

## oGlimpse Questions:

A Java constructor cannot be abstract, static, final, and synchronized

Restrictions for the static method

There are two main restrictions for the static method. They are:

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

Why can we not override static method?

It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

## why spring is introduced –

Spring Framework was introduced to address the challenges faced by Java developers in building enterprise-level applications. Here's why it was introduced:

1. **Simplification of Enterprise Java Development:** Traditional Java EE (Enterprise Edition) development involved a lot of boilerplate code, configuration, and complexity. Spring aimed to simplify this process by providing a lightweight container, inversion of control (IoC), and aspect-oriented programming (AOP) capabilities, making development more streamlined and efficient.
2. **Loose Coupling:** Spring promotes loose coupling between components, making it easier to manage dependencies and swap out implementations without impacting other parts of the application. This enables better modularity and testability of code.
3. **Dependency Injection (DI):** Spring introduced the concept of DI, where objects define their dependencies through constructor arguments or setters. This allows for better decoupling of components and facilitates easier testing and maintenance.
4. **Aspect-Oriented Programming (AOP):** AOP is a programming paradigm that allows developers to modularize cross-cutting concerns such as logging, security, and transaction management. Spring integrates AOP seamlessly into the application, reducing code duplication and improving maintainability.

5. **Integration with Existing Technologies:** Spring provides integration with existing Java technologies like JDBC, Hibernate, JPA, and JMS, among others. This allows developers to leverage their existing skills and investments while benefiting from the features provided by Spring.
6. **Lightweight Container:** The Spring container is lightweight and non-invasive, unlike traditional Java EE containers. It doesn't rely on heavy-weight application servers, making it suitable for standalone applications or microservices.
7. **Support for Enterprise Features:** Spring offers support for various enterprise features such as transaction management, security, caching, messaging, and scheduling, making it well-suited for building robust, scalable, and secure applications.

## spring vs springboot

differences between spring vs springboot

ChatGPT

**ChatGPT**

Certainly! Let's delve into the key differences between Spring and Spring Boot:

### 1. **Purpose:**

- **Spring Framework:** It is a comprehensive framework for building Java applications, providing a wide range of features such as dependency injection, aspect-oriented programming, data access, transaction management, and more. Spring Framework requires more manual configuration and setup.
- **Spring Boot:** It is built on top of the Spring Framework and aims to simplify the process of building stand-alone, production-grade Spring-based applications. Spring Boot follows the convention over configuration principle, reducing the amount of boilerplate code and configuration required.

### 2. **Configuration:**

- **Spring Framework:** Developers need to manually configure various components and dependencies using XML or Java-based configuration.
- **Spring Boot:** It provides auto-configuration, which automatically configures beans based on the classpath and opinionated defaults. Developers can override or customize the auto-configuration as needed.

### 3. **Dependency Management:**

- **Spring Framework:** Developers need to manage dependencies manually, including resolving conflicts and versioning.
- **Spring Boot:** It simplifies dependency management through the use of starters, which are pre-configured sets of dependencies for common use cases. Starters help in quickly setting up the required dependencies for specific tasks, such as web applications, data access, security, etc.

#### 4. **Embedded Servers:**

- **Spring Framework:** Developers need to configure and deploy applications to external servlet containers like Tomcat, Jetty, or others.
- **Spring Boot:** It includes embedded servers like Tomcat, Jetty, or Undertow, allowing applications to be run as stand-alone JAR files without the need for external servlet containers.

#### 5. **Convention over Configuration:**

- **Spring Framework:** It requires explicit configuration for most components and features.
- **Spring Boot:** It follows the convention over configuration principle, providing sensible defaults and configurations out-of-the-box. Developers can override these defaults as needed, but in most cases, minimal configuration is required.

#### 6. **Microservices:**

- **Spring Framework:** While it can be used to build microservices, it requires more manual setup and configuration.
- **Spring Boot:** It is well-suited for building microservices due to its opinionated nature, embedded servers, and support for containerization technologies like Docker.

## What is IOC container?

1. **IoC Container:** An IoC container is a framework or mechanism that manages the lifecycle of objects and their dependencies. It creates, configures, and manages instances of classes (often referred to as beans) and injects their dependencies as needed. Examples of IoC containers in the Java world include the Spring Framework's ApplicationContext and Java EE's CDI (Contexts and Dependency Injection) container.

## What is Inversion of controller?

Inversion of Control (IoC) is a design principle in software engineering where the control of object creation, lifecycle management, and dependency injection is shifted from the

application code to a container or framework. Instead of components explicitly creating and managing their dependencies, IoC containers are responsible for resolving dependencies and injecting them into the components.

## What is dependency?

A dependency is when one object relies on another object to perform its function. For example, a class may depend on another class to provide database access or perform a specific task.

## What is Dependency Injection

**Dependency Injection (DI):** Dependency Injection is a technique used in IoC, where the dependencies of a component are injected into it by an external entity (typically an IoC container) rather than the component creating them itself. This promotes loose coupling between components and makes the code more modular, testable, and maintainable.

## post vs put vs patch

### 1. POST:

- **Purpose:** POST is used to submit data to a specific resource, often causing a change in state or side effects on the server.
- **Idempotent:** POST requests are not idempotent, meaning that performing the same POST request multiple times may result in different outcomes or create duplicate resources.
- **Use Cases:** POST is commonly used for creating new resources on the server, such as creating a new user account, submitting a form, or adding an item to a collection.

### 2. PUT:

- **Purpose:** PUT is used to update or replace a resource or create a new resource if it does not exist at the specified URI.

- **Idempotent:** PUT requests are idempotent, meaning that performing the same PUT request multiple times will have the same effect as performing it once.
- **Use Cases:** PUT is commonly used for updating existing resources with a complete representation of the resource, such as updating a user's profile or replacing an entire document in a document store.

### 3. PATCH:

- **Purpose:** PATCH is used to apply partial modifications to a resource, allowing clients to update specific fields or properties of the resource.
- **Idempotent:** PATCH requests are not guaranteed to be idempotent, meaning that the same PATCH request may produce different outcomes if applied multiple times.
- **Use Cases:** PATCH is commonly used when clients only need to update specific fields or properties of a resource, rather than replacing the entire resource. For example, updating the status of an order or changing the email address of a user.

In summary, POST is used for creating new resources, PUT is used for updating or replacing resources, and PATCH is used for applying partial modifications to resources. The choice between them depends on the specific requirements of the API and the desired semantics for creating, updating, or modifying resources.

## save saveall saveandflush

- **save:** Persists a single entity to the database. It either inserts a new record or updates an existing one.
- **saveAll:** Persists a collection of entities to the database. It iterates through the collection and persists each entity individually.
- **saveAndFlush:** Persists a single entity to the database and immediately flushes the changes to the database.

In general, **save** and **saveAll** are used for normal persistence operations, while **saveAndFlush** is used when you need to ensure that changes are immediately synchronized with the database.

## types of repository

In the context of Spring Data, which is commonly used with Spring Boot for data access, there are several types of repositories:

### 1. **CrudRepository:**

- **CrudRepository** provides CRUD (Create, Read, Update, Delete) operations for the entity it manages.
- It includes methods like **save**, **findById**, **findAll**, **delete**, etc.
- This is the most basic repository interface provided by Spring Data.

### 2. **PagingAndSortingRepository:**

- **PagingAndSortingRepository** extends **CrudRepository** and adds additional methods for pagination and sorting of results.
- It includes methods like **findAll(Pageable pageable)**, **findAll(Sort sort)**, etc., to fetch results in a paginated or sorted manner.

### 3. **JpaRepository:**

- **JpaRepository** extends **PagingAndSortingRepository** and adds JPA-specific features.
- It includes methods for flushing changes to the database (**flush()**), deleting entities in a batch (**deleteInBatch()**), etc.
- **JpaRepository** is commonly used in applications that use JPA (Java Persistence API) for data access.

## pagination and sorting

Pagination and sorting are techniques used in software development, particularly in the context of databases and APIs, to manage and present large sets of data in a more manageable and user-friendly way:

### 1. **Pagination:**

- Pagination is the process of dividing a large set of data into smaller, discrete pages.

- Instead of loading and displaying all the data at once, pagination allows users to view one page of data at a time, typically with a limited number of records per page.
- Pagination is commonly used in web applications, where it enhances performance by reducing the amount of data transferred between the server and the client, and improves usability by providing a more structured and navigable interface.

## 2. **Sorting:**

- Sorting is the process of arranging data in a specific order based on one or more criteria.
- Sorting can be performed in ascending or descending order, depending on the desired result.
- Sorting is commonly used to present data in a more organized and meaningful way, making it easier for users to find the information they need.
- Sorting can be applied to various types of data, such as text, numbers, dates, and more.

In the context of database queries and APIs:

- Pagination is often achieved by using SQL queries with **LIMIT** and **OFFSET** clauses in relational databases, or similar mechanisms in other types of databases.
- Sorting is achieved by specifying the **ORDER BY** clause in SQL queries, which sorts the result set based on one or more columns in ascending or descending order.

For APIs:

- Pagination is typically implemented using query parameters to specify the page number and the number of records per page.
- Sorting is implemented using query parameters to specify the sorting criteria (e.g., field name) and the sort order (ascending or descending).

Overall, pagination and sorting are essential techniques for managing large datasets and improving the user experience in applications by presenting data in a structured, navigable, and meaningful way.

# springprofile,actuator->health,memory,app is up or not

## generation strategies

### 1. Identity/Auto-increment:

- In this strategy, the database generates primary key values automatically when a new row is inserted into the table.
- Typically used in databases like MySQL, PostgreSQL, SQL Server, etc., where an auto-incrementing column is defined as the primary key.

### 2. Sequence:

- Some databases, like Oracle and PostgreSQL, use sequences to generate unique identifier values.
- A sequence is a database object that generates a sequence of unique numbers, which can be used as primary key values for different tables.

### 3. Table:

- In this strategy, a separate table is used to maintain a counter for generating unique identifier values.
- Each time a new row is inserted, the application fetches a new value from this table and uses it as the primary key.

