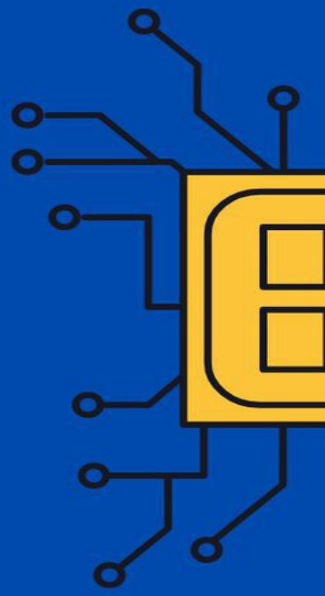
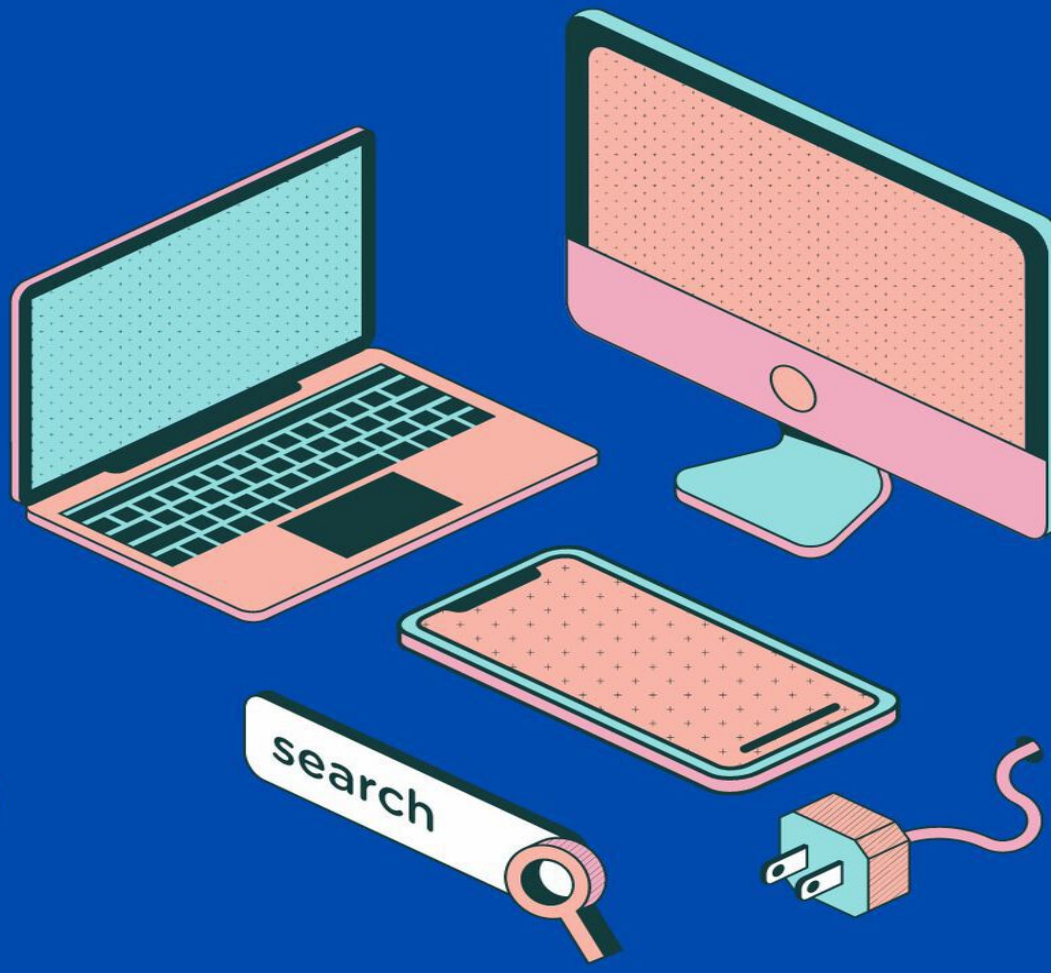
GUIDE TO CLEAR JAVA DEVELOPER INTERVIEW



Contains 250+ Java, SpringBoot,Microservice Interview Questions



AJAY RATHOD

@ajtheory

CONTENTS

What is the difference between abstract and interface keyword in java?

What is the difference between Static method and Default methods in Java 8?

How to create an Object in java? Explain Different ways to do it.

How to create an immutable class?

How to restrict object creation in java?

How to create a custom class loader in java?

Can we write the main method without static?

Explain the diamond problem in java and how to resolve it.

How to write a wrapper class in java?

How does HashMap work internally?

Explain the internal implementation of HashMap.

Which Classes are eligible to be used inside the resource block?

How does HashSet works internally?

What is the difference between HashMap and HashTable?

What is the difference between HashMap and Linked-HashMap?

What is the difference between HashMap and ConcurrentHashmap?

Can we insert the null key in the HashMap and HashTable?

How to create an immutable map in Java?

What are the in-built immutable classes in java?

What is index in java? Advantages and disadvantagesin Database?

What is the difference between CompletableFuture and a Callable and Runnable Future?

In Java, there are several ways to perform asynchronous operations:

How to parse XML file with JSON in java?

How to parse JSON to a HashMap?

What are fail-safe and fail-fast iterators?

What is the object class in Java? what are the methods in it?

Why and how to use clone method in Java?

What parsing libraries you have used so far?

What is the difference between comparable and comparator?

Explain the classpath exception?

What is the hierarchy of exceptions?

Explain Throw, Throws, and Throwable keywords in java.

What is a String in java?

Why String is immutable?

Where is a new string is stored?

Where do strings get stored and where does the reference get stored?

Can we create a customized immutable String class, how to achieve it?

What is the difference between String, StringBuffer and StringBuilder?

Is StringBuffer synchronized? Where is synchronized used in StringBuffer?

Why are Java substrings bad?

What is a Runtime exception and how they are they implemented?

Draw the collection hierarchy?

What is the difference between these syntaxes?

What collection will we use for manipulation (ArrayList or LinkedList)?

What is the use of an iterator in Java?

What is the default capacity of HashMap?

How does HashMap behaves when it reaches its maximum capacity?

How to create a custom object as key in HashMap?

Does HashMap store value in ordered way or not?

What is HashSet and TreeSet?

How to get values from HashSet?

What is the difference between them, which one will compile and what is the best way to declare?

Difference between ArrayList and LinkedList?

Difference between Set and List collection?

Difference between HashSet and HashMap?

Why does HashMa not maintain the order like Linked-HashMap?

How does LinkedHashMap is able to maintain the insertion order?

Difference between LinkedHashMap and Priority queue?

Write a Hashcode implementation and what is return type of it?

What is ConcurrentHashmap?

What is the Internal implementation of ConcurrentHashmap?

Is it possible to modify ConcurrentHashmap using iterator?

What is the concurrent collection?

Can we insert Null in ConcurrentHashmap?

What is a Concurrent Modification exception, and how to prevent that?

What is serialization?

Uses of serialization? Why is this needed?

When to use ArrayList and when to use LinkedList?

What is Garbage Collection?

What is System.gc in java?

What kind of algorithm is used in the garbage collector?

What are fast and fail-safe in collection framework?

Where are static methods or static variables stored in Java memory?

How to create custom exceptions in Java?

What is the difference between Class and Instance variables?

CHAPTER 7: SPRING-BOOT

CHAPTER 8: MICROSERVICE

Under what circumstances is the Microservice architecture are not preferable to you?

What are the design principles of Microservice?

Do you know the 12-factor methodology to build a Microservice?

Why are Microservice stateless?

What is the advantage of Microservice using Spring Boot Application + Spring Cloud?

How to share a database with multiple microservices?

Do you know Distributed tracing? What is its us?

How distributed tracing is done in Microservice?

How to connect internal and external services in microservices.

Which Microservice design pattern have you used so far and why?

Which design patterns are used for database design in Microservice?

What is the SAGA Microservice pattern?

Explain the CQRS concept?

Which Microservice pattern will you use for read-heavy and write-heavy applications?

Explain the Choreography pattern in Microservice?

What are the types of fault tolerance mechanisms in Spring Microservice?

What is circuit breaker pattern? What are examples of it?

Explain the annotations used to implement circuit breaker in spring boot?

Which library have you used to implement circuit breaker in spring boot?

How to call methods Asynchronously, in the spring framework how can we do that?

How to call another microservice asynchronously?

How to communicate between two microservices?

CHAPTER 12: SQL/DATABASE/HIBERNATE-JPA

CHAPTER 13: CODING

Write a Program to find the duplicates in an array using stream API.

How to sort the employee list in ascending and descending order using java 8 streams API?

Find the highest salary of an employee from the HR department using Java stream API.

Find an average of even numbers using Java 8 stream API?

How to use sorting in Java-8?

Write a program using stream API - Find the employee count in each department in the employee list?

Find employees based on location or city and sort in alphabetical manner using stream API?

Find the occurrence of names of employees from the List<Employee>, and find the frequency of each name.

Write a Program to print only numbers from an alphanumeric char array using stream API in java-8.

Write a program to find the sum of the entire array result using java 8 streams?

Write a program to find even numbers from a list of integers and multiply by 2 using stream java 8?

Write a program to find the occurrence of each word in a given string in java?

Write a Program to find a common element from three integer ArrayList. eg. arr1, arr2, and arr3.

Write a program to convert string to integer in java without any API?

Write a program to find the first occurrence of a character in a string in java?

Write a program to find the missing number in an Array in java.

Write a Program to Find a possible combination of the given string "GOD"?

Write a program for valid parenthesis in java?

CHAPTER 16: KAFKA

CHAPTER 17: MISCELLANEOUS

Can we write j-units for static methods?

How to resolve this Mockito exception “Mockito cannot mock this class”?

What is binary search tree?

OVERVIEW

Welcome to the Ultimate Guide for Mastering Java Developer Interviews!

Whether you're just starting out in Java development or have been at it for up to 10 years, this book is here to help you prepare for your dream job. It's designed to be your go-to companion.

Inside, you'll find a handpicked collection of important interview questions based on my own experiences. But it's not just questions – I've also included detailed and relevant answers to each one.

This guide covers a wide range of topics to make sure you're well-prepared. From the basics like Object-Oriented Programming and Core Java to more advanced topics like Java-8, Spring Framework, Spring-Boot, Microservice architecture, Memory Management in Java, REST principles, Design Patterns, System Design, SQL and Hibernate-JPA, and various Coding and Programming Questions – it's all covered!

I've even included Scenario-Based Interview Questions to test your problem-solving skills in practical situations. And there's a section on Miscellaneous topics to make sure you're knowledgeable in all the essential areas.

The book also explores Multithreading, an area often focused on in interviews to assess your concurrent programming skills.

By the end of this guide, you'll walk into your interview with confidence and expertise. The knowledge you gain here will set you apart from the competition.

So, embrace this opportunity and start your journey toward interview success with enthusiasm. Best of luck!

Best Regards,

Ajay Rathod

HOW TO PREPARE FOR A JAVA DEVELOPER INTERVIEW

I will be sharing my preparation strategy for techies who want to hunt for their next tech job. Over the course of three months, I extensively prepared for a Java developer role. This preparation helped me clear many interviews, including those with BIG FOUR companies. Developers with 0–10 years of experience can apply this strategy.

Disclaimer: This article is not intended for FAANG companies; I am referring to general tech companies ranging from MNCs to small-scale and large-scale entities within the Indian Job Market.

In this article, I will elaborate on interview preparation and share my strategies on how to maximize your chances of being selected for an interview.

Typical Interview Format

- Technical Round 1 (L1 round)
- Technical Round 2 (L2 round)
- Manager Round
- HR round

Usually, if you can clear the technical rounds, you are well on your way to receiving an offer letter, as the manager and HR rounds primarily involve discussions. Hence, our focus should be on preparing for the technical rounds.

Technical Round Preparation Steps

The interview typically begins with an introduction, and the interviewer may inquire about the project you have been working on.

Step 1: Know Your Current Project Inside and Out

a. Understand the functionality of the project, including its purpose and the problems it solves for customers. Essentially, you should have a comprehensive overview of your project's functions. b. Familiarize yourself with the project's architecture and technical stack. You can also delve deeper to comprehend the end-to-end flow. c. Discuss the technical stack used in the project. For instance, specify the front-end framework (e.g., Angular, React), the backend technology (e.g., Java, Python), and the database (e.g., PostgreSQL, Dynamo DB). d. Gain insight into the CI/CD model employed. Many developers are unaware of the deployment process, so this is a crucial aspect to understand.

Thoroughly studying the aforementioned project-related aspects will empower you to steer the interview in your favor. Remember, your responses often guide the interview.

Step 2: Java Developer Competencies

As a Java Developer, you should be well-versed in the following topics, which will significantly enhance your likelihood of being selected.

- Object-Oriented Programming principles, including SOLID principles (prepare for inheritance puzzles)
- Multithreading and Concurrency (prepare for Executor framework and concurrency API)
- Collection framework (Comprehend the internal workings of each collection data structure, such as HashMap, ConcurrentHashMap, HashSet)
- Serialization (Understand its functioning)
- Design Patterns (Familiarize yourself with at least 4–5 design patterns, including creational, behavioral, and structural patterns)
- Garbage Collection
- Java Generics

- Java 8 (Stream and Functional Programming—prepare to write Java 8 code for stream operations)
- SQL Queries (Be ready to write queries involving JOINS and employee tables, such as retrieving the highest salary)
- Coding Practice (Solve a variety of Array and String problems)
- Memory Management (Stay informed about memory management changes in Java 8 and above)

Proficiency in the aforementioned areas is crucial for interview success. Candidates are typically evaluated based on their practical knowledge and ability to write programs and SQL queries using Java. These skills significantly contribute to interview performance.

Remember, diligent preparation and a strong grasp of these concepts will greatly improve your chances of excelling in your Java Developer interview. Good luck!

HOW I MASTERED JAVA IN DEPTH FOR TECHNICAL INTERVIEWS AND TRIPLED MY SALARY

Learning Java in-depth for Java Developer Technical interviews was a transformative journey, driven by my personal experience. As I share my takeaways, you'll discover how this process enriched my knowledge and significantly boosted my earnings. With the high demand for Java professionals in the market, my insights can benefit Java Developer professionals seeking to excel in their current or future roles.

Initial Steps: From Junior Java Developer to Depth Mastery

Starting as a Junior Java developer, I heavily relied on internet searches for coding solutions. A mentor's guidance would have been invaluable, but I lacked that privilege. My quest for a new job led to rejection due to my superficial knowledge.

Key Lessons:

1. **Java API Documentation and Understanding**: Delving into Java API documentation unveiled the inner workings of JDK, aiding me in comprehending their internal mechanisms, time, and space complexities.

2. **Foundational Reading**: Engaging with key Java books, such as "Head First Java" and "Effective Java," fortified my grasp of Object-oriented programming and laid a solid foundation.

3. **Coding Efficiency**: Shifting from brute-force coding to optimized practices became vital. Solving coding problems on platforms like LeetCode and HackerRank honed my coding skills.

4. **Design Patterns and System Design**: Recognizing the importance of design patterns and system design, I incorporated these concepts into my repertoire.

5. **Unit Testing Proficiency**: Embracing thorough knowledge of the JUnit framework was essential, as interviewers assessed unit testing skills.

Topics Explored in Depth:

1. **Core Java API**: Thorough exploration of Collection framework, Streams, Java Lang, Java Util, Java Time, Java IO, Java net, and Java SQL packages.

2. **Design Patterns**: In-depth understanding of design patterns like Builder, Factory, Proxy, Adaptor, Facade, and Observer, supported by practical examples.

3. **SOLID Principles**: Application of SOLID design principles in coding, promoting clean and effective code writing.

4. **Clean Code Practices**: Influence from "Clean Code" and "Clean Coder" by Uncle Bob contributed to cleaner coding habits and improved code reviews.

5. **Frameworks Mastery**: Proficiency in Spring Framework, Hibernate framework, and JPA facilitated comprehensive development.

6. **Unit Testing Frameworks**: Mastery of JUnit, Mockito, and PowerMock ensured comprehensive code coverage, aligning with pipeline criteria.

Results and Acknowledgment:

By immersing myself in these concepts, I navigated technical interview rounds successfully, achieving a substantial salary increase and recognition as a proficient technical resource. A manager's feedback commended my technical prowess and quality code delivery.

In Closing:

My journey from superficial Java knowledge to depth mastery underscores the transformative impact of deliberate learning. This narrative aims to inspire fellow developers to embrace a

comprehensive approach to skill enhancement. For those seeking guidance, my articles offer valuable insights.

CHAPTER 1: HOW TO INTRODUCE YOURSELF AND ANSWER INITIAL QUESTIONS

In this chapter, we'll talk about why it's important to introduce yourself in an interview. When you introduce yourself, you can give the interviewer a quick overview of your work experience and technical skills.

When you introduce yourself, it's crucial to only talk about things you're very sure about, things you know 100 percent. For example, if you're not familiar with a technology like JavaScript, it's best not to mention it because the interviewer might ask you about it.

Remember, you can influence the interview by choosing what topics to talk about. Try to focus on the things you're really good at and be ready to answer questions about them.

Let’s dive into the interview questions.

Tell me about yourself, tell me about your skills.

"Tell me about yourself" and "Tell me about your skills" are common interview questions that can be challenging to answer. Here are some tips on how to tackle these questions:

Prepare ahead of time: Spend some time thinking about your strengths, experiences, and achievements that are relevant to the position you are applying for. Write them down and practice talking about them out loud.

Keep it relevant: When answering the question "Tell me about yourself," focus on your professional experiences and achievements that are relevant to the position you are applying for. You can also mention your personal interests if they relate to the job.

Be concise: Keep your answers brief and to the point. Don't ramble on or share irrelevant information. Stick to the main points and be

clear and concise.

Highlight your skills: When asked about your skills, provide specific examples of how you have used them in the past to achieve success. Talk about your strengths and how they will benefit the company. Be sure to include both technical and soft skills, such as problem-solving, communication, and teamwork.

Be honest: It's important to be truthful when answering these questions. Don't exaggerate or make up skills or experiences that you don't possess.

Practice: It's a good idea to practice answering these questions with a friend or family member to help you feel more comfortable and confident during the actual interview.

Please tell me about your project and its architecture. Please explain it and draw the architecture, framework, and technology used.

When answering the question "Tell me about your project and architecture," it's important to provide a clear and concise overview of the project and the underlying architecture. Here are some tips to help you tackle this question:

Start with an overview: Begin by giving a brief overview of the project and the business problem it was designed to solve. This will help provide context for the architecture you will describe.

Discuss the architecture: Explain the underlying architecture that was used in the project. This should include a high-level overview of the components and how they interact with each other. You can also discuss the rationale for choosing this architecture and any trade-offs that were made.

Describe the design decisions: Talk about the design decisions that were made during the project. This could include how the architecture was designed to meet specific performance requirements, how the system was designed to be scalable, or how it was designed to be maintainable.

Discuss the implementation: Describe how the architecture was implemented and any challenges that were encountered during implementation. This could include any optimizations that were made or any trade-offs that were made to meet specific requirements.

Highlight your role: Be sure to discuss your role in the project and how you contributed to the architecture design and implementation. This could include any specific tasks you performed or any technical challenges you helped overcome.

Use visual aids: If possible, use diagrams or other visual aids to help illustrate the architecture and design decisions. This can help the interviewer better understand your explanation and provide a more comprehensive answer.

Remember to stay focused on the most relevant aspects of the project and architecture, and be sure to highlight your role and contributions to the project. Be clear and concise in your explanation, and use examples and visual aids where possible to help support your answer.

What are the best practices to follow while developing an application?

There are several best practices to follow during software development to ensure a high-quality product and efficient development process. Here are some of the most important ones:

Plan and prioritize: Before starting development, make sure to plan the project thoroughly and prioritize tasks based on their importance and urgency.

Follow a consistent process: Use a consistent development process, such as agile or waterfall, to ensure a structured and predictable development process.

Use version control: Use a version control system, such as Git, to manage and track changes to the codebase.

Test early and often: Test the software early and often to catch bugs and errors before they become more difficult to fix.

Use automation: Use automation tools, such as continuous integration and deployment (CI/CD) pipelines, to automate repetitive tasks and ensure consistency.

Write clean and modular code: Write clean, modular, and maintainable code to make it easier to maintain and extend the software over time.

Document the code: Document the code thoroughly, including comments and documentation, to make it easier for other developers to understand and work with the code.

Use appropriate tools and technologies: Use appropriate tools and technologies that are well-suited for the project requirements and team's skills.

Collaborate effectively: Foster effective collaboration within the team, including communication, code reviews, and regular meetings to ensure everyone is aligned and working towards the same goals.

Continuously improve: Continuously evaluate and improve the development process and incorporate feedback from users to improve the software over time.

By following these best practices, software development teams can create high-quality software that is maintainable, scalable, and efficient to develop.

What challenging tasks have you accomplished so far? Could you provide examples?

Interviewers are usually interested in hearing about the most challenging technical tasks you've tackled. They'll likely ask in-depth questions about these challenges, so it's a good idea to pick one area where you have extensive experience and explain it thoroughly.

Share information about any specific functionality or module you've worked on, and if you've been involved in creating MVPs (Minimum Viable Products), be sure to mention those too.

Please explain the code and the flow of the project, both within it and outside of it.

When asked to explain the code and flow of a project, it's important to provide a clear and concise overview of the project and how it works. Here are some tips to help you answer this question:

Start with an overview: Begin by giving a brief overview of the project and its purpose. This will help provide context for the code and flow you will describe.

Discuss the architecture: Provide an explanation of the underlying architecture that was used in the project. This should include a high-level overview of the components and how they interact with each other.

Describe the code flow: Describe how the code flows through the different components of the project. This should include an explanation of the different modules, functions, and classes that make up the codebase.

Explain the logic: Explain the logic behind the code and how it implements the functionality of the project. This should include an explanation of the algorithms and data structures used in the code.

Use visual aids: If possible, use diagrams or other visual aids to help illustrate the code flow and architecture. This can help the listener better understand your explanation and provide a more comprehensive answer.

Highlight your contributions: Be sure to discuss your contributions to the project and how you contributed to the code and flow. This could include any specific tasks you performed or any technical challenges you helped overcome.

Provide examples: Provide specific examples of how the code and flow work in different scenarios. This can help illustrate the functionality of the project and provide a more concrete understanding of how it works.

Remember to stay focused on the most relevant aspects of the project and code flow, and be sure to highlight your role and

contributions to the project. Be clear and concise in your explanation, and use examples and visual aids where possible to help support your answer.

CHAPTER 2: OBJECT ORIENTED PROGRAMMING

In this chapter, we'll dive into the world of object-oriented programming (OOP). If you're new to programming, just starting your career, or have less than five years of experience, it's quite common for your interviews to focus on the fundamentals of OOP.

The interviewer's goal is to assess your grasp of this topic, evaluating your strength in designing code based on these principles, especially since Java is an object-oriented language. It's essential to be well-versed in key OOP concepts like abstraction, encapsulation, inheritance, and polymorphism. Real-life examples can help you explain these concepts better, and it's also important to understand the SOLID principles.

Failing to provide satisfactory answers to these questions can lead to rejection because no one wants to hire a software engineer who lacks proficiency in these fundamental aspects.

Let’s dive into the interview questions,

What are the four principles of OOP?

The four principles of Object-Oriented Programming (OOP) are:

Encapsulation: This refers to the practice of hiding the internal workings of an object and exposing only the necessary functionality. The data and behaviour of an object are encapsulated within the object, and can only be accessed through well-defined interfaces.

Inheritance: Inheritance allows objects to inherit properties and behaviours from other objects. Inheritance allows for the creation of hierarchical relationships between classes, with parent classes passing down their characteristics to their child classes.

Polymorphism: Polymorphism refers to the ability of objects to take on many forms, and is achieved through the use of inheritance,

overloading and overriding methods, and interfaces. Polymorphism allows for greater flexibility and reuse of code.

Abstraction: Abstraction refers to the process of identifying common patterns and extracting essential features of objects, creating classes from these patterns. Abstraction allows for the creation of higher-level concepts that can be used in multiple contexts, and can simplify complex systems.

What is the difference between an abstract class and an interface?

Feature	Abstract Class	Interface
Purpose	Partial abstraction, shared implementation	Complete abstraction, contract
Methods	Can have abstract and non-abstract methods	Traditionally only abstract methods (can have default and static methods in Java 8+)
Inheritance	Supports single inheritance	Supports multiple inheritance
Variables	Can have static, non-static, final, non-final	Only static final variables
Implementation	Subclass must implement all abstract methods	Implementing class must provide code for all methods
Instantiation	Cannot be instantiated directly	Cannot be instantiated directly
Constructors	Can have constructors	Cannot have constructors
Main method	Can have a main method	Cannot have a main method

When to Use:

Abstract classes:

- When you have some common implementation to share among subclasses.
- When you want to enforce a hierarchy and prevent direct instantiation of the base class.

- When you need to control access to members using access modifiers.

Interfaces:

- When you want to define a contract that multiple unrelated classes can implement.
- When you want to achieve loose coupling between classes.
- When you need to support multiple inheritance.

What is the use of constructor in an abstract class?

While abstract classes cannot be instantiated directly, their constructors play a crucial role in object initialization within inheritance hierarchies. Here are the key purposes of constructors in abstract classes:

Initializing Member Variables:

- Abstract classes often have member variables that need to be initialized for proper object state. Constructors perform this initialization, ensuring consistent setup for all subclasses.
- Example: An abstract Shape class might have a color property initialized in its constructor.

Enforcing Invariants and Constraints:

- Constructors can enforce rules and constraints that must hold true for all objects in the hierarchy. This ensures data integrity and validity.
- Example: A BankAccount abstract class might require a non-negative initial balance in its constructor.

Shared Initialization Logic:

- Common initialization steps for all subclasses can be consolidated in the abstract class constructor, reducing code duplication.

- Example: An Employee abstract class might initialize a hireDate property in its constructor, shared by all employee types.

Controlling Instantiation:

- Constructors can be made private or protected to control how subclasses are created, ensuring they are instantiated through specific mechanisms or helper methods.
- Example: A Singleton abstract class might have a private constructor to enforce a single instance.

Calling Superclass Constructors:

- Subclasses must call their superclass constructor (implicitly or explicitly) during their own construction. This ensures proper initialization of inherited state.
- Example: A SavingsAccount subclass must call the BankAccount constructor to initialize shared account properties.

Key Points:

- Abstract class constructors are not used for object creation directly, but they are invoked when a subclass object is created.
- They ensure consistent initialization and enforce class invariants, promoting code reusability and maintainability.
- Understanding constructor behaviour in abstract classes is essential for effective object-oriented design.

What is abstraction, and what are its advantages?

(Concrete class doing the same what is the advantage over concrete class?)

Abstraction is a fundamental concept in programming and many other fields. In simplest terms, it's the act of focusing on the essential details of something while hiding away the unnecessary complexity. Here's a breakdown of abstraction in programming:

What is it?

- Abstraction allows you to break down complex systems into smaller, easier-to-understand pieces.
- You define interfaces or classes that expose only the relevant functionalities, hiding the inner workings.
- This lets you work with the system at a higher level, without getting bogged down in the low-level details.

Think of it this way:

- Imagine a car. You don't need to understand the intricate mechanics of the engine to drive it. You just need to know how to steer, accelerate, and brake.
- Similarly, when using an abstraction in programming, you don't need to know how it works internally. You simply call its functions or methods and interact with it at a higher level.

Advantages of abstraction:

- Reduced complexity: It makes your code easier to understand, write, and maintain by breaking down large problems into smaller, more manageable chunks.
- Increased productivity: You can focus on the logic and functionality of your program without getting bogged down in implementation details.
- Improved reusability: Abstracted components can be used in multiple parts of your program, reducing code duplication and promoting modularity.
- Enhanced readability: Code becomes more concise and less cluttered, making it easier for others to understand and collaborate on.
- Flexibility and adaptability: Abstractions allow you to change the underlying implementation without affecting the code that uses them.

Here are some specific examples of abstraction in programming:

- Functions: They hide the implementation details of a specific task and provide a simple interface for other parts of your program to interact with.
- Classes: They bundle related data and functionality, making it easier to manage and reuse complex data structures.
- Libraries and frameworks: They provide pre-built abstractions for common tasks, saving you time and effort.

What is the difference between abstraction and encapsulation?

Abstraction:

- Focus: What the object does, hiding implementation details.
- Goal: Simplifying complex systems by exposing only essential features.
- Mechanisms: Abstract classes, interfaces, functions.

Encapsulation:

- Focus: How the object's data and behavior are bundled together.
- Goal: Protecting data integrity and controlling access.
- Mechanisms: Access modifiers (public, private, protected), getters and setters.

Key Differences:

- Scope: Abstraction operates at a higher level, focusing on the overall design and interface. Encapsulation works at the object level, managing internal data and implementation.
- Purpose: Abstraction aims to simplify complexity and promote reusability. Encapsulation aims to protect data and manage dependencies.

- Implementation: Abstraction is often achieved through abstract classes or interfaces. Encapsulation is typically implemented using access modifiers and methods to control access to data.

What is the difference between Abstraction and polymorphism?

Abstraction

- Focus: Hides the internal complexity of an object, exposing only the essential features and functionalities that users need to interact with.
- Think of it as: A map that shows the important landmarks of a city without getting bogged down in the details of every street and alleyway.

Benefits:

- Simplifies code by reducing cognitive load and making it easier to understand.
- Promotes code reusability by focusing on general functionalities that can be applied in different contexts.
- Improves maintainability by making it easier to change the implementation details without affecting the code that uses the abstraction.

Mechanisms:

- Abstract classes: Define a blueprint for subclasses with shared functionality and abstract methods that must be implemented.
- Interfaces: Specify contracts that classes must adhere to, defining methods without implementation.
- Functions: Hide the internal logic of a specific task, providing a simple interface for other parts of the program to interact with.

Polymorphism

- Focus: Enables an object to exhibit different behaviors depending on its actual type at runtime.
- Think of it as: A chameleon that can change its color to blend in with its surroundings.

Benefits:

- Makes code more flexible and adaptable by allowing different objects to respond differently to the same message.
- Promotes code reusability by enabling generic functions and methods that can work with different types of objects.
- Improves maintainability by making it easier to add new types of objects without modifying existing code.

Mechanisms:

- Method overloading: Allows a class to define multiple methods with the same name but different parameter types or numbers.
- Method overriding: Allows subclasses to provide their own implementation of a method inherited from a superclass.
- Interfaces: Can define abstract methods with common behavior that different classes can implement in their own way.

Feature	Abstraction	Polymorphism
Focus	What an object does	How an object behaves
Goal	Simplify complexity, hide internal details	Provide flexibility, adapt behavior based on type
Mechanisms	Abstract classes, interfaces, functions	Method overloading, overriding, interfaces
Benefits	Reduced complexity, improved reusability, maintainability	Increased flexibility, adaptability, reusability

What is the difference between Inheritance and Composition?

Inheritance allows a class (called a subclass) to inherit properties and behaviors from another class (called a superclass). The subclass can then add or modify these properties and behaviors as needed. It's useful for creating hierarchies of related classes and sharing code and functionality.

For example, if we have an Animal class, a Mammal class, and a Cat class, the Cat class can inherit properties and behaviors from both Animal and Mammal classes while adding its own specific methods.

Benefits:

- Promotes code reuse by sharing common functionalities among related classes.
- Provides code organization by structuring classes in a hierarchy.
- Enables specialization by adding specific features to subclasses.

Composition allows a class to be composed of other objects. This means that a class can have references to other objects as its properties and use them to delegate tasks or behaviors. It's useful for creating complex objects from simpler ones and enabling dynamic composition at runtime.

For instance, a Car class can be composed of objects such as an Engine, Wheels, Seats, etc. The Car class can then utilize these objects to perform various tasks.

Benefits:

- Loose coupling between classes – changes in one class usually don't affect the other.
- Greater flexibility – allows using functionalities from any class, not just parent-child hierarchy.
- Promotes modularity and code clarity.

Feature	Inheritance	Composition

Relationship	"is-a"	"has-a"
Implementation	Subclasses inherit from superclass	Member variables hold other objects
Benefits	Code reuse, organization, specialization	Loose coupling, flexibility, modularity
Drawbacks	Tight coupling, limited flexibility, duplication	Complexity, lifecycle management

What are Composition and Aggregation with examples?

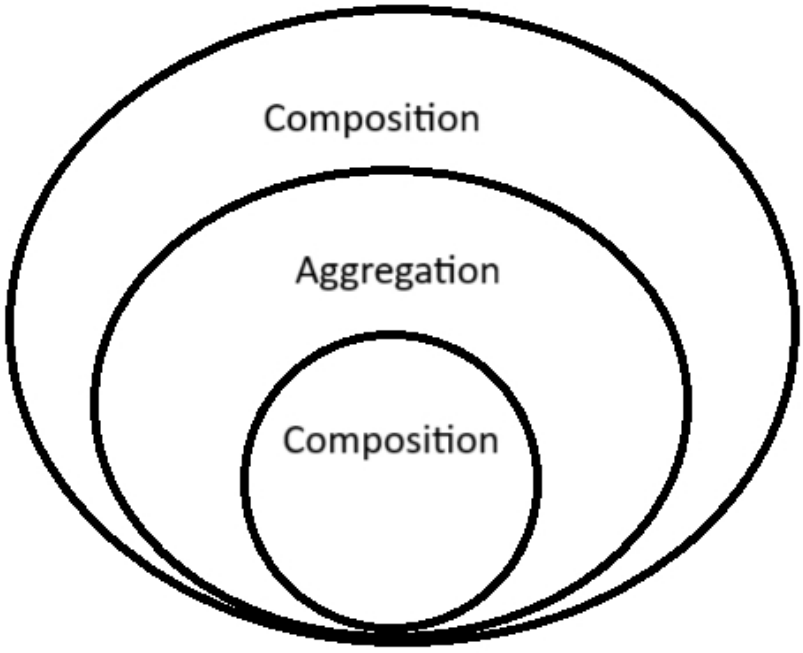
Composition and aggregation are two types of object-oriented programming concepts that describe the relationship between objects.

Composition is a strong type of association where an object is made up of one or more objects of other classes. For example, a car is composed of various parts such as wheels, engine, transmission, etc. The car class has an object of the wheel class, engine class, and transmission class as its member variables.

Aggregation is a weak type of association where an object contains a reference to one or more objects of other classes. For example, a university class has a collection of student classes as its member variable. The student class has an object of the university class as its member variable.

What is aggregation, composition, and inheritance?

To check if the current code contains examples of aggregation, composition, and inheritance, you need to look for the relevant syntax and usage patterns in the code.



Here are some pointers for identifying these concepts in code:

Inheritance: Look for classes that extend or inherit from other classes. This is typically indicated by the extends keyword in Java, for example: public class Car extends Vehicle {...}. Inheritance is used to create a hierarchy of classes where subclasses inherit properties and methods from their parent classes.

Composition: Look for objects that contain other objects as instance variables. This is typically indicated by object instantiation within another object's constructor, for example:

```
public class Person {
        private Job job;

        public Person(Job job) {
        this.job = job;
        }
}
```

Composition is used to build complex objects by combining simpler objects.

Aggregation: Look for objects that have references to other objects as instance variables, but do not own or create them. This is typically indicated by a "has-a" relationship between objects,

for example:

```
public class University {
        private List<Student> students;

        public University(List<Student> students) {
        this.students = students;
        }
}
```

Aggregation is used to represent relationships between objects without tightly coupling them together.

To get a better understanding of how these concepts are used in code, you may want to read through the codebase and look for patterns that indicate their usage. Additionally, you may want to search for specific keywords and syntax related to inheritance, composition, and aggregation, such as extends, implements, and new. Finally, you may also want to talk to other developers or review documentation to understand how these concepts are being used in the codebase.

Can you explain multilevel inheritance in Java?

Multilevel inheritance is a type of inheritance in object-oriented programming (OOP) where a derived class (subclass) is created from another derived class, which itself was derived from a base class (superclass).

In multilevel inheritance, each derived class inherits the characteristics of the class above it in the hierarchy. This means that a subclass not only has all the features of its immediate superclass, but also those of all its ancestors up the hierarchy chain.

Here's an example to illustrate multilevel inheritance:

class Animal {


```
    void eat() {
        System.out.println("Eating...");
    }
```


```
}
class Dog extends Animal {
```

```
    void bark() {
        System.out.println("Barking...");
```


```
    }
}
class Bulldog extends Dog {
    void guard() {
        System.out.println("Guarding...");
    }
}
```

In this example, Animal is the base class, Dog is a derived class from Animal, and Bulldog is a derived class from Dog.

Animal has a single method eat(). Dog inherits eat() from Animal and adds a new method bark(). Bulldog inherits both eat() and bark() from Dog and adds a new method guard().

Now, an instance of Bulldog can access all the methods of its immediate superclass (Dog), as well as all the methods of its

ancestor superclass (Animal). For example:

```
Bulldog bulldog = new Bulldog();
bulldog.eat();   // output: Eating...
bulldog.bark();  // output: Barking...
bulldog.guard(); // output: Guarding...
```

This example demonstrates how multilevel inheritance can be used to create a hierarchy of classes that inherit and extend behavior from each other. However, it is important to use inheritance judiciously to avoid creating overly complex and tightly-coupled class hierarchies.

When do you use encapsulation and abstraction in your project?

Encapsulation and abstraction are two important concepts in object-oriented programming, and they are used in different ways in different parts of a project.

Encapsulation is used to protect the internal state of an object and to control how other objects can access or modify that state. It is typically used in data modelling, where we define classes that represent real-world entities and their properties.

For example, if we were building a system to manage a library, we might define a Book class that has properties like title, author, and isbn. We would use encapsulation to ensure that these properties are not accessible or modifiable from outside the Book class, except through carefully designed methods like getTitle() and setAuthor().

Abstraction, on the other hand, is used to hide the implementation details of a class or component and to present a simpler, higher-level interface to other parts of the system. It is typically used in system design and architecture, where we define components and their interfaces.

For example, if we were building a web application, we might define a UserService component that provides methods for creating, updating, and retrieving user accounts. We would use abstraction to

ensure that other components in the system do not need to know how the UserService is implemented, but can simply use its interface to perform the necessary actions.

