services.

However, after Netflix discontinued Hystrix, the industry has moved towards other libraries and frameworks, such as:

- Resilience4j: A lightweight fault tolerance library inspired by Hystrix but with more modern features and support for functional programming.

- Sentinel: A powerful and versatile library for flow control, circuit breaking, and adaptive system protection in distributed systems.

- Spring Cloud Circuit Breaker: Integrates various circuit breaker implementations (including Resilience4j and Hystrix) into Spring Cloud projects.

### Handling Fault Tolerance in My Application:

As an AI language model, I don't have a personal application or direct use of specific libraries. However, in hypothetical scenarios, I might consider using Resilience4j or Spring Cloud Circuit Breaker for handling fault tolerance.

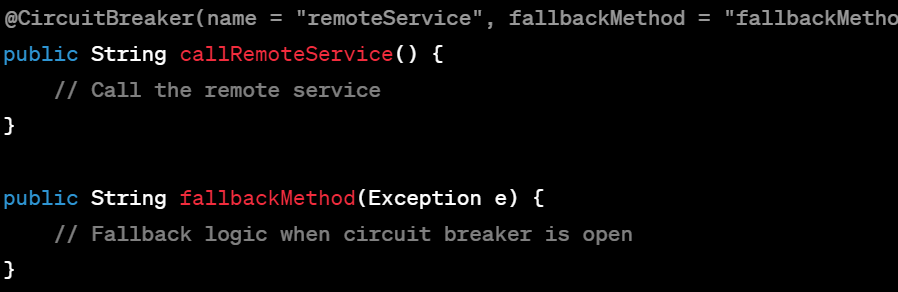
For example, in a Spring Boot microservices architecture, I would integrate Resilience4j by adding its dependencies and configuring circuit breakers using annotations or configuration. Then, annotate or wrap remote service calls with circuit breaker annotations or code to provide fault tolerance.

### Steps for Using Resilience4j:

1. Add Resilience4j dependencies to your project.

2. Configure circuit breakers using Resilience4j's annotations or programmatically.

3. Apply circuit breaker logic to remote service calls.



This setup allows the circuit breaker to intercept and handle failures when calling the remote service, invoking the fallback method when the circuit is open due to repeated failures.

Selecting the right circuit breaker library depends on the specific requirements, compatibility with your technology stack, and community support for ongoing maintenance and updates.

**6)what is Configserver in Microservices?what is the use of it?**

In a microservices architecture, a Config Server is a centralized component that manages the configuration settings of all the microservices in the system. It stores configuration details, such as properties, environment-specific settings, feature toggles, or any other configuration required by the services.

### Use of Config Server:

1. Centralized Configuration Management:

- Config Servers store configurations in a central location, allowing easy management and modification of configurations for multiple services without redeploying them.

2. Dynamic Configuration Updates:

- Services can fetch their configuration details from the Config Server at runtime. This enables dynamic updates to configurations without restarting or redeploying services.

3. Environment-specific Configuration:

- Config Servers allow for managing environment-specific configurations (e.g., dev, test, prod) and help ensure consistency across different deployment environments.

4. Versioning and History:

- Config Servers often support versioning and history tracking of configurations, allowing rollbacks to previous configurations if needed.

5. Security and Encryption:

- They provide mechanisms for managing sensitive information like passwords or API keys, enabling encryption or secure storage of such data.

### Example: Spring Cloud Config Server:

In a Spring Boot/Spring Cloud-based microservices setup, Spring Cloud Config Server is a popular choice. It provides a server-side component for centralized externalized configuration management. Here's a simplified example:

1. Setup Config Server:

- Create a Spring Boot application and include `spring-cloud-config-server` dependency.

- Configure the server to connect to a version-controlled repository (e.g., Git) where service configurations are stored.

2. Service Configuration Retrieval:

- Other microservices (clients) use `spring-cloud-starter-config` dependency and configure their `bootstrap.properties` or `bootstrap.yml` to specify the Config Server's location.

3. Fetching Configurations:

- Services fetch their configurations (e.g., properties, YAML files) from the Config Server at startup.

4. Dynamic Updates:

- Configurations can be updated in the central repository (e.g., Git), and services can fetch the updated configurations dynamically without restarting.

### Benefits of Config Server:

- Centralized Management: Simplifies configuration management across multiple services.

- Dynamic Updates: Enables real-time changes to configurations without service restarts.