### 4. UUID/GUID:

- Universally Unique Identifiers (UUIDs) or Globally Unique Identifiers (GUIDs) are generated using algorithms that guarantee uniqueness across different systems and databases.
- UUIDs/GUIDs are typically generated by the application code and assigned to entities before they are persisted to the database.

### 5. Assigned:

- In this strategy, the application code assigns primary key values to entities before persisting them to the database.
- The application is responsible for ensuring the uniqueness of assigned primary key values.



In Hibernate, you can specify the generation strategy for primary keys using annotations or XML configuration. For example, `@GeneratedValue(strategy = GenerationType.IDENTITY)` specifies the identity/auto-increment strategy, while `@GeneratedValue(strategy = GenerationType.SEQUENCE)` specifies the sequence strategy.

The choice of generation strategy depends on factors such as the database being used, performance considerations, portability, and requirements for uniqueness and predictability of primary key values.

## default methods in functional interface—

In summary, default methods in functional interfaces provide a way to extend interfaces with new methods without breaking existing implementations, making it easier to evolve APIs over time. They are a powerful feature introduced in Java 8 to enhance the flexibility and extensibility of interfaces.

## backward compatibility

Backward compatibility, often abbreviated as "backwards compatibility," is a concept in software development and engineering that refers to the ability of a system, product, or component to maintain compatibility with earlier versions or interfaces, ensuring that existing functionality continues to work as expected even after changes or updates are made.

Here are some key points about backward compatibility:

1. **Preservation of Functionality:** Backward compatibility ensures that existing features and functionalities of a software system remain intact and operational when newer versions are introduced. This means that applications or components built using older versions of the software should continue to function without requiring modifications or updates.
2. **Smooth Upgrades:** Backward compatibility facilitates smooth upgrades and migrations to newer versions of software. Users can adopt new versions of software gradually, without the need to immediately replace or update all components of their systems.
3. **API Stability:** In the context of APIs (Application Programming Interfaces), backward compatibility ensures that changes to the API do not break existing client code. This means that existing applications that rely on the API can continue to function seamlessly even after updates or modifications are made to the API.
4. **Version Interoperability:** Backward compatibility allows systems with different versions or implementations to interoperate smoothly. For example, a client application built using an older version of a protocol should still be able to communicate with servers implementing newer versions of the protocol.

5. **Maintaining User Trust:** Backward compatibility is crucial for maintaining user trust and satisfaction. Users often rely on software systems to perform critical tasks, and any unexpected changes or disruptions to functionality can lead to frustration and loss of confidence in the software.
6. **Trade-offs:** While backward compatibility is important for ensuring continuity and smooth transitions, it can sometimes impose limitations on innovation and the introduction of new features. Maintaining backward compatibility may require compromises or additional effort to ensure that new functionalities do not interfere with existing ones.

In summary, backward compatibility is a fundamental principle in software development that emphasizes the importance of preserving existing functionality and ensuring seamless transitions when introducing changes or updates to software systems. It plays a significant role in user experience, system stability, and long-term sustainability of software products and platforms.

## two interfaces having same default method/

When two interfaces that a class implements each declare a default method with the same signature, it leads to a conflict known as "diamond problem" in multiple inheritance. When we try to compile **MyClass**, we'll encounter a compile-time error:

## Steps to Resolve Daimond problem

1. **Explicit Override:** In class C, you must override the `display()` method. You can call the desired implementation from either interface using `InterfaceName.super.methodName()`

## Springbootapplication

A Spring Boot application is a Java application that is built using the Spring Boot framework. It is the process of building and deploying Spring applications by providing opinionated defaults, embedded servers, auto-configuration, and production-ready features. It is widely used in the Java community for building web applications, microservices, and various other types of applications.

## how to create a bean

In the context of Spring Framework, a bean is simply an object that is managed by the Spring container. There are several ways to create a bean in Spring:

### 1. Using Component Scan and Stereotype Annotations:

- Annotate a class with `@Component`, `@Service`, `@Repository`, or `@Controller` (or their meta-annotations) to mark it as a Spring-managed bean.

- Spring will automatically detect and create a bean for any class annotated with one of these stereotypes during component scanning.
- Example:

java

Copy code

```
public class MyBean
```

## 2. Using XML Configuration:

- Define a bean configuration in an XML file (e.g., `applicationContext.xml`) using the `<bean>` element.
- Specify the class name, bean ID, and any dependencies required by the bean.
- Example:

xml

Copy code

```
id "myBean" class "com.example.MyBean"
```

## 3. Using Java Configuration:

- Create a configuration class annotated with `@Configuration`.
- Define bean creation methods within the configuration class using `@Bean` annotation.
- Example:

java

Copy code

```
public class AppConfig {
    public MyBean myBean() {
        return new MyBean();
    }
}
```

## 4. Using Factory Methods:

- Define a method that creates and returns an instance of the bean in a configuration class.
- Annotate the method with `@Bean`.
- Example:

java

Copy code

```
public class AppConfig {
    public MyBean myBean() {
        return new MyBean();
    }
}
```

## 5. Using Constructor Injection:

- Define a bean using constructor-based dependency injection, where dependencies are passed as constructor arguments.
- Spring will automatically create and initialize the bean with the specified dependencies.
- Example:

```
java
Copy code
public class MyBean {
    private
    public
    MyBean
    this
```

These are some of the common ways to create beans in a Spring application. The choice of method depends on the specific requirements and preferences of the application.

## can we create a method object//lambda

In Java, you cannot create an instance of a method directly as an object. However, you can obtain a reference to a method using reflection and then invoke it dynamically. Here's how you can do it:

```
import java.lang.reflect.Method;

public class Main {
    public static void main(String[] args) throws Exception {
        // Get a reference to the method using reflection
        Method method = MyClass.class.getMethod("myMethod");

        // Create an instance of the class containing the method
        MyClass obj = new MyClass();

        // Invoke the method on the object
        method.invoke(obj);
    }
}

class MyClass {
    public void myMethod() {
        System.out.println("Hello, World!");
    }
}
```

1. We obtain a reference to the `myMethod` method of the `MyClass` class using the `getMethod` method from the `Class` class. This method takes the name of the method as a string and can optionally take parameter types if the method is overloaded.

2. We create an instance of the `MyClass` class.
3. We invoke the `myMethod` method on the instance of `MyClass` using the `invoke` method from the `Method` class. This method takes the object on which to invoke the method (in this case, `obj`) and any arguments required by the method (none in this case, as `myMethod` has no parameters).

While you cannot create an instance of a method directly as an object, using reflection allows you to work with methods dynamically at runtime, which can be useful in certain scenarios such as building frameworks, libraries, or testing tools. However, reflection should be used judiciously due to its potential performance overhead and complexity.

## Streams

In Java, streams are a powerful and versatile feature introduced in Java 8 as part of the `java.util.stream` package. Streams provide a declarative way to process collections of elements, enabling functional-style operations such as filtering, mapping, reducing, and more. Streams are designed to facilitate concise and expressive code for data manipulation and transformation.

### list to map

converting a List to a Map provides flexibility and efficiency in organizing, accessing, and processing data, making it a valuable technique in Java programming, especially when dealing with key-value pairs or grouped data

### multivalue map

A `MultiValueMap`, as the name suggests, is a map that can hold multiple values for a single key. In Java, it is commonly used in scenarios where you need to associate multiple values with a single key, such as HTTP request parameters, query parameters, or headers.

In the Spring Framework, specifically in the `org.springframework.util.MultiValueMap` interface, a `MultiValueMap` represents a Map where each key can be associated with multiple values. This interface is commonly used in Spring MVC for handling HTTP requests and responses.

## Limit- >

**collections.groupingby->** The `Collectors.groupingBy()` method is a powerful feature introduced in Java 8 as part of the `java.util.stream.Collectors` class. It allows you to group elements of a stream into a Map based on a classification function.

`groupBy()` is a versatile collector that enables you to perform complex data transformations and aggregations based on specific criteria. It's widely used in Java streams for tasks such as data analysis, reporting, and organizing data.

## Transactional

In the context of databases and software systems, "transactional" refers to a set of operations that are treated as a single unit of work. These operations either succeed as a whole or fail as a whole, ensuring data consistency and integrity.

Here are some key aspects of transactional behavior:

1. **ACID Properties:** Transactions are designed to adhere to the ACID properties:

- **Atomicity:** All operations within a transaction are treated as a single indivisible unit. Either all operations are successfully completed, or none of them are applied.
- **Consistency:** Transactions maintain the consistency of the database, ensuring that it moves from one valid state to another valid state. The database remains in a consistent state before and after the transaction.
- **Isolation:** Transactions are isolated from each other, meaning that the intermediate states of one transaction are not visible to other transactions until it is completed.
- **Durability:** Once a transaction is committed, its changes are permanently saved to the database and cannot be lost, even in the event of a system failure.

2. **Transactional Support:** Most modern database management systems (DBMS) provide support for transactions. They offer mechanisms such as transaction logs, locks, and isolation levels to ensure transactional integrity and reliability.

3. **Transactional Control:** Transactions can be explicitly controlled and managed by the application code. Transactional control typically involves marking certain operations or methods as transactional, defining transaction boundaries, and specifying transactional attributes such as isolation level and propagation behavior.

4. **Transactional Behavior:** In application frameworks like Spring Framework, the `@Transactional` annotation is used to declare transactional behavior for methods or classes. When a method annotated with `@Transactional` is invoked, a transaction is automatically started before the method execution and

committed (or rolled back) after the method completes. If an exception occurs during the method execution, the transaction is rolled back to maintain data consistency.

Transactional behavior is essential in ensuring data integrity, reliability, and consistency in modern software systems, especially in scenarios where multiple operations need to be performed as a single unit of work. It helps maintain the reliability and correctness of data even in the presence of concurrent access, failures, or errors.

**Isolation** Isolation ensures that the intermediate states of one transaction are not visible to other transactions until it is completed, thereby preventing interference, conflicts, and data inconsistencies.