In general, encapsulation and abstraction are used together in object-oriented programming to create robust, maintainable, and scalable systems. Encapsulation is used to protect the internal state of objects and to control how other objects can access or modify that state, while abstraction is used to hide the implementation details of components and to present a simpler, higher-level interface to other parts of the system.

How do you achieve encapsulation?

Encapsulation is achieved in Java through the use of access modifiers and getter and setter methods.

Access modifiers control the visibility of variables and methods in a class. There are three access modifiers in Java: public, private, and protected.

Public: Public variables and methods can be accessed from anywhere in the program.

Private: Private variables and methods can only be accessed within the same class.

Protected: Protected variables and methods can be accessed within the same class, and by subclasses and classes in the same package.

By default, if you don't specify an access modifier, the variable or method is considered to have "package" or "default" access, which means it can be accessed within the same package.

Here's an example of how to use access modifiers to achieve encapsulation:

```
public class Person {
    private String name;
    private int age;
    public String getName() {
```

```
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be
negative");
        }
        this.age = age;
    }
}
```

In this example, the Person class has two private variables, name and age. These variables are not directly accessible from outside the class, which means that other classes cannot modify or access them directly.

To allow other classes to access these variables, we provide public getter and setter methods for name and age. The getter methods allow other classes to retrieve the values of these variables, while the setter methods allow other classes to modify their values.

Note that we can also add validation logic to the setter methods to ensure that the values being set are valid. In this example, the setAge method throws an exception if the age is negative.

By using access modifiers and getter and setter methods, we can achieve encapsulation in Java. This allows us to protect the data and

behavior of our objects and prevent other objects from accessing or modifying them directly, which makes our code more robust and maintainable.

What is polymorphism, and how can it be achieved?

Polymorphism is the ability of objects of different classes to be treated as if they are of the same type. It allows us to write code that can work with objects of different types in a uniform way, without needing to know the specific class of each object.

In Java, polymorphism is achieved through two mechanisms: method overloading and method overriding.

Method overloading is when a class has two or more methods with the same name, but different parameters. When a method is called, the compiler determines which method to call based on the number and types of the arguments passed to it.

```
public class Calculator {
    public int add(int x, int y) {
        return x + y;
    }
    public double add(double x, double y) {
        return x + y;
    }
}
```

In this example, the Calculator class has two methods named add, one that takes two integers and one that takes two doubles. When the add method is called, the compiler determines which version of the method to call based on the types of the arguments passed to it.

Method overriding is when a subclass provides its own implementation of a method that is already defined in its superclass. The subclass method must have the same name, return type, and parameter list as the superclass method.

```
public class Animal {
    public void speak() {
        System.out.println("Animal speaks");
```

```
    }
}
public class Dog extends Animal {
    public void speak() {
        System.out.println("Dog barks");
    }
}
```

In this example, the Animal class has a method named speak. The Dog class extends the Animal class and provides its own implementation of the speak method. When we call the speak method on a Dog object, the Dog version of the method is called instead of the Animal version.

Polymorphism allows us to write code that can work with objects of different types in a uniform way, without needing to know the specific class of each object. It makes our code more flexible and easier to maintain, and is a key feature of object-oriented programming.

What are Method Overloading and Overriding?

Method overloading and overriding are two concepts in object-oriented programming that describe how methods in a class can be used.

Method overloading is the ability of a class to have multiple methods with the same name but with different parameters. This is also known as "compile-time polymorphism" or "function overloading" in some languages. For example, a class Calculator might have multiple methods with the name add, but with different parameters such as add (int a, int b) and add (double a, double b).

Method overriding is the ability of a subclass to provide a different implementation of a method that is already provided by its superclass. This is also known as "runtime polymorphism" or "function overriding". The method in the subclass has the same name, return type and parameters as the method in the superclass.

The purpose of method overriding is to change the behaviour of the method in the subclass.

Here's an example of Overloading:

```
public class Calculator {
    public int add(int x, int y) {
        return x + y;
    }
    public int add(int x, int y, int z) {
        return x + y + z;
    }
    public double add(double x, double y) {
        return x + y;
    }
}
```

In this example, Calculator defines three different add() methods with the same name but different parameters. The first method takes two int arguments, the second takes three int arguments, and the third takes two double arguments. The compiler decides which method to call based on the number and type of arguments passed to it.

Method overriding, on the other hand, is the ability to define a method in a subclass that has the same name and parameters as a method in its superclass. When the method is called on an object of the subclass, the method in the subclass is executed instead of the method in the superclass.

Here's an example overriding:

```
public class Animal {
    public void makeSound() {
        System.out.println("The animal makes a sound");
    }
```

```
}
public class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("The dog barks");
    }
}
```

In this example, Animal defines a makeSound() method that prints a message. Dog overrides this method with its own implementation that prints a different message. When makeSound() is called on a Dog object, the overridden method in Dog is executed, producing the output "The dog barks".

What is Method overriding with an example?

In Java, method overriding is when a subclass provides its own implementation of a method that is already defined in its superclass. The subclass method must have the same name, return type, and parameter list as the superclass method.

Access specifiers determine the visibility of a method, and they can also be used when overriding methods. When overriding a method, the access specifier of the overriding method cannot be more restrictive than the access specifier of the overridden method. In other words, if the overridden method is public, the overriding method must also be public or less restrictive.

Here is an example of method overriding with access specifiers:

```
public class Animal {
    public void speak() {
        System.out.println("Animal speaks");
    }
    protected void eat() {
        System.out.println("Animal eats");
```

```
    }
}
public class Dog extends Animal {
    @Override
    public void speak() {
        System.out.println("Dog barks");
    }
    @Override
    protected void eat() {
        System.out.println("Dog eats");
    }
}
```

In this example, the Animal class has a method named speak that is public, and a method named eat that is protected. The Dog class extends the Animal class and provides its own implementations of the speak and eat methods.

The speak method in the Dog class overrides the speak method in the Animal class and is also public. The eat method in the Dog class overrides the eat method in the Animal class and is also protected. Since the eat method in the Animal class is also protected, the access specifier of the eat method in the Dog class can be the same or less restrictive, but not more restrictive.

In this way, we can use method overriding to provide our implementations of methods that are defined in a superclass, while also adhering to the access specifiers of the original methods. This allows us to customize the behavior of our subclasses while maintaining the structure and visibility of the superclass.

What is Method overriding in terms of exception handling, with an example?

In Java, when overriding a method, the overridden method can throw exceptions, and the overriding method can choose to throw

the same exceptions or a subset of them. Here's an example of method overriding with exception handling:

```
public class Animal {
    public void speak() throws Exception {
        System.out.println("Animal speaks");
    }
    public void eat() throws Exception {
        System.out.println("Animal eats");
    }
}
public class Dog extends Animal {
    @Override
    public void speak() throws IOException {
        System.out.println("Dog barks");
        throw new IOException("Exception from Dog.speak");
    }
    @Override
    public void eat() {
        System.out.println("Dog eats");
    }
}
```

In this example, the Animal class has two methods: speak and eat. Both methods are declared to throw an Exception.

The Dog class extends the Animal class and overrides both the speak and eat methods.

The speak method in the Dog class overrides the speak method in the Animal class and throws an IOException. The IOException is a subclass of Exception, so this is allowed.

The eat method in the Dog class overrides the eat method in the Animal class but does not throw any exceptions.

When calling these methods, we can catch the exceptions that are thrown. For example:

```
public static void main(String[] args) {
    Animal animal = new Dog();
    try {
        animal.speak();
    } catch (IOException e) {
        System.out.println("Caught IOException: " + e.getMessage());
    } catch (Exception e) {
        System.out.println("Caught Exception: " + e.getMessage());
    }
    try {
        animal.eat();
    } catch (Exception e) {
        System.out.println("Caught Exception: " + e.getMessage());
    }
}
```

In this example, we create an instance of the Dog class and assign it to a variable of type Animal. We then call the speak and eat methods on this object.

Since the speak method in the Dog class throws an IOException, we catch that exception specifically and print out its message. If the speak method in the Dog class threw a different type of exception, such as RuntimeException, it would not be caught by this catch block.

The eat method in the Dog class does not throw any exceptions, so the catch block for Exception will not be executed. If the eat method

in the Dog class did throw an exception, it would be caught by this catch block.

Can we have overloaded methods with different return types?

No, we cannot have overloaded methods with only different return types. Overloaded methods must have the same method name and parameters, but they can have different return types only if the method parameters are also different. The reason for this is that the return type alone is not enough information for the compiler to determine which method to call at compile time. For example, consider the following code:

```
public class Calculator {
    public int add(int x, int y) {
        return x + y;
    }
    public double add(int x, int y) {
        return (double) (x + y);
    }
}
```

In this example, we have two add() methods with the same parameters, but different return types (int and double). This will cause a compilation error because the compiler cannot determine which method to call based on the return type alone.

Can we override the static method? Why Can't we do that?

No, it is not possible to override a static method in Java. A static method is associated with the class and not with an object, and it can be called directly on the class, without the need of creating an instance of the class.

When a subclass defines a static method with the same signature as a static method in the superclass, the subclass method is said to hide the superclass method. This is known as method hiding. The

subclass method is not considered an override of the superclass method, it is considered a new method and it hides the superclass method, but it doesn't override it.

In summary, since a static method is associated with the class, not an object and it is directly callable on the class, it is not possible to override a static method in Java, instead, it is hidden by subclass method with the same signature.

What are SOLID principles, with example?

SOLID is an acronym that represents five principles of object-oriented design that aim to make software more maintainable, flexible, and easy to understand. Here are the explanations of each of the five SOLID principles, along with examples:

Single Responsibility Principle (SRP): This principle states that a class should have only one reason to change. In other words, a class should have only one responsibility or job. For example, if we have a class named User, it should only be responsible for handling user-related functionalities such as authentication, user data management, etc. It should not be responsible for other unrelated functionalities like sending emails or managing payment transactions.

Open/Closed Principle (OCP): This principle states that a class should be open for extension but closed for modification. This means that we should be able to add new functionalities or behaviors to a class without changing its existing code. For example, instead of modifying an existing Payment class to add support for a new payment method, we can create a new PaymentMethod class that implements a Payment interface and inject it into the Payment class.

Liskov Substitution Principle (LSP): This principle states that derived classes should be able to replace their base classes without affecting the correctness of the program. In other words, if we have a base class Animal and a derived class Dog, we should be able to use the Dog class wherever we use the Animal class. For example, if we have a method that takes an Animal parameter and performs

some action, we should be able to pass a Dog object to that method without any issues.

Interface Segregation Principle (ISP): This principle states that a class should not be forced to implement interfaces it does not use. In other words, we should separate interfaces that are too large or general into smaller and more specific interfaces. For example, instead of having a single Payment interface that includes all payment methods, we can have separate interfaces like CreditCardPayment, PayPalPayment, etc.

Dependency Inversion Principle (DIP): This principle states that high-level modules should not depend on low-level modules. Instead, both should depend on abstractions. This means that we should rely on abstractions, rather than concrete implementations. For example, instead of depending on a specific database implementation in a class, we should depend on a database interface, which can be implemented by different databases. This allows for easier testing, maintenance, and scalability.

What is Cohesion and Coupling?

Cohesion refers to the degree to which the elements within a module or component work together to achieve a single, well-defined purpose. High cohesion means that the elements within a module are strongly related and work together towards a common goal, while low cohesion means that the elements are loosely related and may not have a clear purpose.

Coupling, on the other hand, refers to the degree of interdependence between modules or components. High coupling means that a change in one module or component will likely affect other modules or components, while low coupling means that changes in one module or component will have minimal impact on other modules or components.

In general, software design principles strive for high cohesion and low coupling, as this leads to code that is more modular, maintainable, and easier to understand and change.

What does Static keyword signify?

In Java, the static keyword is used to create variables, methods, and blocks that belong to the class, rather than to an instance of the class. When a variable or method is declared as static, it is associated with the class and not with individual objects of the class. Here are some common uses of the static keyword:

Static variables: A static variable is a variable that belongs to the class and is shared by all instances of the class. Static variables are declared using the static keyword and are often used for constants or for variables that need to be shared across instances of the class.

```
public class Example {
    public static int count = 0;
}
```

In this example, the count variable is declared as static, so it belongs to the Example class and is shared by all instances of the class.

Static methods: A static method is a method that belongs to the class and can be called without creating an instance of the class. Static methods are declared using the static keyword and are often used for utility methods that do not depend on the state of an instance.

```
public class Example {
    public static void printMessage(String message) {
        System.out.println(message);
    }
}
```

In this example, the printMessage() method is declared as static, so it belongs to the Example class and can be called without creating an instance of the class.

Static blocks: A static block is a block of code that is executed when the class is loaded. Static blocks are used to initialize static variables or to perform other one-time initialization tasks.

```
public class Example {
    static {
        System.out.println("Initializing Example class");
    }
}
```

In this example, the static block is executed when the Example class is loaded and prints a message to the console.

In summary, the static keyword is used to create variables, methods, and blocks that belong to the class, rather than to an instance of the class. This allows these elements to be shared by all instances of the class and to be accessed without creating an instance of the class.

What is the difference between static variable and an instance variables?

Static variables and instance variables are both types of variables used in programming, but they differ in their scope and lifetime.

Static variables are declared using the "static" keyword and are shared across all instances of a class. They are initialized only once, when the class is loaded, and retain their value throughout the execution of the program. Static variables are typically used to store data that is common to all instances of a class, such as a constant or a count of objects created.

Instance variables, on the other hand, are declared without the "static" keyword and are unique to each instance of a class. They are initialized when an object is created and are destroyed when the object is destroyed. Instance variables are typically used to store data that is specific to each instance of a class, such as the name or age of a person.

In summary, the main differences between static variables and instance variables are:

Scope: Static variables have class scope, while instance variables have object scope.

Lifetime: Static variables are initialized once and retain their value throughout the execution of the program, while instance variables are created and destroyed with the objects they belong to.

Usage: Static variables are used to store data that is common to all instances of a class, while instance variables are used to store data that is specific to each instance of a class.

What is Covariant type?

Covariant type refers to the ability to use a subclass type in place of its superclass type. In other words, it allows a subclass to be used in place of its superclass. This feature is supported by some programming languages such as Java and C#.

For example, if class B is a subclass of class A, then an object of class B can be used wherever an object of class A is expected. Here is an example in Java:

```
class A { }

class B extends A { }

A a = new A();

B b = new B();

a = b; // valid
```

In the above example, the variable "a" is of type A, and the variable "b" is of type B. However, the assignment "a = b" is valid, because B is a subclass of A.

Covariant return types allow a method to return a subclass type in place of its superclass type.

```
class A { }

class B extends A { }

class C {

    A getA() { return new A(); }

    B getB() { return new B(); }
```

}

In the above example, the method getA() returns an object of type A, and the method getB() returns an object of type B. Because B is a subclass of A, the method getB() can be overridden to return B instead of A.

In summary, Covariant type refers to the ability to use a subclass type in place of its superclass type, this feature is supported by some programming languages such as Java and C#. It allows a subclass to be used in place of its superclass and covariant return types allow a method to return a subclass type in place of its superclass type.

Can Java interface have no method in it?

An interface without any methods is called a "marker interface". It is used to mark a class as having some specific behavior or property, without specifying any methods for that behavior or property.

For example, the Serializable interface in Java has no methods:

public interface Serializable {

}

However, classes that implement the Serializable interface gain the ability to be serialized and deserialized, which is the behavior that the Serializable marker interface indicates.

Here's an example of using a marker interface:

In this example, we define a marker interface MyMarkerInterface with no methods. We then define a class MyClass that implements the MyMarkerInterface interface. This indicates that MyClass has some specific behavior or property that is indicated by the MyMarkerInterface marker interface. However, since MyMarkerInterface has no methods, MyClass does not need to implement any methods as a result of implementing the MyMarkerInterface interface.

What are the exception rules for overriding?

When overriding a method in Java, there are some rules that must be followed regarding exceptions:

The overriding method can throw the same exceptions as the overridden method, or any subset of those exceptions.

The overriding method can also throw unchecked exceptions, even if the overridden method does not.

The overriding method cannot throw checked exceptions that are not in the same class hierarchy as the exceptions thrown by the overridden method. This means that the overriding method cannot throw checked exceptions that are more general than those thrown by the overridden method. However, it can throw more specific checked exceptions or unchecked exceptions.

If the overridden method does not throw any exceptions, the overriding method cannot throw checked exceptions.

Here's an example to illustrate these rules:

```
class Parent {
    void foo() throws IOException {
        // implementation goes here
    }
}
class Child extends Parent {
    // this is a valid override
    void foo() throws FileNotFoundException {
        // implementation goes here
    }
    // this is not a valid override
    void foo() throws Exception {
        // implementation goes here
    }
```

```
    // this is a valid override
    void foo() throws RuntimeException {
        // implementation goes here
    }
    // this is not a valid override
    void foo() throws SQLException {
        // implementation goes here
    }
    // this is a valid override, since it does not throw any exceptions
    void foo() {
        // implementation goes here
    }
}
```

In this example, Parent has a method foo() that throws an IOException. Child overrides this method and follows the rules for exception handling:

The first override is valid, since FileNotFoundException is a subclass of IOException.

The second override is not valid, since Exception is a more general exception than IOException.

The third override is valid, since RuntimeException is an unchecked exception and can be thrown even if the overridden method does not throw any exceptions.

The fourth override is not valid, since SQLException is not in the same class hierarchy as IOException.

The fifth override is valid, since it does not throw any exceptions.

What are the different types of access modifiers?

There are four types of access modifiers in Java:

public: The public access modifier is the most permissive access level, and it allows access to a class, method, or variable from any other class, regardless of whether they are in the same package or not.

protected: The protected access modifier allows access to a class, method, or variable from within the same package, as well as from any subclass, even if they are in a different package.

default (no modifier): If no access modifier is specified, then the class, method, or variable has package-level access. This means that it can be accessed from within the same package, but not from outside the package.

private: The private access modifier is the most restrictive access level, and it allows access to a class, method, or variable only from within the same class. It cannot be accessed from any other class, even if they are in the same package.

Here is an example of how access modifiers can be used:

```
public class MyClass {
    public int publicVar;
    protected int protectedVar;
    int defaultVar;
    private int privateVar;
    public void publicMethod() {
    }
    protected void protectedMethod() {
    }
    void defaultMethod() {
    }
    private void privateMethod() {
    }
}
```

In this example, the MyClass class has four instance variables and four instance methods, each with a different access modifier. The publicVar and publicMethod() can be accessed from any other class, while protectedVar and protectedMethod() can be accessed from any subclass and from within the same package. defaultVar and defaultMethod() can be accessed from within the same package only, while privateVar and privateMethod() can be accessed only from within the same class.

What is the difference between private and protected access modifiers?

The main difference between private and protected access modifiers in Java is that private members are only accessible within the same class, while protected members are accessible within the same class and its subclasses, as well as within the same package.

Here are some more differences between private and protected:

Visibility: Private members are only visible within the same class, while protected members are visible within the same class and its subclasses, as well as within the same package.

Access: Private members cannot be accessed outside the class, while protected members can be accessed by subclasses and other classes in the same package.

Inheritance: Private members are not inherited by subclasses, while protected members are inherited by subclasses.

Overriding: Private members cannot be overridden in subclasses, while protected members can be overridden in subclasses.

Here is an example to illustrate the difference between private and protected access modifiers:

```
public class MyClass {
    private int privateVar;
    protected int protectedVar;
    public void myMethod() {
        privateVar = 1; // OK, can be accessed within the same class
```

```
        protectedVar = 2; // OK, can be accessed within the same class
    }
}
public class MySubclass extends MyClass {
    public void mySubMethod() {
        // privateVar = 3; // Error, cannot be accessed in subclass
        protectedVar = 4; // OK, can be accessed in subclass
    }
}
public class MyOtherClass {
    public void myOtherMethod() {
        MyClass obj = new MyClass();
        // obj.privateVar = 5; // Error, cannot be accessed outside //the class
        // obj.protectedVar = 6; // Error, cannot be accessed //outside the package or subclass
    }
}
```

In this example, we have a class MyClass with a private variable privateVar and a protected variable protectedVar. The myMethod() method of MyClass can access both variables. We also have a subclass MySubclass of MyClass, which can access the protectedVar variable, but not the privateVar variable. Finally, we have another class MyOtherClass, which cannot access either variable, because they are not visible outside the class or its package.

What is the use of protected members?

Protected members in Java are used to provide access to class members within the same package and to subclasses of the class, even if they are in a different package. The protected access

modifier is more restrictive than public, but less restrictive than private.

The protected access modifier is useful when you want to expose certain methods or variables to subclasses, while still hiding them from other classes in the same package or outside the package. For example, you might have a superclass with some variables and methods that are not intended to be used outside the class or package, but should be accessible to subclasses. In this case, you can declare those variables and methods as protected.

Here is an example of how protected members can be used:

```
package com.example.package1;

public class Superclass {
    protected int protectedVar;

    protected void protectedMethod() {
        // ...
    }
}
```

```
package com.example.package2;

import com.example.package1.Superclass;

public class Subclass extends Superclass {
    public void someMethod() {
        protectedVar = 42;
        protectedMethod();
    }
}
```

In this example, we have two packages, com.example.package1 and com.example.package2. Superclass is defined in package1, and has a protected variable protectedVar and a protected method protectedMethod(). Subclass is defined in package2 and extends

Superclass. In the someMethod() method of Subclass, we can access protectedVar and protectedMethod() from Superclass, even though they are protected, because Subclass is a subclass of Superclass.

CHAPTER 3: CORE JAVA

This chapter deals with Core Java Interview question, All the questions that I am writing here throughout the books are really important. These are the question those are appeared in an interview.

Even with questions you will get the idea like which topic is important and which is not. As an interviewee we should be focussing on the hot topics to prepare better.

In Core Java, these topics are most important.

- String
- Collection framework (HashMap, Concurrent HashMap)
- Concepts of immutability
- Exception
- Serialization
- Garbage collectors
- Multithreading
- Executor Framework
- Lambdas
- Stream
- Java 8 all new features
- Optional
- Functional interface

What is JIT in java?

JIT stands for "Just-In-Time" compiler in Java. It is a feature of the Java Virtual Machine (JVM) that improves the performance of Java applications by compiling bytecode into native machine code at runtime. The JIT compiler examines the bytecode of a method and compiles it into machine code if it determines that the method is executed frequently. This allows for faster execution of the compiled code, rather than interpreting the bytecode each time it is executed.

What is the difference between abstract and interface keyword in java?

In Java:

1. abstract: keyword is used to define an abstract class, which can have both abstract and concrete methods. Abstract classes can have instance variables and constructors.
2. interface: keyword is used to declare an interface, which can only contain abstract method signatures and constants (public static final fields). Interfaces cannot have instance variables or constructors.

What is the difference between Static method and Default methods in Java 8?

In Java 8 and later versions, interfaces can have two types of methods: default and static methods.

A default method in an interface provides a default implementation for the method, which is used when no implementation is provided by a class that implements the interface. Default methods are marked with the default keyword, and can be overridden by a class that implements the interface. They are useful for adding new methods to an interface without breaking the existing implementations of that interface.

A static method in an interface is a method that is associated with the interface itself, rather than with any particular instance of the interface. Static methods are marked with the static keyword, and can be called directly on the interface, without the need for an instance of the interface. They are useful for providing utility methods related to the interface.

Here are some key differences between static and default methods in Java interfaces:

A default method is an instance method, while a static method is a class method.

A default method can be overridden by a class that implements the interface, while a static method cannot be overridden.

Default methods can access instance variables of the implementing class, while static methods cannot.

Default methods can be called using an instance of the implementing class, while static methods can be called directly on the interface.

Static methods cannot access instance variables or methods of the implementing class, while default methods can.

In general, default methods are used to add new functionality to an interface, while static methods are used to provide utility methods that are related to the interface.

How to create an Object in java? Explain Different ways to do it.

In Java, an object is an instance of a class and can be created using the new keyword. The new keyword is used to create an object of a class by calling its constructor. Here's an example:

```
public class MyClass {
    public MyClass() {
        // constructor implementation
    }
}

MyClass myObject = new MyClass();
```

In this example, MyClass is the class and myObject is the object that is created using the new keyword and calling the constructor of the class MyClass(). Once an object is created, you can use it to call methods and access fields of the class.

It's also possible to create an object using the Class.forName() method, which returns the Class object associated with the class or interface with the given string name. With the Class object, you can create an instance of the class using the newInstance() method.

```
Class<?> myClass = Class.forName("com.example.MyClass");
MyClass myObject = (MyClass) myClass.newInstance();
```

It's worth mentioning that when an object is created, the memory for the object is allocated on the heap, and the object's constructor is called to initialize the object's state. When the object is no longer being used, the memory for the object is reclaimed by the garbage collector.

How to create an immutable class?

To make a class immutable, you need to ensure that its state cannot be modified once it has been initialized. Here are some steps to create an immutable class:

1. Declare all instance variables as private and final.
2. Provide only getter methods for the instance variables, and make sure they only return the value without allowing modification.
3. Do not provide any setter methods or any other methods that can modify the state of the object.
4. Ensure that the class cannot be subclassed by declaring it as final or making its constructor private.

Here is an example of an immutable class:

```
public final class ImmutableClass {
   private final String name;
   private final int age;
   public ImmutableClass(String name, int age) {
      this.name = name;
      this.age = age;
   }
   public String getName() {
      return name;
   }
   public int getAge() {
      return age;
```

```
    }
}
```

How to restrict object creation in java?

In Java, there are several ways to restrict the creation of objects:

Making the constructor private: By making the constructor private, the class can only be instantiated within the class. This means that the object of the class cannot be created by any other class.

```
public class MyClass {
    private MyClass() {
        // constructor implementation
    }
}
```

Using the singleton pattern: This pattern is used to create a single instance of a class, and the same instance is returned every time the class is instantiated.

```
public class MySingleton {
    private static MySingleton instance = null;
    private MySingleton() {
        // constructor implementation
    }
    public static MySingleton getInstance() {
        if (instance == null) {
            instance = new MySingleton();
        }
        return instance;
    }
}
```

Using an abstract class or an interface: An abstract class or an interface cannot be instantiated, so objects of an abstract class or interface cannot be created.

```
abstract class MyAbstractClass {
    // class implementation
}
interface MyInterface {
    // interface methods
}
```

Using a factory method: A factory method is a method that creates and returns an object of a class. This method can be used to create an object of a class only if certain conditions are met.

```
public class MyFactory {
    public static MyClass createMyClass(String parameter) {
        }
```

How to create a custom class loader in java?

In Java, a class loader is responsible for loading classes into the JVM at runtime. The JVM includes several built-in class loaders, such as the bootstrap class loader and the system class loader. In some cases, you may need to create a custom class loader to load classes from a specific location or with specific characteristics. Here's an example of how to create a custom class loader:

```
public class MyClassLoader extends ClassLoader {

    public MyClassLoader(ClassLoader parent) {
        super(parent);
    }

    public Class loadClass(String name) throws
ClassNotFoundException {
        try {
            // Get the bytes of the class
```

```
            byte[] classBytes = loadClassBytes(name);
            if (classBytes == null) {
                throw new ClassNotFoundException();
            }

            // Define the class using the bytes
            return defineClass(name, classBytes, 0, classBytes.length);
        } catch (IOException e) {
            throw new ClassNotFoundException("Could not load class "
+ name, e);
        }
    }

    private byte[] loadClassBytes(String name) throws IOException {
        // Code to load the class bytes from a specific location
        // and return the bytes as a byte[]
    }
}
```

In this example, the MyClassLoader class extends the ClassLoader class and overrides the loadClass method. The loadClass method uses the loadClassBytes method to load the class bytes from a specific location and then calls the defineClass method to define the class using the bytes.

To use the custom class loader, you can create an instance of the MyClassLoader class and use the loadClass method to load the class:

```
MyClassLoader myClassLoader = new MyClassLoader(getParent());
Class myClass = myClassLoader.loadClass()
```

Can we write the main method without static?

In Java, the main method is the entry point of the program, and is typically declared as "public static void main(String [] args)". The keyword "static" is used to indicate that the main method is a class method and can be called without creating an instance of the class.

It's possible to write the main method without the static keyword, but in order to call it, you would need to create an instance of the class first, then call the main method on that instance.

```
class Main {
    public void main(String [] args) {
        // code
    }
}
```

you would need to create an instance of Main to call the main method like this:

```
Main main = new Main();
main.main(args);
```

It's not typical to use main method without static keyword, as it's not the standard way of running the program and also it will cause confusion among developers.

Explain the diamond problem in java and how to resolve it.

The Diamond problem in Java is a problem that arises when a class inherits from multiple classes that have a common base class. This problem occurs because Java does not support multiple inheritance of classes, but it allows a class to inherit from multiple interfaces.

```
class A {
    public void method1() {
        // implementation
    }
}

interface B extends A {
    public void method2();
}

interface C extends A {
    public void method3();
}
```

```
class D Implements B, C {
    // implementation
}
```

Here, class D inherits from both interfaces B and C, which both inherit from class A. If class A defines a method named method1, it is unclear which version of method1 should be inherited by class D, since it could inherit it from either B or C.

This problem is known as the Diamond problem because of the shape of the inheritance diagram, which looks like a diamond.

There are several ways to solve the diamond problem in Java:

Explicitly specifying which method to inherit: You can explicitly specify which method to inherit by using the super keyword. For example, you can inherit method1 from class A using the following code:

```
class D Implements B, C {
    public void method1() {
        super.A.method1();
    }
    // implementation
}
```

Providing an implementation of the common method in the sub-class: You can provide an implementation of the common method in the sub-class, which will override any implementation provided by the super-class or interfaces.

Using Interfaces: In case of interfaces, Java 8 has introduced a feature called 'default' methods which solves the diamond problem, it allows the interface to provide a default implementation of the method, so that the implementing class can either use the default implementation or provide its own implementation.

It's important to notice that the Diamond problem is specific to class inheritance and not interfaces, as interfaces do not have

implementation, they only have method signatures, this problem is solved by default methods, as the class can use any implementation of the method provided by the interfaces or can provide its own implementation.

How to write a wrapper class in java?

A wrapper class in Java is a class that wraps (or "encapsulates") a primitive data type, such as an int or a double, and provides additional functionality that is not available with the primitive data type. The wrapper classes in Java are located in the java.lang package and are named after the primitive data type they wrap, with the first letter capitalized.

Here is an example of how to write a wrapper class for the int data type:

```
public class IntWrapper {
    private int value;
    public IntWrapper(int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
    public void setValue(int value) {
        this.value = value;
    }
    public void increment() {
        value++;
    }
    public void decrement() {
        value--;
```

```
    }
    public String toString() {
        return Integer.toString(value);
    }
}
```

In the above example, the IntWrapper class wraps an int variable, called "value", and provides additional functionality such as increment() and decrement() methods that can be used to increment or decrement the value, and a toString() method that returns a string representation of the value.

Here's an example of how to use the IntWrapper class:

```
IntWrapper iw = new IntWrapper(5);
System.out.println(iw.getValue()); // 5
iw.increment();
System.out.println(iw.getValue()); // 6
System.out.println(iw); // 6
```

It's worth noting that Java provides wrapper classes for all of the primitive data types like int, char, double, boolean, etc. These classes are called as Integer, Character, Double, Boolean etc. These classes have additional functionalities that we can use like parseInt, parseDouble etc.

In summary, a wrapper class in Java is a class that wraps a primitive data type, such as an int or a double, and provides additional functionality that is not available with the primitive data type, you can write a wrapper class by creating a class with variable of primitive type, and adding additional functionality to the class and use it as per requirement.

How does HashMap work internally?

A HashMap in Java is a data structure that stores key-value pairs and is implemented using a hash table. It uses a hash function to map

keys to indices in an array, called a bucket. The hash function takes a key as input and returns an index, called a hash code, which is used to determine the location of the key-value pair in the bucket.

Internally, a HashMap consists of an array of Entry objects, where each Entry object stores a key-value pair and a reference to the next Entry object in the same bucket. When a key-value pair is added to the HashMap, the hash function is used to calculate the hash code of the key, which is used to determine the index of the bucket in the array. If there is no Entry object at that index, a new Entry object is created and added to the array. If there is already an Entry object at that index, the new key-value pair is added to the linked list of Entry objects at that index.

When a key-value pair is retrieved from the HashMap, the hash function is used to calculate the hash code of the key, which is used to determine the index of the bucket in the array. The linked list of Entry objects at that index is then searched for the key-value pair that has the same key as the one being retrieved.

The HashMap uses an array and linked list, which allows for constant-time O(1) performance for basic operations like put() and get() in an average case, and O(n) in the worst case when there's a high number of collisions. The load factor is a metric that determines when the HashMap should resize the array to maintain good performance. The default load factor is 0.75, which means that when the number of key-value pairs in the HashMap exceeds 75% of the size of the array, the array is resized to prevent excessive collisions.

Explain the internal implementation of HashMap.

HashMap is a popular data structure in Java known for its efficient key-value storage and retrieval. Understanding its internal implementation can shed light on its performance characteristics and trade-offs. Here's a breakdown:

Key Components:

Hash Table: HashMap uses an array of buckets called a hash table to store key-value pairs. The size of this array plays a crucial role in performance.
Nodes: Each key-value pair is stored in a "Node" object. This node contains the key, value, hashcode of the key, and a reference to the next node (for collision resolution).
Hash Function: This function converts the key into an integer index within the hash table range. A good hash function minimizes collisions, where multiple keys map to the same index.
Collision Resolution Strategy: When collisions occur, HashMap employs a strategy to store additional nodes at the same index. Common strategies include chaining (using linked lists) and open addressing (probing for empty slots).

Steps involved in operations:

put(key, value):

- Calculate the hashcode of the key.
- Use the hash function to find the corresponding index in the hash table.
- If no nodes exist at that index, add a new node with the key-value pair.
- If a collision occurs:

    1. Chaining: Add the new node to the linked list at the index.
    2. Open addressing: Probe for an empty slot nearby using a specific strategy.

get(key):

- Calculate the hashcode of the key.
- Use the hash function to find the corresponding index in the hash table.
- Traverse the linked list (if chaining) or probe for the matching key (open addressing).

- If the key is found, return the associated value. Otherwise, return null.

Important:

- HashMap performance relies heavily on a good hash function and efficient collision resolution.
- Chaining is generally preferred for smaller HashMaps due to its simplicity, while open addressing can be faster for larger ones with a well-chosen probing strategy.
- The load factor (number of entries divided by the table size) affects performance. Rehashing occurs when the load factor exceeds a threshold, dynamically resizing the table to maintain efficiency.

Which Classes are eligible to be used inside the resource block?

The try-with-resources statement is a feature introduced in Java 7 that allows you to automatically close resources that are declared in the try-block. This eliminates the need for a finally block to close the resources and ensures that the resources are closed even if an exception is thrown.

To use the try-with-resources statement, a class must implement the AutoCloseable interface. The AutoCloseable interface has a single method close() which is called when the try block exits.

The following classes are eligible to be used inside the resource block:
Any class that implements the AutoCloseable interface, such as: FileInputStream, FileOutputStream, BufferedReader, BufferedWriter, Scanner, PrintWriter, Connection, Statement.

How does HashSet works internally?

A HashSet is a collection class in Java that implements the Set interface. It stores a collection of unique elements, which means

that it does not allow duplicate elements. The HashSet class internally uses a HashMap to store the elements.

When an element is added to the HashSet, it is first passed through a hash function which calculates a unique hash code for the element. This hash code is used as the key in the underlying HashMap, and the element is used as the value.

When an element is retrieved from the HashSet, the hash code of the element is calculated and used to look up the corresponding value in the underlying HashMap. If the value is found, it is returned, otherwise, it means the element is not present in the HashSet.

The HashSet class uses the equals() method to compare the elements for equality. When an element is added to the HashSet, the HashSet calls the equals() method to check if the element is already present in the HashSet. If the element is already present, it is not added to the HashSet, otherwise, it is added to the HashSet.

The HashSet class also uses the hashCode() method to determine the position of an element in the underlying HashMap. The hashCode() method returns an integer value, which is used as the index of the element in the HashMap.

It's important to notice that the performance of HashSet depends on the implementation of the hashCode() method. If the hashCode() method is implemented poorly, it could lead to many elements being stored in the same bucket, which can cause performance issues.

It's also worth mentioning that the HashSet is not thread-safe, if multiple threads are accessing a HashSet at the same time, it's necessary to use synchronization or other thread-safe collections like ConcurrentHashSet.

What is the difference between HashMap and HashTable?

In Java, both HashMap and Hashtable are used to store key-value pairs, but they have some key differences:

Synchronization: Hashtable is synchronized, which means that all its methods are thread-safe. This means that only one thread can access a Hashtable at a time. On the other hand, HashMap is not synchronized, which means that multiple threads can access a HashMap at the same time. This can lead to better performance but also to data inconsistencies if not used properly.

Hashtable<String, String> table = new Hashtable<>();

HashMap<String, String> map = new HashMap<>();

Null Keys and Values: Hashtable does not allow null keys or values, it will throw NullPointerException if you try to insert a null key or value into a Hashtable. While HashMap allows one null key and multiple null values.

Iteration: Hashtable's enumerator is not fail-fast, while HashMap's iterator is fail-fast. Fail-fast means that if the HashMap is modified while an iteration over the HashMap is in progress in any way except through the iterator's own remove method, the iterator will throw a ConcurrentModificationException.

Performance: Hashtable is slower than HashMap because it is synchronized, while HashMap is faster because it is not synchronized.

Legacy: Hashtable is a legacy class and was introduced in the first version of Java, while HashMap was introduced in Java 2.

In general, HashMap is a better choice than Hashtable when you don't need thread-safety and need faster performance, and when you need to use null keys or values. While Hashtable is good when you need thread-safety, but the performance is slower than HashMap.

It's important to notice that since Java 5, ConcurrentHashMap is the recommended alternative for a thread-safe Hashtable and its performance is much better.

What is the difference between HashMap and Linked-HashMap?

HashMap and LinkedHashMap are two different implementations of the Map interface in Java.

HashMap is an implementation of the Map interface that uses a hash table to store the key-value pairs. It provides constant-time performance for most operations, such as put(), get(), and containsKey(). However, the order of the elements in a HashMap is not guaranteed, and may vary between different runs of the application.