- Consistency and Versioning: Ensures consistency and allows tracking configuration changes.

- Security and Encrypted Storage: Supports secure management of sensitive data.

Config Servers play a critical role in maintaining flexibility, consistency, and manageability in a microservices architecture by centralizing configuration management and enabling dynamic updates across the services.

**8)how can you access one service's API's from other service using FeignClient or Webclient?]**

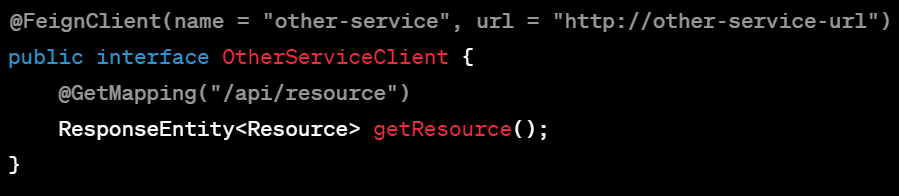
In a microservices architecture, services often need to communicate with each other via APIs. Two common approaches in Java-based applications for consuming APIs between services are using `FeignClient` or `WebClient`.

### Using FeignClient:

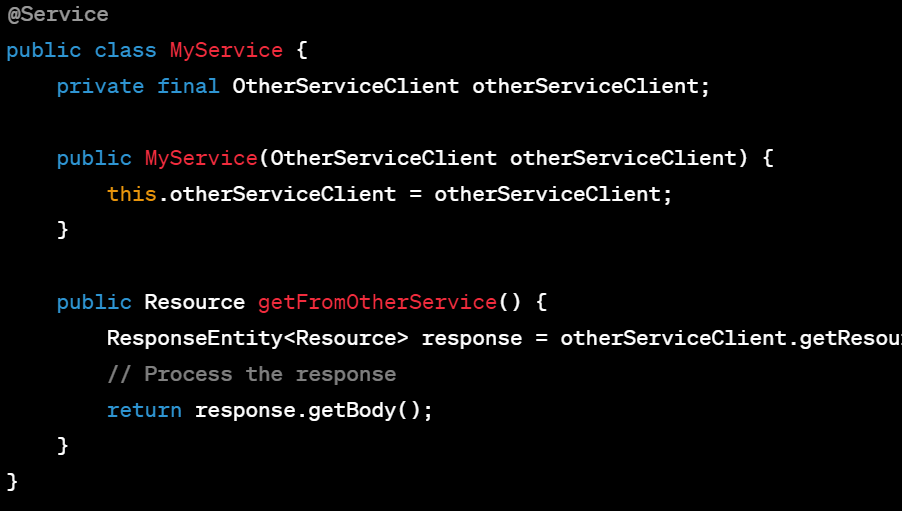
`FeignClient` is a declarative HTTP client provided by Spring Cloud that simplifies making HTTP requests to other services.

1. Dependency: Add the `spring-cloud-starter-openfeign` dependency to your project.

2. Create Feign Interface: Define an interface annotated with `@FeignClient` and specify the target service's name and endpoint details.



3. Inject Feign Interface: Inject the Feign interface into your service and use it to make API calls to the target service.

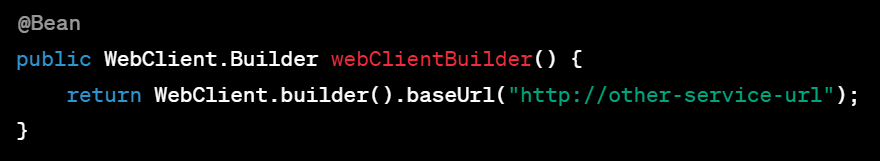


### Using WebClient:

`WebClient` is a non-blocking, reactive web client introduced in Spring WebFlux, suitable for asynchronous and reactive programming.

1. Dependency: Add the appropriate dependencies for Spring WebFlux and WebClient to your project.

2. Create WebClient Instance: Instantiate a `WebClient` bean with the target service's base URL.



3. Use WebClient to Make Requests: Inject the `WebClient.Builder` bean and use it to create a `WebClient` instance to make API calls.



Both `FeignClient` and `WebClient` provide different levels of abstraction and functionality for consuming APIs between services. The choice between them might depend on factors like familiarity with the libraries, project requirements, and performance considerations.