## custom annotation//

A custom annotation in Java is a marker or metadata added to classes, methods, fields, or other program elements to provide additional information or semantics about the element. Custom annotations are often used to convey instructions, configuration parameters, or constraints to the runtime environment, frameworks, or tools.

To create a custom annotation in Java, you define a new annotation type using the `@interface` keyword. Here's the basic syntax for creating a custom annotation:

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE) // Specify the target element types where the annotation
can be applied
public @interface MyAnnotation {
    // Define elements of the annotation (optional)
    String value() default ""; // Example element with a default value
}
```

- **@interface MyAnnotation:** Declares a new custom annotation named `MyAnnotation`.
- **@Retention(RetentionPolicy.RUNTIME):** Specifies the retention policy of the annotation. `RUNTIME` retention allows the annotation to be accessible at runtime via reflection.
- **@Target(ElementType.TYPE):** Specifies the target element types where the annotation can be applied. In this case, `ElementType.TYPE` indicates that the annotation can be applied to classes and interfaces.
- **String value() default "":** Defines an element named `value` within the annotation. This element can optionally be used to specify a single value when using the annotation. The `default` keyword provides a default value for the element.

Here's an example of how you can use the custom annotation:

```
@MyAnnotation(value = "Example")
public class MyClass {
    // Class implementation
}
```

In this example, the `@MyAnnotation` annotation is applied to the `MyClass` class with a specified value.

Custom annotations can also have other types of elements such as primitives, arrays, enums, or other annotation types. Additionally, you can specify multiple retention policies (`SOURCE`, `CLASS`, or `RUNTIME`) and target element types (`TYPE`, `METHOD`, `FIELD`, etc.) depending on your requirements.

Once defined, custom annotations can be processed at compile-time or runtime using reflection, or they can be used by frameworks, libraries, or tools to provide additional functionality or behavior.

## scope of beans//-

In the context of the Spring Framework, the scope of a bean defines the lifecycle and visibility of the bean within the Spring container. Different scopes allow beans to be managed in various ways, depending on factors such as how many instances are created, when they are created, and how long they are available.

Here are some common scopes of beans in Spring:

1. **Singleton (default)**: Beans with the singleton scope are created once per Spring container (per `ApplicationContext`). The Spring container manages a single instance of the bean, and all requests for the bean within the container result in the same instance being returned. Singleton beans are suitable for stateless components or shared resources.
2. **Prototype**: Beans with the prototype scope are created each time they are requested from the Spring container. The Spring container does not manage the lifecycle of prototype beans, and a new instance is returned every time the bean is requested. Prototype beans are suitable for stateful components or objects that need to maintain their state independently.
3. **Request**: Beans with the request scope are created once per HTTP request in a web application. The Spring container manages a single instance of the bean for the duration of the HTTP request, and the same instance is available to all components processing the request.
4. **Session**: Beans with the session scope are created once per HTTP session in a web application. The Spring container manages a single instance of the bean for the



duration of the HTTP session, and the same instance is available to all components within the session.

5. **Global Session:** Beans with the global session scope are created once per global HTTP session in a Portlet-based web application. This scope is similar to the session scope but is specific to Portlet environments.
6. **Application:** Beans with the application scope are created once per ServletContext in a web application. The Spring container manages a single instance of the bean for the entire duration of the web application, and the same instance is available to all components within the application.
7. **WebSocket:** Beans with the WebSocket scope are created once per WebSocket connection. This scope is specific to WebSocket-based applications and is managed by Spring's WebSocket support.

Each scope has its use cases and implications in terms of resource management, performance, and concurrency. By selecting the appropriate scope for your beans, you can ensure that they are managed and accessed correctly within your Spring application.

## bean lifecycle//(container started, bean instantiated, dependencies injected, init, utility, destroy)

### singleton usecase

The Singleton scope in the Spring Framework is the default scope for beans. It means that, by default, if you define a bean without specifying its scope, it will be treated as a Singleton. Here are some common use cases for Singleton-scoped beans:

1. **Stateless Services:** Singleton beans are ideal for stateless services or components that do not maintain any state between method calls. Examples include service classes responsible for business logic, data access objects (DAOs), utility classes, and helper classes.
2. **Shared Resources:** Singleton beans are suitable for managing shared resources such as database connections, thread pools, caches, and configuration settings. By maintaining a single instance of the bean, you ensure that all components in the application share the same resource.
3. **Caching:** Singleton beans can be used to implement caching mechanisms for frequently accessed data or expensive computations. The singleton scope ensures that the cached data is available throughout the application and that multiple requests can benefit from the cached data.

4. **Application-wide Configuration:** Singleton beans are often used to manage application-wide configuration settings or parameters. For example, you might use a Singleton bean to manage global settings, environment variables, or feature toggles that need to be accessed by multiple components.
5. **Singleton Design Pattern:** The Singleton scope is aligned with the Singleton design pattern, which ensures that only one instance of a class is created and provides a global point of access to that instance. Spring's Singleton beans follow this pattern and provide a convenient way to implement singletons in your application.

It's important to note that while Singleton beans offer advantages such as performance optimization and centralized management, they also have considerations regarding thread safety, shared state, and potential for creating tight coupling between components. Care should be taken when using Singleton beans to ensure that they are designed and used appropriately within the application context.

## concurrent hashmap//-

**ConcurrentHashMap** is a thread-safe implementation of the **Map** interface provided by Java's Collections Framework. It is designed to support concurrent access by multiple threads without the need for external synchronization.

Here are some key features and characteristics of ConcurrentHashMap:

1. **Thread-Safe:** ConcurrentHashMap provides thread-safe access to its underlying data structure, allowing multiple threads to read and write to the map concurrently without the need for external synchronization. This makes it suitable for concurrent programming scenarios where multiple threads need to access and modify a shared map.
2. **Partitioned Locking:** ConcurrentHashMap achieves concurrency by partitioning its data structure into segments, each of which is independently locked. This allows multiple threads to concurrently read and write to different segments of the map without contention, improving scalability and reducing contention under heavy concurrent access.
3. **Fine-Grained Locking:** ConcurrentHashMap uses fine-grained locking mechanisms, such as lock striping, to minimize contention and improve concurrency. Lock striping divides the map into smaller segments, each protected by its own lock, allowing multiple threads to access different segments concurrently.

4. **High Performance:** `ConcurrentHashMap` is designed for high throughput and low contention, making it suitable for performance-critical applications and multi-threaded environments. By minimizing synchronization overhead and contention, it provides efficient concurrent access to its data structure.
5. **Scalability:** `ConcurrentHashMap` is scalable under concurrent access, meaning that its performance scales well with the number of threads accessing the map concurrently. This makes it suitable for applications with varying degrees of concurrency and workload.
6. **Consistency Guarantees:** `ConcurrentHashMap` provides strong consistency guarantees under concurrent access, ensuring that updates made by one thread are immediately visible to other threads. However, it does not provide strict consistency for compound operations, such as `putIfAbsent()` or `replace()`.
7. **Iterators:** Iterators returned by `ConcurrentHashMap` are weakly consistent, meaning that they reflect the state of the map at the time of creation and do not throw `ConcurrentModificationException`. However, they may not reflect the most recent updates made to the map by other threads.

`ConcurrentHashMap` is widely used in concurrent and multi-threaded applications where thread safety and performance are important considerations. It is commonly used in scenarios such as caching, data aggregation, and concurrent data processing.

## When is `ConcurrentModificationException` occurs and how to resolve:

`ConcurrentModificationException` is thrown by certain collection classes in Java when an attempt is made to modify a collection while it is being iterated over. This exception typically occurs in single-threaded environments when the structure of the collection is modified (such as adding or removing elements) while iterating over it using an iterator or enhanced for loop.

Here are some common scenarios where you might encounter

`ConcurrentModificationException`:

1. **Using Iterator:** If you modify a collection (add, remove, or modify elements) using methods like `add()`, `remove()`, or `clear()` while iterating over it using an iterator's methods like `next()` or `remove()`, a `ConcurrentModificationException` may be thrown.

**Using Enhanced For Loop:** Similarly, if you modify a collection while iterating over it using an enhanced for loop (`for-each` loop), a `ConcurrentModificationException` may occur.

1. **Multithreaded Environment:** In a multithreaded environment, if one thread modifies a collection while another thread is iterating over it without proper synchronization, it may result in a `ConcurrentModificationException`.

To avoid `ConcurrentModificationException`, you can use one of the following approaches:

- Use an explicit `Iterator` and its `remove()` method to modify the collection while iterating.
- Use a synchronized collection (e.g., `Collections.synchronizedList()`) or a concurrent collection (e.g., `ConcurrentHashMap`) designed for concurrent access.
- Use concurrent collection classes or thread-safe iterators from libraries such as Java's `java.util.concurrent` package or third-party libraries like Google Guava.

By following these approaches, you can prevent `ConcurrentModificationException` and safely modify collections in concurrent or single-threaded environments.

## atomic integer

`AtomicInteger` is a class in Java that provides atomic operations for integers. It is part of the `java.util.concurrent.atomic` package and is designed for use in concurrent programming scenarios where multiple threads may concurrently access and modify an integer variable.

Here are some key features of `AtomicInteger`:

1. **Atomic Operations:** `AtomicInteger` provides atomic operations such as `get()`, `set()`, `getAndSet()`, `incrementAndGet()`, `decrementAndGet()`, and `getAndIncrement()`, which can be used to perform read-modify-write operations atomically without the need for explicit synchronization.
2. **Thread Safety:** `AtomicInteger` operations are thread-safe and guarantee atomicity without the need for external synchronization. This means that multiple threads can concurrently access and modify an `AtomicInteger` instance without causing data races or other synchronization issues.
3. **Compare-and-Set:** `AtomicInteger` supports compare-and-set (CAS) operations through methods like `compareAndSet()`, which atomically checks if the current value of the integer matches an expected value and updates it to a new value if the condition is met.

4. **Memory Visibility:** `AtomicInteger` operations ensure visibility of changes across threads. When a thread modifies the value of an `AtomicInteger`, the new value is immediately visible to other threads without the need for explicit synchronization or memory barriers.
5. **Performance:** `AtomicInteger` operations are typically implemented using low-level atomic hardware instructions or efficient software-based techniques, making them lightweight and suitable for high-performance concurrent programming.