LinkedHashMap is a subclass of HashMap that maintains a linked list of the entries in the map, in the order in which they were added. It provides a predictable iteration order, which is the order in which the entries were added to the map. LinkedHashMap is slightly slower than HashMap because it maintains a doubly-linked list to maintain the order of elements.

Here are some key differences between HashMap and LinkedHashMap:

Order of elements: HashMap does not guarantee the order of the elements, while LinkedHashMap maintains the order of elements in which they were inserted.

Performance: HashMap provides better performance than LinkedHashMap due to fewer overheads of maintaining order.

Iteration: LinkedHashMap provides predictable iteration order while iteration order in HashMap is not guaranteed.

Null keys and values: Both HashMap and LinkedHashMap allow null keys and values.

Memory consumption: LinkedHashMap uses more memory than HashMap due to additional overhead of maintaining the linked list.

In summary, if you require a predictable iteration order or need to maintain the order in which elements were added to the map, use

LinkedHashMap. Otherwise, use HashMap, which provides better performance and lower memory consumption.

What is the difference between HashMap and ConcurrentHashmap?

HashMap and ConcurrentHashmap are both used to store key-value pairs, but they have some key differences:

Synchronization: HashMap is not synchronized, which means that multiple threads can access a HashMap at the same time. This can lead to better performance but also to data inconsistencies if not used properly. On the other hand, ConcurrentHashmap is thread-safe, which means that all its methods are thread-safe. This means that only one thread can access a ConcurrentHashmap at a time, and it will not throw ConcurrentModificationException if one thread is iterating over it while another thread modifies it.

HashMap<String, String> map = new HashMap<>();
ConcurrentHashMap<String, String> cmap = new ConcurrentHashMap<>();

Performance: HashMap is faster than ConcurrentHashMap because it is not synchronized, while ConcurrentHashMap is slower because it is thread-safe.

Concurrent Operations: ConcurrentHashMap is designed to support high-concurrency and it can be used in multi-threaded environments where multiple threads can read and write to it simultaneously. HashMap is not designed to support high-concurrency, and it's not suitable to be used in multi-threaded environments.

Iteration: HashMap's iterator is fail-fast, while ConcurrentHashMap's iterator is weakly consistent. Fail-fast means that if the HashMap is modified while an iteration over the HashMap is in progress in any way except through the iterator's own remove method, the iterator will throw a ConcurrentModificationException. Weakly consistent means that the iterator may reflect some, but not necessarily all, of the changes made to the map since the iterator was created.

Legacy: HashMap is a legacy class and was introduced in the first version of Java, while ConcurrentHashMap was introduced in Java 5.

In general, ConcurrentHashMap is a better choice than HashMap when you need thread-safety and when you need to use the map in a multi-threaded environment. While HashMap is good when you don't need thread-safety and need faster performance.

Can we insert the null key in the HashMap and HashTable?

In Java, both HashMap and Hashtable allow you to insert a null key. However, it is not recommended to use null keys in a HashMap or Hashtable, as it can lead to unexpected behavior and errors.

When a null key is used in a HashMap, the hashCode() method of the key returns 0, which is used to determine the index of the bucket in the array. All the keys that have a hashCode() of 0 will be stored in the same bucket, and this can lead to collisions and poor performance.

When a null key is used in a Hashtable, the hashCode() method of the key returns 0, which is used to determine the index of the bucket in the array. All the keys that have a hashCode() of 0 will be stored in the same bucket, and this can lead to collisions and poor performance.

It's important to notice that even though it's allowed to insert null keys in a HashMap and Hashtable, it's not a good practice, since it can lead to unexpected behavior and errors. Instead, it's recommended to use a sentinel value or a special object as a key to represent a null value, and handle it in a specific way.

How to create an immutable map in Java?

In Java, there are several ways to create an immutable map:

Using the Collections.unmodifiableMap() method: This method returns an unmodifiable view of the specified map. Any attempt to modify the map will throw an UnsupportedOperationException.

```
Map<String, String> map = new HashMap<>();
map.put("key1", "value1");
map = Collections.unmodifiableMap(map);
```

Using the Map.of() method: Java 9 introduced the Map.of() method, which creates an immutable map of the specified key-value pairs.

```
Map<String, String> map = Map.of("key1", "value1", "key2", "value2");
```

Using the Map.ofEntries() method: Java 9 also introduced the Map.ofEntries() method, which creates an immutable map from the specified entries.

```
Map<String, String> map = Map.ofEntries(
    Map.entry("key1", "value1"),
    Map.entry("key2", "value2")
);
```

Using the ImmutableMap.of() method from Guava library: Google's Guava library provides an ImmutableMap class that has a of() method which creates an immutable map of the specified key-value pairs.

```
Map<String, String> map = ImmutableMap.of("key1", "value1", "key2", "value2");
```

Using the ImmutableMap.Builder class from Guava library: You can also use the ImmutableMap.Builder class to build an immutable map by adding key-value pairs to it.

```
Map<String, String> map = new ImmutableMap.Builder<String, String>()
    .put("key1", "value1")
    .put("key2", "value2")
    .build();
```

All of the above methods will create an immutable map, which means that any attempt to modify the map will result in an

exception being thrown.

What are the in-built immutable classes in java?

Java provides several built-in immutable classes that are commonly used in programming:

String: The String class is an immutable class that represents a sequence of characters. Once a String object is created, its value cannot be changed.

Integer, Long, Short, Byte, Character, Double, Float: These are the classes for the primitive data types int, long, short, byte, char, double and float respectively. They are also immutable classes and once object of these classes is created, it's value cannot be changed.

BigInteger and BigDecimal: These are immutable classes that are used to represent large integers and decimal numbers, respectively.

Enum: Enum types are classes that represent enumerated values. They are also immutable classes and once an object of Enum is created, it's value cannot be changed.

LocalDate, LocalTime, LocalDateTime, Instant: These are classes from the java.time package, they represent date, time, datetime and timestamp respectively. They are also immutable classes and once an object of these classes is created, it's value cannot be changed.

Optional: This is a class from the java.util package, it is used to represent an optional value. The Optional class is immutable, and once an Optional object is created, its value cannot be changed.

It's important to notice that even though these classes are immutable, their state can be changed when they are used as fields in a class, to prevent this, it's necessary to make the fields final and private.

What is index in java? Advantages and disadvantagesin Database?

In Java, an index is a data structure that allows efficient lookups of elements in a collection or table. An index is used to speed up the search process by allowing the program to quickly locate a specific element in the collection or table, rather than having to scan through the entire collection or table.

An index can be used in many different types of data structures, such as arrays, lists, and tables. In a database, an index is used to improve the performance of queries by allowing the database to quickly locate the rows that match a specific condition.

There are several types of indexes that can be used in a database, such as:

Primary key index: This is a unique index that is used to enforce the integrity of the primary key constraint.

Unique index: This index is used to enforce the integrity of unique constraints.

Clustered index: This index determines the physical order of data in a table. Each table can have only one clustered index.

Non-clustered index: This index contains a copy of the indexed column(s) along with a pointer to the actual data row. Each table can have multiple non-clustered indexes.

Full-text index: This index is used to support full-text searches.

The advantages of using index in a database are:

- It improves the performance of search queries by allowing the database to quickly locate the rows that match a specific condition.
- It can help to enforce the integrity of the primary key and unique constraints.

- It can help to improve the performance of join operations by allowing the database to quickly locate the related rows.

The disadvantages of using index in a database are:

- It can increase the size of the database and reduce the overall performance if the index is not used properly.
- It can increase the time required to insert, update or delete data in a table as the indexes need to be updated as well.
- In some cases, the index can be less efficient than a table scan, especially if the table is small or the percentage of rows that match the condition is low.

It's important to notice that when creating an index, it's necessary to consider the trade-off between the benefits of improved query performance and the costs of increased storage and update overhead.

What is the difference between CompletableFuture and a Callable and Runnable Future?

In Java, there are several ways to perform asynchronous operations:

Callable: A Callable is similar to a Runnable, but it can return a value and throw a checked exception. It can be submitted to an ExecutorService to be executed and return a Future object. The Future object can be used to check if the computation is complete, wait for its completion, and retrieve the result of the computation.

```
Callable<Integer> myCallable = () -> {
    // perform computation
    return result;
};
```

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

Runnable: A Runnable is a task that can be executed by an Executor. It does not return a value and cannot throw a checked exception. It can be submitted to an ExecutorService to be executed and return a Future object. The Future object can be used to check if the computation is complete and wait for its completion.

```
Runnable myRunnable = () -> {
    // perform computation
};

Future<?> future = executorService.submit(myRunnable);
```

A CompletableFuture can be used to compose multiple asynchronous operations together, handle errors, and provide callbacks. It can be used to perform actions when the computation is complete, such as applying a function to the result, chaining multiple operations together, or handling errors. It also allows to create a future that is already completed with a value or an exception.

How to parse XML file with JSON in java?

Parsing an XML file and converting it to JSON in Java can be done using a library such as Jackson or Gson. Here's an example of how to do it with Jackson:

First, you will need to add the Jackson library to your project. You can do this by adding the following dependency to your build file:

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
    <version>2.11.3</version>
</dependency>
```

Next, you will need to read the XML file and convert it to an object using the XmlMapper class.

```
XmlMapper xmlMapper = new XmlMapper();
```

```
JsonNode jsonNode = xmlMapper.readTree(new
File("path/to/xml/file.xml"));
```

Finally, you can convert the JsonNode object to a JSON string using the writeValueAsString() method.

```
String jsonString = xmlMapper.writeValueAsString(jsonNode);
```

You can also use Gson library for this purpose, it has the feature of handling both XML and JSON.

It's important to notice that this process is not a direct conversion from XML to JSON, it's rather parsing the XML into JsonNode object and then converting that object to a JSON string, this is why the method used is writeValueAsString()

It's also worth mentioning that some of the information from the XML file may not be preserved during this process, such as comments, processing instructions, and text nodes that do not have a corresponding element.

How to parse JSON to a HashMap?

There are several libraries available in Java to parse JSON to a HashMap, some of the most popular libraries are:

Jackson: Jackson is a popular JSON processing library for Java. It provides a simple and easy-to-use API for parsing JSON to a HashMap. Here is an example of how to use Jackson to parse JSON to a HashMap:

```
String jsonString = "{\"name\":\"John\",\"age\":30}";
ObjectMapper mapper = new ObjectMapper();
HashMap<String, Object> map = mapper.readValue(jsonString, new
TypeReference<HashMap<String, Object>>(){});
```

Gson: Gson is another popular JSON processing library for Java. It provides a simple and easy-to-use API for parsing JSON to a HashMap. Here is an example of how to use Gson to parse JSON to a HashMap:

```
String jsonString = "{\"name\":\"John\",\"age\":30}";
Gson gson = new Gson();
HashMap<String, Object> map = gson.fromJson(jsonString, new
TypeToken<HashMap<String, Object>>(){}.getType());
```

org.json: org.json is a built-in Java library for parsing and generating JSON. It provides a simple and easy-to-use API for parsing JSON to a HashMap.

What are fail-safe and fail-fast iterators?

Fail-safe iterators do not throw ConcurrentModificationException if the underlying collection is modified while iterating over it. They create a snapshot of the collection at the time of iteration, so changes made while iterating do not affect the iteration.

Fail-fast iterators, on the other hand, throw a ConcurrentModificationException if the underlying collection is modified while iterating over it. They check for modifications at every iteration, so if a change is made, the exception is thrown immediately.

What is the object class in Java? what are the methods in it?

Java JDK (Java Development Kit) provides many built-in classes with their respective methods. Some commonly used object class methods in Java JDK are:

equals(Object obj): This method compares the current object with the specified object and returns true if they are equal, else it returns false.

hashCode(): This method returns the hash code of the object. The hash code is a unique integer value that is used by hash-based data structures such as HashMap, HashSet, etc.

toString(): This method returns a string representation of the object. It is generally used for debugging and logging purposes.

clone(): This method creates a new object that is a copy of the current object. The clone() method is used to create a copy of an object without modifying the original object.

getClass(): This method returns the class of the current object. It is used to get the runtime class of the object.

wait(): This method causes the current thread to wait until it is notified. It is generally used in multi-threaded applications.

notify(): This method wakes up a single thread that is waiting on the object's monitor.

finalize(): This method is called by the garbage collector when the object is no longer referenced.

These are some of the commonly used object class methods in Java JDK. However, there are many other methods available in Java JDK for different classes, and you can explore them in the Java documentation.

Why and how to use clone method in Java?

The clone() method in Java is used to create a new object that is a copy of the original object. This method creates a new object with the same state as the original object, i.e., it copies all the field values of the original object to the new object.

The clone() method is defined in the Object class and can be used to create a copy of any object, provided that the class implements the Cloneable interface. The Cloneable interface is a marker interface, which means it does not contain any methods, but it indicates that the class is cloneable.

Here are some reasons why you may want to use the clone() method:

To create a backup of an object: You may want to create a backup of an object so that you can restore its state later if needed. The clone() method can be used to create a backup of an object.

To create a copy of an object: You may want to create a copy of an object to modify its state without affecting the original object. The clone() method can be used to create a copy of an object.

To create an object with default values: You may want to create an object with default values, and the clone() method can be used to

create an object with the same default values as the original object.

Here's how you can use the clone() method:

Make sure that the class implements the Cloneable interface.

Override the clone() method in the class.

Inside the clone() method, call the super.clone() method to create a copy of the object.

Cast the returned object to the class type.

Return the cloned object.

Here's an example code snippet that demonstrates how to use the clone() method:

```
class MyClass implements Cloneable {
    private int value;
    public MyClass(int value) {
        this.value = value;
    }
    public void setValue(int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

public class Main {
```

```
    public static void main(String[] args) throws
CloneNotSupportedException {
        MyClass obj1 = new MyClass(10);
        MyClass obj2 = (MyClass) obj1.clone();
        System.out.println("obj1 value: " + obj1.getValue());
        System.out.println("obj2 value: " + obj2.getValue());
        obj2.setValue(20);
        System.out.println("obj1 value after obj2 modification: " +
obj1.getValue());
        System.out.println("obj2 value after modification: " +
obj2.getValue());
    }
}
```

In the above example, we have created a class MyClass that implements the Cloneable interface and overrides the clone() method to create a copy of the object. We have then created two objects of MyClass and cloned one of them using the clone() method. We have then modified the value of the cloned object and checked if it affects the original object.

What parsing libraries you have used so far?

There are several parsing libraries available in Java that can be used to parse various types of data such as XML, JSON, CSV, etc. Some of the popular parsing libraries in Java are:

Jackson: Jackson is a high-performance JSON parser for Java. It can parse JSON data from a file or a stream and map it to Java objects. It provides annotations to customize the mapping process and supports bidirectional mapping between JSON and Java objects.

Gson: Gson is another popular JSON parser for Java. It can parse JSON data into Java objects and vice versa. It also provides support for custom serialization and deserialization of Java objects.

JAXB: JAXB stands for Java Architecture for XML Binding. It is a framework that allows Java developers to map XML schemas to Java classes. It provides tools to generate Java classes from XML schemas and to marshal/unmarshal XML data to/from Java objects.

Jsoup: Jsoup is a Java library for working with HTML documents. It provides a simple API to parse HTML documents and extract information from them. It also supports HTML cleaning and manipulation.OpenCSV: OpenCSV is a CSV parsing library for Java. It can read and write CSV files and provides support for custom mapping between CSV fields and Java objects.

Apache Tika: Apache Tika is a content detection and analysis framework for Java. It can detect the content type of a file and parse its contents into a structured format. It supports parsing of various file formats such as PDF, HTML, XML, Microsoft Office documents, etc.

These are some of the popular parsing libraries in Java that can be used to parse various types of data. The choice of a parsing library depends on the specific requirements and the data format being parsed.

What is the difference between comparable and comparator?

In Java, both Comparable and Comparator are interfaces that are used to compare objects and establish their order in collections like lists or arrays. However, they serve different purposes and are used in different contexts:

1. **Comparable**:

The Comparable interface is used to define the natural ordering of objects. When a class implements the Comparable interface, it indicates that instances of that class have a default way to be sorted. The natural ordering is defined by implementing the compareTo() method, which returns a negative integer if the current object is "less than" the other object, zero if they are "equal," and a positive integer if the current object is "greater than" the other object.

For example, consider a Person class that implements Comparable to sort instances based on their age:

```
public class Person implements Comparable<Person> {
    private String name;
    private int age;
    // Constructor, getters, and setters
    @Override
    public int compareTo(Person otherPerson) {
        return Integer.compare(this.age, otherPerson.age);
    }
}
```

By implementing Comparable, instances of the Person class can be sorted using methods like Collections.sort() or Arrays.sort() without needing to provide a separate comparator.

2. **Comparator**:

The Comparator interface, on the other hand, is used to provide custom comparison logic for objects that may not have a natural ordering or when you want to sort objects based on different criteria than their inherent properties. A Comparator is a separate class that implements the comparison logic. This approach allows you to have multiple ways of sorting objects without modifying their original class.

For instance, let's say you want to sort instances of the Person class not only by age but also by name. You can create a separate NameComparator class that implements the Comparator<Person> interface:

```
import java.util.Comparator;
public class NameComparator implements Comparator<Person> {
    @Override
    public int compare(Person person1, Person person2) {
```

```
        return person1.getName().compareTo(person2.getName());
    }
}
```

Then, you can sort instances of Person using this NameComparator:

```
List<Person> people = new ArrayList<>();
// Add people to the list
Collections.sort(people, new NameComparator());
```

In summary, the key differences between Comparable and Comparator in Java are:

- Comparable is used to define the natural ordering within a class itself.

- Comparator is used to define external comparison logic for classes that may not have a natural ordering or when you want to sort based on different criteria. It allows you to create multiple sorting strategies without modifying the original class.

Explain the classpath exception?

In Java, the Classpath is a parameter that specifies the locations where the Java Virtual Machine (JVM) should look for class files that are needed by a running application. The Classpath can be set as an environment variable or passed as a command-line argument.

A Classpath exception occurs when the JVM is unable to find the required class file in the specified Classpath locations. This can happen for several reasons, such as:

Incorrect Classpath: The Classpath specified may not be correct or may not include the required directory or JAR file.

Missing class files: The required class file may be missing or may have been moved or deleted from the Classpath.

Incompatible versions: The class file may be compiled with a different version of Java than the one being used to run the application, resulting in a version incompatibility error.

Security restrictions: The JVM may be running under a security manager that restricts access to the specified Classpath locations.

Permissions issues: The user running the application may not have sufficient permissions to access the required Classpath locations.

To resolve a Classpath exception, the following steps can be taken:

Verify the Classpath: Verify that the Classpath specified is correct and includes the required directory or JAR file.

Check for missing class files: Check if the required class file is missing or has been moved or deleted.

Check for version compatibility: Ensure that the required class file is compiled with the same version of Java as the one being used to run the application.

Check for security restrictions: Check if the JVM is running under a security manager that restricts access to the specified Classpath locations.

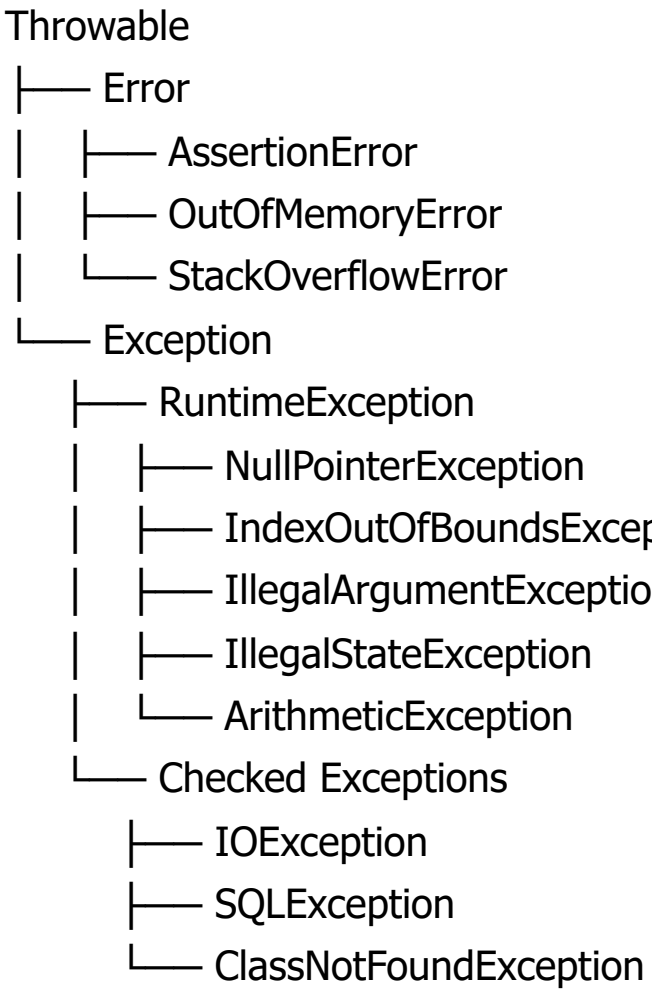
Check permissions: Ensure that the user running the application has sufficient permissions to access the required Classpath locations.

In summary, a Classpath exception occurs when the JVM is unable to find the required class file in the specified Classpath locations. This can be resolved by verifying the Classpath, checking for missing class files, version compatibility, security restrictions, and permissions issues.

What is the hierarchy of exceptions?

In Java, exceptions are organized in a hierarchical manner, with the java.lang.Throwable class at the root of the hierarchy. The Throwable class has two direct subclasses: Error and Exception. The Error class represents unrecoverable errors that usually occur at the system level, such as OutOfMemoryError. The Exception class represents recoverable errors and has several subclasses, such as RuntimeException and IOException.

Here's a hierarchy of some of the most common exception classes in Java:



As shown in the hierarchy above, RuntimeException and its subclasses represent unchecked exceptions that do not need to be declared in a method's throws clause. All other exceptions are checked exceptions, which must be declared in a method's throws clause or handled within the method using a try-catch block.

It is important to note that Java allows the creation of custom exception classes by extending the Exception class or one of its subclasses. By doing so, developers can create their own custom exceptions that represent specific errors or situations in their application.

Explain Throw, Throws, and Throwable keywords in java.

In Java, the keywords "throw", "throws", and "Throwable" are used to handle and manage exceptions in a program:

The "throw" keyword is used to explicitly throw an exception. When a method encounters an exceptional situation and it cannot handle it, it can throw an exception to the calling method. This allows the calling method to handle the exception in an appropriate way. The general form of the throw statement is "throw exception_object;", where exception_object is an instance of a class that extends the Throwable class.

```
public void method() throws Exception {
    if (condition) {
        throw new Exception("Exception occurred");
    }
}
```

The "throws" keyword is used in the method signature to indicate that a method can throw one or more exceptions. The general form of the throws clause is "throws exception_class, exception_class,...", where exception_class is the type of exception that the method can throw. It's important to notice that when a method throws an exception, it's not handling it, it's just indicating that it can happen.

```
public void method() throws Exception {
    // code
}
```

The "Throwable" class is the parent class of all exceptions and errors in Java. It is the superclass of all classes that can be thrown by the Java Virtual Machine (JVM) or the user. The Throwable class defines several methods that can be used to print the stack trace of an exception, get the message of an exception, and get the cause of an exception.
In summary, "throw" keyword is used to throw an exception, "throws" keyword is used to indicate that a method can throw an exception and "Throwable" is the base class for all the exception and errors in Java.

What is a String in java?

In Java, String is a class that represents a sequence of characters. It is one of the most commonly used classes in Java and is included in the java.lang package, which means that it is automatically imported into every Java program.

A String object can be created using a string literal or by using the new keyword and a constructor. For example:

String str1 = "Hello, world!";

String str2 = new String("Hello, world!");

Both of these statements create a String object that contains the sequence of characters "Hello, world!".

Once a String object is created, its value cannot be changed. In other words, String is an immutable class. Therefore, any operation on a String object creates a new String object.

The String class provides many useful methods for manipulating strings, such as charAt(), indexOf(), substring(), toUpperCase(), toLowerCase(), trim(), length(), and many others.

String objects are also widely used in Java as parameters to method calls, in concatenation operations, and as return values from methods.

In summary, String is a class in Java that represents a sequence of characters and is widely used for string manipulation and processing.

Why String is immutable?

Here are the key reasons why String is immutable in Java and other languages:

1. Security:

Strings are often used to store sensitive information like passwords, URLs, and database connection strings. Immutability makes them tamper-proof, preventing accidental or malicious modification of sensitive data. This enhances security and reduces the risk of vulnerabilities.

2. Caching and Performance:

Java caches String literals in a String pool for efficient reuse. If Strings were mutable, changes to one String could affect other Strings using the same literal, leading to unpredictable behavior. Immutability ensures that String values remain consistent and predictable, even when shared across multiple references.

3. Synchronization and Thread Safety:

Immutable objects are inherently thread-safe, as their state cannot be modified by multiple threads concurrently. This eliminates the need for synchronization mechanisms (like locks) when working with Strings in multithreaded environments, simplifying code and improving performance.

4. Class Loading:

Java uses String objects for class names and resource paths. Immutability ensures that these references remain stable and reliable throughout the application's lifecycle, preventing issues with class loading and resource access.

5. Use as Keys in Hash-Based Data Structures:

Strings are commonly used as keys in hash-based data structures like HashMap and HashSet. Immutability guarantees that the hashcode of a String remains consistent throughout its lifetime, making these data structures work correctly and efficiently.

6. Substring Operations:

While String objects themselves are immutable, Java provides methods to create new String objects that are modified versions of existing ones. For example, the substring() method returns a new String that is a portion of the original String, without altering the original object.

7. StringBuilder and StringBuffer:

For scenarios where you need to modify String content frequently, Java offers mutable alternatives: StringBuilder (for non-thread-safe operations) and StringBuffer (for thread-safe operations). These

classes are designed for efficient String manipulation and modification.

Where is a new string is stored?

The storage location of a new String depends on how it's created:

1. String Literal Pool:

When you create a String using a literal (e.g., String str = "Hello";), Java checks the String pool first.

If an identical String already exists in the pool, it reuses that object for efficiency.

This means multiple variables can refer to the same String object in memory, saving space.

2. Heap Memory:

If the String literal doesn't exist in the pool, a new String object is created and stored in the heap memory.

This happens in two main cases:

When you create a String using the new keyword: String str = new String("Hello");

When you modify an existing String, resulting in a new String object: String str2 = str.concat(" World");

3. String Interning:

You can explicitly place a String in the pool using the intern() method: String internedStr = "Hello".intern();

This ensures that all String variables with the same content refer to the same object in the pool, even if they were created with new.

Where do strings get stored and where does the reference get stored?

Strings in Java are stored in the heap memory. The heap memory is a region of memory that is used to store objects. When you create a string object, the Java Virtual Machine (JVM) allocates space for the object in the heap memory.

The reference to the string object is stored on the stack. The stack is a region of memory that is used to store local variables and method parameters. When you assign a string object to a variable, the JVM stores the reference to the object in the stack memory.

Here is an example of how strings are stored in memory:

String myString = "Hello, world!";

In this example, the string object "Hello, world!" is stored in the heap memory. The reference to the string object is stored in the stack memory, in the variable myString.

The JVM manages the heap memory and the stack memory automatically. This means that you do not need to worry about allocating or freeing memory for string objects.

If you don’t want to use the String class then what is the alternative?

If you don't want to use the String class in Java, you can use the following alternatives:

StringBuilder: The StringBuilder class is a mutable string class that can be used to create and modify strings. StringBuilder objects are thread-safe, meaning that they can be safely used in multithreaded applications.

StringBuffer: The StringBuffer class is similar to the StringBuilder class, but it is not thread-safe. StringBuffer objects are slower than StringBuilder objects, but they are safe to use in multithreaded applications.

Character arrays: Character arrays can be used to represent strings. However, character arrays are not as efficient as string objects, and they are more difficult to use.

Can we create a customized immutable String class, how to achieve it?

In Java, the String class is already an immutable class, so it is not necessary to create a custom String class for immutability. However, it is possible to create a similar custom class that is immutable.

To create a custom immutable string class, you can follow these steps:

Define a private final field of type String to store the string value.

Create a constructor that accepts a String parameter and initializes the private field.

Do not provide any setter methods that can modify the value of the private field.

Override the toString() method to return the value of the private field.

If necessary, override the equals() and hashCode() methods to ensure that objects of this class can be properly compared and used as keys in hash-based collections.

Here's an example implementation of a custom immutable string class:

```
public final class ImmutableString {
    private final String value;
    public ImmutableString(String value) {
        this.value = value;
    }
    public String toString() {
        return value;
    }
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }
```

```
        ImmutableString other = (ImmutableString) obj;
        return Objects.equals(value, other.value);
    }
    public int hashCode() {
        return Objects.hash(value);
    }
}
```

In this implementation, the value field is declared as final, which makes it immutable. The constructor initializes the value field using the provided String parameter. The toString(), equals(), and hashCode() methods are overridden to ensure proper functionality.

By following this approach, you can create a custom class that behaves similarly to the String class but is immutable.

What is the difference between String, StringBuffer and StringBuilder?

In Java, String, StringBuffer, and StringBuilder are classes that are used to manipulate strings, but they differ in their characteristics and usage.

String is an immutable class, which means that once a String object is created, its value cannot be changed. Therefore, any operation on a String object creates a new object. For example, if two String objects are concatenated using the + operator, a new String object is created. This can be inefficient if many string manipulations are needed, as it creates a lot of temporary objects.

StringBuffer and StringBuilder are mutable classes that can be used to perform string manipulation operations efficiently. StringBuffer was introduced in Java 1.0, while StringBuilder was added in Java 1.5. Both classes provide methods for appending, inserting, deleting, and replacing characters in a string.

The main difference between StringBuffer and StringBuilder is that StringBuffer is thread-safe, which means that multiple threads can safely access and modify the same StringBuffer object at the same

time without any issues. On the other hand, StringBuilder is not thread-safe, and therefore should be used in single-threaded environments.

In summary:

String is immutable, so any operation on a String object creates a new object.

StringBuffer and StringBuilder are mutable and provide methods for efficient string manipulation.

StringBuffer is thread-safe, while StringBuilder is not.

Therefore, String is best used for situations where the string value will not change frequently, while StringBuffer or StringBuilder should be used for situations where frequent string manipulations are required. If the code is running in a multi-threaded environment, StringBuffer should be used to avoid concurrency issues. If the code is running in a single-threaded environment, StringBuilder can be used for even better performance.

Is StringBuffer synchronized? Where is synchronized used in StringBuffer?

Yes, StringBuffer is a synchronized class in Java, which means that its methods are thread-safe and can be accessed by multiple threads concurrently without causing data inconsistency or other issues.

In the StringBuffer class, the synchronized keyword is used to make the methods thread-safe. Specifically, the synchronized keyword is used to make the following methods synchronized:

append()

insert()

delete()

deleteCharAt()

replace()

substring()

charAt()

setCharAt()

length()

capacity()

ensureCapacity()

trimToSize()

toString()

By making these methods synchronized, multiple threads can access them safely without interfering with each other.

It's worth noting that in Java 5, a new class called StringBuilder was introduced, which is similar to StringBuffer but is not synchronized. If you do not need thread-safety, you can use StringBuilder instead of StringBuffer, as it can be faster in some cases. However, if you need to access a mutable string from multiple threads concurrently, you should use StringBuffer to ensure thread-safety.

Why are Java substrings bad?

The substring() method in Java is a useful method that is used to extract a portion of a string. However, it is important to use it carefully to avoid potential errors and performance issues.

One common issue with the substring() method is that it creates a new string object every time it is called. This can lead to performance problems if it is called repeatedly in a loop or in performance-critical code. To avoid this, you can use the StringBuilder or StringBuffer classes to build a string gradually instead of using substring().

Another potential issue with substring() is that it can throw an IndexOutOfBoundsException if the starting index or ending index is out of bounds. To avoid this, you should always check the length of the string before calling substring() and ensure that the indices are within the valid range.

It's also important to note that the substring() method returns a new string object that shares the same character array as the original string object. This means that if you modify the substring, it will also

modify the original string. To avoid this, you can create a new string object from the substring.

In summary, the substring() method is a useful method for extracting a portion of a string, but it should be used carefully to avoid potential errors and performance issues.

What is a Runtime exception and how they are they implemented?

In Java, runtime exceptions are a type of exception that can occur during the execution of a program. They are not checked at compile-time, and they do not need to be declared in the method signature using a throws clause. Instead, they are thrown implicitly by the JVM when an error condition occurs at runtime.

Runtime exceptions are implemented as subclasses of the RuntimeException class, which itself is a subclass of the Exception class. Some examples of runtime exception classes in Java include NullPointerException, ArrayIndexOutOfBoundsException, and ArithmeticException.

Runtime exceptions can be caused by a variety of factors, such as invalid input, incorrect usage of APIs, or unexpected conditions such as a divide-by-zero error. When a runtime exception occurs, the JVM will throw an instance of the corresponding exception class, which can then be caught and handled by the program if necessary.

To catch runtime exceptions in a Java program, you can use a try-catch block. For example:

```
try {
    // code that may throw a runtime exception
} catch (NullPointerException e) {
    // handle the null pointer exception
} catch (ArrayIndexOutOfBoundsException e) {
    // handle the array index out of bounds exception
} catch (Exception e) {
```

```
    // handle any other exception
}
```

It's important to note that while runtime exceptions do not need to be declared in the method signature, they should still be handled properly in the program to avoid unexpected termination or other issues.

Draw the collection hierarchy?

In Java, the Collection Framework is a set of classes and interfaces that provides a unified architecture for storing and manipulating groups of objects. The Collection Framework includes several key interfaces and classes, arranged in a hierarchical manner:

Collection: This is the top-level interface in the Collection Framework. It represents a group of objects and provides methods for adding, removing, and querying the elements of the collection. The Collection interface has two main sub-interfaces:

List: This interface extends the Collection interface and represents an ordered collection of elements. List allows duplicates and provides methods for accessing elements by their index.

Set: This interface also extends the Collection interface, but it represents a collection of unique elements. Set does not allow duplicates and provides methods for testing whether a particular element is present in the set.

Queue: This interface extends the Collection interface and represents a collection of elements that can be accessed in a specific order. Queue provides methods for adding, removing, and accessing elements from the collection based on the order in which they were added.

Map: This interface represents a collection of key-value pairs. Map allows you to store and retrieve elements based on their associated key. Map does not extend the Collection interface, but it is still considered part of the Collection Framework.

There are also several classes in the Collection Framework that provide implementations of the various interfaces, such as ArrayList

and LinkedList for the List interface, HashSet and TreeSet for the Set interface, and HashMap and TreeMap for the Map interface. These classes provide different performance characteristics and are designed for different use cases.

Overall, the Collection Framework in Java provides a powerful and flexible set of tools for storing and manipulating groups of objects, and its hierarchical structure allows for easy organization and use of the various interfaces and classes.

What is the difference between these syntaxes?

List list = new Arraylist<>();

Arraylist alist = new Arraylist<>();

Both syntaxes create an instance of the ArrayList class in Java, but they differ in the type of reference variable that is used to store the reference to the object.

List list = new ArrayList<>();

This syntax creates a new ArrayList object and assigns it to a reference variable of type List. This is an example of programming to an interface, which is a best practice in Java. By using List instead of ArrayList, the code becomes more flexible and easier to maintain, as the implementation class can be changed without affecting the rest of the code.

ArrayList alist = new ArrayList<>();

This syntax creates a new ArrayList object and assigns it to a reference variable of type ArrayList. This is an example of programming to an implementation, which is generally less flexible than programming to an interface. While it may be appropriate in some cases to use a specific implementation class, it can make the code more difficult to maintain if changes need to be made in the future.

In general, it's recommended to use the first syntax (List list = new ArrayList<>();) to create instances of collection classes in Java, unless there is a specific reason to use the implementation class

directly (ArrayList alist = new ArrayList<>();). This allows for greater flexibility and maintainability of the code.

What collection will we use for manipulation (ArrayList or LinkedList)?

ArrayList and LinkedList are both implementations of the List interface in Java, but they have some important differences in their implementation and performance characteristics:

Data structure: ArrayList is implemented as a resizable array, while LinkedList is implemented as a doubly-linked list. This means that ArrayList can access elements by index in constant time (O(1)), while LinkedList has to traverse the list to access an element, which takes linear time (O(n)).

Insertion and deletion: Insertion and deletion operations are faster in LinkedList because they only involve modifying the pointers of adjacent elements, while in ArrayList, elements have to be shifted to maintain the order of the list.

Random access: Random access is faster in ArrayList because it can access elements by index in constant time (O(1)), while LinkedList has to traverse the list to access an element, which takes linear time (O(n)).

Memory usage: ArrayList uses less memory than LinkedList because it only needs to store the elements and a backing array, while LinkedList needs to store the elements and pointers to the previous and next elements.

In general, ArrayList is a better choice if you need to access elements frequently by index and if you don't need to insert or delete elements frequently. On the other hand, LinkedList is a better choice if you need to insert or delete elements frequently and if you don't need to access elements frequently by index.

What is the use of an iterator in Java?

In Java, an Iterator is an interface that provides a way to iterate over a collection of objects, such as a List, Set, or Map. It allows you

to traverse the elements in a collection one by one and perform various operations on them.

The Iterator interface defines three methods:

hasNext(): Returns true if there are more elements in the collection, and false otherwise.

next(): Returns the next element in the collection.

remove(): Removes the last element returned by next() from the collection.

By using an Iterator, you can iterate over the elements in a collection without having to know the specific implementation of the collection. This makes your code more flexible and reusable.

Here's an example of using an Iterator to iterate over the elements in an ArrayList:

```
ArrayList<String> list = new ArrayList<>();
list.add("apple");
list.add("banana");
list.add("cherry");
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    String element = iterator.next();
    System.out.println(element);
}
```

This code creates an ArrayList of strings, adds some elements to it, and then creates an Iterator for the list. The while loop uses the hasNext() and next() methods of the Iterator to iterate over the elements in the list and print them to the console.

What is the default capacity of HashMap?

The default capacity of a Java HashMap is 16. This means that when you create a new HashMap object without specifying a capacity, it will be initialized with a capacity of 16.

However, it's important to note that the capacity of a HashMap can be increased or decreased dynamically based on the number of key-value pairs in the map and the load factor, which is another parameter that determines when the HashMap should resize itself.

If the number of key-value pairs in the HashMap grows beyond a certain threshold based on the load factor, the capacity of the HashMap will be automatically increased to maintain efficient performance. Conversely, if the number of key-value pairs in the map decreases, the capacity may also be reduced to save memory.

How does HashMap behaves when it reaches its maximum capacity?

When a HashMap reaches its maximum capacity, it will automatically resize itself to accommodate more key-value pairs. This process is called rehashing.

During rehashing, a new internal array is created with twice the capacity of the original array. Each key-value pair from the old array is then hashed again and added to the new array at a new index, based on the new array size and the hash code of the key.

Rehashing is necessary to maintain the performance of the HashMap. As the number of key-value pairs in the map grows, the likelihood of hash collisions increases, which can slow down the performance of the map. By resizing the map when it reaches its maximum capacity, the likelihood of collisions is reduced, and the map can continue to perform efficiently.

However, rehashing can be an expensive operation, as it involves iterating through all the key-value pairs in the map and recalculating their hash codes. To minimize the number of rehashing operations, it's important to choose an appropriate initial capacity and load factor for the HashMap based on the expected number of key-value pairs that it will hold.

How to create a custom object as key in HashMap?

To use a custom object as a key inside a HashMap, you need to ensure that the custom object implements the hashCode() and

equals() methods.

The hashCode() method is used by the HashMap to compute a hash value for the key, which is used to determine the index in the internal array where the key-value pair will be stored. The equals() method is used by the HashMap to compare keys for equality, which is necessary to resolve collisions that occur when different keys have the same hash code.

Here is an example of how to create a custom object as a key inside a HashMap:

```
public class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    // Implement the hashCode() method based on the object's fields
    @Override
    public int hashCode() {
        int result = 17;
        result = 31 * result + name.hashCode();
        result = 31 * result + age;
        return result;
    }
    // Implement the equals() method to compare objects based on the object's fields
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
```

```
        if (!(obj instanceof Person)) return false;
        Person other = (Person) obj;
        return name.equals(other.name) && age == other.age;
    }
}
// Create a HashMap with Person objects as keys
Map<Person, String> people = new HashMap<>();
Person john = new Person("John", 30);
Person sarah = new Person("Sarah", 25);
people.put(john, "555-1234");
people.put(sarah, "555-5678");
// Retrieve a value using a Person object as the key
String johnsPhone = people.get(new Person("John", 30));
```

In this example, the Person class implements the hashCode() and equals() methods to use the name and age fields as the basis for comparison. A HashMap is then created using Person objects as keys, and key-value pairs are added to the map. Finally, a value is retrieved from the map using a new Person object with the same name and age fields as the key.

Does HashMap store value in ordered way or not?

No, a HashMap does not store its values in any particular order. The order in which the key-value pairs are stored in a HashMap is not guaranteed, and may change over time as the internal structure of the HashMap is modified due to resizing or other operations.

The order of the key-value pairs in a HashMap is determined by the hash code of the keys, which is used to compute the index where each key-value pair is stored in the internal array of the HashMap. Because the hash codes of the keys are used to determine the storage location of the values, there is no inherent ordering of the key-value pairs based on their values.

If you need to maintain a specific ordering of the key-value pairs in a collection, you should consider using a different data structure such as a LinkedHashMap, which maintains the insertion order of its elements, or a TreeMap, which maintains a natural ordering of its elements based on their keys.

What is HashSet and TreeSet?

Both HashSet and TreeSet are implementations of the Set interface in Java.

A HashSet is an unordered collection of unique elements. It uses a hash table to store the elements, which allows for constant-time performance for basic operations such as add, remove, and contains. However, because the elements are not ordered, the order in which they are stored is not guaranteed.

Here is an example of how to create a HashSet and add elements to it:

```
Set<String> set = new HashSet<>();

set.add("apple");

set.add("banana");

set.add("orange");
```

A TreeSet, on the other hand, is an ordered collection of unique elements. It is implemented as a self-balancing binary search tree, which allows for log(n) performance for basic operations such as add, remove, and contains. Because the elements are ordered, the order in which they are stored is guaranteed according to their natural ordering or a custom comparator.

Here is an example of how to create a TreeSet and add elements to it:

```
Set<String> set = new TreeSet<>();

set.add("apple");

set.add("banana");

set.add("orange");
```

In general, you should choose a HashSet when you don't care about the order of the elements and need fast performance for basic operations, and a TreeSet when you need to maintain a specific ordering of the elements or perform range queries over the elements.

How to get values from HashSet?

To get the values in a HashSet in Java, you can use an iterator or a for-each loop.

Here is an example of how to use an iterator to get the values in a HashSet:

```
Set<String> set = new HashSet<>();
set.add("apple");
set.add("banana");
set.add("orange");
Iterator<String> iterator = set.iterator();
while (iterator.hasNext()) {
    String value = iterator.next();
    System.out.println(value);
}
```

In this example, an iterator is obtained from the HashSet using the iterator() method, and the hasNext() method is called to check if there are more elements to iterate over. If there are, the next() method is called to retrieve the next element in the set, and the value is printed to the console.

Alternatively, you can use a for-each loop to iterate over the elements of the HashSet:

```
Set<String> set = new HashSet<>();
set.add("apple");
set.add("banana");
set.add("orange");
```

```
for (String value : set) {
    System.out.println(value);
}
```

In this example, a for-each loop is used to iterate over the elements of the HashSet. The loop variable value is set to each element in turn, and the value is printed to the console.

What is the difference between them, which one will compile and what is the best way to declare?

List ls = new List ().

List ls = new ArrayList ().

Arraylist arr = new ArrayList ().

The first line of code is not valid in Java, as List is an interface and cannot be directly instantiated.

The second line of code creates an ArrayList object and assigns it to a List reference variable:

List<Object> ls = new ArrayList<>();

This code creates an empty ArrayList that can store objects of any type, and assigns it to the ls reference variable of type List. This is a common practice in Java, as it allows for greater flexibility in the code, since you can switch to a different List implementation (such as LinkedList) without changing the rest of the code.

The third line of code creates an ArrayList object and assigns it to an ArrayList reference variable:

ArrayList<Object> arr = new ArrayList<>();

Difference between ArrayList and LinkedList?

Both ArrayList and LinkedList are implementations of the List interface in Java, but they have different characteristics that make them suitable for different use cases. Here are some key differences between ArrayList and LinkedList:

Data Structure: ArrayList is based on a dynamic array, while LinkedList is based on a doubly-linked list.

Memory Allocation: ArrayList allocates memory in chunks, while LinkedList allocates memory for each element separately.

Indexing: ArrayList provides fast random access to elements using an index, while LinkedList provides slower access because it needs to traverse the list from the beginning or end to reach a specific element.

Insertion/Deletion: ArrayList is slower for inserting or deleting elements in the middle of the list, because it requires shifting elements to fill the gap. LinkedList is faster for these operations, because it only requires updating the links between nodes.

Iteration: ArrayList is faster for iterating over all elements in the list, because it can use an index to access elements directly. LinkedList is slower for this operation, because it needs to traverse the list using its links.

Here is an example to illustrate the difference between ArrayList and LinkedList:

```
List<String> arrayList = new ArrayList<>();
arrayList.add("one");
arrayList.add("two");
arrayList.add("three");
arrayList.add("four");
List<String> linkedList = new LinkedList<>();
linkedList.add("one");
linkedList.add("two");
linkedList.add("three");
linkedList.add("four");
// Random access using index
String element1 = arrayList.get(1); // O(1)
String element2 = linkedList.get(1); // O(n)
// Insertion in the middle of the list
```

```
arrayList.add(2, "two-and-a-half"); // O(n)
linkedList.add(2, "two-and-a-half"); // O(1)
// Iteration over all elements
for (String element : arrayList) { // O(n)
    System.out.println(element);
}
for (String element : linkedList) { // O(n)
    System.out.println(element);
}
```

Difference between Set and List collection?

In Java, Set and List are both interfaces that represent collections of objects. However, they have different characteristics and are used for different purposes.

The main differences between Set and List are:

Duplicates: Set does not allow duplicate elements, while List does. If you try to add a duplicate element to a Set, it will not be added, while in a List it will be added as a new element.

Order: List maintains the order of elements as they are added to the list, while Set does not guarantee any specific order of elements.

Indexing: List provides indexed access to its elements using an integer index, while Set does not. You can access an element in a List using its index, while in a Set you need to iterate over the elements to find the one you want.

Iteration: Both List and Set provide ways to iterate over their elements, but the order of iteration is guaranteed for List and not for Set.

Here is an example to illustrate the difference between Set and List:

```
List<String> list = new ArrayList<>();
list.add("one");
list.add("two");
```

```
list.add("three");
list.add("two");
Set<String> set = new HashSet<>();
set.add("one");
set.add("two");
set.add("three");
set.add("two");
System.out.println(list); // prints [one, two, three, two]
System.out.println(set); // prints [one, two, three]
```

In this example, we create a List and a Set with the same elements. We add a duplicate element ("two") to both collections. When we print the collections, we see that the List contains the duplicate element, while the Set does not. This is because the Set does not allow duplicates, while the List does.

Overall, you should use a List when you need to maintain the order of elements and allow duplicates, and a Set when you don't care about the order of elements and need to ensure that there are no duplicates.

Difference between HashSet and HashMap?

In Java, both HashSet and HashMap are used to store collections of objects, but they have different characteristics and are used for different purposes.

The main differences between HashSet and HashMap are:

Key-Value pairs: HashMap stores key-value pairs, while HashSet only stores values. In other words, HashMap allows you to associate a value with a key, while HashSet only stores individual values.

Duplicates: HashSet does not allow duplicate elements, while HashMap allows duplicate values but not duplicate keys. If you try to add a duplicate value to a HashSet, it will not be added, while in a HashMap it will be added as a new value. If you try to add a

duplicate key to a HashMap, the existing value will be replaced by the new value.

Ordering: HashMap does not guarantee any specific order of its elements, while HashSet does not maintain the order of its elements. If you need to maintain the order of elements in a collection, you should use LinkedHashMap or LinkedHashSet.

Retrieval: In HashMap, you can retrieve values using a key, while in HashSet you need to iterate over the elements to find the one you want.

Here is an example to illustrate the difference between HashSet and HashMap:

```
HashMap<String, Integer> hashMap = new HashMap<>();
hashMap.put("one", 1);
hashMap.put("two", 2);
hashMap.put("three", 3);
HashSet<Integer> hashSet = new HashSet<>();
hashSet.add(1);
hashSet.add(2);
hashSet.add(3);
System.out.println(hashMap.get("two")); // prints 2
System.out.println(hashSet.contains(2)); // prints true
```

In this example, we create a HashMap with key-value pairs and a HashSet with individual values. We retrieve a value from the HashMap using a key and check if a value exists in the HashSet.

Overall, you should use HashMap when you need to associate a value with a key and allow duplicate values, and use HashSet when you need to store unique values and don't need to associate them with keys.

Why does HashMa not maintain the order like Linked-HashMap?

HashMap does not maintain the order of keys because it is designed to be as fast and memory-efficient as possible. Maintaining the insertion order of keys would require additional overhead, both in terms of time and space.

HashMap uses a hash table to store its entries. A hash table is a data structure that maps keys to values by using a hash function to convert each key to a unique index. This allows HashMap to quickly find the value for a given key.

To maintain the insertion order of keys, HashMap would need to use a different data structure, such as a linked list. A linked list is a data structure that stores items in a linear sequence. Each item in a linked list has a pointer to the next item in the sequence. This allows linked lists to maintain the order in which items are added.

How does LinkedHashMap is able to maintain the insertion order?

LinkedHashMap is able to maintain the insertion order of keys by using a doubly-linked list to store its entries. A doubly-linked list is a data structure that stores items in a linear sequence. Each item in a doubly-linked list has a pointer to the next item in the sequence and a pointer to the previous item in the sequence. This allows doubly-linked lists to maintain the order in which items are added.

When you add an entry to a LinkedHashMap, it is added to the end of the doubly-linked list. When you iterate over a LinkedHashMap, the entries are returned in the order in which they were added, because the iterator traverses the doubly-linked list.

```
LinkedHashMap {
  // Hash table
  key1 -> value1
  key2 -> value2
  key3 -> value3
```

```
  // Doubly-linked list
  head -> key1 -> key2 -> key3 -> tail
}
```

Difference between LinkedHashMap and Priority queue?

LinkedHashMap and PriorityQueue are two different types of collections in Java with different characteristics and use cases.

LinkedHashMap is a type of HashMap that maintains the insertion order of elements. In other words, the elements in a LinkedHashMap are stored in the order they were added to the map. It uses a doubly linked list to maintain the order of elements, which makes it slightly slower than a regular HashMap.

On the other hand, PriorityQueue is an implementation of the Queue interface that orders its elements according to their natural ordering or a custom comparator. The elements are stored in a heap data structure, which allows for efficient insertion and removal of elements in logarithmic time complexity.

The main difference between LinkedHashMap and PriorityQueue is their ordering strategy. LinkedHashMap maintains the insertion order of elements, while PriorityQueue maintains a priority order based on a sorting strategy. Additionally, LinkedHashMap is a map, which means it associates keys with values, while PriorityQueue is a queue that stores elements in a particular order.

Here is an example to illustrate the difference between LinkedHashMap and PriorityQueue:

```
LinkedHashMap<String, Integer> linkedHashMap = new LinkedHashMap<>();

linkedHashMap.put("one", 1);

linkedHashMap.put("two", 2);

linkedHashMap.put("three", 3);

PriorityQueue<Integer> priorityQueue = new PriorityQueue<>();

priorityQueue.add(3);
```

```
priorityQueue.add(1);
priorityQueue.add(2);
System.out.println(linkedHashMap); // prints {one=1, two=2, three=3}
System.out.println(priorityQueue); // prints [1, 3, 2]
```

In this example, we create a LinkedHashMap with three elements and a PriorityQueue with three elements. We print both collections to show the difference in their ordering strategy.

Overall, you should use LinkedHashMap when you need to maintain the insertion order of elements and access them by key, while PriorityQueue should be used when you need to maintain a priority order of elements and access them in a first-in-first-out (FIFO) order.

Write a Hashcode implementation and what is return type of it?

hashCode() is a method defined in the Object class in Java that returns an integer hash code for the object. The hash code is typically used by hash-based data structures like HashMap, HashSet, and Hashtable to quickly look up objects and improve performance.

Here is an example implementation of hashCode() for a class Person:

```
public class Person {
    private String name;
    private int age;
    private String address;
    // constructor, getters, setters
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + age;
```

```
        result = prime * result + ((address == null) ? 0 :
address.hashCode());
        result = prime * result + ((name == null) ? 0 :
name.hashCode());
        return result;
    }
}
```

In this implementation, we use the prime number 31, which is commonly used in Java hash code implementations. We then multiply the result variable by prime and add each member variable of the class to the hash code calculation. We use the hashCode() method of each member variable that is an object to get its hash code, and we check for null values to avoid NullPointerExceptions.

The return type of hashCode() is int. The returned hash code should ideally be unique for each object, but it is not required to be unique. Instead, it should be consistent with the equals() method of the class, which is used to determine if two objects are equal. If two objects are equal according to equals(), they should have the same hash code. If two objects are not equal according to equals(), they can have the same hash code (this is called a hash code collision), but this can decrease the performance of hash-based data structures.

What is ConcurrentHashmap?

ConcurrentHashmap is a thread-safe implementation of the Map interface in Java. It was introduced in Java 5 to provide a high-performance, scalable, and concurrent hash table that can be used in multi-threaded environments.

The main feature of ConcurrentHashmap is its ability to allow multiple threads to access and modify the map concurrently without the need for external synchronization. This is achieved by dividing the map into multiple segments, each of which is protected by a separate lock. This allows multiple threads to read and write to

different segments of the map simultaneously, improving the performance of concurrent operations.

ConcurrentHashmap provides the same basic operations as a regular HashMap, such as put(), get(), remove(), and containsKey(). It also provides additional atomic operations, such as putIfAbsent(), remove(), and replace(), which can be used to perform atomic updates to the map.

ConcurrentHashMap is particularly useful in applications where multiple threads need to access a shared map concurrently, such as in web servers, database systems, and other multi-threaded applications.

Here's an example of how to use ConcurrentHashMap:

```
ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();

// Add elements to the map

map.put("one", 1);

map.put("two", 2);

map.put("three", 3);

// Retrieve elements from the map

System.out.println(map.get("one")); // prints 1

// Remove an element from the map

map.remove("two");

// Check if a key exists in the map

System.out.println(map.containsKey("two")); // prints false
```

In this example, we create a ConcurrentHashMap and add three elements to it. We then retrieve an element by key, remove an element by key, and check if a key exists in the map.

Note that ConcurrentHashMap does not provide any guarantees about the order in which elements are inserted or accessed, so if you need to maintain ordering, you should use a different type of map, such as LinkedHashMap.

What is the Internal implementation of ConcurrentHashmap?

In Java 8, the introduction of default and static methods in interfaces allows for new functionality and more flexibility in the way that interfaces can be used. The main differences between static and default methods are:

Static methods: A static method is a method that is associated with the interface itself, rather than with any instance of the interface. They can be called directly on the interface, without needing an instance of the class that implements it.

Default methods: A default method is a method that has a default implementation in the interface. Classes that implement the interface are not required to override the default method, but can choose to do so if they need to provide a different implementation.

Access Modifiers: Static methods can have any access modifiers like public, private, protected, default. But for default methods, the access modifiers can only be public or default.

Overriding: Classes that implement the interface can override the default methods to provide their own implementation, but they cannot override static methods.

Purpose: The main purpose of static methods in interfaces is to provide utility methods that can be called directly on the interface without needing an instance. The main purpose of default methods is to provide a default implementation of an interface method that can be used by classes that implement the interface, without needing to override the method.

Use case: Static methods are useful when you want to provide utility methods that are not tied to any particular instance of a class. Default methods are useful when you want to provide a default implementation for a method that is common to all classes that implement the interface, but can be overridden if needed.

Is it possible to modify ConcurrentHashmap using iterator?

Modifying a ConcurrentHashMap using an iterator is not recommended, as it can lead to race conditions and other concurrency issues.

When you use an iterator to iterate over a ConcurrentHashmap, the iterator provides a snapshot of the current state of the map, and any modifications made to the map while the iterator is active may not be reflected in the iterator's view of the map.

To modify a ConcurrentHashmap, it is generally recommended to use the map's built-in thread-safe methods, such as putIfAbsent, replace, or remove. These methods are designed to handle concurrent modifications to the map safely, without requiring the use of iterators.

If you do need to modify a ConcurrentHashMap while iterating over it, one approach is to use the ConcurrentHashMap's keySet() method to obtain a set of keys, and then iterate over the set while making modifications to the map using the built-in thread-safe methods. This approach can help avoid concurrency issues, but it may not be suitable for all use cases, depending on the specific requirements of your application.

What is the concurrent collection?

Concurrent collections are a type of data structure in Java that are designed to be used in multi-threaded environments, where multiple threads can access and modify the data structure concurrently. They provide thread-safe and efficient access to data, and are essential in building scalable and high-performance multi-threaded applications.

The following are some of the commonly used concurrent collections in Java:

ConcurrentHashMap: A thread-safe implementation of the Map interface that provides efficient and scalable access to key-value pairs.

ConcurrentSkipListMap: A thread-safe implementation of the NavigableMap interface that maintains its elements in a sorted order,

and provides efficient and scalable access to key-value pairs.

CopyOnWriteArrayList: A thread-safe implementation of the List interface that provides efficient and scalable access to elements, and allows concurrent iteration over the list without the risk of ConcurrentModificationException.

LinkedBlockingQueue: A thread-safe implementation of the BlockingQueue interface that provides efficient and scalable access to a queue of elements, and allows multiple threads to add and remove elements concurrently.

ConcurrentLinkedQueue: A thread-safe implementation of the Queue interface that provides efficient and scalable access to a queue of elements, and allows multiple threads to add and remove elements concurrently.

ConcurrentSkipListSet: A thread-safe implementation of the SortedSet interface that maintains its elements in a sorted order, and provides efficient and scalable access to elements.

These concurrent collections are essential in building high-performance and scalable multi-threaded applications, and they provide a wide range of functionalities to suit different use cases.

Can we insert Null in ConcurrentHashmap?

It depends on the version of Java you are using.
In Java 8 and below, it's not allowed to insert a null key or value in a ConcurrentHashMap, it will throw a NullPointerException.

However, starting with Java 9, ConcurrentHashMap has been updated to allow for null keys and values. However, it's not recommended to use null keys or values in a ConcurrentHashMap, since it can lead to unexpected behaviour and errors.

It's important to notice that, even though it's allowed to insert null keys and values, it's not a good practice, since concurrent data structures like ConcurrentHashMap are designed to handle concurrent operations and null keys and values can lead to unexpected behaviour and errors.

It's also worth mentioning that if you use the putIfAbsent method, it does not accept null keys or values, and it throws NullPointerException when trying to insert null keys or values.

What is a Concurrent Modification exception, and how to prevent that?

A Concurrent Modification exception is a runtime exception that occurs when multiple threads try to modify a collection (such as a list, set, or map) at the same time. The exception is thrown because the collection is not designed to be modified by multiple threads simultaneously, and as a result, the collection's state can become inconsistent.

To prevent a Concurrent Modification exception, you can use one of the following strategies:

Synchronization: You can use the synchronized keyword to synchronize access to the collection, so that only one thread can access the collection at a time.

```
List<String> list = new ArrayList<>();
synchronized (list) {
    list.add("item");
    list.remove("item");
}
```

Using a thread-safe collection: Java provides thread-safe collections, such as ConcurrentHashMap and CopyOnWriteArrayList, that are designed to be modified by multiple threads simultaneously. These collections use locks internally to ensure that the state of the collection remains consistent.

```
List<String> list = new CopyOnWriteArrayList<>();
list.add("item");
list.remove("item");
```

Using an Iterator: Using an Iterator to iterate over the collection and modify it, Iterator has a fail-fast behaviour, and it throws ConcurrentModificationException if it detects that the collection has been modified while iterating over it.

```
List<String> list = new ArrayList<>();
Iterator<String> it = list.iterator();
while (it.hasNext()) {
    String item = it.next();
    if (item.equals("item")) {
        it.remove();
    }
}
```

Using a for-each loop: Using a for-each loop to iterate over the collection and modify it, it will throw ConcurrentModificationException when the collection is modified.

```
List<String> list = new ArrayList<>();
for (String item : list) {
    if (item.equals("item")) {
        list.remove(item);
    }
}
```

It's important to note that all of the above solutions are based on the collection type, the number of threads and the read-write operations that you are going to perform. The best solution is the one that fits the best with the project requirements.

What is serialization?

Serialization is the process of converting an object into a stream of bytes that can be stored or transmitted over a network, and later reconstructed to create a new object with the same properties as the original. The process of serialization is commonly used in Java

for data persistence, inter-process communication, and distributed computing.

To make an object serializable, it must implement the Serializable interface, which is a marker interface that indicates to the Java Virtual Machine (JVM) that the object can be serialized. When an object is serialized, all of its instance variables and non-transient fields are written to a stream of bytes, along with information about the object's class and superclasses.

Java provides two main mechanisms for serializing and deserializing objects: ObjectOutputStream and ObjectInputStream. ObjectOutputStream is used to write the serialized object to an output stream, while ObjectInputStream is used to read the serialized object from an input stream.

Here's an example of how to serialize an object in Java:

```
import java.io.*;

public class SerializationDemo {
    public static void main(String[] args) {
        try {
            // create an object to be serialized
            Person person = new Person("John", 25);

            // serialize the object to a file
            FileOutputStream fileOut = new
FileOutputStream("person.ser");
            ObjectOutputStream out = new
ObjectOutputStream(fileOut);
            out.writeObject(person);
            out.close();
            fileOut.close();
            System.out.println("Serialized data is saved in person.ser");
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }
}
class Person implements Serializable {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
}
```

In this example, we create a Person object and serialize it to a file named person.ser. The Person class implements the Serializable interface, which allows it to be serialized using ObjectOutputStream. Once the object is serialized, it can be deserialized and reconstructed later using ObjectInputStream.

Serialization is a powerful tool in Java, but it also has some limitations and potential drawbacks. Serialized objects can take up a lot of disk space or network bandwidth, and they may not be compatible with different versions of the same class. It's important to carefully design and test serialization code to ensure that it meets the requirements of the application.

Uses of serialization? Why is this needed?

In addition to serialization, there are a few other use cases for Java object streams:

Remote Method Invocation (RMI): RMI is a mechanism for making remote method calls in Java. When a remote method call is made, the arguments and return values are serialized and sent over the network using object streams.

Caching: Object streams can be used for caching objects in memory or on disk. By serializing and deserializing objects, we can save the object state to a file or database and then reload it later as needed.

Copying Objects: In some cases, it may be useful to create a copy of an object. By serializing and deserializing the object, we can create an independent copy with the same state as the original.

Deep Cloning: Object streams can also be used to create deep clones of objects. By serializing and deserializing an object, we can create a new object with the same state, but with new references to all of its fields.

Messaging: Object streams can be used for messaging between different parts of a Java application. By serializing and deserializing objects, we can send messages that contain complex data structures over a messaging system.

Overall, object streams are a powerful and flexible tool in Java, with many potential use cases beyond just serialization.

When to use ArrayList and when to use LinkedList?

ArrayList and LinkedList are both classes that implement the List interface in Java and provide a way to store and manipulate collections of elements. However, their internal implementations are different.

ArrayList:

It uses an array as its underlying data structure.

Random access is fast since array provides constant time for get and set operations.

Insertions and deletions are slow since arrays are of fixed size and when an element is inserted or deleted, all the elements after the insertion or deletion point have to be shifted.

It can be used when the number of read operations are more than the write operations.

LinkedList:

It uses a doubly-linked list as its underlying data structure.

Each element in a linked list contains a reference to the next and previous element.

Insertions and deletions are faster since only the references need to be updated.

Random access is slow since it requires traversing the linked list starting from the head or tail.

It can be used when the number of write operations are more than the read operations.

It's important to note that, the choice between ArrayList and LinkedList depends on the use case and the operations that will be performed on the collection. If you need fast random access, go for ArrayList, if you need fast insertions and deletions and don't mind slower random access, go for LinkedList.

Also, it's worth mentioning that, LinkedList also implements the Deque interface and can be used as a double-ended queue, while ArrayList doesn't have that capability.

What is Garbage Collection?

Garbage collection is a process in which a program's runtime system automatically manages memory allocation and deallocation by freeing up memory that is no longer being used by the program. In other words, it is the automatic process of freeing memory occupied by objects that are no longer being referenced by the program.

In Java, memory is allocated to objects at runtime by the JVM. When an object is no longer being used by the program, it becomes eligible for garbage collection. The garbage collector runs

periodically to identify and remove objects that are no longer being used, freeing up memory for use by other objects.

The garbage collector works by identifying all objects that are no longer being used, marking them as garbage, and then reclaiming the memory occupied by those objects. The garbage collector uses various algorithms to identify and reclaim memory, such as reference counting, mark-and-sweep, and copying.

Garbage collection is important because it helps to prevent memory leaks and ensures that the program uses memory efficiently. Without garbage collection, a program would need to manually manage memory allocation and deallocation, which can be time-consuming and error-prone.

In Java, garbage collection is an integral part of the language and is managed automatically by the JVM, allowing programmers to focus on writing code rather than managing memory.

What is System.gc in java?

In Java, System.gc() is a method that is used to suggest to the JVM that it should run the garbage collector to free up unused memory.

The System.gc() method is not guaranteed to immediately run the garbage collector, as the JVM may decide to delay or ignore the request based on various factors such as system load, memory usage, or garbage collection algorithms. Therefore, it is generally not recommended to rely on this method for precise memory management, as the JVM will automatically run the garbage collector as needed.

It's worth noting that calling System.gc() unnecessarily can actually have a negative impact on performance, as it can cause the JVM to spend time running the garbage collector when it's not necessary. Therefore, it's generally best to let the JVM manage memory automatically and only use System.gc() when there is a specific need to force garbage collection, such as when profiling or debugging a program.

What kind of algorithm is used in the garbage collector?

mark-and-sweep algorithm and parallel GC are one of the algorithms used in garbage collector mechanism.

The mark-and-sweep algorithm is called a tracing garbage collector because it traces out the entire collection of objects that are directly or indirectly accessible by the program.

Parallel garbage collection - It uses mark-copy in the Young Generation and mark-sweep-compact in the Old Generation.

What are fast and fail-safe in collection framework?

In the collection framework, fast and fail-safe refer to two different types of iterators.

Fast iterators are designed to throw a ConcurrentModificationException if the collection is modified while the iterator is in use. This is because fast iterators keep track of the current position in the collection using a modification count. If the collection is modified, the modification count is incremented. When the iterator checks the modification count, it will throw an exception if it has changed, indicating that the collection has been modified since the iterator was created.

Fail-safe iterators are designed to not throw an exception if the collection is modified while the iterator is in use. This is because fail-safe iterators operate on a copy of the collection, not the original collection. When the iterator is created, it takes a snapshot of the collection. This snapshot is then used to iterate over the collection. If the collection is modified while the iterator is in use, the iterator will not be affected.

examples of fast and fail-safe iterators in the Java collection framework:

Fast iterators:

ArrayList iterator

HashMap iterator

Fail-safe iterators:

CopyOnWriteArrayList iterator

ConcurrentHashMap iterator

Where are static methods or static variables stored in Java memory?

Static methods and static variables in Java are stored in the Metaspace memory area. Metaspace is a region of memory that is used to store class metadata, such as the class name, its methods and fields, and its superclass. Static methods and variables are stored in Metaspace because they are associated with the class itself, rather than with any particular instance of the class.

Before Java 8, static methods and variables were stored in a separate area of memory called the PermGen. However, the PermGen was a fixed size, which could lead to problems if a program created a lot of classes. As a result, the PermGen was removed in Java 8 and replaced with Metaspace. Metaspace is a more flexible memory area that can grow or shrink as needed.

How to create custom exceptions in Java?

By Extending Exception or Runtime Exception class you can create custom exception class and write your custom implementation.

e.g.

```
public class MyCheckedException extends Exception {
    public MyCheckedException(String message) {
        super(message);
    }
}
```

What is the difference between Class and Instance variables?

Characteristic	Class variable	Instance variable
Scope	Shared by all instances of the class	Not shared by other instances of the class
Declaration	Declared with the static keyword	Declared without the static keyword

Initialization	Initialized once	Initialized for each instance of the class
Access	Accessed using the class name	Accessed using the object reference

What is the difference between Throw and Throws?

Characteristic	throw	throws
Purpose	To explicitly throw an exception from a method or any block of code.	To declare that a method may throw a specific type of exception.
Usage	Used within a method or any block of code.	Used in the method signature.
Exception types	Can be used to throw either checked or unchecked exceptions.	Can only be used to declare checked exceptions.

What is the difference between try/catch block and throws?

Characteristic	Try/catch block	Throws
Purpose	To handle exceptions that occur within the block	To declare that a method may throw a specific type of exception
Usage	Used within code blocks	Used in method declarations
Control flow	If an exception occurs within the try block, the program will jump to the corresponding catch block, which will handle the exception	If an exception is thrown by a method, the caller of the method is responsible for handling it

What is thedifference between HashMap and LinkedHashMap?

Feature	HashMap	LinkedHashMap
Maintains insertion order of keys	No	Yes
Internal implementation	Hash table	Hash table + doubly-linked list
Performance	Faster	Slower

Memory usage	Less	More

What is the difference between == and equals?

The == operator in Java is used to compare the reference equality of two objects. This means that it compares the memory addresses of the two objects. If the two objects have the same memory address, then the == operator will return true, otherwise it will return false.

The equals() method in Java is used to compare the logical equality of two objects. This means that it compares the values of the two objects. If the two objects have the same values, then the equals() method will return true, otherwise it will return false.

For primitive types, the == operator and the equals() method are equivalent. However, for object types, the == operator and the equals() method are not equivalent. The == operator will only return true if the two objects have the same memory address, while the equals() method will return true if the two objects have the same values.

Here is an example of how to use the == operator and the equals() method to compare strings:

If an exception is declared in throws and if an exception is encountered what will happen?

If an exception is declared in throws and if an exception is encountered, the exception will be thrown to the caller of the method. The caller of the method is then responsible for handling the exception.

```
public class MyClass {
    public void myMethod() throws MyException {
        // Code that may throw a MyException
    }
}
```

The throws declaration in the myMethod() method tells the caller of the method that it may throw a MyException. The caller of the method must then handle the exception if it occurs.

How to achieve inheritance without using an interface?

You can achieve inheritance without using an interface by using the extends keyword to extend a class. When you extend a class, you inherit all of the public and protected fields and methods of the base class.

```
public class Animal {
    private String name;
    public Animal(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}
public class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }
    public void bark() {
        System.out.println("Woof!");
    }
}
```

CHAPTER 4: MULTITHREADING

What is Multithreading?

Multithreading is a programming concept that involves the execution of multiple threads in a single process or program. A thread is an independent path of execution within a program that can run concurrently with other threads.

In a single-threaded program, the program executes instructions in a linear fashion, with each instruction completing before the next one begins. In a multithreaded program, multiple threads can run concurrently, with each thread executing a different part of the program at the same time.

Multithreading is used to achieve concurrency in a program, which can lead to improved performance and responsiveness. For example, a multithreaded program can allow one thread to handle user input and another thread to perform a long-running task in the background, without blocking the user interface.

However, multithreading also introduces new challenges, such as thread synchronization and race conditions. Proper synchronization mechanisms need to be implemented to ensure that threads can access shared resources safely and prevent data corruption or unexpected program behaviour.

Multithreading is widely used in modern software development, particularly in applications that require high performance or responsiveness, such as web servers, video games, and scientific simulations.

What is a ThreadPool In java?

In Java, a thread pool is a collection of pre-initialized threads that are used to execute a set of tasks. Thread pools are used to

optimize the performance of concurrent programs by reducing the overhead of creating and destroying threads.

The main advantage of using a thread pool is that it allows multiple tasks to be executed concurrently by reusing threads from a pool, rather than creating a new thread for each task. This reduces the overhead of creating and destroying threads, which can be expensive in terms of memory and CPU usage.

The Java concurrency API provides a built-in thread pool implementation called ExecutorService. ExecutorService is an interface that provides methods to submit tasks to a thread pool and manage its lifecycle. The Executors class provides utility methods for creating different types of ExecutorService implementations, such as a fixed thread pool, cached thread pool, or scheduled thread pool.

Here is an example of how to create and use a thread pool in Java:

```
ExecutorService executor = Executors.newFixedThreadPool(5);
for (int i = 0; i < 10; i++) {
    Runnable task = new Task(i);
    executor.submit(task);
}
executor.shutdown();
```

In this example, a fixed thread pool with a maximum of 5 threads is created using the newFixedThreadPool method of the Executors class. Ten tasks are then submitted to the thread pool using the submit method of the ExecutorService interface. Finally, the shutdown method is called to initiate a graceful shutdown of the thread pool.

How to create a Thread Pool and how to use it in the database connection pool?

A thread pool is a collection of worker threads that can be used to execute multiple tasks concurrently. A thread pool can be used to improve the performance of an application by reducing the overhead of creating and destroying threads for each task.

```
int numberOfThreads = 10;
Executor executor =
Executors.newFixedThreadPool(numberOfThreads);
```

The above code creates a fixed-size thread pool with 10 worker threads. The Executors.newFixedThreadPool() method takes an integer argument that specifies the number of worker threads in the thread pool.

Once the thread pool is created, you can submit tasks to be executed by the worker threads using the Executor.execute() method:

```
executor.execute(new MyTask());
```

A Thread Pool can also be used to create a database connection pool; this is a technique used to maintain a pool of open connections to a database. When a connection is requested, a connection from the pool is returned. When the connection is no longer needed, it is returned to the pool, rather than being closed.

This approach can improve the performance of the application by reducing the overhead of creating and closing connections to the database.

Here's an example of how to create a connection pool using the Apache DBCP library:

```
BasicDataSource dataSource = new BasicDataSource();
dataSource.setDriverClassName("com.mysql.jdbc.Driver");
dataSource.setUrl("jdbc:mysql://localhost:3306/mydb");
dataSource.setUsername("user");
dataSource.setPassword("password");
dataSource.setInitialSize(10);
dataSource.setMaxTotal(50);
```

Once the connection pool is created, you can retrieve a connection from the pool using the BasicDataSource.getConnection() method.