**9)Is maintanance really good with Microservices?if so how?**

Maintaining microservices can bring several advantages, although it's important to acknowledge that while microservices offer flexibility and scalability, they also introduce complexities that need careful management. Here's how maintenance can be advantageous in a microservices architecture:

### Granular Updates and Scaling:

- Modularity: Microservices allow updating or scaling individual components without affecting the entire system. This modularity facilitates easier maintenance as changes are confined to specific services.

### Independent Development and Deployment:

- Parallel Development: Teams can work independently on different services, enabling faster development cycles.

- Frequent Updates: Services can be updated and deployed independently, allowing faster delivery of new features or bug fixes.

### Isolation of Failures:

- Fault Containment: Failures in one service typically don't cascade to other services, improving fault isolation. This makes diagnosing and fixing issues more manageable.

### Technology Flexibility:

- Technology Adoption: Different services can use technologies best suited for their specific purposes, enabling the adoption of newer, more efficient technologies without impacting the entire system.

### Challenges to Consider:

- Increased Complexity: Managing multiple services introduces complexities in deployment, monitoring, and communication between services.

- Distributed Systems Complexity: Handling network communication and ensuring data consistency across services can be challenging.

- Operational Overhead: Requires robust monitoring, logging, and management tools to handle the increased operational complexity.

### Best Practices for Maintenance in Microservices:

1. Robust Monitoring and Logging: Implement comprehensive monitoring and logging to track the health and performance of individual services.

2. Automated Deployment and CI/CD: Set up automated deployment pipelines and continuous integration/continuous deployment (CI/CD) processes to streamline updates.

3. Service-Level Agreements (SLAs): Establish SLAs for services to define expectations for availability, response times, and reliability.

4. Resilience and Fault Tolerance: Design services with fault tolerance in mind. Implement resilience patterns like circuit breakers, retries, and fallback mechanisms.

5. Versioning and Documentation: Maintain versioning of services and APIs, and ensure comprehensive documentation for each service.

6. Testing Strategies: Employ thorough testing practices, including unit tests, integration tests, and end-to-end tests to ensure service reliability.

Microservices offer advantages in terms of agility, scalability, and independent development. However, maintaining a microservices-based system requires careful consideration of the increased complexity and the adoption of robust practices and tools to effectively manage and address the challenges they present.

**10)any design pattern you are followed/implemented in Microservices?**

In a microservices architecture, various design patterns are commonly used to address specific challenges related to distributed systems, scalability, resilience, and communication between services. Here are several design patterns often implemented in microservices:

### 1. Service Registry & Discovery:

- Pattern: Service Registry and Service Discovery pattern using tools like Eureka, Consul, or etcd to register services and allow dynamic service discovery.

### 2. API Gateway:

- Pattern: API Gateway pattern for a centralized entry point to handle client requests, route them to appropriate services, and provide API composition.

### 3. Circuit Breaker:

- Pattern: Circuit Breaker pattern using libraries like Hystrix, Resilience4j, or Spring Cloud Circuit Breaker to prevent cascading failures and manage faults between services.

### 4. Event-Driven Architecture:

- Pattern: Event-Driven Architecture pattern using tools like Kafka or RabbitMQ to implement asynchronous communication between services via events or messages.

### 5. Saga Pattern:

- Pattern: Saga pattern to manage distributed transactions across multiple services, ensuring eventual consistency without relying on two-phase commit protocols.

### 6. Bulkhead Pattern:

- Pattern: Bulkhead pattern to isolate failures by partitioning services into separate thread pools or resource pools, preventing one service's failure from affecting others.

### 7. Database per Service:

- Pattern: Database per Service pattern to have dedicated databases for each microservice, ensuring loose coupling and independence but potentially requiring careful data consistency management.

### 8. Backend for Frontend (BFF):

- Pattern: Backend for Frontend pattern to create specialized APIs for specific client types or user interfaces, tailoring responses for optimal client experience.

### 9. Retry Pattern:

- Pattern: Retry pattern to handle transient failures by implementing retry logic when calling remote services or APIs.

### 10. Strangler Pattern:

- Pattern: Strangler pattern for migrating from monolithic systems to microservices gradually, by gradually replacing functionalities in a controlled manner.

### Implementation Considerations:

- Adaptation to Business Context: Apply patterns based on the specific needs, complexities, and scale of the application.

- Tool and Technology Selection: Choose tools and libraries that align with the selected patterns and the technology stack of the microservices architecture.

- Documentation and Education: Ensure teams understand the patterns being employed and their respective roles in the architecture.