```
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicIntegerExample {
    public static void main(String[] args) {
        AtomicInteger atomicInteger = new AtomicInteger(0);

        // Increment the atomic integer
        int newValue = atomicInteger.incrementAndGet();
        System.out.println("New value: " + newValue);

        // Compare and set operation
        int expectedValue = 1;
        int updatedValue = 10;
        boolean success = atomicInteger.compareAndSet(expectedValue,
updatedValue);
        if (success) {
            System.out.println("Value updated successfully");
        } else {
            System.out.println("Value update failed");
        }
    }
}
```

## Synchronised

In Java, the `synchronized` keyword is used to create synchronized blocks or methods, which are used to control access to critical sections of code in a multithreaded environment. Synchronization prevents multiple threads from executing certain code blocks simultaneously, ensuring that only one thread can execute a synchronized block or method at a time.

## Volatile

In Java, the `volatile` keyword is used to declare a variable as volatile. When a variable is declared as volatile, it indicates to the Java Virtual Machine (JVM) that the variable's value may be modified by multiple threads asynchronously. This ensures that changes made to the variable by one thread are visible to other threads immediately, without the need for synchronization.

1. **No Caching:** The JVM ensures that volatile variables are not cached in thread-local memory or CPU caches. Instead, volatile variables are always read from

and written to main memory, ensuring that changes made by one thread are immediately visible to other threads.

2.

## Garbagecollector

A garbage collector (GC) is a part of a runtime environment or programming language's implementation that automatically manages the memory used by a program by reclaiming memory occupied by objects that are no longer in use or reachable by the program. Garbage collection is a key feature of modern programming languages, such as Java, C#, and Python, that use automatic memory management.

Here's how a typical garbage collector works:

1. **Memory Allocation:** When a program creates objects, memory is allocated from the heap to store those objects. The heap is a region of memory managed by the garbage collector.
2. **Object Lifetimes:** Each object has a certain lifetime during which it is accessible and in use by the program. Once an object is no longer needed or referenced by the program, it becomes eligible for garbage collection.
3. **Identifying Garbage:** The garbage collector periodically scans the heap to identify objects that are no longer reachable or referenced by the program. This process is known as garbage collection or garbage collection cycle.
4. **Reclaiming Memory:** Once unreachable objects are identified, the garbage collector reclaims the memory occupied by those objects, making it available for future allocations. The reclaimed memory is then added back to the heap and can be used to allocate new objects.
5. **Garbage Collection Algorithms:** Different garbage collectors use various algorithms to identify and reclaim garbage, such as mark-and-sweep, generational collection, reference counting, and others. These algorithms vary in their efficiency, complexity, and trade-offs in terms of memory overhead and pause times.
6. **Tuning and Configuration:** Some runtime environments allow developers to configure and tune the behavior of the garbage collector to optimize performance and memory usage for specific applications. This includes settings such as heap size, garbage collection frequency, and collection algorithms.

Garbage collection provides several benefits, including automatic memory management, prevention of memory leaks, and simplification of memory management for developers. However, it also introduces overhead in terms of runtime performance and latency due to periodic garbage collection cycles, which can lead to temporary pauses or interruptions in application execution.

Overall, garbage collectors play a crucial role in managing memory efficiently in modern programming environments, allowing developers to focus on writing code without worrying about manual memory management and resource deallocation.

## Finalize

In Java, `finalize()` is a method defined in the `Object` class that is called by the garbage collector before reclaiming an object's memory. It's a part of Java's automatic memory management mechanism.

## markerinterface//-

Some common examples of marker interfaces in Java include `Serializable`, `Cloneable`, and `RandomAccess`. These interfaces indicate that classes implementing them support serialization, cloning, or random access, respectively.

## oauth//

OAuth (Open Authorization) is a widely used protocol that allows third-party applications to access a user's resources without sharing their credentials. In the context of Java, OAuth is often implemented to facilitate secure authorization for web applications and

## services.Springboot starter data security:

`@EnableWebSecurity`

extend `WebSecurityConfigurerAdapter`

Override the `configure(HttpSecurity http)` method to define security rules for your application:

**lambda ex--:**

`@FunctionalInterface`

interface `Operation`{

```
    //method providing definition of lambda expression
    public double calculations(double num1, double num2);
//Line 1
}
public class Calculator {
    public static void main(String[] args) {

        //creating lambda expression
        Operation adder = (double x, double y)-> x + y;           //Line 2
        //executing lambda expression
        double result = adder.calculations(5, 6);                 //Line 3
    }
}
```

```

        System.out.println(result);
    }
}

```

**function takes a single input value and return single value or object**

```

Pageable p=PageRequest.of(int page,int size)
PageRequest of(int page,int size,Sort sort)
findAll(Sort sort)
Sort sort=Sort.by("TransactionDate").descending();

```

primary key generation strategies:

Identity,table,sequence,auto

Identity:-

```
@GeneratedValue(strategy=GenerationType.IDENTITY)
```

```
@GeneratedValue(strategy=GenerationType.TABLE)
```

one to one

```
@OneToOne(Cascade=CascadeType.ALL)
```

```
@JoinColumn(name="address_id",unique=true)
```

```
@ManyToOne(cascade=CascadeType.ALL)
```

```
@JoinColumn(name="cust_id")
```

```
@OneToMany(cascade=CascadeType.ALL)
```

```
@JoinColumn(name="cust_id")
```

## Controller:

Controllers are responsible for handling incoming requests from users, processing the input, interacting with the model layer to retrieve or modify data, and then returning an appropriate response to the user.

**RestController** is a combination of controller and

ResponseBody(it is used for automatic conversion of response to a json or XML)

**@CrossOrigin(origins = "\*", allowedHeaders = "\*"):**

Cross-Origin Resource Sharing (CORS) is a security feature implemented by web browsers that allows or restricts web pages from making requests to a different domain than the one that served the web page. This is important for maintaining security and preventing malicious sites from accessing sensitive data.

### Why CORS?

When a web application tries to make a request to a different origin (domain, protocol, or port), the browser restricts this action to protect the user. CORS provides



a way to bypass this restriction by allowing servers to specify which origins are permitted to access their resources.

```
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
@RestController
public class MyController {

    @CrossOrigin(origins = "http://example.com")
    @GetMapping("/api/data")
    public String getData() {
        // Method logic
    }
}
```

**to get synchronized hash map--**

```
Collections.synchronizedMap(mapName);
```

**to get concurrent hash map--**

```
ConcurrentHashMap<String,String> cMap=new
```

```
ConcurrentHashMap<String,String>();
```

it does not throw any concurrent modification exception

concurrent hashmap does not allow nullkeys and null values

```
keys=map.keySet();
```

```
values=map.values();
```

**Scope--**

**@Scope("singleton")**

**@Scope("prototype")**

```
public double calculations(double num1,double num2);
```

```
Operation adder=(num1,num2)->num1+num2;
```

```
mapName.keySet();
```

```
mapName.values();
```

```
Set<Entry<Integer,String>> valueSet=mapName.entrySet();
```

```
for(Entry<Integer,String> i:valueSet){
```

```
syso(i.getKey()+""+i.getValue());
```

```
}
```

**Singleton Class in java:**

**in class Abc:-**

```
static Abc a =new Abc();
```

```
private Abc(){ --Private constructor
```

```

}
public static getInstance(){
return a;
}

```

jdk=jre+development Tools

jre=jvm+libraryfiles

**jvm->converts byte code into machine language**

**@Qualifier is used near @Autowired**

**@Primary used near bean declaration;**

**fail fast and fail safe are collections in java.**

the failfast system terminates the operation as fast as possible that are exposing the failures.it stops the entire operation

the failsafe system doesn't terminate the operations that are exposing failures.the fail safe system tries to avoid raising failures as much as possible.

the failsafe iterator avoids failures or errors by creating a copy of the original collection and modifying the copy instead of original

we need extra space for this

**jit-->**

javasourcecode is convrted into byte code with help of jdk.

**intrpreter** is used to convert this byte code into machine code

but basically interpreter is slow

with the help of jit it converts byte code into machine code just by compiling.

**methods of object class--**

clone --creates a copy of object

equals--checks whether two objects are same(==)

**finalize--** called by garbage collector on an object when garbage collector determines that there are no more references to the object.

**Logger** logger=LoggerFactory.getLogger(className.class);

```
logger.error("msg");  
logger.info("msg");  
logger.debug("msg");  
logger.warn("msg");  
logger.trace("msg");
```

spring-boot-starter-logging

we can set logging level in application.properties

```
logging.level.root=warn;
```

spring-boot-starter-log4j2

in application.properties:

```
logging.pattern.console=
```

```
logging.pattttern.file=
```

logging.file=/path --- we will specify a path here and a log file will be generated at the corresponding location path

for adding log data to console we can use appenders configuration in xml file

## **Git:**

clone-- git clone url

add-- git add filename

commit-- git commit -m "commit message"

log-- git log ---used to fetch the committed project history

push-- git push branchName -- this command is used for pushing our local repository code to remote repository

push all git push --all this command is used for pushing all branches to the server repository

pull git pull url it fetchs and merges the changes from remote directory to ur working directory

branch git branch it is used to fetch all the branches available in the repository

merge git merge branchname it is used for merging the specified branches into the current branch

## **Caching-**

@Cacheable in Repository layer

@EnableCaching near @SpringBootApplication