```
Connection connection = dataSource.getConnection();
```

What is the lifecycle of thread in java?

The life cycle of a thread in Java includes several states:

New: The thread is in the new state when it is first created using the new Thread() constructor, but before the start() method is called.

Runnable: The thread is in the runnable state when the start() method is called. It's now eligible to run and can be scheduled by the JVM to execute.

Running: The thread is in the running state when it is currently executing.

Blocked: The thread is in the blocked state when it is waiting for a resource, such as a lock or a semaphore.

Waiting: The thread is in the waiting state when it is waiting for another thread to perform a specific action.

Timed Waiting: The thread is in the timed waiting state when it is waiting for a specific period of time.

Terminated: The thread is in the terminated state when it has completed execution or when it has been interrupted by another thread.

A thread pool, on the other hand, is a collection of worker threads that can be used to execute multiple tasks concurrently. When a task is submitted to the thread pool, it is added to a queue and a worker thread from the pool is assigned to execute the task.

The life cycle of a thread in a thread pool includes the following states:

Idle: The thread is in the idle state when it is first created and is waiting for a task to be assigned.

Running: The thread is in the running state when it is executing a task.

Completed: The thread is in the completed state when it has finished executing a task and is returned to the idle state.

When a thread pool is created, a fixed number of worker threads are created and added to the pool. These worker threads remain in the pool until the thread pool is shut down. When a task is submitted to the thread pool, a worker thread is picked from the pool, the task is executed and the thread is returned to the pool again. This process is repeated for each task that is submitted to the thread pool.

How to do Thread dump analysis in java?

Thread dump analysis is the process of examining the state of threads in a Java program to identify and diagnose issues such as deadlocks, high CPU usage, or performance bottlenecks. Here are the steps to perform a thread dump analysis in Java:

Take a thread dump: A thread dump is a snapshot of the state of all threads in a Java program. To take a thread dump, you can use the jstack command-line tool or a Java profiler such as VisualVM or YourKit. For example, to take a thread dump using jstack, you can run the following command:

jstack <pid>

Where <pid> is the process ID of the Java program.

Analyze the thread dump: Once you have a thread dump, you can analyze it to identify potential issues such as deadlocks or high CPU usage. Look for threads that are blocked or waiting, as these can indicate potential issues. Pay attention to the stack traces of each thread, as they can provide valuable information about what the thread is doing and what resources it is waiting for.

Identify the root cause: Based on the information gathered from the thread dump analysis, you can identify the root cause of the issue and take appropriate action to address it. For example, if you identify a deadlock, you may need to modify the code to avoid acquiring locks in a circular order or use a timeout on locks to avoid indefinite blocking.

In addition to thread dumps, there are other tools and techniques available for thread analysis in Java, such as profiling tools like VisualVM, JProfiler, or YourKit, or logging frameworks like log4j or

SLF4J. These tools can provide more detailed information about thread activity and performance in a Java program.

Why is a Threadpool needed in multithreading?

Thread pools are useful in multithreaded programming because they provide a way to manage and optimize the performance of concurrent programs. Here are some reasons why thread pools are needed:

Reduced overhead: Creating and destroying threads can be expensive in terms of memory and CPU usage. Thread pools provide a way to reuse threads for multiple tasks, reducing the overhead of thread creation and destruction.

Increased scalability: By using a thread pool, you can increase the number of tasks that can be executed concurrently without having to create a new thread for each task. This can help to improve the scalability of a program by allowing it to handle more concurrent requests.

Improved resource management: Thread pools provide a way to limit the number of threads that can be created, which can help to prevent resource exhaustion and improve the overall performance of a program.

Better performance: Thread pools can improve the performance of a program by reducing the amount of time it takes to create and destroy threads, and by allowing tasks to be executed concurrently.

Simplified concurrency management: Thread pools provide a higher-level abstraction for managing concurrent tasks, making it easier to write and maintain multithreaded code.

Overall, thread pools are an important tool for optimizing the performance and scalability of concurrent programs, and are widely used in Java and other programming languages.

What is deadlock I multithreading?

Deadlock in Java is a situation that occurs when two or more threads are blocked and waiting for each other to release the resources they

hold. As a result, none of the threads can proceed with their execution, leading to a complete halt in the program.

A typical scenario for a deadlock involves two or more threads acquiring locks on multiple resources in different orders. For example, Thread A may acquire a lock on Resource X, while Thread B acquires a lock on Resource Y. If Thread A then attempts to acquire a lock on Resource Y while Thread B attempts to acquire a lock on Resource X, both threads will be blocked, waiting for the other thread to release the lock. This situation is known as a deadlock.

Deadlocks can be difficult to detect and diagnose, as they typically do not result in any error messages or exceptions. They can lead to significant performance degradation or even complete program failure if not properly handled.

To prevent deadlocks, it's important to follow best practices for concurrent programming, such as avoiding nested locks, acquiring locks in a consistent order, and using timeouts on locks to avoid indefinite blocking. Additionally, tools such as deadlock detection algorithms and thread profiling tools can be used to identify and diagnose deadlocks in a program.

How to check if there is deadlock and how to prevent it?

To check if there is a deadlock in a Java program, you can use various tools and techniques. One of the most common ways to detect a deadlock is by analyzing a thread dump. A thread dump is a snapshot of the current state of all threads in a Java program. You can use the jstack command-line tool or a Java profiler to capture a thread dump and analyze it to check for deadlocks.

To prevent deadlocks in a Java program, you can use several techniques, including:

Acquire locks in a consistent order: One of the main causes of deadlocks is when multiple threads acquire locks on resources in different orders. To prevent this, you can define a consistent order for acquiring locks and ensure that all threads follow the same order.

Use timeouts on lock acquisition: To prevent deadlocks caused by thread contention for a lock, you can use timeouts on lock acquisition. This allows threads to wait for a lock for a limited time, after which they release the lock and try again later.

Avoid nested locks: Nested locks, where a thread acquires one lock while holding another lock, can increase the likelihood of deadlocks. To prevent this, you can try to design your code to avoid nested locks wherever possible.

Use higher-level concurrency abstractions: Higher-level concurrency abstractions, such as semaphores, barriers, or thread-safe data structures, can help to simplify the management of concurrent code and reduce the likelihood of deadlocks.

Test and debug your code: Testing and debugging your code using tools such as jUnit, JMeter, or debuggers can help you identify and fix potential issues before they cause deadlocks in production.

By using these techniques, you can prevent deadlocks and ensure the reliability and performance of your Java program.

What is the difference between deadlock and Livelock?

Characteristic	Deadlock	Livelock
Definition	Two or more threads are waiting for each other to release a resource in order to proceed.	Two or more threads are continuously changing their state in response to each other's actions, but none of the threads are making any progress towards their goals.
Progress	None of the threads can make progress.	The threads are making progress, but not towards their goals.
Resolution	Identify and break the deadlock cycle.	Identify and eliminate the source of the livelock.

What are the Symptoms of deadlock?

A deadlock in a Java program can manifest in several ways, but there are some common symptoms that you can look for to identify

a deadlock. Here are some of the most common symptoms of a deadlock in Java:

Threads appear to be stuck or unresponsive: When a deadlock occurs, one or more threads may appear to be stuck or unresponsive, which can cause the program to become unresponsive as well.

The program hangs or stops responding: If a deadlock occurs, the program may hang or stop responding, even though it appears to be running normally.

CPU usage spikes: A deadlock can cause a spike in CPU usage, as the program may be using more CPU resources than necessary to execute a task.

Threads are waiting for resources: In a deadlock situation, one or more threads may be waiting for resources, such as locks or shared data, that are held by other threads that are waiting for resources held by the first thread.

Thread dump analysis shows a circular wait: When analyzing a thread dump, you may see a circular wait, where one thread is waiting for a resource held by another thread, which is in turn waiting for a resource held by the first thread.

If you suspect that your Java program is experiencing a deadlock, you can use various tools and techniques, such as thread dump analysis or profiling, to identify and fix the issue.

What is Static synchronization in java?

In Java, the keyword "static" is used to indicate that a method or variable belongs to the class rather than to an instance of the class. A static method or variable can be accessed without creating an instance of the class.

Static synchronization is a mechanism used to synchronize the access to a static method or variable by multiple threads. In Java, a static method or variable can be accessed by multiple threads simultaneously, which can lead to data inconsistencies if not used

properly. To prevent this, a static method or variable can be synchronized, so that only one thread can access it at a time.

To synchronize a static method in Java, you can use the keyword "synchronized" before the method declaration.

```
public static synchronized void myStaticMethod() {
    //method body
}
```

To synchronize a static variable in Java, you can use the keyword "synchronized" before the variable declaration or you can use a class level lock to synchronize the block of code that access the variable.

```
public static int myStaticVariable;
public static void addToStaticVariable(int value) {
    synchronized (MyClass.class) {
        myStaticVariable += value;
    }
}
```

It's important to notice that when you synchronize a static method or variable in Java, you are synchronizing the access to that method or variable across all instances of the class. This can lead to poor performance if the synchronized block of code is accessed frequently by multiple threads, it's important to make sure that the synchronization is used only when it's necessary and make sure that the synchronized block of code is as small as possible.

Which exception can be thrown from the threads run method?

The run() method of a Thread class in Java can throw an unchecked exception, ThreadDeath. Additionally, any exception thrown by the code inside the run() method will propagate out of the run() method and can be caught by an appropriate exception handler.

ThreadDeath is a special exception that is used by the Java Virtual Machine (JVM) to terminate a thread. It is not intended to be caught or handled by application code, and typically indicates that the thread has completed its execution.

It's important to note that ThreadDeath is an unchecked exception, which means that it does not need to be declared in a throws clause or caught by a catch block.

It's always a good practice to include try-catch block in the run method, it will handle any unexpected exception and prevent the thread from getting terminated abruptly.

What is thread-local?

Thread-local is a Java class that allows you to store data that is specific to a given thread.

Some of the Use case:

- Sharing data between different parts of the same thread without having to pass it around explicitly.
- Storing data that is specific to a particular user or request.
- Implementing the singleton pattern.

What is thread-local, weak references, volatile, finalize, finally andserialization?

1. Thread Local: Thread-local variables are unique to each thread and do not share their values with other threads.

2. Weak References: Weak references allow objects to be eligible for garbage collection when no strong references exist.

3. Volatile: The volatile keyword in Java ensures that a variable's value is always read and written from/to main memory, preventing thread-specific caching.

4. Finalize: The finalize method in Java is called by the garbage collector before an object is reclaimed. It's rarely used due to its unpredictability.

5. Finally: finally is a block in exception handling that is executed regardless of whether an exception is thrown or not.

6. Serialization: Serialization is the process of converting objects into a byte stream, often used for storage or network transmission.

CHAPTER 5: JAVA-8

What are the features of Java 8 and Java 11?

Java 8 was a major release of the Java programming language and platform, and it introduced several new features and improvements. Some of the most notable features of Java 8 include:

Lambda expressions: A way to define and pass around blocks of code as if they were objects, which allows for more concise, functional-style code.

Functional interfaces: Interfaces that have exactly one abstract method, which allows for behaviour parameterization and the ability to pass behaviour as a method argument.

Streams: A new API for processing collections of data that allows for operations such as filtering, mapping, and reducing to be performed in a more functional and readable way.

Date and time API: A new API for working with date and time, which replaces the legacy java.util.Date and java.util.Calendar classes.

Concurrent Accumulators: A set of classes designed for use with parallel streams, which allow for the efficient accumulation of values.

Java 11, released in 2018, is a long-term support release and it brings several important changes and improvements over Java 8. Some of the most notable features of Java 11 include:

Local-variable type inference: A new syntax that allows you to infer the type of a variable from the value being assigned to it, which can make your code more readable and concise.

What are lambda expressions and their use in java 8?

Lambda expressions are a new feature introduced in Java 8 that allow developers to write more concise, functional-style code. They are a way to define and pass around blocks of code as if they were objects.

A lambda expression is composed of three parts:
A list of parameters (or none) enclosed in parentheses.
The "arrow" token ->
The body of the lambda expression, which can be a single expression or a block of code.
Here is an example of a simple lambda expression that takes two integers and returns their sum:

```
(int a, int b) -> {return a + b; }
```

Lambda expressions can be used to define functional interfaces, which are interfaces that have a single abstract method. The java.util.function package in Java 8 includes several functional interfaces such as Consumer, Function, Predicate and Supplier.

Lambda expressions can also be passed to methods or used as arguments for functional interfaces. For example, the forEach method of the java.util.stream.Stream class takes a Consumer functional interface as an argument, allowing you to pass in a lambda expression to perform a specific action on each element in the stream.

Lambda expressions can also be used with other features of Java 8 such as streams and the new date and time API to perform operations such as filtering, mapping and reducing collections of data, in a more functional and readable way.

It's worth noting that, although lambda expressions can help make your code more concise and readable, they can also make it more difficult to understand if they are not used correctly. It's important to use them in a way that makes the code easy to understand and

maintain.

What are the Java 8 Interface changes?

Java 8 introduced several changes to the way interfaces work, including the addition of default methods and static methods. These changes were made to allow interfaces to provide more functionality and to make it easier to add new methods to existing interfaces without breaking existing code.

Default methods: Java 8 introduced the concept of default methods, which are methods that have a default implementation in an interface. This allows interfaces to provide a default implementation for methods, without requiring the classes that implement the interface to provide one.

Static methods: Java 8 also introduced the ability for interfaces to have static methods, which are methods that can be called on the interface itself, rather than on an instance of the interface.

Functional interface: Java 8 also introduced functional interface, an interface that has exactly one abstract method. This is used to create lambda expressions, which are used to implement the single abstract method of the functional interface.

Private methods: Java 9 introduced the ability to define private methods within interfaces. This feature allows the interfaces to have more encapsulation and organization and allows the interface to provide more functionality.

These changes to interfaces in Java 8 and later have made it possible to add new functionality to existing interfaces in a backwards-compatible way, and have also made it easier to create more functional and modular code.

What is a Functional interface in Java-8?

In Java 8, a functional interface is an interface that has exactly one abstract method. The "functional" in the name refers to the fact that

the interface can be used as the target of a lambda expression or method reference.

Functional interfaces are also known as Single Abstract Method Interfaces or SAM Interfaces. A functional interface can have any number of default and static methods.

Functional interfaces are annotated with @FunctionalInterface annotation.

The main use of functional interfaces is to create lambda expressions, which are used to implement the single abstract method of the functional interface.

For example,

```
@FunctionalInterface
interface MyFunctionalInterface {
    public void myMethod();
}
```

This is a functional interface because it has only one abstract method, myMethod().

A functional interface can be implemented using a lambda expression, like this:

```
MyFunctionalInterface myObject = () -> {
    // code here
};
```

Java 8 library has many functional interface such as:

java.util.function.Function<T,R>: Represents a function that takes an argument of type T and returns an argument of type R.
java.util.function.Consumer<T>: Represents an operation that takes a single input argument and returns no result.
java.util.function.Predicate<T>: Represents a predicate (boolean-valued function) of one argument.
java.util.function.Supplier<T>: Represents a supplier of results.

The above are examples of functional interfaces which are widely used in the Java 8 Stream API and other functional programming constructs in Java 8.

What are the types of Functional interfaces?

There are several types of functional interfaces in Java 8, each with a specific purpose. Some of the most commonly used functional interfaces include:

Consumer<T>: Represents an operation that takes a single input argument and returns no result. This interface is typically used to perform some operation on an object, such as printing it to the console.

Supplier<T>: Represents a supplier of results. This interface is typically used to create a new object or retrieve a value from a data source.

Predicate<T>: Represents a predicate (boolean-valued function) of one argument. This interface is typically used to test a condition and return a boolean value.

Function<T, R>: Represents a function that takes an argument of type T and returns an argument of type R. This interface is typically used to transform an object from one type to another.

UnaryOperator<T>: Represents an operation on a single operand that produces a result of the same type as its operand. It is a specialization of Function for the case where the operand and result are of the same type.

BinaryOperator<T>: Represents an operation upon two operands of the same type, producing a result of the same type as the operands.

BiConsumer<T, U>: Represents an operation that accepts two input arguments and returns no result.

BiFunction<T, U, R>: Represents a function that takes two arguments and produces a result.

BiPredicate<T, U>: Represents a predicate (boolean-valued function) of two arguments.

Runnable: Represents a command that can be executed.

These are some of the most common functional interfaces, but there are many others in the Java standard library, each with its own specific use case.

What is Method Reference in Java 8?

In Java 8, a method reference is a shorthand notation for a lambda expression that simply invokes an existing method. The basic syntax for a method reference is:

ClassName::methodName

For example, if you have a class called "MyClass" with a method called "myMethod", you could use a method reference to invoke that method like this:

MyClass::myMethod

You can also use method references with constructors and array constructors. The basic syntax for a constructor reference is:

ClassName::new

For example, if you have a class called "MyClass", you could use a constructor reference to create a new instance of that class like this:

MyClass::new

And the basic syntax for a array constructor reference is:

TypeName[]::new

For example, if you want to create an array of integers, you could use an array constructor reference like this:

int[]::new

Method references can be used in situations where a lambda expression would be used to invoke an existing method, such as when passing a method as an argument to a higher-order function.

What is Optional in java?

In Java, the Optional class is a container object which may or may not contain a non-null value. It is introduced in Java 8 as a part of the java.util package. It is used to represent a value that may not be present, and to prevent null pointer exceptions.

The main methods of the Optional class are:
of(T value): Creates an Optional instance with the given non-null value.
ofNullable(T value): Creates an Optional instance with the given value, which can be null.
empty(): Creates an empty Optional instance.
isPresent(): Returns true if the Optional contains a value, false otherwise.
get(): Returns the contained value, if present. If the Optional is empty, it throws a NoSuchElementException.
orElse(T other): Returns the contained value if present, otherwise returns the given default value.
orElseGet(Supplier<? extends T> supplier): Returns the contained value if present, otherwise returns the result of the given supplier function.
orElseThrow(Supplier<? extends X> exceptionSupplier): Returns the contained value if present, otherwise throws the exception provided by the given supplier function.
ifPresent(Consumer<? super T> consumer): If a value is present, invoke the specified consumer with the value, otherwise do nothing.
It is best practice to use Optional when the return type of a method can return null as it forces to handle the null case explicitly.

For example,

```
Optional<String> optional = Optional.ofNullable(null);
if(optional.isPresent()) {
    System.out.println(optional.get());
} else {
    System.out.println("No value");
}
```

In this example, the value of the optional is null, so the output would be "No value".

What are the Intermediate and terminal operations in java 8?

In Java 8, the Stream API is used to process collections of data in a functional manner. The Stream API provides two types of operations: intermediate and terminal.

Intermediate operations are operations that are performed on a stream, but do not produce a final result. They are used to transform the elements of a stream in some way, and return a new stream that contains the transformed elements. Examples of intermediate operations include filter, map, and flatMap.

Terminal operations are operations that produce a final result or a side-effect. They are used to consume the elements of a stream and produce a final result, such as a count, a sum, or a list. Examples of terminal operations include forEach, reduce, and collect.

Intermediate operations are lazy, meaning that they are not executed until a terminal operation is called. This allows multiple intermediate operations to be chained together, with the result of one operation being passed as the input to the next.

For example,

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

int sum = numbers.stream()
                .filter(n -> n % 2 == 0)
                .map(n -> n * 2)
```

```
        .reduce(0, Integer::sum);
```

In this example, filter is an intermediate operation that filters the stream of numbers to keep only even numbers. map is an intermediate operation that transforms each number in the stream by doubling it. reduce is a terminal operation that sums the numbers in the stream and returns the result.
It is important to note that once a terminal operation is called, the stream is considered consumed and it can't be reused.

What is parallel processing in Java-8, and what are its uses?

Parallel processing in Java 8 refers to the ability to perform operations on a stream in parallel, using multiple threads. The Java 8 Stream API provides the parallel() method, which can be used to create a parallel stream from an existing sequential stream.

A parallel stream automatically splits the data into smaller chunks and assigns each chunk to a separate thread for processing. The results from each thread are then combined to produce the final result.

Parallel processing can be useful for improving the performance of certain types of operations, such as filtering and mapping, on large data sets. It can also be used to perform complex computations in parallel, such as reducing a large data set to a single value.

For example,

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
int sum =
numbers.parallelStream().mapToInt(Integer::intValue).sum();
```

In this example, the parallelStream() method is used to create a parallel stream of the numbers, and the mapToInt() and sum() methods are used to calculate the sum of the numbers in parallel.

It's important to note that parallel processing may not always be beneficial and it is dependent on the size of data and nature of

operation. It's always good to check the performance of the operation in both parallel and sequential mode and compare the results.

What is the difference between Flat and flat-map methods in Java-8?

flatMap is a method in Java Streams that is used to convert a stream of collections or arrays into a single flattened stream. In contrast, the flat method is not a standard method in Java Streams.

Here is an example of using flatMap to flatten a stream of collections:

```
List<List<Integer>> nestedList = Arrays.asList(
    Arrays.asList(1, 2),
    Arrays.asList(3, 4),
    Arrays.asList(5, 6)
);
List<Integer> flattenedList = nestedList.stream()
    .flatMap(Collection::stream)
    .collect(Collectors.toList());
System.out.println(flattenedList); // Output: [1, 2, 3, 4, 5, 6]
```

In this example, we start with a List of List objects. We use the flatMap method to convert each inner List into a stream of integers, and then concatenate all the streams into a single stream of integers. Finally, we collect the resulting stream into a new List object.

The flat method, on the other hand, is not a standard method in Java Streams. It may be implemented as a custom method or library method, but its behavior would depend on the implementation.

What is default method its use?

A default method is a method defined in an interface that has a default implementation. Default methods were introduced in Java 8

to allow interfaces to be extended without breaking existing implementations.

Prior to Java 8, interfaces could only contain method signatures, which meant that any class that implemented an interface was required to provide an implementation for all of its methods. This could be problematic when you wanted to add new methods to an existing interface, because it would break all of the existing implementations.

With default methods, you can provide a default implementation for a method in an interface, which means that classes that implement the interface are not required to provide their own implementation. If a class does not provide its own implementation for a default method, it will use the default implementation defined in the interface.

Default methods are useful for extending existing interfaces without breaking existing implementations. They can also be used to provide a common implementation for a method that is applicable to all classes that implement the interface.

For example, consider an interface for a collection of items:

```
public interface Collection<T> {
    void add(T item);
    boolean contains(T item);
    int size();
    default boolean isEmpty() {
        return size() == 0;
    }
}
```

This interface defines three methods for adding items to the collection, checking if an item is contained in the collection, and getting the size of the collection. It also defines a default method, isEmpty(), that returns true if the size of the collection is 0.

Classes that implement this interface are not required to provide their own implementation for isEmpty(), because a default implementation is already provided in the interface. However, they can override the default implementation if they need to provide a different behaviour.

What is default and static methods in Java-8?

Default and static methods are two new features that were introduced in Java 8.

Default methods allow you to add new methods to interfaces without breaking existing code. This is done by providing a default implementation of the method in the interface. Classes that implement the interface can override the default implementation, or they can simply use the default implementation.

example of a default method:

```
public interface Animal {
  default void eat() {
    System.out.println("I am eating.");
  }
}
```

Any class that implements the Animal interface will have access to the eat() method, even if the class does not explicitly implement the eat() method.

Static methods are methods that can be declared in interfaces. Static methods belong to the interface itself, not to any specific instance of the interface. Static methods can be called without creating an instance of the interface.

```
public interface Animal {
  static void makeSound() {
    System.out.println("I am making a sound.");
  }
}
```

The makeSound() method can be called without creating an instance of the Animal interface:

Animal.makeSound(); // prints "I am making a sound."

Default and static methods can be used to improve the design of Java applications in a number of ways. For example, default methods can be used to add new functionality to existing interfaces, and static methods can be used to provide utility methods that are available to all classes that implement a particular interface.

What are the memory changes that happened in java8?

The following are some of the memory changes that happened in Java 8:

Metaspace: Java 8 introduced Metaspace to replace PermGen. Metaspace is a region of memory that is used to store class metadata, such as class names, field and method names, and method bytecode. Metaspace is part of the native memory heap, which means that it is not limited by the maximum heap size.

G1 garbage collector: Java 8 introduced the G1 garbage collector as the default garbage collector. The G1 garbage collector is a concurrent garbage collector, which means that it can collect garbage while the application is still running. This can improve the performance of applications that have large heaps.

CompressedOops: Java 8 introduced CompressedOops, which is a technique that can reduce the memory footprint of Java objects. CompressedOops works by compressing object pointers from 64 bits to 32 bits on 64-bit platforms. This can reduce the memory footprint of Java objects by up to 50%.

String deduplication: Java 8 introduced String deduplication, which is a technique that can reduce the memory footprint of String objects. String deduplication works by storing a single copy of each unique String object in memory. This can reduce the memory footprint of String objects by up to 50%.

Overall, the memory changes in Java 8 have made Java applications more memory-efficient. This is important for applications that run on

devices with limited memory, such as mobile devices and embedded systems.

What is the new Java 8 changes in HashMap?

Java 8 made the following changes to HashMap:

New hash function for Strings: Java 8 introduced a new hash function for Strings that is more resistant to hash collisions. This can improve the performance of HashMap when it is used to store Strings.

Treeification: Java 8 added a new feature called "treeification" to HashMap. Treeification automatically converts a linked list of entries in a bucket to a red-black tree when the number of entries in the bucket exceeds a certain threshold. This can improve the performance of HashMap when there are a large number of hash collisions.

ConcurrentHashMap: Java 8 introduced a new concurrent implementation of HashMap called ConcurrentHashMap. ConcurrentHashMap is designed to be safe for concurrent access by multiple threads.

Why are the variable inside lambda function final in java?

Variables inside lambda functions are final in Java because it helps to prevent concurrency problems. Lambda functions are often used to capture variables from the surrounding scope. If these variables were not final, then it would be possible for multiple threads to modify the variables at the same time, which could lead to unexpected results.

```
int x = 0;

Runnable runnable = () -> {

  x++; // This would cause a concurrency problem if multiple threads were executing this lambda function at the same time.

};

Thread thread1 = new Thread(runnable);

Thread thread2 = new Thread(runnable);
```

```
thread1.start();
thread2.start();
```

If the x variable were not final, then it is possible that both threads would increment the x variable at the same time, and the final value of x would be unpredictable.

By making variables inside lambda functions final, Java can ensure that these variables cannot be modified by multiple threads at the same time. This helps to prevent concurrency problems and makes Java code more robust.

Here is an example of how to use a lambda function without causing a concurrency problem:

```
int x = 0;
Runnable runnable = () -> {
  // This is safe because the variable x is final.
  int y = x + 1;
  System.out.println(y);
};
Thread thread1 = new Thread(runnable);
Thread thread2 = new Thread(runnable);
thread1.start();
thread2.start();
```

In this example, the x variable is final, so it cannot be modified by multiple threads at the same time. This ensures that both threads will read the same value for the x variable, and the output of the program will be predictable.

Overall, making variables inside lambda functions final is a good practice that can help to prevent concurrency problems and make Java code more robust.

CHAPTER 6: SPRING-FRAMEWORK

What is dependency injection?

Dependency injection is a design pattern used in software development that involves separating the creation of an object from its dependencies. It allows for a more flexible and testable code by decoupling the components of a software system.

In simple terms, dependency injection is a technique for providing the dependencies of an object from the outside, rather than having the object itself create or find them. This is achieved by injecting the dependencies into the object's constructor or by using a dedicated dependency injection framework.

By using dependency injection, software components become more modular and reusable. Changes to one component can be made without affecting the other components of the system, making it easier to maintain and extend the software. Additionally, it promotes better testing practices, as dependencies can be easily mocked or replaced during testing.

Overall, dependency injection is an important tool for creating well-structured and maintainable software systems.

What are the types of dependency injection and what benefit we are getting using that?

Dependency injection : Dependency injection (DI) is a design pattern that allows objects to be supplied with their dependencies, rather than having to create them themselves. There are several types of dependency injection, each with its own benefits:

Constructor injection: In this type of injection, the dependencies are passed to the constructor of the class when it is instantiated. This ensures that the class always has the required dependencies and can be useful for enforcing class invariants.

Setter injection: In this type of injection, the dependencies are passed to setter methods of the class after it has been instantiated. This allows the class to be reused in different contexts, as the dependencies can be changed at runtime.

Field injection: In this type of injection, the dependencies are injected directly into the fields of the class. This can be useful for simple classes with a small number of dependencies.

Method injection: In this type of injection, the dependencies are passed to a method of the class after it has been instantiated. This allows the class to be reused in different contexts, as the dependencies can be changed at runtime.

Each type of dependency injection has its own benefits, and the choice of which one to use will depend on the specific requirements of the application.

Constructor injection is useful when a class needs to be in a specific state when it is created. It makes the class more robust and less susceptible to bugs caused by incomplete initialization.

Setter injection allows the class to be reusable, as the dependencies can be changed at runtime, making it easy to test the class with different dependencies.

Field injection is the simplest way of injecting dependencies and it doesn't require any additional methods or constructors.

Method injection allows the class to be reusable, as the dependencies can be changed at runtime and it can be used to configure objects that need to be initialized with specific values.

Overall, dependency injection allows for more flexible and maintainable code by decoupling the implementation of a class from the creation and management of its dependencies. This makes it easier to test, understand, and evolve the code over time.

Which type of dependency injection do you prefer?

The preferred type of dependency injection depends on the specific use case and the requirements of the application.

Here are some benefits and considerations for each type of dependency injection:

Constructor injection:

Preferred when a bean has a mandatory dependency that must be provided at instantiation.

Constructor injection ensures that all dependencies are provided and valid at instantiation.

Constructor injection makes the code more readable and self-explanatory.

Constructor injection makes the code more testable, as the dependencies are explicit.

Setter injection:

Preferred when a bean has optional dependencies that can be provided later.

Setter injection allows the bean to be instantiated without all of its dependencies.

Setter injection makes the code more readable and self-explanatory.

Setter injection makes the code more testable, as the dependencies are explicit.

Field injection:

Preferred when a bean has a mandatory dependency that must be provided at instantiation.

Field injection is less verbose than constructor injection.

Field injection can make the code more difficult to read and understand.

Field injection can make the code more difficult to test, as the dependencies are not explicit.

Ultimately, the choice of which type of dependency injection to use depends on the specific requirements of your application and your team's coding style.

How does inversion of control works inside the Spring Container?

Inversion of Control (IoC) is a design pattern that allows control to be transferred from the application code to an external container. In the context of a Java application, this container is often referred to as an IoC container or a dependency injection (DI) container.

IoC containers are responsible for creating and managing objects, and they do this by relying on a set of configuration rules that define how objects are created and wired together.

Here's how IoC works inside an IoC container:

- **Configuration**: In order to use an IoC container, you need to configure it with a set of rules that define how objects should be created and wired together. This configuration is typically done using XML or Java annotations.
- **Object creation**: When your application requests an object from the container, the container uses the configuration rules to create a new instance of the requested object.
- **Dependency injection**: The container injects any required dependencies into the newly created object. These dependencies are typically defined in the configuration rules.
- **Object lifecycle management**: The container manages the lifecycle of the objects it creates. This means that it's responsible for creating, initializing, and destroying objects as required by the application.
- **Inversion of control**: By relying on the container to create and manage objects, the application code no longer has direct control over the object creation process.

Instead, the container takes on this responsibility, and the application code simply requests the objects it needs from the container.

Overall, the IoC container is responsible for managing object creation and lifecycle management, while the application code is responsible for defining the rules that govern how objects are created and wired together. This separation of concerns allows for greater flexibility and modularity in the application, as the application code can be easily modified without affecting the underlying object creation and management processes.

What is the difference Between BeanFactory and ApplicationContext?

In Spring Framework, both the BeanFactory and the ApplicationContext are used to manage the lifecycle and dependencies of beans, but they have some key differences.

BeanFactory: BeanFactory is the root interface for accessing a Spring container. It is the basic container providing only configuration management, without advanced features like internationalization or event propagation. BeanFactory is lightweight and suitable for simple applications, but it does not provide some advanced features like internationalization, event handling, and AOP support.

ApplicationContext: The ApplicationContext interface is a sub-interface of BeanFactory. It provides additional features such as support for internationalization (I18N) messages, application-layer specific contexts such as the WebApplicationContext for use in web applications, and the ability to publish application events to interested event listeners. It also provides support for AOP and can automatically publish events to listeners.

In summary, the BeanFactory is a more lightweight and simple container, while the ApplicationContext is a more advanced container that provides additional features such as internationalization, event handling, and AOP support. If your application needs only the basic

functionality of a container, the BeanFactory may be a better choice, while if your application needs more advanced features, the ApplicationContext may be a better choice.

What is difference between application context and bean context?

In Spring Framework, both the application context and bean context represent the context in which Spring-managed beans live. However, there are some key differences between these two concepts:

Scope: The application context is the top-level context for a Spring application, and it manages the lifecycle of all beans within the application. The bean context, on the other hand, is a child context that is created for a specific set of beans, typically defined within a module or subsystem of the application.

Configuration: The application context is responsible for configuring the entire application, and it can be configured using XML, annotations, or Java code. The bean context, on the other hand, is typically configured using XML or annotations, and it only contains the configuration for the beans within that context.

Lifecycle: The application context is responsible for managing the lifecycle of the entire application, including starting up and shutting down the application. The bean context, on the other hand, only manages the lifecycle of the beans within that context.

Accessibility: The application context is accessible throughout the entire application, while the bean context is only accessible within the context in which it is created.

In summary, the application context is the top-level context that manages the entire Spring application, while the bean context is a child context that manages a specific set of beans within the application. The application context is responsible for configuring and managing the lifecycle of the entire application, while the bean context only manages the beans within its scope.

What is the Spring bean lifecycle?

In Spring Framework, a bean is an object that is managed by the Spring IoC container. The lifecycle of a bean is the set of events that occur from its creation until its destruction.

The Spring bean lifecycle can be divided into three phases: instantiation, configuration, and destruction.

- **Instantiation**: In this phase, Spring IoC container creates the instance of the bean. Spring Framework supports several ways of instantiating a bean, such as through a constructor, a static factory method, or an instance factory method.

- **Configuration**: In this phase, Spring IoC container configures the newly created bean. This includes performing dependency injection, applying any bean post-processors, and registering any initialization and destruction call-backs.

- **Destruction**: In this phase, Spring IoC container destroys the bean instance. It is the last phase of the Spring bean lifecycle.

In addition to these three phases, Spring Framework also provides several callbacks that allow developers to specify custom initialization and destruction logic for a bean. These callbacks include:

@PostConstruct: Invoked after the bean has been constructed and all dependencies have been injected
init-method: Specifies a method to be called after the bean has been constructed and all dependencies have been injected
destroy-method: Specifies a method to be called just before the bean is destroyed.
@PreDestroy: Invoked before the bean is destroyed.

The Spring bean lifecycle is controlled by the Spring IoC container, which creates, configures, and manages the lifecycle of the beans. Developers can take advantage of the bean lifecycle callbacks to add custom initialization and destruction logic to their beans, making it easier to manage the lifecycle of their objects and ensuring that resources are properly.

What are bean scopes? What are prototype and request bean scopes?

In Spring Framework, a bean scope defines the lifecycle and the visibility of a bean within the Spring IoC container. Spring Framework provides several built-in bean scopes, each with a specific purpose and behaviour.

The following are the most commonly used bean scopes in Spring Framework:

- **singleton**: This is the default scope for a bean. A singleton bean is created only once per Spring IoC container and is shared by all the clients that request it.
- **prototype**: A prototype bean is created every time it is requested by a client. This scope is useful for beans that are stateful, and the state should not be shared between clients.
- **request**: A request-scoped bean is created for each HTTP request and is only available to the beans that are involved in handling that request.
- **session**: A session-scoped bean is created for each HTTP session and is only available to the beans that are involved in handling that session.
- **application**: An application-scoped bean is created for the entire lifetime of the web application, and is available to all beans throughout the application.

- **websocket**: A websocket-scoped bean is created for the duration of a WebSocket session, and is available to all beans that are involved in handling that session.

It's important to note that the scope of a bean affects the lifecycle and visibility of that bean within the Spring IoC container. By choosing the appropriate scope for a bean, developers can control how and when the bean is created and how it interacts with other beans in the application.

What is the stateless bean in spring? name it and explain it.

A stateless bean in Spring Framework is a bean that does not maintain any state between method invocations. This means that the bean does not store any information about the previous invocations, and each method call is handled independently.

Stateless beans are typically used for services that perform actions or calculations, but do not maintain any state between invocations. This can include services that perform mathematical calculations, access external resources, or perform other tasks that do not require the bean to maintain state.

Stateless beans can be implemented as singleton beans, and multiple clients can share the same instance of the bean. Since stateless beans do not maintain any state, they can be easily scaled horizontally by adding more instances of the bean to handle the increased load.

Stateless beans also have the advantage of being simpler and easier to reason about, since they do not have to worry about maintaining state between invocations. Additionally, since stateless beans do not maintain any state, they can be easily serialized and replicated for high availability and scalability.

How is the bean injected in spring?

In Spring, a bean is injected (or wired) into another bean using the Dependency Injection (DI) pattern. DI is a design pattern that allows

a class to have its dependencies provided to it, rather than creating them itself.

Spring provides several ways to inject beans into other beans, including:

Constructor injection: A bean can be injected into another bean by passing it as a constructor argument. Spring will automatically create an instance of the dependent bean and pass it to the constructor.

```
public class BeanA {
    private final BeanB beanB;
    public BeanA(BeanB beanB) {
        this.beanB = beanB;
    }
}
```

Setter injection: A bean can be injected into another bean by passing it as a setter method argument. Spring will automatically call the setter method and pass the dependent bean.

```
public class BeanA {
    private BeanB beanB;
    @Autowired
    public void setBeanB(BeanB beanB) {
        this.beanB = beanB;
    }
}
```

Field injection: A bean can be injected into another bean by annotating a field with the @Autowired annotation. Spring will automatically set the field with the dependent bean.

```
public class BeanA {
    @Autowired
    private BeanB beanB;
}
```

Interface injection: A bean can be injected into another bean by implementing an interface. Spring will automatically set the field with the dependent bean.

```
public class BeanA implements BeanBUser {
    @Autowired
    private BeanB beanB;
}
```

It's important to note that, you can use any combination of the above methods, but you should choose the appropriate one depending on your use case.

Also, Spring uses a technique called Autowiring to automatically wire beans together, Autowiring can be done by type, by name, or by constructor.

By default, Spring will try to autowire beans by type, but if there are multiple beans of the same type, it will try to autowire by name using the bean's name defined in the configuration file.

How to handle cyclic dependency between beans?

Let's say for example: Bean A is dependent on Bean B and Bean B is dependent on Bean A. How does the spring container handle eager & lazy loading?

A cyclic dependency between beans occurs when two or more beans have a mutual dependency on each other, which can cause issues with the creation and initialization of these beans.

There are several ways to handle cyclic dependencies between beans in Spring:

Lazy Initialization: By using the @Lazy annotation on one of the beans involved in the cycle, it can be initialized only when it is actually needed.

```
@Lazy
@Autowired
private BeanA beanA;
```

Constructor injection: Instead of using setter or field injection, you can use constructor injection, which will make sure that the

dependencies are provided before the bean is fully initialized.

```
public class BeanA {
    private final BeanB beanB;

    public BeanA(BeanB beanB) {
        this.beanB = beanB;
    }
}
```

Use a proxy: A proxy can be used to break the cycle by delaying the initialization of one of the beans until it is actually needed. Spring AOP can be used to create a proxy for one of the beans involved in the cycle.

Use BeanFactory: Instead of injecting the bean directly, you can use BeanFactory to retrieve the bean when it's actually needed.