Implementing these patterns in a microservices-based system helps address various challenges associated with distributed architectures, ensuring scalability, resilience, and maintainability while accommodating the system's evolving needs.

**11)How can you handle messaging/events in microservices?**

Handling messaging or events in microservices involves implementing patterns and tools for asynchronous communication between services. Event-driven architectures facilitate loosely coupled systems where services communicate through events or messages. Here's how messaging/events can be managed in microservices:

### 1. Message Broker or Event Bus:

- Use a message broker like Kafka, RabbitMQ, or Amazon SQS as a centralized platform to publish and subscribe to events/messages.

### 2. Event-Driven Communication:

- Implement an event-driven approach where services produce events and other services subscribe to these events based on interest.

### 3. Event Schema and Contract:

- Define clear schemas for events/messages to ensure consistency and interoperability between services.

### 4. Event Sourcing:

- Use event sourcing to store the state of a system as a sequence of events. Services can consume and react to these events to maintain their own state.

### 5. Publish-Subscribe (Pub/Sub):

- Use Pub/Sub messaging patterns where publishers publish events to topics, and subscribers receive events from these topics.

### 6. Reliable Messaging:

- Ensure message delivery guarantees using acknowledgment mechanisms (like acknowledgments in RabbitMQ or Kafka's message offsets).

### 7. Event Choreography or Orchestration:

- Implement event choreography to handle business processes by allowing services to react to events and orchestrate complex workflows.

### 8. Error Handling and Dead Letter Queues:

- Set up dead letter queues or error handling mechanisms to handle failed events or messages, ensuring they're logged or retried appropriately.

### 9. Integration Patterns:

- Employ integration patterns like event sourcing, CQRS (Command Query Responsibility Segregation), and event-driven architecture to manage communication and state changes.

### Example Workflow:

1. Service A produces an event when a specific action occurs (e.g., Order Placed).

2. The event is published to a message broker (e.g., Kafka topic or RabbitMQ queue).

3. Service B and Service C, interested in Order Placed events, subscribe to the topic/queue and consume/process these events independently.

4. Services react to the event, updating their own state or triggering further actions (e.g., Order Processed, Payment Initiated).

### Considerations:

- Message Durability: Ensure message persistence and durability to prevent data loss in case of failures.

- Scaling and Performance: Choose message brokers that support scalability and performance requirements.

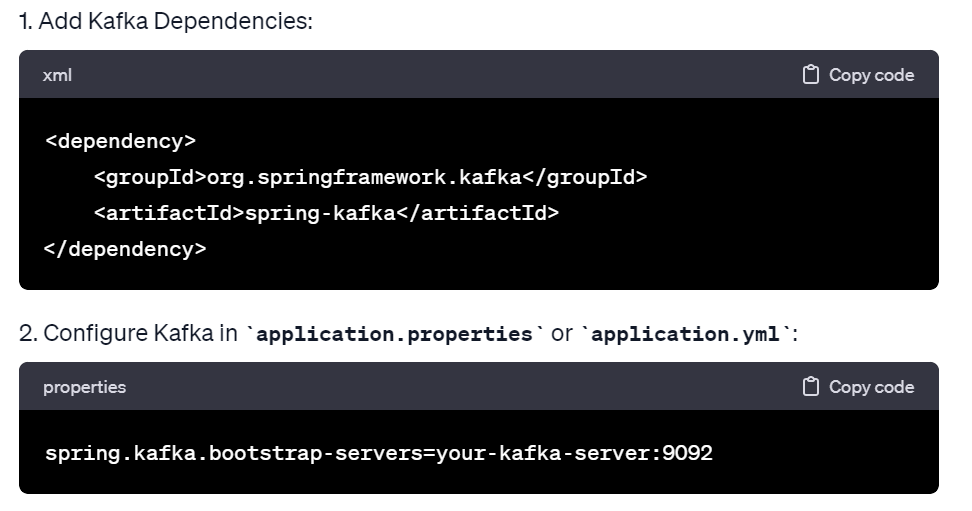
- Monitoring and Logging: Implement monitoring and logging to track message processing, ensuring reliability and troubleshooting.