```
@Cacheable("customer")
    public Customer getCustomerDetails(Integer customerId)
throws InfyBankException {
java techie
comparator
streams sorting
Intermediate functions--filter,map,sorted
Terminal--count,forEach,reduce
```

\

## **Multithreading--**

we can create thread by  
extending Thread class--override run method  
implementing runnable interface--implement the run method--create  
an object for class implementing runnable--pass the object reference  
for new thread(ref)

## **thread methods:**

start--begins the thread execution  
run--contains code which need to be executed by threads  
sleep--it suspends execution for specified duration  
alive--check whether thread is alive or not  
join-- waits for thread to die  
yield--pauses the execution of thread temporarily and allows other  
thread with same priority to continue  
states of thread--  
newborn--when a thread object is created  
runnable--when start method is invoked on it .it is in runnable state  
when yield is executed thread state changes from running to  
runnable  
running--when thread is scheduled and is being executed by the  
processor  
waiting -- when thread needs a resource which is used by another  
thread it will go into waiting state and once the resource is available  
it will go back to runnable state

sleeping--when we invoke the sleep method .after the interval it go backs to runnable

blocked--if thread encounters a I/O request it will go to blocked state

dead-- when all the operations are completd it reaches dead state

thread scheduling techniques:

preemptive--priority based

time sliced--round robin scheduling--each thread gets cpu memory equally

## **Synchronization--**

only one thread can access a shared resource at a time

## **daemon thread--**

it is a thread that runs in the background and serves other threads

program wont wait for daemon thread to complete its execution

garbage collector is an example of daemon thread

**stack memory--**localvariables,local references,method invocations

**heap memory--** objects and arrays stored here,instance variables

## **garbage collection criteria--**

object reference is null

when reference of one object is set to new object and old object

doesn't have any refernce

local object refernce in method

**finalize method--**used for garbage collection

**if an object is of read mode and if we trying to invoke write methods on it .it will cause I/O exception callable:**

In Java, **Callable** is a functional interface introduced in the **java.util.concurrent** package as part of the Java Concurrency Utilities. It's similar to **Runnable** but differs in that it can return a result and throw a checked exception.

## Lazy loading:

`spring.main.lazy-initialization=true`

`(application.properties)`

## Reddis is like a json document having key value pairs

## Java 14- Record,detailed exception message, switch (case with lambda)

programming language. Here are some of the key features of Java 14:

1. **Switch Expressions Enhancements:**
  - Java 14 introduced further enhancements to switch expressions, making them more versatile and easier to use. These enhancements include allowing multiple constants in a single case label, and the ability to return a value from a switch expression.
2. **Pattern Matching for instanceof:**
  - This feature simplifies the code for type casting and conditional extraction of components from objects. It allows the combination of type testing and casting into a single, more concise and readable expression.
3. **Records (Preview):**
  - Records provide a compact syntax for declaring classes which are essentially data carriers. They automatically generate the constructors, accessors, and other methods for data encapsulation. In Java 14, records were introduced as a preview feature.
4. **Text Blocks (Preview):**
  - Text Blocks simplify the writing of multi-line string literals in Java, enhancing readability and maintainability of code. They allow for cleaner embedding of HTML, SQL, or any other multiline strings directly into Java code.
5. **JFR Event Streaming:**
  - Java Flight Recorder (JFR) can now stream events from a running JVM in real-time, enabling external monitoring tools to consume and process these events without needing to rely on disk-based recordings.
6. **Packaging Tool (Incubator):**
  - This tool was introduced as an incubator feature to simplify the packaging of Java applications as platform-specific executable files, including self-contained applications.
7. **Non-Volatile Mapped Byte Buffers:**
  - This feature introduces a new implementation of `java.nio.channels.FileChannel` that maps a region of a file directly into memory in a non-volatile way, allowing for more efficient I/O operations.

What is garbage collector , types and how it works

In Java, garbage collection (GC) is a crucial aspect of memory management that helps automatically reclaim memory that is no longer in use, thereby reducing the risk of memory leaks and improving the efficiency of the application. Java's garbage collectors are built into the Java Virtual Machine (JVM), and several types of garbage collectors are available to handle different performance needs.

## How Garbage Collection Works in Java

1. **Memory Allocation:** When objects are created in Java, they are allocated in the heap memory. The JVM manages this memory and keeps track of which objects are in use.
2. **Object Reachability:** The garbage collector identifies which objects are still reachable from the root references (e.g., local variables, static fields, and references from threads). Objects that are not reachable are considered garbage and eligible for collection.
3. **Marking Phase:** The GC starts by marking all reachable objects. It traverses the object graph from the roots, marking all objects that can be reached.
4. **Sweeping Phase:** After marking, the GC sweeps through the heap memory to collect and reclaim memory used by unmarked (unreachable) objects.
5. **Compaction (Optional):** Some garbage collectors also compact the heap memory to reduce fragmentation by moving objects around to make space for new objects.

## Types of Garbage Collectors in Java

Java provides several types of garbage collectors, each optimized for different use cases. Here's a summary of the main types:

1. **Serial Garbage Collector**
  - **Description:** The Serial GC uses a single thread for garbage collection. It is designed for applications with small heaps and single-threaded environments.
  - **How it Works:** It performs all garbage collection tasks (minor and major collections) using a single thread, which can lead to application pauses during GC.
  - **Use Case:** Suitable for small applications or applications where pauses are not a significant issue.
  - **Command-Line Option:** `-XX:+UseSerialGC`
2. **Parallel Garbage Collector (Throughput Collector)**
  - **Description:** The Parallel GC uses multiple threads to perform garbage collection, which can reduce pause times by parallelizing the work.
  - **How it Works:** It performs minor and major collections in parallel, thus improving the overall throughput and efficiency of memory management.
  - **Use Case:** Suitable for applications with large heaps and multi-threaded environments where throughput is important.
  - **Command-Line Option:** `-XX:+UseParallelGC`
3. **Concurrent Mark-Sweep (CMS) Collector**
  - **Description:** The CMS collector aims to minimize pause times by performing most of the garbage collection work concurrently with the application.
  - **How it Works:** It performs a majority of the garbage collection work concurrently with the application's threads, reducing the pauses associated with GC.
  - **Use Case:** Suitable for applications where low pause times are critical.
  - **Command-Line Option:** `-XX:+UseConcMarkSweepGC`

- **Note:** CMS has been deprecated in newer versions of Java in favor of G1 and other collectors.
- 4. **Garbage-First (G1) Collector**
  - **Description:** The G1 collector is designed to provide high throughput and low pause times by dividing the heap into regions and focusing on collecting regions with the most garbage.
  - **How it Works:** It performs incremental garbage collection and prioritizes regions that have the most garbage, aiming to keep pause times predictable and manageable.
  - **Use Case:** Suitable for applications requiring a balance between high throughput and low pause times.
  - **Command-Line Option:** -XX:+UseG1GC
- 5. **Z Garbage Collector (ZGC)**
  - **Description:** ZGC is designed for very low pause times, even with large heaps. It performs most of the garbage collection work concurrently and uses techniques like load barriers to achieve minimal pauses.
  - **How it Works:** It employs techniques to minimize pauses and perform garbage collection concurrently with application threads.
  - **Use Case:** Suitable for applications requiring very low pause times with very large heaps.
  - **Command-Line Option:** -XX:+UseZGC
- 6. **Shenandoah Garbage Collector**
  - **Description:** Shenandoah GC aims to reduce pause times by performing concurrent garbage collection, similar to ZGC but with different implementation details.
  - **How it Works:** It performs many of the garbage collection tasks concurrently with the application's threads to minimize pause times.
  - **Use Case:** Suitable for applications requiring low latency and shorter pause times.
  - **Command-Line Option:** -XX:+UseShenandoahGC

## Choosing a Garbage Collector

The choice of garbage collector depends on the application's requirements:

- **For low-latency applications:** G1, ZGC, or Shenandoah.
- **For high-throughput applications:** Parallel GC.
- **For older applications or compatibility:** CMS (though deprecated).

In summary, Java's garbage collection mechanisms are designed to optimize performance based on different application needs, from reducing pause times to increasing throughput. Each garbage collector has its own strengths and trade-offs, making it important to choose the right one based on the specific requirements of your application.

Shallow copy and deep copy and its user examples

## Shallow Copy

A **shallow copy** of an object is a new object that is a copy of the original object, but it does not create copies of the objects referenced by the original object. Instead, it copies the



references to those objects. Thus, changes to the referenced objects in the copy will also reflect in the original object.

## Deep Copy

A **deep copy** of an object creates a new object and also recursively copies all objects referenced by the original object. This means that the copied object and the original object do not share references to any objects. Changes in the copied object will not affect the original object.

## Summary

- **Shallow Copy:** Copies the object but not the objects it references. Changes in the referenced objects affect both the original and the copy.
- **Deep Copy:** Creates a new instance of the object and recursively copies all objects referenced by it. Changes in the copied object do not affect the original object.

Deep copying generally requires custom logic, either by implementing the Cloneable interface and overriding the clone() method or by using copy constructors. Shallow copying is simpler but less safe when dealing with mutable objects that are referenced by the object being copied.

## what is main advantages of using feignclient rather than rest template while connecting two micro services?

When connecting microservices in a Java application, choosing between different HTTP client libraries can significantly impact the performance, usability, and maintainability of your code. **FeignClient** and **RestTemplate** are two common options in the Spring ecosystem for making HTTP requests. Here are the main advantages of using **FeignClient** over **RestTemplate**:

### 1. Declarative API

**FeignClient**:

- **Advantage:** FeignClient allows you to declare HTTP clients using a declarative approach. You define an interface and annotate it with HTTP method annotations (e.g., `@GetMapping`, `@PostMapping`). FeignClient generates the implementation at runtime, simplifying the client code.

- **Example:**

```
```java
```

```
@FeignClient(name = "user-service")
```

```
public interface UserServiceClient {  
  
    @GetMapping("/users/{id}")  
  
    User getUserById(@PathVariable("id") String id);  
  
}  
...
```

**\*\*RestTemplate\*\*:**

- **\*\*Advantage\*\*:** RestTemplate requires you to manually construct and manage the HTTP requests and responses. This often involves more boilerplate code and can be less intuitive compared to the declarative approach of FeignClient.

### ### 2. **\*\*Integration with Ribbon\*\***

**\*\*FeignClient\*\*:**

- **\*\*Advantage\*\*:** FeignClient integrates seamlessly with Ribbon for client-side load balancing, making it easier to implement load-balanced HTTP requests. With FeignClient, you can leverage Ribbon's capabilities directly within your HTTP client interfaces.