```
public class BeanA {
    private BeanB beanB;

    @Autowired
    public BeanA(BeanFactory beanFactory) {
        this.beanB = beanFactory.getBean(BeanB.class);
    }
}
```

It's important to note that, the best way to handle cyclic dependencies will depend on the specific requirements of your application. Therefore, you should carefully analyze the problem and choose the approach that best suits your needs.

What method would you call a before starting/loading a Spring boot application?

In Spring Boot, there are several methods that can be called before starting or loading a Spring Boot application. Some of the most commonly used methods are:

main() method: The main() method is typically the entry point of a Spring Boot application. It is used to start the Spring Boot application by calling the SpringApplication.run() method.

@PostConstruct method: The @PostConstruct annotation can be used to mark a method that should be called after the bean has been constructed and all dependencies have been injected. This can be used to perform any necessary initialization before the application starts.

CommandLineRunner interface: The CommandLineRunner interface can be implemented by a bean to run specific code after the Spring Application context has been loaded.

ApplicationRunner interface: The ApplicationRunner interface can be implemented by a bean to run specific code after the Spring Application context has been loaded and the Application arguments have been processed.

@EventListener : The @EventListener annotation can be used to register a method to listen to specific Application events like ApplicationStartingEvent, ApplicationReadyEvent and so on.

It's important to note that the choice of method will depend on the specific requirements of the application, such as whether the method needs to be called after the application context has been loaded or after specific Application events.

How to handle exceptions in the spring framework?

There are several ways to handle exceptions in the Spring Framework:

try-catch block: You can use a try-catch block to catch and handle exceptions in the method where they occur. This approach is useful for handling specific exceptions that are likely to occur within a particular method.

@ExceptionHandler annotation: You can use the @ExceptionHandler annotation on a method in a @Controller class to handle exceptions that are thrown by other methods in the same class. This approach is useful for handling specific exceptions in a centralized way across multiple methods in a controller.

@ControllerAdvice annotation: You can use the @ControllerAdvice annotation on a class to define a global exception handler for multiple controllers in your application. This approach is useful for handling specific exceptions in a centralized way across multiple controllers.

HandlerExceptionResolver interface: You can implement the HandlerExceptionResolver interface to create a global exception handler for your entire application. This approach is useful for handling specific exceptions in a centralized way across the entire application.

ErrorPage: You can define an ErrorPage in your application to redirect to a specific page when a certain exception occurs. This approach is useful for displaying a user-friendly error page when an exception occurs.

@ResponseStatus annotation: You can use the @ResponseStatus annotation on an exception class to define the HTTP status code that should be returned when the exception is thrown.

How does filter work in spring?

In Spring Framework, a filter is a component that can be used to pre-process and post-process requests and responses in a web application. Filters are executed before and after the request is handled by the controller. They can be used for various purposes such as:

- Logging and auditing
- Authentication and Authorization
- Encoding and Decoding

- Compression
- Caching

A filter in Spring can be implemented as a class that implements the javax.servlet.Filter interface. This interface defines three methods: init(FilterConfig), doFilter(ServletRequest, ServletResponse, FilterChain), and destroy(). The init() method is called when the filter is first initialized, the doFilter() method is called for each request, and the destroy() method is called when the filter is being taken out of service.

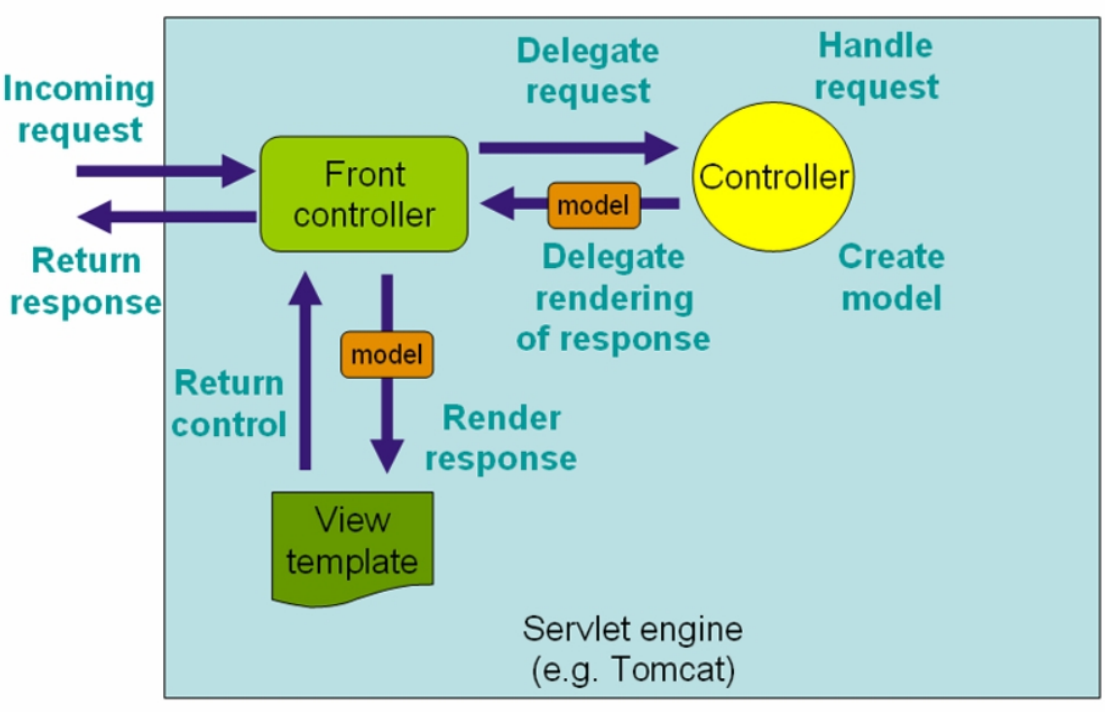
To use a filter in a Spring application, you can register the filter using FilterRegistrationBean or @WebFilter annotation. Once a filter is registered, it can be mapped to a specific URL pattern or servlet.

The doFilter() method of a filter is where the actual processing takes place. The method is passed a ServletRequest, a ServletResponse, and a FilterChain object. The FilterChain object represents the chain of filters that are executed for a particular request. The doFilter() method can choose to pass the request and response to the next filter in the chain by calling the doFilter() method on the FilterChain object, or it can choose to handle the request itself and not pass the request and response to the next filter.

In summary, filters are a powerful way to add pre-processing and post-processing to requests and responses in a Spring web application. They can be used for various purposes such as logging, authentication, encoding, compression, and caching. A filter is a class that implements the javax.servlet.Filter interface, it can be registered using FilterRegistrationBean or @WebFilter annotation, and it can be mapped to a specific URL pattern or servlet.

What is the Spring-MVC flow?

Spring MVC is a popular web framework for building Java web applications. It provides a Model-View-Controller architecture that separates the application logic into three components: the model, the view, and the controller.



The requesting processing workflow in Spring Web MVC (high level)

The Spring MVC flow involves the following steps:

Client sends a request: The user sends a request to the Spring MVC application through a browser or any other client application.

DispatcherServlet receives the request: The DispatcherServlet is a central controller in the Spring MVC architecture. It receives the request from the client and decides which controller should handle the request.

HandlerMapping selects the appropriate controller: The HandlerMapping component maps the request URL to the appropriate controller based on the URL pattern configured in the Spring configuration file.

Controller processes the request: The controller handles the request and performs the necessary processing logic. It may interact with the model component to retrieve data or update the data.

Model updates the data: The model component manages the data and provides an interface for the controller to retrieve or update the data.

ViewResolver selects the appropriate view: The ViewResolver component maps the logical view name returned by the controller to the actual view template.

View renders the response: The view template is rendered to generate the response. It may include data from the model component.

DispatcherServlet sends the response: The DispatcherServlet sends the response back to the client through the appropriate view technology, such as JSP, HTML, or JSON.

The Spring MVC flow is a cyclical process, as the client may send additional requests to the application, and the cycle repeats.

Can singleton bean scope handle multiple parallel requests?

A singleton bean in Spring has a single instance that is shared across all requests, regardless of the number of parallel requests. This means that if two requests are processed simultaneously, they will share the same bean instance and access to the bean's state will be shared among the requests.

However, it's important to note that if the singleton bean is stateful, and the state is shared among requests, this could lead to race conditions and other concurrency issues. For example, if two requests are trying to modify the same piece of data at the same time, it could lead to data inconsistencies.

To avoid these issues, it's important to make sure that any stateful singleton beans are designed to be thread-safe. One way to do this is to use synchronization or other concurrency control mechanisms such as the synchronized keyword, Lock or ReentrantLock classes, or the @Transactional annotation if the bean is performing database operations.

On the other hand, if the singleton bean is stateless, it can handle multiple parallel requests without any issues. It can be used to provide shared functionality that doesn't depend on the state of the bean.

In conclusion, a singleton bean can handle multiple parallel requests, but it's important to be aware of the state of the bean and to ensure that it's designed to be thread-safe if it has shared state.

Tell me the Design pattern used inside the spring framework.

The Spring Framework makes use of several design patterns to provide its functionality. Some of the key design patterns used in Spring are:

Inversion of Control (IoC): This pattern is used throughout the Spring Framework to decouple the application code from the framework and its components. The IoC container is responsible for managing the lifecycle of beans and injecting dependencies between them.

Singleton: A singleton pattern is used to ensure that there is only one instance of a bean created in the Spring IoC container. The singleton pattern is used to create a single instance of a class, which is shared across the entire application.

Factory: The factory pattern is used in Spring to create objects of different classes based on the configuration. Spring provides a factory pattern to create beans, which is based on the factory method design pattern.

Template Method: The template method pattern is used in Spring to provide a common structure for different types of operations. Spring provides several template classes such as JdbcTemplate, Hibernate Template, etc. that provide a common structure for performing database operations.

Decorator: The decorator pattern is used in Spring to add additional functionality to existing beans. The Spring AOP (Aspect-Oriented Programming) module uses the decorator pattern to add additional functionality to existing beans through the use of proxies.

Observer: The observer pattern is used in Spring to notify other beans of changes to the state of a bean. Spring provides the

ApplicationEvent and ApplicationListener interfaces, which can be used to implement the observer pattern.

Command: The command pattern is used in Spring to encapsulate the execution of a particular piece of code in a command object. This pattern is used in Spring to create reusable and testable code.

Façade: The façade pattern is used in Spring to simplify the interface of a complex system. The Spring Framework uses the façade pattern to provide a simplified interface for interacting with its components.

These are just a few examples of the design patterns used in Spring, there are many more. Spring framework makes use of these patterns to provide a consistent and simple way to build applications, making it easier to manage complex systems.

Is singleton bean scope thread-safe?

By default, a singleton bean in Spring is thread-safe, as only one instance of the bean is created and shared among all requests. However, it's important to note that the thread-safety of a singleton bean depends on the implementation of the bean and the way it's being used.

If the singleton bean is stateless, it can handle multiple parallel requests without any issues, as it does not maintain any state.

If the singleton bean is stateful, and the state is shared among requests, it could lead to race conditions and other concurrency issues if not designed properly. For example, if two requests are trying to modify the same piece of data at the same time, it could lead to data inconsistencies. To avoid these issues, it's important to make sure that any stateful singleton beans are designed to be thread-safe by using synchronization or other concurrency control mechanisms such as the synchronized keyword, Lock or ReentrantLock classes, or the @Transactional annotation if the bean is performing database operations.

In summary, a singleton bean is thread-safe by default, but the thread-safety of a singleton bean depends on the implementation of the bean and the way it's being used. If the bean is stateless it can handle multiple parallel requests without issues, if it's stateful it should be designed to be thread-safe in order to handle multiple parallel requests correctly.

How do factory design patterns work in terms of the spring framework?

In Spring, the factory design pattern is used to create objects of different classes based on the configuration. The Spring IoC container uses the factory pattern to create beans, which is based on the factory method design pattern.

The factory method is a design pattern that provides a way to create objects of different classes based on a factory interface. In Spring, the IoC container acts as the factory, and the factory interface is represented by the BeanFactory or ApplicationContext interfaces.

The IoC container is responsible for creating and managing the lifecycle of beans. When you define a bean in the configuration, the IoC container will use the factory pattern to create an instance of the bean. The IoC container will then manage the lifecycle of the bean, including injecting dependencies, initializing the bean, and destroying the bean when it is no longer needed.

Here's an example of how you can define a bean in Spring using the factory design pattern:

```
@Configuration
public class MyConfig {
    @Bean
    public MyService myService() {
        return new MyService();
    }
}
```

In this example, the myService() method is annotated with @Bean. This tells Spring to create an instance of the MyService class when the IoC container is created. The IoC container will use the factory pattern to create the instance and manage its lifecycle.

Another way to use factory pattern in spring is to use FactoryBean interface, which allows you to create beans that are created by a factory method, it's a factory of bean. The FactoryBean interface defines a single method, getObject(), which returns the object that should be exposed as the bean in the Spring application context.

In summary, the factory design pattern is used in the Spring Framework to create objects of different classes based on the configuration. The Spring IoC container acts as the factory, and the factory interface is represented by the BeanFactory or ApplicationContext interfaces, creating and managing the lifecycle of beans, and also can be used by implementing FactoryBean interface to create beans in a factory method.

How the proxy design pattern is used in spring?

The proxy design pattern is used in Spring to add additional functionality to existing objects. The Spring Framework uses the proxy pattern to provide AOP (Aspect-Oriented Programming) functionality, which allows you to add cross-cutting concerns, such as logging, security, and transaction management, to your application in a modular and reusable way.

In Spring, AOP proxies are created by the IoC container, and they are used to intercept method calls made to the target bean. This allows you to add additional behaviour, such as logging or security checks, before or after the method call is made to the target bean.

AOP proxies are created using one of three proxy types: JDK dynamic proxies, CGLIB proxies, or AspectJ proxies.

JDK dynamic proxies: This is the default proxy type in Spring, and it is used to proxy interfaces.

CGLIB proxies: This proxy type is used to proxy classes, and it works by creating a subclass of the target bean.
AspectJ proxies: This proxy type uses the AspectJ library to create proxies, and it allows you to use AspectJ pointcuts and advice in your application.
Spring uses the proxy pattern to provide AOP functionality by generating a proxy object that wraps the target bean. The proxy object will intercept method calls made to the target bean, and it will invoke additional behavior, such as logging or security checks, before or after the method call is made to the target bean.

Here's an example of how you can use Spring AOP to add logging to a bean:

```
@Aspect
@Component
public class LoggingAspect {
    @Before("execution(* com.example.service.*.*(..))")
    public void logBefore(JoinPoint joinPoint) {
        log.info("Started method: " +
joinPoint.getSignature().getName());
    }
}
```

In this example, the LoggingAspect class is annotated with @Aspect and @Component to make it a Spring bean. The @Before annotation is used to specify that the logBefore() method should be executed before the method call is made to the target bean. The logBefore() method uses the JoinPoint argument to log the name of the method that is being called.
In summary, the proxy design pattern is used in Spring to add additional functionality to existing objects by intercepting method calls made to the target bean and invoke additional behavior before or after the method call using AOP functionality. The proxy objects are generated by the IoC container using one of three proxy types: JDK dynamic proxies, CGLIB proxies, or AspectJ proxies.

What if we call singleton bean from prototype or prototype bean from singleton How many objects returned?

When a singleton bean is called from a prototype bean or vice versa, the behavior depends on how the dependency is injected.

If a singleton bean is injected into a prototype bean, then each time the prototype bean is created, it will receive the same instance of the singleton bean. This is because the singleton bean is only created once during the startup of the application context, and that same instance is then injected into the prototype bean each time it is created.

On the other hand, if a prototype bean is injected into a singleton bean, then each time the singleton bean is called, a new instance of the prototype bean will be created. This is because prototype beans are not managed by the container, and a new instance is created each time a dependency is injected.

Here's an example to illustrate this:

```
@Component
@Scope("singleton")
public class SingletonBean {
   // code for singleton bean
}
@Component
@Scope("prototype")
public class PrototypeBean {
   @Autowired
   private SingletonBean singletonBean;
   // code for prototype bean
}
```

In this example, when a prototype bean is created and injected with the singleton bean, it will receive the same instance of the singleton

bean each time it is created. However, if the singleton bean is created and injected with the prototype bean, it will receive a new instance of the prototype bean each time it is called.

It's important to note that mixing singleton and prototype scopes in a single application context can lead to unexpected behavior and should be avoided unless necessary. It's best to use one scope consistently throughout the application context.

What is the difference between Spring boot and spring?

why choose one over the other? Here are some reasons to choose Spring Framework:

You need a comprehensive set of features and capabilities for your application.

You want to build a modular application where you can pick and choose only the components that you need.

You need a high degree of flexibility and customization in your application.

Here are some reasons to choose Spring Boot:

You want to quickly set up a stand-alone Spring application without needing to do a lot of configuration.

You want to take advantage of pre-configured dependencies and sensible defaults.

You want to easily deploy your application as a self-contained executable JAR file.

Overall, both Spring and Spring Boot are powerful frameworks that can be used to build enterprise-level applications. The choice between them depends on the specific needs of your application and the level of flexibility and customization that you require.

How can you create a prototype bean?

A prototype bean in Spring can be created by setting the "scope" attribute of the bean definition to "prototype". This tells the Spring framework to create a new instance of the bean each time it is

requested from the application context, instead of returning a single shared instance as is the case with a singleton-scoped bean.

Here's an example of how you can create a prototype bean using XML configuration:

```
<bean id="prototypeBean" class="com.example.PrototypeBean" scope="prototype">
  <!-- property definitions go here -->
</bean>
```

And here's an example using Java configuration:

```
@Configuration
public class AppConfig {

  @Bean(name="prototypeBean")
  @Scope("prototype")
  public PrototypeBean prototypeBean() {
    return new PrototypeBean();
  }

}
```

In both cases, each time you request the prototype bean from the application context, you will get a new instance of the PrototypeBean class.

What is Method overloading and method overriding? Where it has been used in the spring framework?

Method overloading and method overriding in the Spring Framework:

Method overloading and method overriding are used extensively in the Spring Framework. For example, the @Autowired annotation can be used to autowire dependencies into a Spring bean. The @Autowired annotation can be overloaded to support different types of dependencies, such as field dependencies, constructor dependencies, and setter dependencies.

Another example of method overloading in the Spring Framework is the getBean() method of the ApplicationContext interface. The getBean() method can be overloaded to accept different types of parameters, such as the bean name, the bean type, and the bean qualifiers.

Method overriding is also used extensively in the Spring Framework. For example, the AbstractBeanDefinitionReader class defines a loadBeanDefinitions() method that is overridden by different bean definition readers, such as the ClassPathBeanDefinitionScanner class and the XmlBeanDefinitionReader class.

If I want my class should not be extended and the instance cannot be created by other classes then how to declare class?

1.Declare the class as final. This will prevent other classes from extending it.

2.Make the constructor private. This will prevent other classes from creating instances of the class.

CHAPTER 7: SPRING-BOOT

Tell me About Spring-Boot's Entry point and how @SpringbootApplication annotation works?

In Spring Boot, the entry point of a web application is a class with the @SpringBootApplication annotation, which is typically located in the main package of the application. This class contains the main() method, which is executed when the application starts.

Here's an example of a typical entry point class for a Spring Boot application:

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

In this example, the MyApplication class is the entry point of the application and the main() method starts the Spring Boot application by calling the SpringApplication.run() method.

The @SpringBootApplication annotation is a combination of several other annotations, like @Configuration, @EnableAutoConfiguration, and @ComponentScan, which are used to configure the Spring application.

The @Configuration annotation indicates that the class is a source of bean definitions for the application context.

The @EnableAutoConfiguration annotation tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.

The @ComponentScan annotation tells Spring to look for other components, configurations, and services in the package, allowing it to find the controllers.

It's important to note that, you can use the SpringApplication.run() method to run a Spring application from any class in your code, but the class with the @SpringBootApplication annotation is typically used as the entry point because it provides a convenient way to configure a Spring application using Spring Boot features.

Explain below Spring-Boot annotations?

@Component:

The @Component annotation is a fundamental annotation in the Spring Framework and is used in Spring Boot to mark a class as a Spring-managed component. Spring components are objects that are managed by the Spring IoC container and can be injected with dependencies, managed in the container, and wired together with other components.

When a class is annotated with @Component, Spring Boot automatically detects it and creates an instance of the class as a bean in the Spring application context. The @Component annotation is a meta-annotation, which means that it can be used to create other annotations such as @Service, @Repository, and @Controller.

Here's an example of a class annotated with @Component in Spring Boot:

```
@Component
public class MyComponent {
  // class implementation
}
```

In the above example, MyComponent is a Spring-managed component that can be injected with dependencies,

managed by the Spring IoC container, and wired together with other components.

In summary, the @Component annotation is a fundamental annotation in Spring Boot that marks a class as a Spring-managed component. It is used to create beans in the Spring application context, which can be injected with dependencies and wired together with other components.

@Autowired:

The @Autowired annotation is used in Spring Boot to automatically wire beans together based on their dependencies. When a class is annotated with @Autowired, Spring Boot automatically injects the dependent beans into the class.

Here's an example of a class using @Autowired to inject a dependent bean:

```
@Service
public class MyService {
    private final MyRepository myRepository;
    @Autowired
    public MyService(MyRepository myRepository) {
        this.myRepository = myRepository;
    }
    // Class implementation
}
```

In the above example, the MyService class is a Spring-managed service that has a dependency on a MyRepository bean. The @Autowired annotation is used in the constructor to automatically inject the MyRepository bean into the class.

When Spring Boot starts up, it scans the application context for beans that have dependencies and automatically wires them

together. This allows you to write modular and maintainable code by decoupling the components of your application.

The @Autowired annotation can also be used on fields and setter methods, but constructor injection is generally considered best practice as it ensures that all dependencies are provided when the object is created.

In summary, the @Autowired annotation is used in Spring Boot to automatically wire beans together based on their dependencies. It can be used on constructors, fields, and setter methods to inject dependent beans into a class.

@GetMapping:

The @GetMapping annotation is used in Spring Boot to map HTTP GET requests to a method in a controller class. When a GET request is received for a specific URL, Spring Boot looks for a method in the controller class annotated with @GetMapping that matches the URL and executes the method.

Here's an example of a method in a controller class annotated with @GetMapping:

```
@RestController
public class MyController {
    @GetMapping("/hello")
    public String helloWorld() {
        return "Hello, World!";
    }
    // Other methods
}
```

In the above example, the helloWorld method is annotated with @GetMapping("/hello"), which maps the URL /hello to the method. When a GET request is received for the URL /hello, Spring Boot executes the helloWorld method and returns the string "Hello, World!" as the response.

The @GetMapping annotation can also be used with URL variables to extract values from the URL and use them in the method. Here's an example:

In the above example, the hello method is annotated with @GetMapping("/hello/{name}"), which maps the URL /hello/{name} to the method. The @PathVariable annotation is used on the name parameter to extract the value of the name variable from the URL and use it in the method.

In summary, the @GetMapping annotation is used in Spring Boot to map HTTP GET requests to a method in a controller class. It can be used with or without URL variables to extract values from the URL and use them in the method.

@PostMapping:

The @PostMapping annotation is used in Spring Boot to map HTTP POST requests to a method in a controller class. When a POST request is received for a specific URL, Spring Boot looks for a method in the controller class annotated with @PostMapping that matches the URL and executes the method.

Here's an example of a method in a controller class annotated with @PostMapping:

```
@RestController
public class MyController {
    @PostMapping("/submit")
    public String submitForm(@RequestBody FormObject form) {
        // process the form data
        return "Form submitted successfully!";
    }
    // Other methods
}
```

In the above example, the submitForm method is annotated with @PostMapping("/submit"), which maps the URL /submit to the

method. When a POST request is received for the URL /submit, Spring Boot executes the submitForm method and passes the form data as a FormObject object to the method.

The @RequestBody annotation is used to tell Spring Boot to deserialize the request body into a FormObject object. This allows you to easily handle complex form data in your controller methods.

The @PostMapping annotation can also be used with URL variables to extract values from the URL and use them in the method. Here's an example:

```
@RestController
public class MyController {
    @PostMapping("/submit/{id}")
    public String submitForm(@PathVariable Long id, @RequestBody FormObject form) {
        // process the form data and the id
        return "Form submitted successfully for ID " + id + "!";
    }
    // Other methods
}
```

In the above example, the submitForm method is annotated with @PostMapping("/submit/{id}"), which maps the URL /submit/{id} to the method. The @PathVariable annotation is used on the id parameter to extract the value of the id variable from the URL and use it in the method.

In summary, the @PostMapping annotation is used in Spring Boot to map HTTP POST requests to a method in a controller class. It can be used with or without URL variables to extract values from the URL and use them in the method. The @RequestBody annotation is used to deserialize the request body into a Java object.

@Repository:

The @Repository annotation is used in Spring Framework to indicate that the annotated class is a repository, which is a class that provides data access operations (e.g., reading, writing, and querying data) for a specific domain object or set of domain objects.

By annotating a class with @Repository, Spring Boot will automatically create an instance of that class and manage its lifecycle as a Spring bean. This means that the repository can be injected into other Spring-managed components (e.g., services or controllers) using the @Autowired annotation.

Here's an example of a repository class annotated with @Repository:

```
@Repository
public class MyRepository {
    public void save(MyObject myObject) {
        // save the object to the database
    }
    public MyObject findById(Long id) {
        // find the object with the given id in the database
        return null;
    }
    // other data access methods
}
```

In the above example, the MyRepository class is annotated with @Repository, which tells Spring Boot that this class is a repository. The class provides two data access methods, save and findById, which can be used to save or retrieve instances of MyObject from the database.

The @Autowired annotation can be used to inject the repository into other Spring-managed components. For example, here's a service class that uses the MyRepository class:

```
@Service
```

```
public class MyService {
    @Autowired
    private MyRepository myRepository;
    public void save(MyObject myObject) {
        myRepository.save(myObject);
    }
    public MyObject findById(Long id) {
        return myRepository.findById(id);
    }
    // other service methods
}
```

In the above example, the MyService class is annotated with @Service, which tells Spring Boot that this class is a service. The @Autowired annotation is used to inject the MyRepository bean into the MyService class. The service methods save and findById use the repository to perform data access operations.

In summary, the @Repository annotation is used in Spring Boot to indicate that a class is a repository, which provides data access operations for a specific domain object or set of domain objects. Spring Boot automatically creates and manages instances of @Repository-annotated classes as Spring beans.

@Service:

The @Service annotation is used in Spring Framework to indicate that the annotated class is a service, which is a class that provides business logic operations or acts as an intermediary between the controller and the repository layers.

By annotating a class with @Service, Spring Boot will automatically create an instance of that class and manage its lifecycle as a Spring bean. This means that the service can be injected into other Spring-managed components (e.g., other services or controllers) using the @Autowired annotation.

Here's an example of a service class annotated with @Service:

```
@Service
public class MyService {
    @Autowired
    private MyRepository myRepository;
    public void save(MyObject myObject) {
        myRepository.save(myObject);
    }
    public MyObject findById(Long id) {
        return myRepository.findById(id);
    }
    // other service methods
}
```

In the above example, the MyService class is annotated with @Service, which tells Spring Boot that this class is a service. The class provides two service methods, save and findById, which use the MyRepository class to perform data access operations.

The @Autowired annotation is used to inject the MyRepository bean into the MyService class. This allows the service to use the repository to perform data access operations.

The @Service annotation is typically used to annotate classes that provide business logic or coordinate data access operations between multiple repositories. Services can also be used to implement other application-specific operations.

In summary, the @Service annotation is used in Spring Boot to indicate that a class is a service, which provides business logic or acts as an intermediary between the controller and repository layers. Spring Boot automatically creates and manages instances of @Service-annotated classes as Spring beans, which can be injected into other Spring-managed components using the @Autowired annotation.

@Controller:

The @Controller annotation is used in Spring Framework to indicate that the annotated class is a controller, which is a class that handles incoming web requests and returns an HTTP response.

By annotating a class with @Controller, Spring Boot will automatically create an instance of that class and manage its lifecycle as a Spring bean. This means that the controller can be injected into other Spring-managed components (e.g., services) using the @Autowired annotation.

Here's an example of a controller class annotated with @Controller:

```
@Controller
public class MyController {
    @Autowired
    private MyService myService;
    @GetMapping("/my-path")
    public String handleRequest(Model model) {
        MyObject myObject = myService.findById(1L);
        model.addAttribute("myObject", myObject);
        return "my-view";
    }
    // other controller methods
}
```

In the above example, the MyController class is annotated with @Controller, which tells Spring Boot that this class is a controller. The class provides a single controller method, handleRequest, which handles incoming GET requests to the /my-path path.

The @Autowired annotation is used to inject the MyService bean into the MyController class. This allows the controller to use the service to perform business logic operations.

The handleRequest method retrieves a MyObject from the MyService, adds it to the Model object, and returns the name of a view (my-view) that will be rendered to the client.

In summary, the @Controller annotation is used in Spring Boot to indicate that a class is a controller, which handles incoming web requests and returns an HTTP response. Spring Boot automatically creates and manages instances of @Controller-annotated classes as Spring beans, which can be injected into other Spring-managed components using the @Autowired annotation.

What is the Difference between @Component ,@Service ,@Repository and @Controller annotations?

In Spring Framework, the @Component, @Service, @Repository, and @Controller annotations are all used to mark classes as Spring beans, but they are typically used for different types of classes.

- **@Component**: This is a general-purpose annotation that can be used to mark any class as a Spring bean. It is typically used for classes that do not fit into any of the other categories, such as utility classes or classes that perform generic tasks.

- **@Service**: This annotation is used to mark classes that provide business services, such as service classes that perform business logic and interact with repositories to retrieve and persist data.

- **@Repository**: This annotation is used to mark classes that provide data access and storage services, such as classes that interact with databases, file systems, or other data sources.

- **@Controller**: This annotation is used to mark classes that handle incoming HTTP requests, such as classes that define REST controllers or MVC controllers in a web application.

It's worth noting that these annotations are not mutually exclusive, you can use multiple annotations on a single class to provide more context about the class and its function. Also, it's important to note that these annotations are part of the spring-stereotype package, which is a package of stereotypes for annotating classes that play a specific role within your application.

In summary, the main difference between these annotations is their intended use, @Component is a general-purpose annotation while @Service, @Repository, and @Controller are intended to be used in specific roles.

What is the use of component scan?

The @ComponentScan annotation is used in Spring to enable automatic scanning of packages for classes annotated with Spring's stereotype annotations like @Component, @Service, @Repository, and @Controller. It tells Spring to search for and register these classes as beans in the application context.

When the @ComponentScan annotation is used on a class or package, Spring will scan the specified package and its sub-packages for classes annotated with stereotype annotations and register them as beans in the application context. This allows you to easily create and manage the objects of these classes, without having to manually create and configure them.

Here's an example of a class that uses the @ComponentScan annotation:

```
@Configuration
@ComponentScan("com.example.myapp.services")
public class MyConfig {
    // ...
}
```

In this example, the @ComponentScan annotation is used to scan the package "com.example.myapp.services" for classes annotated

with stereotype annotations, and register them as beans in the application context.

It's important to note that the @ComponentScan annotation can be used on a class or package level, if it's used on a class, Spring will only scan the package of the class, if it's used on a package, Spring will scan the package and its sub-packages.

The @ComponentScan annotation is used in conjunction with other annotations like @Configuration and @EnableAutoConfiguration to configure the Spring application.

Also, you can use the basePackages or basePackageClasses attribute to specify the packages to scan, instead of using the default package of the class where the annotation is used.

How does the Spring boot auto-detect feature works?

Spring Boot's auto-detection feature is a mechanism that allows Spring Boot to automatically configure and wire up various components of a Spring application based on the dependencies that are present on the classpath.

When Spring Boot starts up, it automatically scans the classpath of the application for certain annotations, such as @Component, @Service, and @Repository, and registers any classes that are annotated with these annotations as beans in the Spring application context.

Additionally, Spring Boot also automatically detects and configures certain components based on the presence of specific libraries on the classpath. For example, if the application has the spring-data-jpa library on the classpath, Spring Boot will automatically configure and enable JPA-based repositories in the application.

It also supports auto-configuring other components such as security, data source, web, etc based on the libraries present in the classpath.

This feature allows developers to quickly and easily set up a new Spring application without having to manually configure each individual component. It also makes it easy to add or remove features from the application by simply adding or removing the appropriate libraries from the classpath.

However, it's important to note that if you want to customize the auto-configured feature or want to use a different version of a library, you can override the auto-configured settings by providing your own configuration.

What is the difference between @Controller and @RestController annotation?

The @Controller and @RestController annotations are both used in Spring to handle incoming HTTP requests. However, they have some differences in their behavior and use cases.

The @Controller annotation is used to indicate that a class defines a Spring MVC controller. This means that the class is responsible for handling HTTP requests and returning responses. Typically, a @Controller class will have methods annotated with @RequestMapping (or other similar annotations), which define the URL paths and HTTP methods that the controller should handle. The methods in a @Controller class often return a view name or a ModelAndView object, which is then rendered by a view resolver to generate the final response HTML.

On the other hand, the @RestController annotation is a specialized version of @Controller. It combines @Controller and @ResponseBody annotations, which means that all methods in a @RestController class return a response body directly to the client. This response body is usually formatted as JSON or XML, and it can be a simple object, a collection, or any other serializable data. @RestController is typically used to build RESTful web services that expose data APIs.

To summarize:

@Controller is used for building web pages or returning views, where the response can be a view name or a ModelAndView object.

@RestController is used for building RESTful web services, where the response is serialized and returned as a response body.

Both annotations can be used to handle incoming HTTP requests, but the choice between them depends on the use case and the type of response that is required.

What does @ResponseBody Annotations signify?

The @ResponseBody annotation is a Spring annotation used in a controller to indicate that the return value of a method should be bound to the web response body. When a method is annotated with @ResponseBody, Spring will automatically convert the returned value to JSON or XML and write it to the response body.

This annotation can be used on a method level or on a class level, in the latter case all the methods of the controller will return the response body.

Here's an example of a controller method that uses the @ResponseBody annotation:

```
@Controller
public class MyController {
    @ResponseBody
    @RequestMapping("/example")
    public Object example() {
        return new Object();
    }
}
```

In this example, the example() method returns an object, and the @ResponseBody annotation tells Spring to convert the object to JSON or XML and write it to the response body.

It's important to note that, if you use this annotation on a method, Spring will not try to resolve a view for the request and instead it will

directly return the response body.

Also, Spring provide the @RestController annotation as a convenient alternative, which is a combination of @Controller and @ResponseBody, that is typically used for creating RESTful web services.

How to exclude any configuration?

In Spring Boot, you can exclude a configuration by using the exclude attribute of the @EnableAutoConfiguration annotation. This attribute is used to specify a list of classes that should be excluded from the auto-configuration process.

Here's an example of a class that uses the exclude attribute to exclude a specific configuration:

```
@SpringBootApplication(exclude =
{SecurityAutoConfiguration.class})
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

In this example, the SecurityAutoConfiguration class is excluded from the auto-configuration process.

You can also exclude configurations via the application.properties or application.yml file by setting the spring.autoconfigure.exclude property to a comma-separated list of configuration class names or package names.

```
spring.autoconfigure.exclude=org.springframework.boot.autoconfigu
re.security.servlet.SecurityAutoConfiguration
```

It's important to note that, excluding a configuration may cause other configurations to stop working if they depend on the excluded configuration. Therefore, you should carefully analyze the impact of

excluding a configuration and ensure that your application still works as expected.

Also, you can use the spring.autoconfigure.exclude property to exclude any configuration, not only security-related.

How to make the post method idempotent inside spring boot?

In Spring Boot, the HTTP methods (GET, POST, PUT, DELETE, etc.) are mapped to specific Java method handlers using the @RequestMapping annotation. To make a POST method idempotent, you would need to change the behavior of the method so that it can be safely called multiple times without changing the result beyond the initial application of the method.

One way to achieve this is to use the same unique identifier for the resource being created for each subsequent identical request, and return the existing resource if it already exists. This way, the same resource will be created only once and subsequent identical requests will return the existing resource.

Here is an example of how you could implement an idempotent POST method in Spring Boot:

```
@PostMapping("/resource")
public ResponseEntity<Resource> createResource(@RequestBody
Resource resource) {
    Resource existingResource =
resourceService.findByUniqueId(resource.getUniqueId());
    if (existingResource != null) {
        return new ResponseEntity<>(existingResource,
HttpStatus.OK);
    } else {
        Resource createdResource =
resourceService.create(resource);
        return new ResponseEntity<>(createdResource,
HttpStatus.CREATED);
    }
```

```
}
```

In the example above, the createResource method checks if a resource with the same unique identifier already exists before creating a new one, and returns the existing resource if it does.

It's also important to consider the cache-control headers in the response, to ensure that the request is not cached, so that every request to the server is done, and the server can check the idempotence.

Note that this is just one way to make a POST method idempotent, and other methods can be used depending on the requirements of your specific application.

What is spring-boot profile?

In Spring Boot, profiles are used to configure different environments or runtime scenarios of an application. A profile is a set of configurations that can be used to customize an application's behaviour in various environments such as development, production, or testing.

Spring Boot allows you to define different profiles for your application, each with its own set of configuration properties. For example, you can define a "development" profile for local development, a "production" profile for deployment, and a "testing" profile for automated testing.

You can activate a profile by specifying it as a command-line argument or by setting the "spring.profiles.active" property in your application's configuration file. When a profile is activated, Spring Boot will load the corresponding configuration properties and use them to configure the application.

Profiles in Spring Boot are a powerful tool for managing the configuration of your application in different environments. They make it easy to switch between different configurations and ensure that your application is properly configured for each environment.

How to set the properties across different environments like Dev, QA and PROD?

There are several ways to set properties across different environments like Dev, QA, and Prod in Spring Boot. Some of the most common approaches include:

Using profiles: Spring Boot allows you to define different sets of properties for different environments using profiles.

Profiles are activated using the spring.profiles.active or spring.profiles.include properties in the application.properties file. Y

ou can create a separate application-{profile}.properties file for each profile, where {profile} is the name of the profile. For example, you can create an application-dev.properties file for the dev profile and an application-prod.properties file for the prod profile.

Using command line arguments: You can pass environment-specific properties to your Spring Boot application using command line arguments. For example, you can run the application with the --spring.profiles.active=prod option to activate the prod profile.

Using environment variables: Spring Boot can also read properties from environment variables. You can set environment variables for different environments and reference them in the application.properties file.

Using externalized configuration: Spring Boot allows you to externalize configuration by storing properties in a file outside of the application. You can store the properties for different environments in different files and then specify the location of the file when running the application.

Using ConfigServer: You can use Spring Cloud Config Server to manage externalized configuration, it will allow you to store configuration properties in a central place and retrieve them from your application based on the environment.