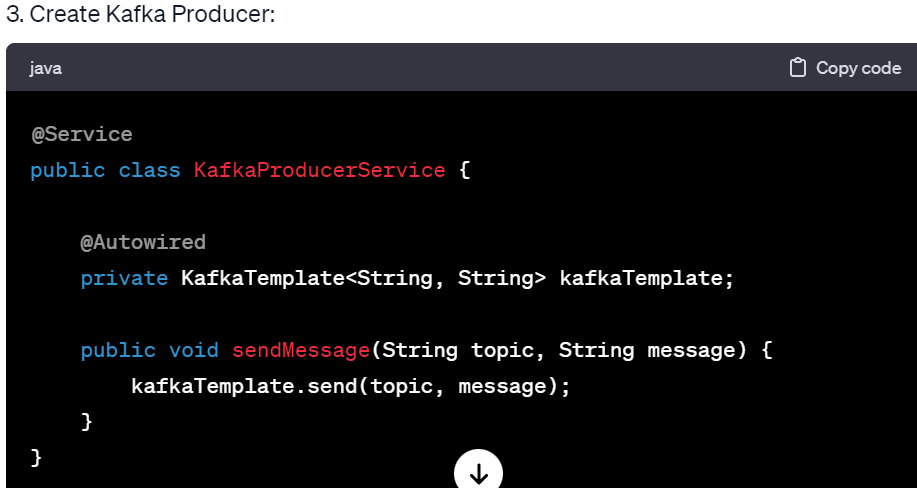
Implementing messaging/events in microservices enables asynchronous communication, decoupling services, improving scalability, and facilitating better handling of complex interactions and workflows in distributed systems.

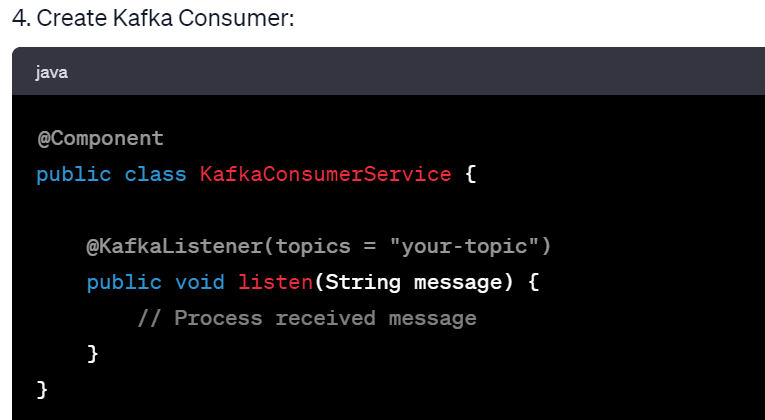
**12)how to integrate kafka/RabbitMQ with springboot in microservices?**

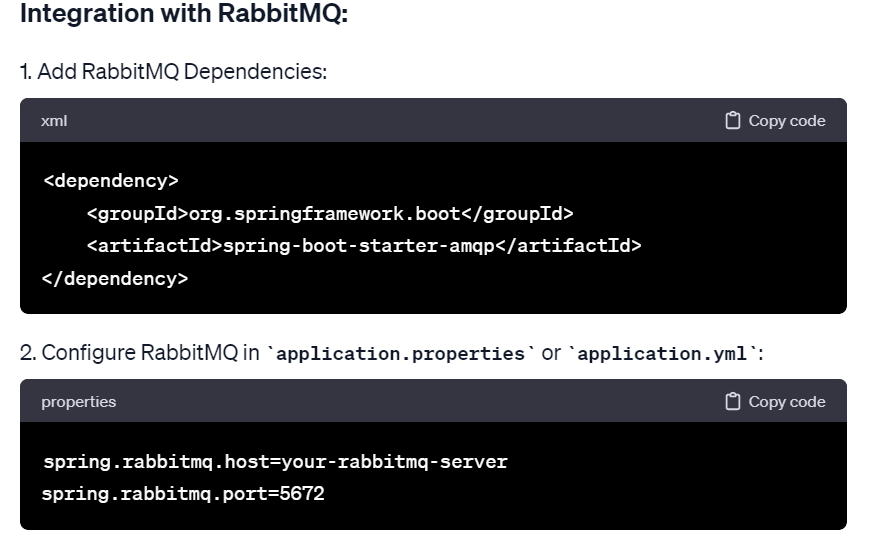
Integrating Kafka or RabbitMQ with Spring Boot in a microservices architecture involves using Spring's support for messaging frameworks to enable communication between services via message brokers. Here's a basic guide for integrating Kafka or RabbitMQ with Spring Boot:

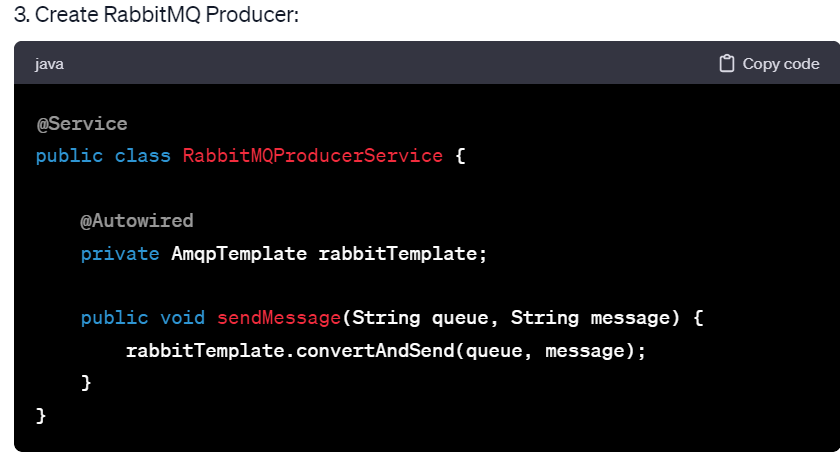
### Integration with Kafka:

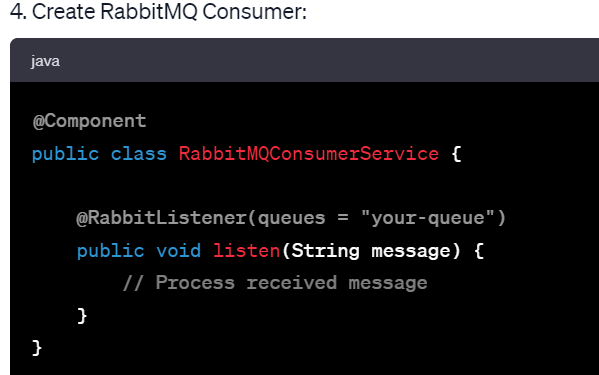












### Additional Configuration:

- For both Kafka and RabbitMQ, you may need to configure serializers/deserializers, security settings, and additional properties based on your setup.

- Ensure proper error handling, logging, and monitoring for message processing and handling.

These examples provide a basic setup for integrating Kafka or RabbitMQ with Spring Boot. Actual configurations may vary based on specific requirements, security settings, and the complexity of your microservices architecture.

**16)How can you deploy microservice in any platform?**

Deploying microservices on various platforms involves several steps depending on the platform being used. Here's a general outline of the deployment process:

### 1. Choose Deployment Platform:

- On-Premises Servers: Directly deploy services on physical or virtual machines within your organization's infrastructure.

- Cloud Platforms (e.g., AWS, Azure, Google Cloud): Use cloud services to deploy and manage microservices.

- Container Orchestration (e.g., Kubernetes, Docker Swarm): Utilize container orchestration platforms to manage and deploy services in containers.

### 2. Build and Package Services:

- Build Microservices: Compile, package, or containerize each microservice. This might involve using build tools like Maven, Gradle, or Docker.

### 3. Configure Infrastructure:

#### On-Premises or Cloud Platforms:

- Provision Servers/Instances: Set up servers or virtual machines with the necessary OS, networking, and security configurations.

#### Container Orchestration Platforms:

- Deploy Container Orchestration: Set up the container orchestration platform (e.g., Kubernetes cluster) and configure nodes, networking, and security.

### 4. Define Deployment Configuration:

#### On-Premises or Cloud Platforms:

- Manual Deployment: Copy and deploy microservice artifacts to the designated servers or instances.

#### Container Orchestration Platforms:

- Container Deployment: Define deployment configurations using YAML files (e.g., Kubernetes Deployment YAML) or orchestrator-specific tools.

### 5. Deploy Microservices:

- Manual Deployment: Execute deployment scripts or commands to start microservices on the designated servers or instances.

- Container Orchestration: Apply deployment configurations to deploy microservices into containers managed by the orchestrator.

### 6. Monitor and Manage Deployed Services:

- Monitoring: Implement monitoring tools (e.g., Prometheus, New Relic) to track microservice health, performance, and logs.

- Scaling and Maintenance: Manage service scaling, updates, and maintenance to ensure optimal performance and reliability.