- **\*\*Example\*\*:** If you use FeignClient with Spring Cloud, Ribbon is automatically integrated for load balancing.

**\*\*RestTemplate\*\*:**

- **\*\*Advantage\*\*:** You need to manually configure and manage load balancing with RestTemplate, often using a `LoadBalancerInterceptor` or other mechanisms.

### ### 3. **\*\*Simplified Error Handling\*\***

**\*\*FeignClient\*\*:**

- **\*\*Advantage\*\*:** FeignClient provides built-in support for handling HTTP errors through exception handling mechanisms. You can define custom error decoders to process and handle errors more effectively.

- **Example**: Custom error handling can be set up using `ErrorDecoder`.

**RestTemplate**:

- **Advantage**: RestTemplate requires you to manually handle HTTP error responses and exceptions. This typically involves more code to handle different HTTP statuses and map errors to exceptions.

#### ### 4. Automatic Encoding/Decoding

**FeignClient**:

- **Advantage**: FeignClient automatically handles encoding and decoding of request and response bodies using registered encoders and decoders. This simplifies working with different media types and serialization formats.

- **Example**: By default, FeignClient handles JSON serialization/deserialization using Jackson.

**RestTemplate**:

- **Advantage**: You need to manually configure message converters for encoding and decoding request and response bodies. This can involve additional setup and configuration.

#### ### 5. Configuration and Extensibility

**FeignClient**:

- **Advantage**: FeignClient supports configuration through annotations and properties files, making it easier to customize behavior (e.g., timeouts, retry mechanisms). You can also extend FeignClient by implementing custom requests and response interceptors.

- **Example**: You can configure FeignClient with properties such as `connectTimeout` and `readTimeout`.

**RestTemplate**:

- **Advantage**: RestTemplate provides a more traditional way to configure clients, often requiring more explicit setup for custom configurations and interceptors.

### 6. Integration with Spring Cloud

#### **FeignClient**:

- **Advantage**: FeignClient is deeply integrated with Spring Cloud, providing features like service discovery, load balancing, and fault tolerance through the Spring Cloud ecosystem.
- **Example**: FeignClient works seamlessly with Spring Cloud Netflix Eureka for service discovery and Ribbon for load balancing.

#### **RestTemplate**:

- **Advantage**: While RestTemplate can also work with Spring Cloud features, it often requires more configuration and manual setup to achieve similar integrations.

### 7. Ease of Use

#### **FeignClient**:

- **Advantage**: FeignClient is generally easier to use and requires less boilerplate code compared to RestTemplate. Its declarative nature allows you to focus more on defining service interfaces rather than managing HTTP requests manually.

#### **RestTemplate**:

- **Advantage**: RestTemplate offers more control over HTTP requests and responses, which can be useful for more complex scenarios but comes with additional complexity.

### Summary

- **FeignClient** provides a higher-level, declarative API for making HTTP requests, integrates seamlessly with Spring Cloud features like Ribbon and Eureka, simplifies error handling, and reduces boilerplate code.
- **RestTemplate** offers a more traditional and lower-level approach to making HTTP requests, giving you more manual control but often requiring more code and configuration.

For most modern Spring applications, particularly those using Spring Cloud for microservices architecture, FeignClient is preferred due to its simplicity, ease of use, and integration with other Spring Cloud components.

### how many ways we can a object in java?

- ☐ **new Keyword:** Standard way to create objects.
- ☐ **Factory Methods:** Provides controlled object creation.
- ☐ **Constructor Overloading:** Allows multiple constructors with different parameters.
- ☐ **Reflection:** Creates objects dynamically at runtime.
- ☐ **Deserialization:** Creates objects from their serialized form.
- ☐ **Cloning:** Creates copies of objects if the class implements Cloneable.
- ☐ **Dependency Injection:** Managed by frameworks like Spring.
- ☐ **Anonymous Inner Classes:** Instantiates classes with one-time implementations.

### @Primary Annotation

The @Primary annotation is used to designate a default bean when multiple beans of the same type are available. It indicates that this bean should be preferred over others when injecting dependencies.

### When to Use @Primary

- **Default Bean Choice:** Use @Primary when you want one bean to be the default choice for injection in case there are multiple candidates.
- **Common Scenario:** In scenarios where a particular bean is used most frequently, and you want to avoid specifying which bean to use every time.

### @Qualifier Annotation

The @Qualifier annotation is used to specify which exact bean to inject when multiple beans of the same type are present. It helps resolve ambiguity by explicitly indicating the desired bean.

## When to Use @Qualifier

- **Specific Bean Selection:** Use @Qualifier when you need to inject a specific bean out of multiple available beans.
- **Custom Behavior:** When different beans have different implementations or configurations, and you need to select one for particular use cases.

## Spring Boot Security Workflow

1. **Request Processing:** When an HTTP request is made, Spring Security intercepts it through a series of filters in the filter chain.
2. **Authentication:** The UsernamePasswordAuthenticationFilter or other authentication filters process the request to authenticate the user. If the authentication is successful, an Authentication object is created and stored in the SecurityContext.
3. **Authorization:** The access decision manager checks whether the authenticated user has the required permissions to access the requested resource. Access is granted or denied based on the user's roles and permissions.
4. **Session Management:** Manages user sessions, including session creation and concurrency control, based on the configured policies.
5. **Response Handling:** After processing the request, the response is sent back to the client. Any configured security headers or CSRF tokens are included.

**CompletableFuture** was introduced in Java 8 as part of the `java.util.concurrent` package. It allows for asynchronous programming and provides a way to write non-blocking code in a more readable and maintainable manner compared to traditional Future implementations.

Here's a detailed overview of how CompletableFuture can be used in Java 8:

### Key Concepts

1. **Asynchronous Execution:**
  - CompletableFuture allows you to run tasks asynchronously using methods like `supplyAsync` or `runAsync`.
2. **Completion Stages:**
  - You can define stages of computation to be executed after the initial computation completes, using methods like `thenApply`, `thenAccept`, and `thenRun`.
3. **Exception Handling:**
  - It provides a way to handle exceptions that may occur during asynchronous computation with methods like `exceptionally` or `handle`.
4. **Combining Futures:**
  - You can combine multiple CompletableFuture instances to run tasks in parallel or sequence, using methods like `allOf` and `anyOf`.

A **circular dependency** occurs when two or more components (classes, modules, or packages) depend on each other, creating a loop. This can lead to several issues in software development, such as difficulties in understanding the code, problems with modularity, and

issues with dependency injection frameworks. Understanding and resolving circular dependencies is crucial for maintaining clean and manageable codebases.

**RESOLVE** : and using techniques like setter injection, you can effectively resolve these dependencies and improve the maintainability of your application.

## Sequential Stream

A **sequential stream** processes elements one at a time, in a single-threaded manner. The operations are performed in a single sequence, which means they are executed in the order they appear in the stream pipeline.

### Characteristics

- **Single Thread**: Uses a single thread for processing, so the operations are executed in order.
- **Order Preservation**: Maintains the order of elements as they appear in the stream.
- **Simplicity**: Easier to reason about and debug since all operations are executed sequentially.

## Parallel Stream

A **parallel stream** processes elements in parallel, using multiple threads to perform operations. This can potentially lead to performance improvements for certain types of tasks, especially those that are CPU-bound and involve a large amount of data.

### Characteristics

- **Multiple Threads**: Utilizes a common ForkJoinPool to process elements concurrently.
- **Order Not Guaranteed**: The order of elements is not guaranteed unless explicitly specified (e.g., using ordered operations).
- **Performance**: Can lead to performance improvements, but the overhead of thread management and potential contention must be considered.

## Differences Between Sequential and Parallel Streams

1. **Performance**:
  - **Sequential**: Generally, sequential streams are more efficient for small datasets or simpler operations due to lower overhead.
  - **Parallel**: Parallel streams can improve performance for large datasets and complex operations by utilizing multiple cores. However, they also introduce overhead due to context switching and thread management.
2. **Order**:
  - **Sequential**: Preserves the order of elements.
  - **Parallel**: Does not guarantee the order of elements unless explicitly handled.
3. **Use Cases**:
  - **Sequential**: Suitable for small datasets, simple tasks, or when maintaining the order of processing is important.

- **Parallel:** Beneficial for large datasets or computationally intensive tasks where the cost of parallelism is outweighed by the performance gains.
- 4. **Overhead:**
  - **Sequential:** Minimal overhead since only one thread is used.
  - **Parallel:** Includes overhead for thread management and splitting tasks, which can negate performance gains for smaller datasets or tasks with low complexity.

## When to Use Which

- **Use Sequential Streams:**
  - When working with small collections.
  - When the operations are simple and the performance gain from parallelism is minimal.
  - When maintaining order is crucial.
- **Use Parallel Streams:**
  - When working with large collections or computationally expensive operations.
  - When the overhead of parallelism is justified by the performance improvements.
  - When the order of processing is not important.

## Summary

Sequential streams and parallel streams both offer powerful ways to process data in Java. Sequential streams are straightforward and maintain order, making them suitable for simpler or smaller tasks. Parallel streams can potentially offer significant performance benefits for large datasets or complex computations, but they come with the complexity of managing multiple threads and potential issues related to order and overhead. Understanding these trade-offs will help you choose the appropriate type of stream for your specific needs.

## 1. Constructor Injection

**Constructor Injection** involves providing dependencies through a class constructor. This approach ensures that all required dependencies are provided when the object is created, making the object immutable and ensuring that it is always in a valid state.

Dependency Injection (DI) is a design pattern used in software development to manage dependencies between objects. It promotes loose coupling and enhances testability by allowing objects to be injected with their dependencies at runtime rather than having them hardcoded. This separation of concerns simplifies managing complex applications and facilitates easier maintenance and testing.