It's important to note that, the best approach to set properties across different environments will depend on the specific requirements of your application and the infrastructure you have.

Therefore, you should carefully analyze the problem and choose the approach that best suits your needs.

Describe the AOP concept and which annotations are used. How do you define the point cuts?

AOP (Aspect-Oriented Programming) is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns. AOP allows you to define reusable modules of code that can be "woven" into the main program logic at runtime.

AOP is built on top of the traditional Object-Oriented Programming (OOP) model and allows you to define and apply additional behavior to objects and classes in a non-invasive way.

AOP is based on the following concepts:

Aspect: An aspect is a module of code that encapsulates a cross-cutting concern, such as logging, security, or transaction management.

Join point: A join point is a point in the execution of a program, such as the execution of a method or the handling of an exception.

Advice: An advice is the action taken by an aspect at a specific join point. There are several types of advice, such as before, after, and around advice.

Pointcut: A pointcut is a predicate that identifies the join points where advice should be applied.

Weaving: Weaving is the process of applying aspects to the program, which is typically done at runtime.

AOP provides a way to modularize cross-cutting concerns, it allows to separate the core business logic of an application from the aspects that provide additional functionality such as logging, security, and transaction management. This results in a more

modular and maintainable codebase, as well as reducing code duplication.

Spring Framework provides support for AOP through the Spring AOP module, which allows you to easily implement AOP in your Spring-based applications.

What is Spring-transaction management?

Spring provides a comprehensive and consistent transaction management framework that allows you to declaratively manage transactions in your application. The Spring Framework provides a consistent programming model for transaction management that can be used across different transaction APIs, such as JDBC, Hibernate, JPA, and JTA.

The main components of the Spring transaction management framework are:

PlatformTransactionManager: This is the central interface in the Spring transaction management framework, and it is responsible for managing transactions. Spring provides several implementations of this interface for different transaction APIs, such as DataSourceTransactionManager, HibernateTransactionManager, JpaTransactionManager, and JtaTransactionManager.

@Transactional annotation: This annotation is used to mark methods or classes as transactional. When a method or class is marked with this annotation, the Spring Framework will automatically start and commit a transaction before and after the method is called.

TransactionDefinition and TransactionStatus: These interfaces are used to define and manage the properties of a transaction, such as the isolation level, timeout, and rollback rules.
Here's an example of how you can use the @Transactional annotation to mark a method as transactional:

```
@Service
public class MyService {
```

```
    @Transactional
    public void updateData() {
        // Perform database updates
    }
}
```

In this example, the updateData() method is marked with the @Transactional annotation. When this method is called, the Spring Framework will automatically start.

How to use transaction management in spring boot?

Spring Boot provides several options for transaction management, including declarative and programmatic approaches. The most common approach is to use the declarative transaction management provided by Spring's @Transactional annotation.

Here's an example of how you can use the @Transactional annotation to manage transactions in a Spring Boot application:

```
@Service
public class MyService {
    @Autowired
    private MyRepository myRepository;

    @Transactional
    public void updateData(Long id, String data) {
        MyEntity myEntity = myRepository.findById(id);
        myEntity.setData(data);
        myRepository.save(myEntity);
    }
}
```

In the example above, the updateData method is annotated with @Transactional, which tells Spring to start a new transaction before executing the method. When the method completes, Spring will automatically commit the transaction. If an exception is thrown, Spring will automatically roll back the transaction.

You can also configure the @Transactional annotation to specify the transaction isolation level, the propagation behavior, and other properties. Here's an example of how you can configure the @Transactional annotation:
@Transactional(isolation = Isolation.READ_COMMITTED, timeout = 30)

In this example, the isolation level is set to READ_COMMITTED, which means that the current transaction can only read data that has been committed by other transactions. The timeout is set to 30 seconds, which means that the transaction will automatically rollback if it takes longer than 30 seconds to complete.

Another way to manage transactions in Spring Boot is to use the PlatformTransactionManager interface and the TransactionTemplate class. This is called programmatic transaction management. Here's an example of how you can use the TransactionTemplate class to manage transactions:

```
@Service
public class MyService {

    @Autowired
    private MyRepository myRepository;

    @Autowired
    private TransactionTemplate transactionTemplate;

    public void updateData(Long id, String data) {
        transactionTemplate.execute(new
TransactionCallbackWithoutResult() {
            @Override
            protected void
doInTransactionWithoutResult(TransactionStatus status) {
                MyEntity myEntity = myRepository.findById(id);
                myEntity.setData(data);
                myRepository.save(myEntity);
            }
```

```
        });
    }
}
```

In this example, the updateData method uses the TransactionTemplate class to execute a transaction. The code that needs to be executed within a transaction is placed in the doInTransactionWithoutResult method. The TransactionTemplate class will automatically start a new transaction before executing the method, and will automatically commit or roll back the transaction based on the outcome of the method.

It's important to note that when using declarative transaction management, you need to configure a PlatformTransactionManager bean that will be used by the @Transactional annotation, which is automatically done by Spring Boot when using a relational database.

Also, you need to make sure that the transaction management configuration is consistent with the underlying data source and the ORM framework you use.

How to handle a transaction and the isolation levels of the transaction?

In a Spring application, you can handle transactions using the Spring Framework's transaction management abstraction, which is built on top of the Java Transaction API (JTA).

To start a transaction, you can use the @Transactional annotation on the service method or class level. This annotation tells Spring to start a new transaction before the method is executed and to commit or rollback the transaction after the method is executed.

```
@Service
public class ExampleService {

    @Transactional
    public void exampleMethod() {
        // Code that needs to be executed in a transaction
```

```
    }
}
```

Isolation level controls how the data is isolated between different transactions. The isolation levels are:

READ UNCOMMITTED: A transaction can read data that has not been committed by other transactions. This is the lowest level of isolation.

READ COMMITTED: A transaction can only read data that has been committed by other transactions. This is a higher level of isolation.

REPEATABLE READ: A transaction can read data that has been committed by other transactions, but other transactions cannot modify or insert data that the current transaction has read.

SERIALIZABLE: A transaction can read data that has been committed by other transactions, and other transactions cannot modify or insert data that the current transaction has read. Additionally, no two transactions can read or write data at the same time. This is the highest level of isolation.

You can set the isolation level of a transaction using the isolation attribute of the @Transactional annotation. For example, to set the isolation level to READ COMMITTED, you can do the following:

```
@Transactional(isolation = Isolation.READ_COMMITTED)
public void exampleMethod() {
    // Code that needs to be executed in a transaction
}
```

It's important to note that the isolation level that you choose will depend on your application's specific requirements, and that different isolation levels can have different performance impacts.

Also, Spring provides several options to configure the transaction manager and to handle the transaction such as JpaTransactionManager and DataSourceTransactionManager, which

allows you to use different strategies to handle transactions and isolation levels depending on the data access technology you are using in your application.

How to handle security in spring-boot?

Spring Boot provides several options for handling security in a web application. The most common approach is to use the Spring Security framework, which is a powerful and highly customizable authentication and access-control framework.

Here's an example of how you can configure Spring Security in a Spring Boot application:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private UserDetailsService userDetailsService;

    @Autowired
    private PasswordEncoder passwordEncoder;

    @Override
    protected void configure(AuthenticationManagerBuilder auth)
throws Exception {
        auth.userDetailsService(userDetailsService).passwordEncoder(
passwordEncoder);
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers("/admin/**").hasRole("ADMIN")
            .antMatchers("/user/**").hasRole("USER")
            .anyRequest().permitAll()
            .and()
            .formLogin();
```

```
    }
}
```

In this example, the SecurityConfig class is annotated with @Configuration and @EnableWebSecurity to enable Spring Security. The class extends WebSecurityConfigurerAdapter which provides a convenient base class for customizing the security configuration.

The configure(AuthenticationManagerBuilder auth) method is used to configure the authentication manager. In this example, it is configured to use a UserDetailsService and a PasswordEncoder to authenticate users.

The configure(HttpSecurity http) method is used to configure the security for web requests. In this example, it is configured to require a role of "ADMIN" for requests to the "/admin/" path, a role of "USER" for requests to the "/user/" path and permit all other requests. The method also enables form-based authentication.

You can also use other authentication methods such as OAuth2, JWT, etc.

It's important to note that the security configuration will only be effective if the spring-security-web and spring-security-config modules are on the classpath. When using Spring Boot, these modules are included by default in the spring-boot-starter-security starter.

Also, it's important to keep in mind that security is a complex topic and it's important to always keep the system updated and to test the security measures in place, to ensure that the system is secure and to fix any vulnerabilities that may arise.

What is a JWT token and how does spring boot fetch that information?

JWT (JSON Web Token) is a compact, URL-safe means of representing claims to be transferred between two parties. It is often used to authenticate users and exchange information securely. JWT consists of three parts: a header, a payload, and a signature. The

header and payload are Base64Url encoded JSON strings, and the signature is a digital signature that ensures the authenticity of the token.

A typical JWT token looks like this:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

In Spring Boot, you can use the spring-security-jwt library to handle JWT tokens. The library provides a JwtTokenProvider class that you can use to generate and validate JWT tokens.

Here's an example of how you can use the JwtTokenProvider class in a Spring Boot application:

```
@Service
public class JwtTokenProvider {

    @Value("${security.jwt.token.secret-key}")
    private String secretKey;

    @Value("${security.jwt.token.expire-length}")
    private long validityInMilliseconds;

    public String createToken(String username, List<Role> roles) {
        Claims claims = Jwts.claims().setSubject(username);
        claims.put("roles", roles.stream().map(s -> new SimpleGrantedAuthority(s.getAuthority())).filter(Objects::nonNull).collect(Collectors.toList()));
        Date now = new Date();
        Date validity = new Date(now.getTime() + validityInMilliseconds);
        return Jwts.builder()
                .setClaims(claims)
                .setIssuedAt(now)
                .setExpiration(validity)
                .signWith(SignatureAlgorithm.HS256, secretKey)
```

```
                .compact();
    }

    public Authentication getAuthentication(String token) {
        UserDetails userDetails =
this.userDetailsService.loadUserByUsername(getUsername(token));
        return new
UsernamePasswordAuthenticationToken(userDetails, "",
userDetails.getAuthorities());
    }

    public String getUsername(String token) {
        return
Jwts.parser().setSigningKey(secretKey).parseClaimsJws(token).getBo
dy().getSubject();
    }

    public boolean validateToken(String token) {
        try {
            Jws<Claims> claims =
Jwts.parser().setSigningKey(secretKey).parseClaimsJws(token);
            if (claims.getBody().getExpiration().before(new Date())) {
                return false;
            }
            return true;
        } catch (JwtException | IllegalArgumentException e) {
            throw new InvalidJwtAuthenticationException("Expired or
invalid JWT token");
        }
    }
```

How does the JWT token work internally?

(you should know the flow of it, and how the token is used internally).

A JWT (JSON Web Token) is a JSON object that is used to securely transmit information between parties. The information can be

verified and trusted, because it is digitally signed. JWT tokens are often used to authenticate and authorize users in a RESTful API.

JWT tokens consist of three parts:
Header: The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

Payload: The payload contains the claims. Claims are statements about an entity (typically, the user) and additional metadata. There are three types of claims: registered, public, and private claims.

Signature: To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that. For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

```
HMACSHA256(
base64UrlEncode(header) + "." +
base64UrlEncode(payload),
secret)
```

The JWT token is passed to the client, usually in the form of an HTTP-only and secure cookie, which is then passed back to the server with every request. The server can then use the signature to verify the authenticity of the token and the claims.

It's important to note that, JWT tokens are stateless, which means that the server does not need to maintain a session or other state for the client. This makes JWT tokens an attractive option for RESTful APIs and Single Page Applications (SPAs) that need to authenticate and authorize users.

How does Transaction work in Spring boot Microservice, how to achieve that?

(Basically, Microservice are mostly stateless and they don’t maintain the state that is required for transactions.

In Spring, maintaining state in a transaction typically involves storing state in the application's context and then using that state during the transaction.

Here are a few ways to maintain state in a Spring Boot transaction:

ThreadLocal: You can use a ThreadLocal variable to store state in the current thread and then use that state in the transaction. The ThreadLocal variable can be accessed using the TransactionSynchronizationManager.getResource() method.

```
ThreadLocal<MyObject> myObjectThreadLocal = new ThreadLocal<>();
myObjectThreadLocal.set(new MyObject());
TransactionSynchronizationManager.bindResource(MyObject.class.getName(), myObjectThreadLocal);
```

Custom synchronization: You can create a custom synchronization object that implements the org.springframework.transaction.support.TransactionSynchronization interface and stores the state in the beforeCommit() method. This object can then be registered with the transaction using the TransactionSynchronizationManager.registerSynchronization() method.

```
TransactionSynchronizationManager.registerSynchronization(new MyTransactionSynchronization(myObject));
```

Transaction attribute: You can use the TransactionAttribute annotation to specify the transaction behavior on methods. By using this annotation, you can access the current transaction, and keep the state in it.

```
@Transactional
public void myMethod(MyObject myObject){
   TransactionStatus status = transactionManager.getTransaction(new DefaultTransactionDefinition());
}
```

How does Oauth2.0 works?

OAuth 2.0 is an authorization framework that lets users grant applications access to their data on other websites or services, without sharing their password with the application. It acts as a secure middleman between you, the application you want to use, and the website or service that holds your data. Here's how it works in a nutshell:

1. You want to use an application: Let's say you want to connect your Facebook account to a new fitness app.

2. Application redirects you: The fitness app directs you to the Facebook login page.

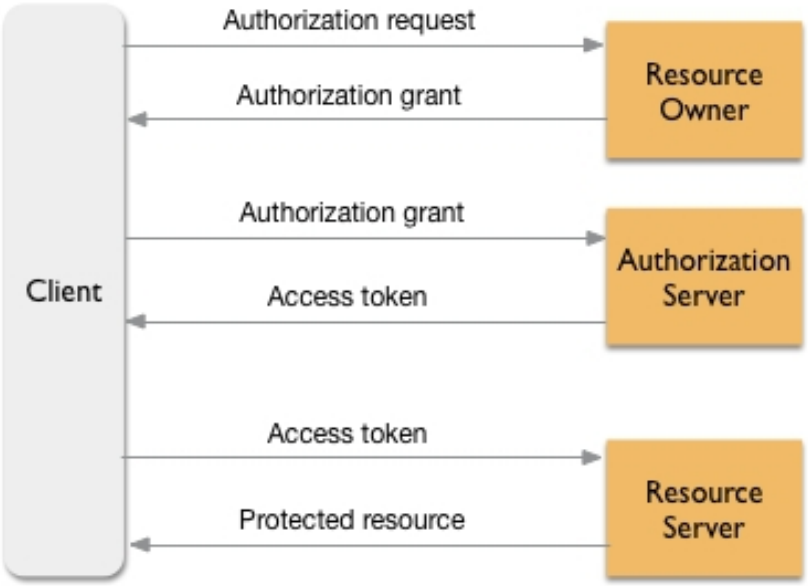
3. You log in to Facebook: You enter your Facebook username and password.

4. Facebook prompts authorization: Facebook asks you if you want to grant the fitness app access to your profile information and activities.

5. You grant or deny access: You review the requested permissions and choose to grant or deny the app access.

6. Facebook sends token to app: If you grant access, Facebook sends the fitness app an access token, which is like a temporary key that allows the app to access your data on Facebook.

7. Application uses token: The fitness app uses the access token to read your Facebook data, such as your profile information and activities, and integrate it into your fitness experience.



Key benefits of OAuth 2.0:

Improved security: You don't share your password with the application, reducing the risk of it being compromised.

Granular control: You can choose what data the application can access.

Privacy-friendly: The application only gets the data you explicitly authorize.

Widely supported: Many popular websites and services use OAuth 2.0.

How to ensure that token has not been tampered with?

To make sure that a JWT token has not been tampered with, you need to check its signature.

The signature of a JWT token is created by taking the encoded header, the encoded payload, a secret, and the algorithm specified in the header, and signing that. The signature is then added to the JWT token as the third part.

When the token is received by the server, the server will decode the token to retrieve the header and payload, and then it will re-create

the signature using the same secret and algorithm specified in the header.

If the re-created signature matches the signature in the token, it means that the token has not been tampered with. But if the re-created signature does not match the signature in the token, it means that the token has been tampered with.

It's important to note that, keeping the secret key safe is important to the security of the JWT token, it should be stored in a secure location and should be rotated regularly.
Also, JWT tokens should be used over an SSL/TLS-secured connection to prevent man-in-the-middle attacks, and you should also validate the claims within the JWT token, such as expiration time, audience, and issuer.

How to use @ControlAdvice for the exception handler?

@ControllerAdvice is a Spring annotation that is used to create a global exception handler for your application. It can be used to apply the same exception handling logic across multiple controllers. It can be used to centralize the handling of exceptions that are thrown by multiple controllers in your application.

When you use @ControllerAdvice on a class, Spring will automatically detect it and use it as a global exception handler. You can then use the @ExceptionHandler annotation on a method inside the @ControllerAdvice class to specify the type of exception that the method should handle.

Here is an example of how you can use @ControllerAdvice and @ExceptionHandler to create a global exception handler:

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<ErrorResponse>
handleCustomException(CustomException ex) {
```

```
        ErrorResponse error = new ErrorResponse("Error",
ex.getMessage());
        return new ResponseEntity<>(error,
HttpStatus.BAD_REQUEST);
    }
}
```

In the example above, the @ControllerAdvice annotation is used to indicate that this class is a global exception handler. The @ExceptionHandler annotation is used to specify that the handleCustomException method should handle CustomException exceptions. The method takes the CustomException object as an argument and returns a ResponseEntity object with a custom error response.

You can also use @ControllerAdvice to handle exceptions thrown by methods in a specific package. You can do this by providing a basePackages or value attribute on the @ControllerAdvice annotation, like this:

```
@ControllerAdvice(basePackages =
"com.example.myapp.controllers")
```

This will only apply the exception handling logic to controllers in the specified package, while all other controllers will not be affected by this.

You can use @ControllerAdvice in combination with @RestControllerAdvice to handle exceptions thrown by rest controllers specifically, or with @ExceptionHandler on a method inside a controller to handle specific exceptions for a specific controller.

How to handle exceptions in Spring boot applications? What are the best practices for doing so?

In Spring Boot, you can handle exceptions in several ways, here are a few options:

Global Exception Handling: You can create a global exception handler class that handles exceptions that are thrown by any controller in the application. You can use the @ControllerAdvice annotation to create a global exception handler and the @ExceptionHandler annotation to specify which exceptions the handler should handle.

```
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(value = Exception.class)
    public ResponseEntity<Object> handleException(Exception ex) {
        // handling logic
        return new ResponseEntity<>(ex.getMessage(),
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Local Exception Handling: You can handle exceptions within the controller methods where they occur. You can use the try-catch block or the @ExceptionHandler annotation to handle exceptions locally.

```
@RestController
public class EmployeeController {
    @GetMapping("/employees/{id}")
    public Employee getEmployeeById(@PathVariable Long id) {
        try {
            return employeeService.getEmployeeById(id);
        } catch (EmployeeNotFoundException ex) {
            throw new
ResponseStatusException(HttpStatus.NOT_FOUND,
ex.getMessage());
        }
    }
```

```
}
```

Custom Exception Handling: You can create custom exceptions that extend the built-in exceptions, and then handle them in the global or local exception handlers.

```
@ResponseStatus(value = HttpStatus.NOT_FOUND, reason =
"Employee not found")
public class EmployeeNotFoundException extends RuntimeException
{
}
```

How to use a custom exception handler in Spring Boot?

In Spring Boot, you can use a custom exception handler to handle specific exceptions that may be thrown by your application. To create a custom exception handler, you will need to create a class that implements the HandlerExceptionResolver interface or extends the AbstractHandlerExceptionResolver class and override the resolveException method.

Here's an example of a custom exception handler class:

```
@ControllerAdvice
public class CustomExceptionHandler extends
ResponseEntityExceptionHandler {

    @ExceptionHandler(CustomException.class)
    public final ResponseEntity<ErrorResponse>
handleCustomException(CustomException ex, WebRequest request)
{
        ErrorResponse error = new ErrorResponse(ex.getMessage(),
request.getDescription(false));
        return new ResponseEntity<>(error,
HttpStatus.BAD_REQUEST);
    }
}
```

The @ControllerAdvice annotation is used to indicate that the class is a global exception handler. The @ExceptionHandler annotation is

used to specify the type of exception that this method should handle. The CustomException in the example above is a custom exception that you have defined in your application.

The handleCustomException method takes the CustomException object and the WebRequest object as arguments, and returns a ResponseEntity object with a custom error response.

You can also create a class that extends ResponseEntityExceptionHandler which is a convenient base class for handling exceptions and providing responses.

Then in your configuration class, you should register your custom exception handler with Spring by using the @Autowired annotation.

```
@Configuration
public class RestConfiguration {
    @Autowired
    private CustomExceptionHandler customExceptionHandler;

    @Bean
    public HandlerExceptionResolver handlerExceptionResolver() {
        return customExceptionHandler;
    }
}
```

With this, your custom exception handler will be registered and used by Spring to handle exceptions in your application.

Write an endpoint in spring boot for getting and saving employees with syntax.

In Spring Boot, you can create endpoints for getting and saving employees using controllers and services. Here's an example of how you could create these endpoints:

EmployeeController: This class defines the REST endpoints for getting and saving employees.

```
@RestController
@RequestMapping("/employees")
public class EmployeeController {

    private final EmployeeService employeeService;

    public EmployeeController(EmployeeService employeeService) {
        this.employeeService = employeeService;
    }

    @GetMapping
    public List<Employee> getAllEmployees() {
        return employeeService.getAllEmployees();
    }

    @GetMapping("/{id}")
    public Employee getEmployeeById(@PathVariable Long id) {
        return employeeService.getEmployeeById(id);
    }

    @PostMapping
    public Employee saveEmployee(@RequestBody Employee
employee) {
        return employeeService.saveEmployee(employee);
    }
}
```

The @RestController annotation is used to indicate that the class is a controller and will handle HTTP requests. The @RequestMapping annotation is used to map the endpoint to a specific URL.

The @GetMapping and @PostMapping annotations are used to indicate that the methods handle GET and POST requests, respectively. The @PathVariable annotation is used to extract a path variable from the URL, and the @RequestBody annotation is used to extract the body of a POST request.

EmployeeService: This class defines the business logic for getting and saving employees.

```
@Service
public class EmployeeService {

    private final EmployeeRepository employeeRepository;

    public EmployeeService(EmployeeRepository
employeeRepository) {
        this.employeeRepository = employeeRepository;
}
}
```

CHAPTER 8: MICROSERVICE

What is Microservice?

Microservice, aka Microservice Architecture, is an architectural style that structures an application as a collection of small autonomous services, modelled around a business domain.

In Microservice architecture, each service is self-contained and implements a single business capability.

Features of Microservice:

Decoupling - Services within a system are largely decoupled, so the application as a whole can be easily built, altered, and scaled.

Componentization - Microservice are treated as independent components that can be easily replaced and upgraded.

Business Capabilities - Microservice are very simple and focus on a single capability.

Autonomy - Developers and teams can work independently of each other, thus increasing speed.

Continuous Delivery - Allows frequent releases of software through systematic automation of software creation, testing, and approval.

Responsibility - Microservice do not focus on applications as projects. Instead, they treat applications as products for which they are responsible.

Decentralized Governance - The focus is on using the right tool for the right job. That means there is no standardized pattern or any technology pattern. Developers have the freedom to choose the best useful tools to solve their problems.

Agility - Microservice support agile development. Any new feature can be quickly developed and discarded again.

What is the advantage of Microservice over monolithic architecture?

Challenges of monolithic architecture

Inflexible - Monolithic applications cannot be built using different technologies.

Unreliable - If even one feature of the system does not work, then the entire system does not work.

Unscalable - Applications cannot be scaled easily since each time the application needs to be updated, the complete system has to be rebuilt.

Blocks Continuous Development - Many features of an application cannot be built and deployed at the same time.

Slow Development - Development in monolithic applications takes a lot of time to be built since each and every feature has to be built one after the other.

Not Fit for Complex Applications - Features of complex applications have tightly coupled dependencies.

What is the disadvantage of Microservice architecture

Complexity: The overall architecture can become complex, especially when dealing with the coordination of many small services.

Increased operational overhead: Managing and deploying many small microservices can increase operational overhead, such as monitoring and testing.

Network latency: Communication between microservices can add network latency and reduce performance if not properly optimized.

Inter-service dependencies: Inter-service dependencies can become complex and difficult to manage, especially when dealing with many small services.

Difficulties in testing: Testing can become more complex and time-consuming with a large number of microservices.

Under what circumstances is the Microservice architecture are not preferable to you?

While microservices have several benefits, there are some circumstances in which they may not be the best option. Here are some situations in which microservices might not be preferable:

Small projects: Microservices architecture is generally recommended for large and complex systems. If your project is small and simple, then it might not be worth the added complexity of implementing a microservices architecture.

Limited resources: Implementing a microservices architecture can require a significant investment of resources, including time, money, and expertise. If you don't have access to the necessary resources, it might not be feasible to use a microservices architecture.

High latency: Since microservices communicate with each other over a network, there can be latency issues. If low latency is critical to your application, then a monolithic architecture might be a better choice.

Interdependent services: If your services are heavily interdependent, then implementing a microservices architecture might not provide significant benefits. It could lead to additional complexity and overhead without delivering much value.

Development team experience: A microservices architecture requires specialized knowledge and experience to design and implement effectively. If your development team is not experienced in microservices, it might not be the best option.

What are the design principles of Microservice?

Modularity: Services should be self-contained and should have a single, well-defined purpose.
Scalability: Services should be able to scale independently to handle increasing load.
Decentralization: The system should be decentralized, allowing for loosely-coupled services.

High Availability: Services should be designed to be highly available to ensure system reliability.
Resilience: Services should be designed to handle failures gracefully.
Data Management: Services should manage their own data and should not share a common database.
Statelessness: Services should be stateless to allow for easy scaling and caching.
Independent Deployment: Services should be deployable independently of other services.
Observability: The system should have built-in monitoring and logging capabilities to allow for visibility into system behaviour.
Automation: Deployment, testing, and scaling should be automated as much as possible.

Do you know the 12-factor methodology to build a Microservice?

The 12-factor methodology is a set of guidelines for building scalable, maintainable, and easily deployable software-as-a-service (SaaS) applications. It's particularly relevant for building Microservice -based applications, since Microservice represent a distributed system and can benefit from the principles of the 12-factor methodology. Here are the 12 factors:

Codebase: One codebase per app, with multiple deploys.

Dependencies: Explicitly declare and isolate dependencies.

Config: Store config in the environment.

Backing services: Treat backing services as attached resources.

Build, release, run: Strictly separate build and run stages.

Processes: Execute the app as one or more stateless processes.

Port binding: Export services via port binding.

Concurrency: Scale out via the process model.

Disposability: Maximize robustness with fast start-up and graceful shutdown.

Dev/prod parity: Keep development, staging, and production as similar as possible.

Logs: Treat logs as event streams.

Admin processes: Run admin/management tasks as one-off processes.

By following the principles of the 12-factor methodology, you can build microservices that are scalable, maintainable, and easily deployable. This can help you to deliver high-quality applications that meet the needs of your customers and users.

Why are Microservice stateless?

Microservice are designed to be stateless for several reasons:

Scalability: Stateless Microservice can be easily scaled horizontally by adding more instances, without having to worry about preserving state across instances. This makes it easier to handle increased traffic and load balancing.

Resilience: Stateless microservices can fail without affecting the system as a whole, since they don't rely on stored state. When a stateless microservice fails, another instance can simply take its place, preserving the overall health of the system.

Portability: Stateless microservices can be deployed to any environment without having to worry about preserving state. This makes it easier to move microservices between environments, such as between development and production, or between data centres.

Simplicity: Stateless microservices are simpler to implement, maintain, and test, since they don't have to manage state. This makes it easier to build, deploy, and manage a system made up of many microservices.

In summary, stateless microservices provide greater scalability, resilience, portability, and simplicity compared to stateful microservices. However, there may be cases where stateful

microservices are necessary, for example, when dealing with user sessions or long-running transactions. In these cases, it's important to carefully manage state and ensure that stateful microservices are still scalable, resilient, and portable.

What is the advantage of Microservice using Spring Boot Application + Spring Cloud?

Advantages of Microservice over Spring Boot Application + Spring Cloud:

Improved Scalability: Microservice architecture allows for better scalability by allowing services to be developed, deployed and scaled independently.
Faster Time-to-Market: By breaking down a monolithic application into smaller, self-contained services, development teams can work in parallel and iterate more quickly.
Resilience: Microservice provide improved resilience by allowing services to fail independently without affecting the entire system.
Better Resource Utilization: Microservice allow for better resource utilization as services can be deployed on the best-suited infrastructure.
Increased Flexibility: Microservice architecture provides increased flexibility as new services can be added or existing services can be updated without affecting the entire system.
Improved Maintainability: Microservice provide improved maintainability by reducing the complexity of the overall system and making it easier to identify and fix problems.
Technology Heterogeneity: Microservice architecture enables technology heterogeneity, allowing for the use of different technologies for different services.
Improved Team Collaboration: Microservice architecture can improve team collaboration by breaking down a monolithic application into smaller, self-contained services that can be developed by smaller, cross-functional teams.

How to share a database with multiple microservices?

When implementing a microservices architecture, sharing a database between multiple microservices can be a common approach. Here are some guidelines on how to do it effectively:

Design the database schema with the microservices in mind: To ensure that the database can be effectively shared between multiple microservices, it's important to design the schema with the microservices architecture in mind. This means creating tables and columns that are specific to the microservices that will be using them, and avoiding dependencies that would create coupling between microservices.

Use a database migration tool: To manage the changes in the database schema over time, it's important to use a database migration tool. This will allow you to apply changes to the database schema in a controlled way, ensuring that all microservices are compatible with the new schema.

Implement a database access layer: To abstract the database access logic from the microservices, it's recommended to implement a database access layer. This layer should handle all database operations and provide a simple interface for the microservices to interact with.

Use a shared database instance: To ensure that all microservices are accessing the same data, it's important to use a shared database instance. This can be a single database server or a cluster of servers, depending on the requirements of the system.

Implement data isolation: To ensure that one microservice does not unintentionally modify data that belongs to another microservice, it's recommended to implement data isolation. This can be done by using different database schemas or database instances for each microservice, or by using row-level security features to restrict access to specific data.

Monitor database performance: When sharing a database between multiple microservices, it's important to monitor the performance of the database. This can be done by monitoring query execution

times, database locks, and resource usage to ensure that the database is performing optimally and not causing performance issues for the microservices.

Regulatory requirements: Some industries have strict regulations around data privacy and security. Implementing a microservices architecture might not be feasible if it does not meet the regulatory requirements of your industry.

Do you know Distributed tracing? What is its us?

Yes, distributed tracing is a technique used in distributed systems to track and monitor the flow of a request or transaction across multiple services or microservices. It is a way to gain visibility into complex, distributed architectures and to diagnose performance and reliability issues.

Distributed tracing involves instrumenting each service in a distributed system to generate and propagate a unique identifier for each incoming request or transaction. This identifier is typically called a "trace ID", and it is used to tie together all the individual spans or segments that make up a single transaction.

As a transaction flows through the different services and components of a distributed system, each component records information about its part of the transaction in a "span" or "segment". Each span includes information such as timing data, error codes, and other relevant metadata. The spans are then collected and correlated across all the services that participated in the transaction, creating a trace of the entire transaction across the distributed system.

Distributed tracing is used to understand the performance of a distributed system, to diagnose issues with specific transactions, and to identify the root causes of failures or errors. By correlating the spans across different services, it is possible to visualize the entire flow of a request and to pinpoint bottlenecks, errors, or other issues. Distributed tracing tools typically provide visualization tools, alerting

mechanisms, and other features to help operators and developers understand the health and performance of their distributed systems.

Distributed tracing can be implemented using open-source tools like Jaeger, Zipkin, and OpenTelemetry, or via commercial products from cloud providers and third-party vendors.

How distributed tracing is done in Microservice?

Distributed tracing is a technique used to track the flow of a request as it travels across multiple Microservice in a distributed system. It helps to understand the performance and behaviour of a Microservice architecture by providing visibility into the interactions between Microservice.

Here are the steps involved in distributed tracing in Microservice:

Instrumentation: Each Microservice is instrumented with tracing information, typically by adding trace headers to the requests sent between Microservice. This information can include a unique trace identifier, the name of the Microservice, and timing information.

Propagation: The trace information is propagated along with each request as it flows between Microservice. This allows the trace information to be captured and recorded by each Microservice that handles the request.

Collection: The trace information is collected by a tracing system, which can be a standalone component or integrated into a log management or monitoring tool.

Analysis: The collected trace information is analysed to gain insights into the performance and behaviour of the Microservice. This can include identifying bottlenecks, tracking the flow of requests through the system, and measuring the response time of individual Microservice.

Visualization: The results of the analysis can be visualized in a way that helps to understand the relationships between microservices and identify any issues or inefficiencies.

Distributed tracing can be performed using dedicated tracing tools, such as Zipkin, Jaeger, or Appdash, or integrated into existing monitoring and log management tools, such as ELK or Datadog.

How to connect internal and external services in microservices.

There are various designs that can be used to connect internal and external services in a microservice architecture, depending on the specific requirements of the system. Some common patterns are:

API Gateway: An API gateway acts as a single entry point for all external requests, forwarding them to the appropriate microservice for handling. The gateway can also perform tasks such as authentication, rate limiting, and caching.

Service Discovery: In this pattern, Microservice register themselves with a central registry, and clients use the registry to find the location of the Microservice they need to interact with. This can be done using a technology such as DNS or a dedicated service discovery tool like Eureka or Consul.

Load Balancer: A load balancer distributes incoming requests to multiple instances of a Microservice, improving reliability and scalability. Load balancing can be performed by a dedicated load balancing tool, or it can be built into the API gateway.

Circuit Breaker: A circuit breaker is a pattern that helps prevent cascading failures in a Microservice architecture by adding resilience to communication between Microservice. The circuit breaker acts as a proxy between the client and the Microservice, monitoring the health of the Microservice and failing over to a backup instance if necessary.

Event-Driven Architecture: In this pattern, Microservice communicate with each other using events, rather than direct requests. This can help decouple Microservice and reduce the coupling between them.

These patterns can be combined and customized as needed to create a suitable solution for your specific use case.

Which Microservice design pattern have you used so far and why?

Service Registry: A Service Registry is a centralized directory that microservices can use to locate each other's endpoints. It helps with service discovery, load balancing, and failover, and makes it easier to manage a distributed system. Examples of service registries include Consul and Eureka.

Circuit Breaker: A Circuit Breaker is a pattern that can help to prevent cascading failures in a distributed system. It monitors requests to a service and can "trip" the circuit when too many requests fail or when the service becomes unresponsive. When the circuit is tripped, subsequent requests can be immediately rejected or diverted to a fallback mechanism.

API Gateway: An API Gateway is a centralized entry point for a set of microservices. It can handle authentication, rate limiting, request routing, and other cross-cutting concerns. The API Gateway helps to simplify the client-side by providing a unified interface to a set of Microservice.

Event-Driven Architecture: An Event-Driven Architecture is a pattern that involves using events to communicate between microservices. Instead of tightly coupling services through synchronous REST APIs, events can be used to decouple services and to provide more flexibility and scalability.

CQRS: CQRS (Command Query Responsibility Segregation) is a pattern that involves separating the responsibility of handling read and write operations into separate services. It can help to simplify the design of a system by separating concerns and reducing complexity. It can also help to improve performance by allowing read and write operations to be optimized separately.

Each of these patterns can help to improve the scalability, reliability, and maintainability of a microservice architecture. The choice of

pattern will depend on the specific requirements and constraints of the system being developed.

Which design patterns are used for database design in Microservice?

Common design patterns used for database design in Microservice are:

Database per Service: Each service has its own database, allowing for a high degree of independence and autonomy.
Shared Database: A shared database is used by multiple services to store data that is commonly used across the system.
Event Sourcing: The state of the system is stored as a series of events, allowing for better scalability and fault tolerance.
Command Query Responsibility Segregation (CQRS): Queries and commands are separated, allowing for improved scalability and performance.
Saga: A long-running transaction is broken down into smaller, autonomous transactions that can be executed by different services.
Materialized View: A pre-computed view of data is used to provide fast access to commonly used data.
API Composition: APIs are composed to provide a unified view of data from multiple services.
Read Replicas: Read replicas are used to offload read requests from the primary database, improving performance and scalability.

What is the SAGA Microservice pattern?

SAGA is a Microservice Pattern:

The Saga pattern is a design pattern used in distributed systems to manage long-running transactions involving multiple services. It is a way to ensure that these transactions maintain data consistency and integrity even in the face of failures and errors.

In the Saga pattern, a transaction is divided into a series of smaller, more granular sub-transactions, also known as "saga steps", each of which can be executed independently by a single service. Each sub-

transaction updates its own local data and sends messages to other services to trigger their corresponding sub-transactions.

If a sub-transaction fails, the Saga pattern uses a compensating action to undo the changes made by the previous steps and maintain consistency across the entire transaction. This can be thought of as a kind of "rollback" mechanism for distributed transactions.

The Saga pattern can be implemented in different ways depending on the specific system and requirements. One common approach is to use a choreography-based saga, where each service is responsible for executing its own sub-transactions and communicating with other services directly. Another approach is to use an orchestration-based saga, where a central coordinator service is responsible for executing and coordinating the different sub-transactions.

The Saga pattern can be a powerful tool for managing long-running distributed transactions, but it also comes with some trade-offs. It can be more complex to implement than simpler transaction models, and it can require careful design and testing to ensure that it can handle all possible failure scenarios. However, in complex distributed systems where data consistency is critical, the Saga pattern can be a valuable tool for maintaining data integrity and avoiding data inconsistencies.

Explain the CQRS concept?

Command Query Responsibility Segregation (CQRS) is a design pattern that separates read and write operations in a system. This means that the operations that retrieve data (queries) are separated from the operations that update data (commands).

The main idea behind CQRS is to improve performance by allowing the read and write operations to be optimized and scaled independently. The read operations can be optimized for read-heavy workloads, while the write operations can be optimized for write-heavy workloads.

By separating read and write operations, CQRS also provides a higher degree of isolation and can simplify the implementation of complex business logic. Additionally, CQRS can improve the ability to handle concurrent access to data, allowing for better scalability and fault tolerance.

CQRS is often used in Microservice architectures and event-driven systems, where different parts of the system can have different requirements for read and write operations.

Which Microservice pattern will you use for read-heavy and write-heavy applications?

For read-heavy applications, you may use the CQRS (Command Query Responsibility Segregation) pattern. This pattern separates the responsibilities of reading and writing data into separate microservices. The write-side microservice is responsible for handling updates and writes to the database, while the read-side microservice is responsible for serving up data for queries.

By separating these responsibilities, you can scale each microservice independently based on the needs of your application. For example, you can scale up the read-side microservice to handle increased read traffic, or scale down the write-side microservice to handle lower write traffic.

For write-heavy applications, you may use the Event Sourcing pattern. This pattern involves storing every change to the state of your application as an event. Each microservice can subscribe to these events and update its own state accordingly. This allows multiple microservices to collaborate and ensures that all changes are captured and recorded.

In both cases, you can also consider using a message queue to handle asynchronous communication between the microservices, and a cache to improve performance for read-heavy applications.

Explain the Choreography pattern in Microservice?

Choreography in Microservice refers to the way in which services communicate and coordinate with each other without the need for a central authority or central point of control. Instead, each service is responsible for handling its own behaviour and communicating with other services as needed.

In a choreographed system, services exchange messages or events to coordinate their behaviour. For example, one service might send an event to another service indicating that a certain action has taken place, and the receiving service can respond as necessary.

The main advantage of choreography is that it provides a more decentralized and flexible system, where services can evolve and change independently. This can lead to improved scalability, as services can be added or removed without affecting the entire system. Additionally, choreography can improve reliability, as a failure in one service does not affect the rest of the system.

Choreography is often used in event-driven systems and is an alternative to the centralized coordination provided by a central authority, such as a service registry or a centralized API gateway.

What are the types of fault tolerance mechanisms in Spring Microservice?

Spring framework provides several mechanisms for implementing fault tolerance in Microservice:

Circuit Breaker: The Spring Cloud Netflix Hystrix library provides a circuit breaker implementation. The circuit breaker acts as a proxy between a Microservice and its dependencies. It monitors the health of the dependencies and opens the circuit if a certain number of failures occur within a defined time window. This prevents further failures and allows the Microservice to degrade gracefully.

Load Balancing: Spring Cloud Netflix Ribbon provides client-side load balancing. It allows a Microservice to distribute incoming

requests across multiple instances of a dependent Microservice. This helps to increase availability and resilience, since the Microservice can still function even if one of its dependencies fails.

Retry: Spring Retry provides declarative control of retry behaviour. It allows a Microservice to automatically retry a failed request to a dependent Microservice, with configurable parameters such as maximum number of retries and back off policies.

Timeouts: Spring Cloud Hystrix provides timeout functionality for dependent Microservice. It allows a Microservice to specify a timeout for a request, and fail fast if the dependent Microservice does not respond within the defined timeout period.

Monitoring and Management: Spring Boot provides built-in support for monitoring and management of Microservice. This includes monitoring of application health, metrics, and logs, as well as management of the application lifecycle, such as starting and stopping the application.

By using these fault tolerance mechanisms, you can build robust and resilient Microservice with Spring framework. This can help you to deliver high-quality applications that are able to withstand failures and handle high levels of traffic and load.

What is circuit breaker pattern? What are examples of it?

The Circuit Breaker pattern is a design pattern used to prevent failures in a distributed system by adding a layer of protection between the calling service and the called service. The Circuit Breaker acts as a switch that can be opened or closed based on the health of the called service.

Here's a simple example of how you can implement the Circuit Breaker pattern in the Spring framework:

Create a Circuit Breaker class that implements the HystrixCircuitBreaker interface:

@Service