### 7. Automate Deployment (Optional):

- CI/CD Pipelines: Set up CI/CD pipelines (e.g., Jenkins, GitLab CI/CD) to automate the build, testing, and deployment of microservices.

### 8. Post-Deployment Testing:

- End-to-End Testing: Conduct tests to verify the functionality and interaction of deployed microservices.

### Considerations:

- Networking and Security: Configure networking, firewalls, and access controls based on deployment environment and security requirements.

- Health Checks and Recovery: Implement health checks and recovery mechanisms to handle service failures and ensure high availability.

The deployment process varies based on the chosen platform, infrastructure, and deployment strategy. Automating deployments, ensuring robust monitoring, and maintaining proper documentation are key to successful microservices deployment in any platform.

**18)Explain microservices components?**

like - Eureka server, Eureka client,API Gateway, Ribbon,Hystrix and Hystrix dashboard(it is now deprecated - Resilience4J,Sentinel,Istio, EnvoyProxy)

**19)What is Cloud ConfigServer in microservices?**

Ans: which is externalizing all the repository code base by connecting github.for this we will write bootstrap.yml file on every service.

**20)how can you provide security your microservice?**

**21)what are some annotations you have used in microservices?**

@EnableEurekaServer,@EnableDiscoveryClient or @EnableEurekaClient,@Loadbalanced,@EnableHystrix and @EnableHystrixDashboard

**22)Where you can implement security in microservices environment?**

at API gateway level or for each and every service level

**4)How can you register your Service in Service Registry?**

Refer return notes in book

**7)how can you deploy a service into a server/instance/node?**

**14)which logging mechanism you have followed in your project?And is which Monitoring tool you have used?**

REST API:

========

1)What are the HTTP methods you have used in your project?

2)difference between 400vs404 and 401vs403 and 200vs201?

3)diference between put and patch methods?

4)How can you test your REST API's?

5)what is nullipotent HTTP method in rest?

Ans:GET is nullipotent

Hibernate or Springboot JPA:

===========================

1)differences between get() and load() methods in Hibernate

2)differences between save() vs persist() & SessionFactory vs Session

3)Hibernate/JPA Mappings?

4)what are the important interfaces you have used in Hibernate/JPA

5)What are the some important Annotations you have used in Hibernate/Spring data JPA?

6)how can you make connections for Mysql8 and Oracle11g databases?

7)Differnces between @JPARepository vs @CRUDRepository?

8)How many tables will be created in @ManytoMany relationship?

Ans: For sure 3 tables

Suggestion: Please read about @JoinTable and @JoinColumn--- like what happens if we use @JoinTable(creates extra table) and @JoinColumn(create extra column in same table)?

9)What if in-built JPA methods are not enough to create tables or map your entities or fetch data from database?

Ans: read about @Query annotation usage.

10)What is the use of @Transactional annotaion?how can you manage transactions in your application?what are isolation and propagation levels you use?

11)what are Hibernate isolation levels ? How can achieve them?

Ans: read about ACID properties and READ\_COMMITTED, READ\_UNCOMMITTED, REPEATABLE\_READ and SERIALIZABLE

12)Find second highest salary from the Employee table?

Ans: select salary from employee order by Desc salary limit(1,1); ///for third limit(2,1)

13)what is the difference between Inner join and Left join? can you write a join query to fetch the data from 3 tables?

14)what is indexing in table

15)what is the difference between drop vs truncate vs delete in sql queries?

16)What are some important interfaces you have used in your project?

Devops & Cloud:

==============

1)which version control system tool you are using?

Ans:Git/Gitlab/bitbucket

2)What is Jenkins?so you know how to configure pipelines by integrating/connecting git? How build mechanism is happening in your project?

like periodically(every one hour or two hours) or every commit

3)do you have an idea on Docker or Mocker?

4)which cloud you are using to deploy your micriservices/application.

5)what is the difference between git fetch vs git pull

git pull origin bakwas

git pull = git fetch + git merge

Git Fetch : The git fetch command fetches the latest changes from a remote repository but does not automatically merge or apply those changes to your current working branch. It updates your local repository's remote-tracking branches, allowing you to see what changes have been made in the remote repository without affecting your working directory.

Git Pull: The git pull command fetches the latest changes from a remote repository and then automatically merges them into your current working branch. It combines the actions of git fetch and git merge into a single command. If you have uncommitted changes, Git will attempt to merge your changes with the remote changes.

6)How can you publish your artifacts/packages once build completes?

7) remaining are based on your experience---