### Ways to Achieve Dependency Injection

There are several ways to implement dependency injection in Java. The choice of method often depends on the project's requirements, complexity, and preferences. Here are the most common approaches:

#### 1. Constructor Injection



2. **Setter Injection**
3. **Interface Injection**
4. **Dependency Injection Frameworks**

## 1. Constructor Injection

**Constructor Injection** involves providing dependencies through a class constructor. This approach ensures that all required dependencies are provided when the object is created, making the object immutable and ensuring that it is always in a valid state.

### Advantages:

- Immutability of the injected dependencies.
- Ensures that dependencies are always provided.
- Suitable for mandatory dependencies.

### Disadvantages:

- Can lead to a large number of constructor parameters in complex scenarios.

## 2. Setter Injection

**Setter Injection** involves providing dependencies through setter methods after the object is created. This allows for more flexibility, as dependencies can be changed or injected at runtime

### Advantages:

- Flexibility to change dependencies.
- Useful for optional dependencies.

### Disadvantages:

- Potential for incomplete initialization if dependencies are not set.
- Can lead to inconsistent state if dependencies are changed.

**removing @Autowired** from fields, setters, or constructors means that Spring will no longer automatically inject dependencies. This can lead to problems like uninitialized fields, increased boilerplate code, and more complex testing and maintenance. Using @Autowired or other DI mechanisms provided by Spring generally simplifies dependency management and enhances maintainability.

❑ **throws**: Indicates that a method can throw exceptions and allows the caller to handle them. It is used for exception propagation and separation of concerns.

❑ **try-catch**: Used to handle exceptions within the method itself, allowing for localized exception handling, resource management, and user feedback.

## why we need to use @Service ,@Repository instead of @component in springboot

□ **@Component**: This is a generic stereotype for any Spring-managed component. It indicates that a class is a Spring bean but does not provide any additional context about its role.

□ **@Service**: This annotation is used for service-layer components. It indicates that the class contains business logic. By using @Service, you make it clear that the class is meant to perform service tasks and potentially contain transactional business logic.

□ **@Repository**: This annotation is specific to the data access layer. It indicates that the class is responsible for accessing data sources (like databases). It also enables features like exception translation, converting database-related exceptions into Spring's `DataAccessException`.

## Benefits of Generics

1. **Type Safety**: Generics allow for compile-time type checking, reducing runtime errors.
2. **Code Reusability**: You can write methods and classes that work with any data type, promoting code reuse.
3. **Elimination of Casts**: With generics, you often don't need to cast objects, making the code cleaner and safer.

**Java memory management** is a crucial aspect of the Java programming language that involves allocating and deallocating memory for objects during the lifecycle of an application. Here's a breakdown of the key components and processes involved in Java memory management:

### 1. Memory Areas

Java memory is divided into several areas, primarily:

- **Heap Memory**: This is where Java objects are allocated. It is managed by the Java Garbage Collector (GC). The heap is divided into:
  - **Young Generation**: Where most new objects are allocated and where garbage collection occurs frequently.
  - **Old Generation (Tenured Generation)**: Where long-lived objects are moved after surviving several garbage collections.
  - **Permanent Generation (Metaspace in Java 8 and later)**: Where metadata about classes and methods is stored.
- **Stack Memory**: Each thread has its own stack, which stores local variables, method calls, and references to objects in the heap. The stack is managed in a last-in-first-out (LIFO) manner.
- **Method Area**: Stores class-level data such as class structures, constants, and static variables.

**The Java Memory Model (JMM)** is a specification that defines how threads in a Java program interact through memory. It provides rules for visibility and ordering of variable access, ensuring consistency across multiple threads. Here's an overview of the key concepts and components of the Java Memory Model:

### **Key Concepts of the Java Memory Model**

#### **1. Memory Visibility:**

- The JMM specifies how changes made by one thread become visible to other threads. It ensures that when one thread updates a variable, other threads can see that update.
- Without proper synchronization, there's no guarantee that one thread's changes will be visible to others, leading to inconsistent states.

#### **2. Thread Interaction:**

- Threads communicate by reading and writing shared variables. The JMM defines the rules for when these reads and writes are guaranteed to be visible to other threads.

**comparable interface??**

**fpl gulf -regions**

**both clients got merged**

**merging two web sites**

**fpl website we got login screen**

**in session storage we use to store cookies**

**Docker is a kind of package software so that you can run it on any hardware**

**DockerFile-it is a blueprint for building docker image**

**DockerImage - It is a template for running docker containers**

**Container - A Container is just a run in process**

**Docker ps-gives all the running containers on your system**

**territory picker--**

**popup --fpl gulf --choose**

**user belongs to both regions**

**crew notified**

**sql --select max(salary) from employee where salary <(select max(salary) from employee);**

**Limit clause (m,n) -after how many rows you need the data to be fetched(m). ( n) how many rows to be fetched.**

**Select salary from employee order by salary desc limit 1,1;**

=====

**Java Design Pattern - Singleton Design Pattern -  
Singleton - how to break singleton pattern  
- factory Design Pattern**

**Spring Boot Design Pattern: Factory design  
pattern**

**SpringMS Design patterns : API Gateway, SAGA  
Design Pattern**

**Fault tolerance in MS - Hystrix/ Resilience4j**

**Spring security -**

**Authentication/Authorization/JWT token**

**Feign Client**

**Service Discovery**

**Kafka implementation --> fault tolerant**

**Try catch block syntax and how you will minimize  
catch blocks counts -> 1. by creating generic  
exceptions**

**try{}**

**catch(Arithmetic | | Null)**

**Streams:**

- 1. Book class -> price, name, author  
Print books for respective author.**

=====

=====

**Ness technology interview**

**Java 8 lambda.**

**Functiona interface.**

**Custome runtime exception.**

**Serialisation.**

**Syncronization.**

**Encapsulation in java why use.**

**Override private method in java.(ans- no)**

**Which are the two method will override to  
object in hashmap(ans-equal and hashCode).**

**Inheritance vs polymorphism.**

**FailFast vs fail safe.**

**Arraylist vs linked list**

**Checked vs unchecked exception.**

**Spring mvc.**

**Spring default scope.**

**Bean scopes.**

**Function vs procedure.**

---

---

**Axis bank**

**1.Join thread**

**2.HashMap internal working**

**3.Arraylist implementation**

**4.difference between exception and e**

**5. Runtime polymorphism**

=====

=====

**Infrasoft**

**Restful verb.**

**Restful protocol.**

**Restful vs soap.**

**Write the code Sort the students name.**

**HashSet vs hashmap.**

**Jsp life cycle.**

**Jsp tag.**

**Jstl tags.**

=====

=====

**Ness technology**

**Java 8 lambda.**

**Functiona interface.**

**Custome runtime exception.**

**Serialisation.**

**Syncronization.**

**Encapsulation in java why use.**

**Override private method in java.(ans- no)**

**Which are the two method will override to  
object in hashmap(ans-equal and hashCode).**

**Inheritance vs polymorphism.**

**FailFast vs fail safe.**

**Arraylist vs linked list**

**Checked vs unchecked exception.**

**Spring mvc.**

**Spring default scope.**

**Bean scopes.**

**Function vs procedure.**

=====

=====



## **E-solutions**

### **Java**

- Fail safe and fail fast**
- Mutable and immutable**
- HashMap internal working**
- join types and full join**
- Java 1.8 features**
- solid principal**

### **SPRING**

- J unit**
- Bean scope**
- AOP**
- DI and IOC it's same**
- spring security**
- spring transaction management**
- spring aggregation context and bean factory**
- spring exception**
- HTTP method support Rest services**
- Spring boot**

=====

=====

### **Odex**

**-Java**

**Java features.**

**What is multithreading.**

**What is abstract class.**

**What is concrete class.**

**Can you use abstract class in main method.( Yes)**

**Abstract class can have constructor.**

**What is interface.**

**What is default access modifier define in interface.**

**Can interface have a final method.(no)**

**What is exception.**

**Throw vs throws.**

**Steps to Create custom exception.**

**String vs stringBuilder and which one is faster.**

**String buffer vs stringBuilder.**

**What is Collection.**

**List vs set.**

**ArrayList vs sortedSet.**

**Can you add multiple zeros in ArrayList.**

**Static keyword.**

**-Hibernate**

**What is hibernate**

**Dailect.**  
**Criteria.**

**-Spring**  
**What is spring.**  
**Spring mvc.**  
**Loose couple.**

=====

=====

**Tech Mahindra**

**@Qualifier**  
**Statement vs prepared statement.**  
**Log4j.xml implementation.**  
**Contract between hashcode and equal.**  
**Override eaual()method without hashcode ,**  
**program.**  
**Reverse string without reverse function.**  
**Singleton class? How to make singleton java.**

**How to rollback query if those is 3 table for insert query and one is insert and second not insert it will get exception then how to rollback.**

**Transaction management.**

**Spring mvc.**

**View resolver type.**

=====

=====

**Nseit**

**OOPs.**

**Abstract.**

**Abstract class has constructor.(ans -yes)**

**Why use constructor in abstract class.**

**Way to define thread, which one is better and why.**

**Thread class vs runnable interface.**

**What is exception handling.**

**If try and catch block available then finally will be executed.**

**If try block are don't have any exception then finally will execute or not.**

=====

=====

**Servosys**

- 1. Hashmap internal working?**
- 2.JSP object**
- 3.Serialization**
- 4.undefined in java script.**
- 5.SQL queries for JOIN operation**
- 6.Throws handle which exception**
- 7.What is linked list**
- 8.What is algorithm**
- 9. Difference between array list and linked list.**
- 10.What is string ? Why it's immutable?**
- 11.OOPS concept in java?**
- 12.Difference between method overloading and method overriding**
- 13.Abstract class vs Interface**
- 14. Collection difference  
Arraylist and linked list**
- 15. What is vector and what is hashtable.**
- 16. Difference between set and map**
- 17. MySQL questions  
2nd highest salary, Aggregate functions,  
What is group by,  
What is view.**
- 18.Explain trigger with example.**
- 19. Write html form**

**20.send data from client to server side.**

**21.Jdbc and servlet connection.**

**22. What is composition**

**23. Hibernate**

**Note : core and advance java, SQL , java script.**

**What is response body. Why we use response body.**

**I want response in XML formate.**

**Whate is media type.**

**What is content type.**

**Request param vs path variable.**

**Controller vs Restcontroller.**

**Can we Use controller to create the rest API.**

**Using the response body it will return object in jsp page.**

**How to convert java object into json.**

**How you implement security means how you secure your application.**

**How to create token. What is the use of token. CSRF.**

**Sort the descending using collection without sorting function (logic).**

**Interface vs abstract.**

**What is the use of constructor in abstract class.**

**Suquery vs corelated query.**

**Cluster index vs non cluster index.**

**Why we use index.**

**By default for index which data structures we use.**

**Techskill-**

**Morgan Stanley**

**1st round-**

**HashMap internally work in java.**

**How we find Middle element in link list.**

**Failsafe vs failfast.**

**Method overloading and method overriding.**

**Implemented abstract class.**

**Executive framework in multithread.**

**Volatile keyword.**

**Automatic integer.**

**How can we implement factory design pattern.**

**Stringbuffer vs stringBuilder.**

**How to make a class immutable.**

**Strong reference vs weak reference.**

**Spring mvc.**

**Init binder.**

**Even row of data in the table SQL.**

**Top 5 Highest salary in SQL.**

**What is Rank and dens\_rank function in database.**

**[30/10, 12:54] Ravi maurya: Spring boot port change.**

**JPA repository.**

**Dependency injection.**

**Types of dependency injection.**

**Spring security.**

**POST vs PUT.**

**Java 7 features.**

**Comparable vs comparator.**

**Student age sort using comparator.**

**Return type of compareTo method.**

**HashMap vs concurrent- HashMap.**

**Way to create the thread.**

**Concurrency package in java 1.8.**

**Truncated vs drop.**

**Indexing.**

**Primary key.**

**Types of joins**



**Inner join example query.**

**Cross join.**

**Aggregate function in Oracle.**

**Jsp life cycle.**

**Servlet vs jsp.**

**Git lab command.**

**Jenkins.**

**@Qualifier**

**Statment vs prepared statement.**

**Log4j.xml implementation.**

**Contract between hashCode and equal.**

**Override eaual()method without hashCode ,  
program.**

**Reverse string without reverse function.**

**Singleton class? How to make singleton java.**

**How to rollback query if those is 3 table for insert  
query and one is insert and second not insert it  
will get exception then how to rollback.**

**Transaction management.**

**Spring mvc.**

**View resolver type.**

**We have a Parent class with 2 methods walk and  
run. We have a child class which overrides both**

**the methods. Both child class methods just call their respective Super implementation.**

**Parent class run method calls walk().**

**class Parent**

**Parent p = new Child();**

**p.run();**

**Tell the order in which each method is called**

**Answer**

**1. Correctly tells the order: child.run -> parent.run -> child.walk -> parent.walk**

**DB question :**

**We have a table called BookAuthor. It has two columns Book and Author, Book being unique column.**

**Write a query to find the names of the authors who have written more than 10 books.**

**Immutable Class**

- 1. Make all fields private and final. .**
- 2. Remove the setter methods. .**

3. Return either a new copy of degrees or an unmodifiableList in the the getter method .(in case List as member of class)
4. In the constructor, make a copy of the List argument passed..
5. Make the class final. .
6. Knows the benefits of immutable class in multi-threaded programs and as a good design practice..

**DSA :**

**Longest Even Length Word:**

**Write a function to return the longest even length word in a sentence.**

**Sample input: Be not afraid of greatness, some are born great, some achieve greatness, and some have greatness thrust upon them.**

**DSA :**

**Given an array of n integers and a number k, find the pairs of numbers in the array such that the difference between the pair is k. Find the optimal solution with and without extra storage**

**Prepare.**

### **1.HashMap(Confirm Question)**

- overriding hashCode and Equal**
- Its Contract**
- Behavior if hashCode returns a constant (including get and put time complexity).**
- Behavior if hashCode is not implemented but equals is .**
- Behavior if hashCode is implemented but equals is not .**
- Making the class immutable is a good idea when using it as a key..**

```
class A {  
    public void synchronized m1()  
    {Thread.sleep(5000);}  
}
```

**We create two objects of this class - o1 and o2.  
We call o1.m1() on one thread and o2.m1() on  
another thread, at the same time. What will be  
the behaviour?**

**Follow up with - how will you force these calls to execute one after the other**

**1. Understands object level synchronization.**

**These two calls do not block each other.**

**2. Suggests a solution using a shared lock object or class level synchronization.**

**We have a Parent class with 2 methods walk and run. We have a child class which overrides both the methods. Both child class methods just call their respective Super implementation.**

**Parent class run method calls walk().**

**class Parent**

**Parent p = new Child();**

**p.run();**

**Tell the order in which each method is called**

**Answer**

**1. Correctly tells the order: child.run ->**

**parent.run -> child.walk -> parent.walk**

**Inheritance, Polymorphism**

**Java Collections**

**Skill Evaluation Advanced    Design Principles and  
Patterns Multithreading and concurrency**

**Spring, AOP, ORM**

**REST, SOAP, Txn, Sessions, IO**

**DB - Normalization, Joins, Simple Queries**

**[30/10, 12:54] Ravi maurya: What is the  
difference between HashMap and  
ConcurrentHashMap? Follow up with - what is  
the difference between CHM and synchronized  
Map.**

**1. Knows that CHM is thread-safe.**

**2. Knows about  
ConcurrentModificationException.**

**3. Knows how CHM allows concurrent read and  
write as opposed to synchronized Map.**

**4. Knows how read operation does not take a  
lock and still is thread safe.**

**GC, Serialization, Reflection, Lambdas**

.....  
.....  
:::: ::DOCKER::

## **History :**

--We are maintaining different server to run application we maintain different ram, processor, OS, application code.

-- Next.. We are using multiple vms in based with single hardware and we maintain our application there.

-- Now we extract out that VM layer and maintain the application as a container and containaize them.

=====

For an application to run we need set of dependencies like java version jar versions.

**If we are using the application in two systems we need to set up the dependencies by our self and make the application to be run.**

**Let's suppose if we are now deploying in the Vm's**

**Then also we maintain the set of dependencies like.... Java, springboot version, and other configs that required.**

**Let's suppose our application got need more vms to scale like 10000 them, then it becomes harder to setup this config in all vms .**

**To resolve this issue, Docker came into picture. To scale the application.**

**Docker File, Image, Container.**

**File : set of instructions to build the docker Image.**

**Image : It's a package that contains application code and application dependencies**



**Container : place where application is executed /  
Runtime instance of our application.**

**Building docker file gives Image  
Running Image gives container**

**We use <finalname>dockerdemo<filename> tag  
inside <build> tag in pom  
Else it docker will combine groupid, artifactid,  
version as jarfilename**

**Steps**

**Create a jar file of application in target  
Mvn clean package**

**Dockerfile**

**FROM openjdk:11 -- tells to install java 11 version**

**COPY target/dockerdemo.jar /usr/app/ -- tells to  
copy the jar file in this location to docker location**

**WORKDIR /usr/app -- setting this as working  
directory**

**ENTRYPOINT ["java", "-jdk", "dockerdemo.jar"] --**  
**used to run the jarfile.**

**Docker commands ::**

**docker images -- to see all images in docker**

**docker build -t sbimage . -- to build image**

**sbimage - name of image**

**dot represents current working directory**

**-t represents tag name**

**docker run -d -p 8000:8080 sbimage**

**-d represents detached mode**

**-p represents port mapping**

**sbimage represent image name**

**8000:8080 -- hostport:containerport mapping**

**docker ps**

**Tells the containers available or running.**

=====

=====

# Spring Annotations

## **\*Core Annotations:\***

- **@SpringBootApplication**
- **@Configuration**
- **@Component**
- **@Controller**
- **@Service**
- **@Repository**
- **@Bean**
- **@Autowired**
- **@Qualifier**
- **@Value**
- **@Primary**
- **@Lazy**
- **@Profile**

## **\*Web Annotations:\***

- **@RestController**
- **@RequestMapping**
- **@GetMapping**
- **@PostMapping**
- **@PutMapping**
- **@DeleteMapping**
- **@PatchMapping**

- **@RequestBody**
- **@ResponseBody**
- **@RequestParam**
- **@PathVariable**
- **@RequestHeader**
- **@CookieValue**
- **@CrossOrigin**
- **@ExceptionHandler**
- **@ControllerAdvice**
- **@ResponseStatus**
- **@ModelAttribute**
- **@SessionAttribute**
- **@SessionAttributes**
- **@ModelAttribute**
- **@Valid**
- **@InitBinder**
- **@RequestPart**

**\*Data Annotations:\***

- **@Entity**
- **@Table**
- **@Id**
- **@GeneratedValue**
- **@Column**
- **@ManyToOne**

- **@OneToMany**
- **@ManyToMany**
- **@JoinColumn**
- **@Embedded**
- **@Transient**
- **@Query**
- **@Repository**
- **@Transactional**
- **@EnableTransactionManagement**
- **@EnableJpaRepositories**

**\*Security Annotations:\***

- **@EnableWebSecurity**
- **@EnableGlobalMethodSecurity**
- **@Secured**
- **@PreAuthorize**
- **@PostAuthorize**
- **@AuthenticationPrincipal**
- **@EnableOAuth2Client**
- **@EnableAuthorizationServer**
- **@EnableResourceServer**
- **@EnableWebFluxSecurity**

**\*Testing Annotations:\***

- **@RunWith**

- **@SpringBootTest**
- **@WebMvcTest**
- **@DataJpaTest**
- **@MockBean**
- **@SpyBean**
- **@Test**
- **@Before**
- **@After**
- **@BeforeClass**
- **@AfterClass**
- **@Mock**
- **@Spy**
- **@InjectMocks**
- **@ConfigurationProperties**
- **@ContextConfiguration**
- **@ActiveProfiles**

**These annotations cover various aspects of Spring Boot application development, including configuration, web development, data access, security, and testing.**