```
public class MyCircuitBreaker implements HystrixCircuitBreaker {
  @Autowired
  private MyService myService;
  @HystrixCommand(fallbackMethod = "defaultResponse")
  public String callService() {
    return myService.doSomething();
  }
  public String defaultResponse() {
    return "Default Response";
  }
}
```

In this example, we use the @HystrixCommand annotation to wrap the call to the MyService class in a circuit breaker. If the call to MyService fails, the defaultResponse method will be called as a fallback.

Configure the Hystrix Circuit Breaker in your Spring configuration:

```
@Configuration
@EnableCircuitBreaker
public class CircuitBreakerConfig {
  @Bean
  public MyCircuitBreaker myCircuitBreaker() {
    return new MyCircuitBreaker();
  }
}
```

In this example, we enable the Circuit Breaker pattern using the @EnableCircuitBreaker annotation.

When the callService method is called, the Hystrix Circuit Breaker will monitor the health of the MyService class and, if necessary, open the circuit and fall back to the defaultResponse method. This helps

to prevent failures from propagating throughout the system and causing widespread disruption.

Explain the annotations used to implement circuit breaker in spring boot?

In Spring Boot, circuit breakers can be implemented using the spring-cloud-starter-circuitbreaker library, which provides support for several different circuit breaker implementations, including Hystrix.

To use the circuit breaker in Spring Boot, you can use the following annotations:

@HystrixCommand: This annotation is used to wrap a method with a circuit breaker. When the circuit breaker trips, the method will return a fallback response instead of the normal response.

@HystrixProperty: This annotation is used to configure the properties of the circuit breaker, such as the timeout and the number of failures before the circuit breaker trips.

Here is an example of how to use the @HystrixCommand and @HystrixProperty annotations to implement a circuit breaker in Spring Boot:

```
@Service
public class MyService {
   @HystrixCommand(fallbackMethod = "fallback",
commandProperties = {
      @HystrixProperty(name =
"execution.isolation.thread.timeoutInMilliseconds", value = "2000"),
      @HystrixProperty(name =
"circuitBreaker.requestVolumeThreshold", value = "5"),
      @HystrixProperty(name =
"circuitBreaker.errorThresholdPercentage", value = "50"),
      @HystrixProperty(name =
"circuitBreaker.sleepWindowInMilliseconds", value = "5000")
   })
   public String callDependency() {
```

```
        // Call the dependent service
    }

    public String fallback() {
        // Return a fallback response
    }
}
```

Which library have you used to implement circuit breaker in spring boot?

There are several libraries and frameworks available for implementing circuit breakers in various programming languages. Some of the popular ones include:

Hystrix (Java): A library developed by Netflix, it is one of the most popular circuit breaker implementations for Java.

Resilience4j (Java): An lightweight, easy-to-use library for fault tolerance in Java.

Polly (.NET): A library for .NET that provides support for circuit breakers, timeouts, and retries.

Ruby Circuit Breaker (Ruby): A library for Ruby that implements the circuit breaker pattern.

Go-Hystrix (Go): A Go implementation of the Hystrix library, providing circuit breaker functionality for Go applications.

Elixir Circuit Breaker (Elixir): An implementation of the circuit breaker pattern for Elixir applications.

These libraries provide a convenient and easy-to-use way to implement circuit breakers in your applications, allowing you to improve the resilience and fault tolerance of your system.

How to call methods Asynchronously, in the spring framework how can we do that?

In the Spring framework, you can call methods asynchronously using the @Async annotation. The @Async annotation marks a method as being executed asynchronously by a task executor.

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

```
@Service
public class MyService {
  @Async
  public CompletableFuture<String> doSomethingAsync() {
    // Perform some task asynchronously
    return CompletableFuture.completedFuture("Task completed");
  }
}
```

In order to use the @Async annotation, you need to configure a task executor in your Spring configuration. Here's an example:

```
@Configuration
@EnableAsync
public class AsyncConfig {
  @Bean(name = "taskExecutor")
  public Executor taskExecutor() {
    return Executors.newFixedThreadPool(10);
  }
}
```

In this example, we create a task executor using a FixedThreadPool with a pool size of 10. When the doSomethingAsync method is called, it will be executed asynchronously by the task executor. The CompletableFuture returned by the method can be used to retrieve the result of the asynchronous task when it becomes available.

How to call another microservice asynchronously?

To call another microservice asynchronously, you can use a message queue. The basic flow is:
One microservice produces a message and sends it to a message queue.

Another microservice consumes the message from the queue and performs the desired action.

The message queue acts as a buffer between the two microservices, allowing them to communicate asynchronously. This approach has several benefits:

Loose coupling: Microservices can communicate with each other without having to know the details of the other microservice's implementation.

Scalability: The message queue can be scaled independently of the microservices, allowing for improved scalability.

Resilience: If one microservice fails, the message queue can hold the messages until the other microservice is able to process them.

There are several message queues available, such as RabbitMQ, Apache Kafka, and ActiveMQ, and you can choose the one that best fits your needs. To use a message queue, you need to write code to produce and consume messages from the queue.

How to communicate between two microservices?

There are several ways to communicate between two microservices:
HTTP/REST: The most common way of communication between microservices is through REST APIs over HTTP. Microservices can expose a RESTful API that other microservices can use to request data or perform actions.

Message Queueing: Microservices can communicate with each other asynchronously through a message queue such as RabbitMQ or Apache Kafka. Microservices can publish messages to the queue

and other microservices can subscribe to the queue and receive messages.

Event-Driven Architecture: Microservices can use event-driven architecture to communicate with each other. In this approach, microservices can publish events to a centralized event bus and other microservices can subscribe to these events and react to them.

gRPC: gRPC is a high-performance, open-source framework for building microservices. It uses a binary format for communication between microservices and can be used for both synchronous and asynchronous communication.

Regardless of the communication method chosen, it's important to ensure that communication between microservices is secure and that only authorized microservices are able to communicate with each other.

How to restrict the Microservice from calling the other Microservice?

(Let's say there are A, B, C, D, and E-services and I want to restrict A form calling C, D, and E. how will you do that?)

One way to restrict a microservice from calling another microservice is to use API gateway to enforce the access control rules. The API gateway can be configured to only allow authorized microservices to make API requests to other microservices. Additionally, you can use authentication and authorization mechanisms such as OAuth, JWT, or API keys to secure the API endpoints and ensure that only authorized microservices are able to access them.

Another way to restrict microservice access is to use network segmentation and firewall rules to restrict network access between microservices. This can help prevent unauthorized microservices from accessing other microservices on the network.

Finally, you can also use code-level access control in each microservice, for example by using role-based access control (RBAC), to control which microservices can call which APIs and what actions they can perform.

How to save your username password in the spring boot-based Microservice application?

(What is the best practice)

In a Spring Boot-based microservices application, it is generally considered best practice to store sensitive information such as username and password in a secure location, separate from the application code. There are several ways to achieve this, here are a few options:

Environment Variables: You can store the username and password as environment variables on the machine where the microservice is running. This approach is considered secure as environment variables are typically stored outside of the file system and are not accessible to unauthorized users.

Configuration files: You can store the username and password in a separate configuration file that is encrypted and stored in a secure location. The configuration file can then be accessed by the microservice at runtime.

Hashicorp Vault: You can use Hashicorp Vault to securely store, manage, and access sensitive data such as username and password. Vault provides a centralized and secure way to store and access secrets.

External Service: You can also use external services like AWS Secrets Manager, Azure KeyVault, Google Cloud Key Management Service (KMS) to store and manage your credentials.

It's important to note that, whatever the approach you choose, you should make sure that the sensitive information is encrypted and stored in a secure location that is only accessible to authorized users and services. Also, you should use the best practice of never hardcoding the sensitive information in the code, this could help to avoid security issues, and make it easy to rotate and manage your credentials.

CHAPTER 9: MEMORY MANAGEMENT IN JAVA

What is Memory management in Java?

Memory management in Java is handled by the Java Virtual Machine (JVM), which automatically manages the allocation and deallocation of memory for objects. The JVM uses a technique called garbage collection to periodically scan the heap (the area of memory where objects are stored) and identify objects that are no longer being used by the application. These objects are then removed from memory, freeing up space for new objects.

Java has several different garbage collectors that can be used, each with its own set of features and trade-offs. The default garbage collector in most JVMs is called the "Serial GC", which is a basic garbage collector that runs serially on a single thread. Other garbage collectors include the "Parallel GC", which uses multiple threads to speed up garbage collection, and the "G1 GC", which is designed for large heap sizes and can help reduce the frequency of long pauses caused by garbage collection.

Java also provides a way for developers to explicitly manage memory through the use of manual memory management techniques such as the new keyword and the finalize() method. However, it's not recommended to use manual memory management in Java, as it can lead to memory leaks and other issues. The JVM's garbage collector does a much better job of managing memory and it's better to let it do its job.

In addition, Java also provides several memory managements related flags that can be used to configure the JVM memory usage and performance. Such as, -Xmx and -Xms to configure the maximum and minimum heap size, -XX:NewRatio to configure the

size of young and old generation, -XX:+UseG1GC to configure the G1 Garbage collector.

It's worth noting that, effective Memory management requires a deep understanding of JVM internals, Garbage Collection and Java Memory Model. Incorrect usage of memory management flags can lead to poor performance and stability issues.

Overall, these changes to the Java Memory Model in Java 8 help to improve the performance and safety of multi-threaded Java applications.

What is Meta-Space in java ? What benefits does it offer?

In Java 8 and later, the Meta-Space is a memory space that is used to store class metadata. It is separate from the Java heap, which is used to store objects and other data.

The MetaSpace is used to store information about classes and interfaces, such as their methods, fields, and annotations. This information is used by the Java Virtual Machine (JVM) to dynamically load and link classes at runtime.

The MetaSpace is allocated a fixed amount of memory, which is specified using the -XX:MaxMetaspaceSize command-line option. When the MetaSpace is full, the JVM will attempt to free up space by unloading classes that are no longer in use. If this is not sufficient, the JVM will throw a java.lang.OutOfMemoryError: Metaspace error.

It is important to monitor the MetaSpace usage and adjust the -XX:MaxMetaspaceSize as necessary to ensure that there is enough space for the classes that are needed at runtime.

It's worth noting that, starting from Java 11, the MetaSpace has been replaced by the "Class Data Sharing" (CDS) feature. CDS allows to share read-only class data across multiple JVMs, reducing the memory footprint and the time required to start the JVM.

What is memory leak in java? how to rectify that in java?

A memory leak in Java occurs when an application continues to hold onto objects that are no longer needed, preventing the garbage collector from freeing up memory. This can lead to the application using more and more memory over time, eventually causing it to crash or become unresponsive. Memory leaks can be caused by a variety of issues, such as incorrect object references, failure to close resources, or using third-party libraries that have memory leaks. To fix a memory leak, you will need to identify the source of the problem and correct it. This can involve using tools such as memory profilers, heap dumps, and thread dumps to help identify the root cause of the leak.

How to use a profiler to find the memory leak?

There are several ways to use a profiler to identify memory leaks in Java:

Use a built-in profiler: Many integrated development environments (IDEs) such as Eclipse and IntelliJ IDEA have built-in profilers that can be used to detect memory leaks. These profilers typically provide information such as heap usage, object references, and garbage collection statistics.

Use a standalone profiler: Standalone profilers such as VisualVM and JProfiler can be used to profile Java applications. These profilers provide more advanced features such as heap dump analysis, thread profiling, and memory leak detection.

Use command-line tools: The Java Virtual Machine (JVM) provides several command-line tools that can be used to profile memory usage, such as jmap, jstat, and jhat. These tools can be used to create heap dumps, monitor garbage collection statistics, and analyze memory usage.

To use a profiler, you will need to first run your application in profiling mode, and then analyze the data that the profiler collects. This may include analyzing heap dumps, looking at object references, and identifying patterns of memory usage. Once you

have identified the source of the leak, you can then take steps to fix the problem.

It's worth noting that, Profilers can be quite complex, it's advisable to have some familiarity with Java Memory Model, Garbage Collection and JVM internals to effectively use them.

What is out of memory error?

An "Out of Memory" error in Java occurs when the application requests more memory from the JVM than is available. This can happen for a number of reasons, such as:

The application is using more memory than is available on the system: The JVM has a maximum limit on the amount of memory it can use, which is determined by the -Xmx command line option. If the application is using more memory than this limit, an Out of Memory error will occur.

Memory leaks: If an application holds onto objects that are no longer needed, it can cause the JVM to run out of memory. This is known as a memory leak.

Insufficient heap size: The heap is the area of memory where the JVM stores objects. If the heap size is not large enough, the JVM may not be able to allocate enough memory for the application's needs, resulting in an Out of Memory error.

High usage of non-heap memory: The JVM also uses non-heap memory for things like class metadata, JIT compilation data and native resources. If the non-heap memory usage is high, it can cause Out of Memory error.

To fix an Out of Memory error, you will need to identify the cause of the problem and take steps to address it. This may include increasing the amount of memory available to the JVM, fixing memory leaks, or optimizing the application's memory usage.

It's worth noting that, Out of Memory errors can be challenging to debug, it's advisable to use a profiler and analyse the heap dump or thread dump to understand the root cause of the error.

CHAPTER 10: REST

What are the HTTP methods in REST?

REST (Representational State Transfer) is an architectural style for building web services, and it supports the following HTTP methods:

GET: Used to retrieve a resource or a collection of resources. The GET method should be used for read-only operations and should not have any side-effects.

POST: Used to create a new resource. The POST method can be used to submit data to the server, such as form data or JSON payloads.

PUT: Used to update an existing resource. The PUT method can be used to submit data to the server, and it should completely replace the resource if it exists.

PATCH: Used to partially update an existing resource. The PATCH method can be used to submit a set of changes to the server, and it should only modify the specified attributes of the resource.

DELETE: Used to delete a resource. The DELETE method should delete the resource if it exists and should have no additional side-effects.

These methods correspond to the CRUD (Create, Read, Update, Delete) operations that can be performed on resources in a RESTful service. The appropriate method should be used depending on the type of operation being performed. Additionally, some RESTful services may support additional methods, such as HEAD and OPTIONS, for performing specific operations.

What are the idempotent methods in REST?

In REST (Representational State Transfer), idempotent methods are HTTP methods that can be safely called multiple times without changing the result beyond the initial application of the method. The following HTTP methods are considered idempotent:

GET
PUT
DELETE

These methods can be called multiple times without any side effects and should always return the same result.
On the other hand, non-idempotent methods, such as POST, can have side effects and should be called only once.

What are the standards to follow to build a rest service?

REST (Representational State Transfer) is a popular architectural style for building web services. To build a RESTful service that adheres to best practices, there are several standards and guidelines that can be followed, including:

Use HTTP Verbs: RESTful services should use HTTP verbs such as GET, POST, PUT, and DELETE to perform operations on resources.

URI Design: RESTful URIs should be designed to identify resources and their relationships. They should be self-descriptive and hierarchical, with nouns being used as resource names and verbs being used as resource actions.

Use HTTP Status Codes: RESTful services should use appropriate HTTP status codes to indicate the result of an operation, such as 200 OK for success, 404 Not Found for a missing resource, and 500 Internal Server Error for a server-side error.

Use HATEOAS: RESTful services should use HATEOAS (Hypermedia as the Engine of Application State) to allow clients to discover and interact with resources. HATEOAS uses links in the response to provide information about available actions and resources.

Statelessness: RESTful services should be stateless, meaning that they do not maintain client state between requests. This helps to improve scalability and reliability, as it eliminates the need for server-side state storage.

Content Negotiation: RESTful services should support content negotiation, allowing clients to request resources in the format they prefer, such as JSON or XML.

Versioning: RESTful services should be versioned to allow for changes in the API over time. This can be done by including the version number in the URI or in the media type.

Security: RESTful services should be secure, with appropriate measures taken to prevent unauthorized access,

What is the difference between POST and PUT methods?

POST Method:

Purpose: POST is primarily used to submit data to be processed to a specified resource. It is commonly used when you want to create a new resource on the server.
Idempotence: It is not idempotent, meaning that multiple identical requests may have different outcomes, especially if used to create new resources each time.
Safety: It is not considered safe because it may cause changes on the server.

PUT Method:

Purpose: PUT is used to update a resource or create it if it doesn't exist at a specified URL. It is often used when you want to fully replace an existing resource with new data.
Idempotence: It is idempotent, meaning that multiple identical requests will have the same effect as a single request. If the resource doesn't exist, PUT will create it, and if it does, it will update it.
Safety: It is considered safe when used for updates because it doesn't create new resources; it only modifies or replaces existing

ones.

What is sent in headers? Can we intercept the header? If yes, how?

Yes, it is possible to intercept the header in a RESTful service. The header is part of an HTTP request and contains metadata about the request, such as the type of request, the format of the payload, and the authentication credentials.

In some cases, it may be necessary to intercept the header in order to process the request or to provide additional information about the request. This can be done by using middleware, which is a software component that sits between the client and the server. Middleware can inspect and modify the request and response headers, and it can also perform other tasks, such as logging, authentication, and authorization.

For example, if a RESTful service needs to authenticate the client, it can use middleware to inspect the header for an authentication token. If the token is present, the middleware can verify its authenticity and allow the request to continue. If the token is not present or is invalid, the middleware can return an error response, such as a 401 Unauthorized.

In summary, intercepting the header is a common practice in RESTful services, and it can be used to implement additional functionality, such as authentication, logging, and request modification.

How to secure REST API?

Securing a REST API involves a combination of different techniques and technologies to ensure that only authorized clients can access the API and that the data transmitted between the client and the server is protected.

Here are some common ways to secure a REST API:

Authentication: This is the process of verifying the identity of the client. REST APIs can use various authentication methods such as Basic Authentication, Token-based Authentication (OAuth2, JWT) or API keys.

Authorization: This is the process of determining whether a client is allowed to perform a specific action on a resource. Authorization can be done using role-based access control (RBAC) or access control lists (ACLs).

Encryption: This is the process of protecting data in transit by encrypting it. REST APIs can use HTTPS to encrypt the data transmitted between the client and the server.

Validation: This is the process of validating the input data to ensure that it meets certain criteria. REST APIs can use input validation libraries to validate data before processing it.

Rate Limiting: This is the process of limiting the number of requests that a client can make to an API within a certain time period. This can help prevent Denial of Service (DoS) attacks.

Logging and Auditing: This is the process of recording API access and activity, which can be used to detect and investigate security incidents.

How to pass a parameter in request, is it via URL or as a JSON object?

There are multiple ways to pass a parameter in a request, and the choice depends on the application's requirements and the API design. Two common ways to pass parameters in a request are:

Via URL: The URL query string can be used to pass parameters in a request. The parameters are appended to the URL after the "?" character, and multiple parameters are separated by "&". For example, http://example.com/api/users?name=john&age=30.

As a JSON object: Another way to pass parameters is to include them in the request body as a JSON object. This is common in

RESTful APIs that use HTTP POST or PUT methods to create or update resources. For example, the request body could be { "name": "john", "age": 30 }.

Both methods have their advantages and disadvantages. Passing parameters via URL is easy to understand and implement, and the parameters can be easily bookmarked or shared. However, passing sensitive data via URL is not secure as it can be easily intercepted and viewed by third parties.

On the other hand, passing parameters as a JSON object in the request body is more secure as it is not visible in the URL. However, it may require more effort to implement and may not be as easy to understand for beginners.

In summary, passing parameters via URL or as a JSON object in the request body are both valid methods, and the choice depends on the application's requirements and the API design.

It's important to note that, securing a REST API requires a holistic approach that considers all aspects of the API and its environment, such as the infrastructure, network, and clients. Also, security best practices and standards such as OWASP and PCI-DSS should be followed.

Also, you can use Spring Security to secure your REST API.

CHAPTER 11: DESIGN PATTERN & SYSTEM DESIGN

Design Rest API for tiny URL application, how many endpoints it requires?

Based on that there is a discussion on it.

1. Endpoint to create a Tiny URL

Request

POST /tinyurls

Body:

```
{
    "long_url": "https://www.example.com/very/long/url"
}
```

Response

Status: 201 Created

Body:

```
{
    "short_url": "http://tiny.url/abc123",
    "long_url": "https://www.example.com/very/long/url"
}
```

2. Endpoint to retrieve the original URL from a Tiny URL

Request

GET /tinyurls/{short_url_id}

Response

Status: 302 Found

Location: https://www.example.com/very/long/url

3. Endpoint to retrieve statistics for a Tiny URL

Request

GET /tinyurls/{short_url_id}/stats

Response

Status: 200 OK

Body:

```
{
    "click_count": 42,
    "last_clicked_at": "2022-01-01T12:00:00Z"
}
```

Note: The implementation details (such as the format of the short URL and the storage backend) and the error handling are omitted for simplicity.

What is a singleton design pattern?

Singleton design pattern:

The singleton design pattern is a creational design pattern that ensures that a class has only one instance while providing a global access point to this instance. The singleton pattern is often used when a single object is needed to coordinate actions across the system.

To implement the singleton pattern, you will need to:

Define a class with a private constructor, so that no other class can instantiate it.
Declare a static variable of the same type as the class, and create an instance of the class in the variable.
Declare a static method that returns the instance of the class.
Here is an example of a singleton class in Java:

```
public class Singleton {
```

```
    private static Singleton instance;

    private Singleton() {}

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

To use the singleton, you would call the getInstance the method is like this:

The singleton pattern is useful when you need to ensure that a class has only one instance, and you need to provide a global access point to that instance. It is also useful when you need to control the number of objects that are created because you can ensure that only one object is created. However, the singleton pattern can make it difficult to test your code, because you cannot use the constructor to create new instances of the singleton class for testing purposes.

How to break the singleton design pattern?

The Singleton design pattern is a creational pattern that ensures that only one instance of a class can exist in a system, and provides a global point of access to that instance. However, in some situations, it may be necessary to break the Singleton pattern and create multiple instances of the class. Here are some common techniques for breaking the Singleton pattern:

Use Dependency Injection: One way to break the Singleton pattern is to use dependency injection to pass an instance of the Singleton class into other classes that require it. This allows multiple instances to be created, and can be useful when testing or when you need to have different instances with different configurations.

Use Reflection: Another way to break the Singleton pattern is to use reflection to access the private constructor of the Singleton class

and create a new instance. This is not recommended in most cases, as it can cause unexpected behavior and can be a security risk.

Use a Subclass: You can also create a subclass of the Singleton class and override its behavior. The subclass can then be used to create multiple instances with different behavior, while the original Singleton class continues to provide a single instance.

Use a Factory: Another way to break the Singleton pattern is to use a factory class that creates instances of the Singleton class. This allows you to create multiple instances with different configurations, while still providing a global point of access to the instances.

It's important to note that breaking the Singleton pattern can have implications for the overall design of the system, so it should be done with caution and only when necessary. The Singleton pattern is often used to enforce a specific behaviour or guarantee a certain level of performance, so breaking it should be done with a clear understanding of the implications.

What is the solution to fix the above problem?

The Singleton design pattern is a creational pattern that ensures that only one instance of a class can exist in a system, and provides a global point of access to that instance. While it's possible to break the Singleton pattern as I described in the previous answer, it's generally best to maintain the Singleton pattern to ensure that the class behaves consistently and as intended.

Here are some techniques to help prevent breaking the Singleton pattern:

Use a Private Constructor: A key part of the Singleton pattern is the use of a private constructor. This prevents other classes from creating new instances of the Singleton class. By ensuring that the constructor remains private, you can help prevent the pattern from being broken.

Use Static Initialization: In many programming languages, static initialization ensures that a static class member is initialized only

once, the first time it is accessed. This can help to ensure that only one instance of the Singleton class is created.

Use a Final Instance Variable: Another way to help prevent breaking the Singleton pattern is to use a final instance variable to store the Singleton instance. This ensures that the instance is only created once and cannot be changed.

Use a Thread-Safe Implementation: In a multi-threaded environment, it's important to ensure that the Singleton instance is thread-safe. This can be achieved by using a thread-safe implementation such as double-checked locking or the Initialization-on-demand holder idiom.

Use a Private Inner Class: Another approach to implementing the Singleton pattern is to use a private inner class to hold the Singleton instance. This ensures that the instance is only created once and cannot be accessed from outside the class.

By using these techniques, you can help prevent the Singleton pattern from being broken and ensure that the class behaves consistently and as intended.

What is a Builder Design pattern?

Builder Design Pattern

The Builder design pattern is a creational design pattern that allows for the step-by-step construction of complex objects using a specific construction process. It separates the construction of an object from its representation so that the same construction process can create different representations.

The Builder pattern is useful when you want to create complex objects, but the construction process for these objects is relatively simple. It allows you to create the object step by step and provides a way to retrieve the final object once it has been constructed.

Here is an example of how the Builder pattern might be implemented in Java:

```
public class BuilderExample {
    public static void main(String[] args) {
        // create the director
        Director director = new Director();

        // create the builder
        Builder builder = new ConcreteBuilder();

        // construct the complex object
        director.construct(builder);

        // retrieve the finished product
        ComplexObject complexObject = builder.getResult();
    }
}

class ComplexObject {
    // fields for the complex object
}

class Director {
    public void construct(Builder builder) {
        // use the builder to construct the complex object
    }
}

abstract class Builder {
    // methods for constructing the complex object
    public abstract ComplexObject getResult();
}

class ConcreteBuilder extends Builder {
    private ComplexObject object = new ComplexObject();

    // methods for constructing the complex object
    public ComplexObject getResult() {
        return object;
    }
```

```
}
```

Which design pattern is used by spring AOP? Explain with logic?

Spring AOP (Aspect-Oriented Programming) is based on the Decorator design pattern. The Decorator pattern is a structural pattern that allows adding new behaviors to existing objects dynamically by placing these objects inside special wrapper objects that contain the behaviors. In AOP, aspects are defined as modules containing certain behaviors that can be combined with other application objects. The aspect is applied to the application objects using proxies. In this way, AOP provides a way to add behaviors to existing objects without affecting the underlying code.

What is Adapter design pattern & Proxy design pattern?

Adaptor design pattern:

The Adapter design pattern is a structural pattern that allows two incompatible interfaces to work together. It converts the interface of one class into another interface that clients expect. This pattern is often used when an existing class's interface does not match the interface required by a client, or when a client needs to work with multiple classes that have different interfaces.

The Adapter pattern involves the following components:

Target Interface: This is the interface that the client expects to work with.

Adaptee: This is the class that has the interface that does not match the target interface.

Adapter: This is the class that adapts the interface of the Adaptee to the Target Interface.

The Adapter pattern can be implemented in two ways: Class Adapter and Object Adapter.

In the Class Adapter approach, the Adapter class extends the Adaptee class and implements the Target Interface. The Adapter

class inherits the behavior of the Adaptee class and adds the behavior required to match the Target Interface.

In the Object Adapter approach, the Adapter class contains an instance of the Adaptee class and implements the Target Interface. The Adapter class delegates the requests from the client to the Adaptee instance and adds the behavior required to match the Target Interface.

Here's an example to illustrate how the Adapter pattern works:

Suppose you have a client that expects a simple interface for a printer that only has a print() method. However, you have an existing class called AdvancedPrinter that has a complex interface with multiple methods. You can create an Adapter class that adapts the interface of the AdvancedPrinter to the simple interface expected by the client. The Adapter class would have a print() method that calls the appropriate methods on the AdvancedPrinter to accomplish the print operation. The client can then use the Adapter class to print documents without having to know about the complex interface of the AdvancedPrinter.

Proxy Design pattern:

The Proxy design pattern is a structural pattern that provides a surrogate or placeholder for another object to control access to it. The Proxy pattern allows you to create a representative object that can act as an intermediary between a client and the real object. The proxy object can perform additional functionality such as security checks, caching, or remote communication.

The Proxy pattern involves the following components:

Subject: This is the interface that both the Real Subject and the Proxy implement. It defines the common interface for the Real Subject and the Proxy so that the client can work with both objects interchangeably.

Real Subject: This is the object that the client wants to access. It implements the Subject interface and provides the real implementation of the operations.

Proxy: This is the object that acts as a surrogate for the Real Subject. It also implements the Subject interface and forwards the requests from the client to the Real Subject. In addition to forwarding requests, the Proxy may also perform additional functionality such as caching, logging, or security checks.

The Proxy pattern can be implemented in several ways, including:

Virtual Proxy: This is a type of Proxy that creates an object on demand. When the client requests an operation, the Virtual Proxy checks whether the Real Subject has been created, and if not, it creates it. This is useful when creating the Real Subject is expensive, and you want to delay its creation until it is actually needed.

Protection Proxy: This is a type of Proxy that checks whether the client has the necessary permissions to access the Real Subject. If the client has the required permissions, the Proxy forwards the request to the Real Subject. Otherwise, it denies the request.

Remote Proxy: This is a type of Proxy that acts as a local representative for a remote object. When the client requests an operation, the Remote Proxy forwards the request to the remote object and returns the result.

Here's an example to illustrate how the Proxy pattern works:

Suppose you have a resource-intensive object that needs to be accessed frequently. You can create a Proxy object that acts as a surrogate for the real object. The Proxy object can cache the results of the operations and return the cached results to the client when the same operation is requested again. This can save time and resources by avoiding the need to recreate the object or perform expensive operations repeatedly.

What is Decorator Pattern?

Decorator Pattern:

The Decorator pattern is a design pattern in object-oriented programming that allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior

of other objects from the same class. It is a structural pattern that allows objects to have additional behavior or responsibilities without the need to create a subclass of the original object.

In the Decorator pattern, a set of decorator classes are created that add new functionality to the original object. These decorators conform to the same interface as the object being decorated, and they contain a reference to the object they are decorating. The decorators can add new behavior to the object by intercepting its method calls and modifying their behavior or adding new functionality.

Here are some key features of the Decorator pattern:

It is a way to extend the functionality of an object without sub classing.

Decorators can be stacked on top of each other to add multiple layers of functionality.

Decorators can be added and removed at runtime, which makes it easy to change an object's behaviour dynamically.

The original object can remain unchanged, which helps to ensure that existing code and unit tests still work as expected.

The Decorator pattern allows for a clear separation of concerns between the object being decorated and the code that adds new behaviour.

A common real-world example of the Decorator pattern is with streaming services. For example, a user might have a basic streaming service that allows them to access a limited library of content. They can then choose to add a set of decorator services, such as one for high-definition video, another for access to live events, and yet another for access to an expanded library of content. Each decorator adds a layer of functionality to the basic streaming service, allowing the user to customize their experience and access additional features without needing to switch to a different service.

What is a facade design pattern?

The facade design pattern is a structural design pattern that provides a unified interface to a set of interfaces in a subsystem. It defines a high-level interface that makes the subsystem easier to use and hides the complexity of the subsystem from the client.

The facade pattern is often used in libraries and frameworks to provide a simplified interface to their functionality. For example, a facade could be used to provide a single method for starting a database connection, initializing a logging system, and configuring a web server.

The facade pattern can also be used to decouple clients from the implementation of a subsystem. This can make the code more flexible and easier to maintain. For example, if a client needs to use multiple classes in a subsystem, the facade can provide a single interface that the client can use. This allows the client to be independent of the implementation of the subsystem.

```
public class DatabaseFacade {
    private DatabaseConnection connection;
    private Logger logger;
    public DatabaseFacade(DatabaseConnection connection, Logger logger) {
        this.connection = connection;
        this.logger = logger;
    }
    public void openConnection() {
        connection.open();
        logger.log("Database connection opened");
    }
    public void closeConnection() {
        connection.close();
```

```
        logger.log("Database connection closed");
    }
    public void executeQuery(String query) {
        // Execute the query
        logger.log("Query executed: " + query);
    }
}
```

The DatabaseFacade class provides a simplified interface to the database subsystem. It hides the complexity of the database subsystem from the client. To use the database, the client simply needs to create an instance of the DatabaseFacade class and call its methods.

The facade pattern is a useful design pattern for simplifying the use of complex subsystems and decoupling clients from the implementation of subsystems.

CHAPTER 12: SQL/DATABASE/HIBERNATE-JPA

Write a SQL query to find 5th max salary from employee table?

The exact SQL query to find the 5th highest salary from an employee table will depend on the database system you are using. Here is a general SQL query that can be used for most SQL-based database systems:

```
SELECT salary
FROM (
  SELECT salary, ROW_NUMBER() OVER (ORDER BY salary DESC) AS row_num
  FROM employee
) AS sub_query
WHERE row_num = 5;
```

This query uses a subquery to assign a row number to each salary in the employee table, ordered by salary in descending order. The outer query then selects the salary where the row number is equal to 5, which gives the 5th highest salary.

Write a SQL query to remove duplicate employee records?

The SQL query to remove duplicate employee records will depend on the definition of "duplicate". Here's a general query that removes duplicate records based on all columns in the table:

```
DELETE FROM employee
WHERE id NOT IN (
  SELECT MIN(id)
  FROM employee
  GROUP BY column1, column2, ...
```

);

In this query, the subquery selects the minimum id for each group of duplicate records. The outer query then deletes all records whose id is not in the subquery result, effectively removing all duplicate records except one.

You can replace column1, column2, ... with the specific columns that define a duplicate record. For example, if two employee records are considered duplicates if they have the same first name, last name, and hire date, the query would look like this:

```
DELETE FROM employee
WHERE id NOT IN (
  SELECT MIN(id)
  FROM employee
  GROUP BY first_name, last_name, hire_date
);
```

Write a query to find employee numbers in each department.

To find the number of employees in each department, you can use the SQL GROUP BY clause with the COUNT() aggregate function. Here's an example query that should work in most relational database management systems (RDBMS):

```
SELECT department, COUNT(*) as num_employees
FROM employees
GROUP BY department
```

In this query, we select the department column from the employees table and count the number of employees in each department using the COUNT(*) aggregate function. We also give the count column an alias num_employees.

The GROUP BY clause groups the results by the department column, so we get one row for each unique department in the employees table.

The output of this query would be a list of departments with the number of employees in each department.

For example, if we have an employees table with columns employee_id, employee_name, department, and salary, and some sample data like this:

```
employee_id | employee_name | department | salary
------------+---------------+------------+-------
1           | Alice         | HR         | 50000
2           | Bob           | IT         | 60000
3           | Charlie       | IT         | 55000
4           | Dave          | HR         | 45000
5           | Eve           | Marketing  | 70000
6           | Frank         | IT         | 65000
7           | Grace         | Marketing  | 60000
8           | Harry         | HR         | 55000
9           | Ivan          | IT         | 70000
```

Then, the output of the query would be:

```
department | num_employees
-----------+--------------
HR         | 3
IT         | 4
Marketing  | 2
```

In this output, we can see that there are three employees in the HR department, four employees in the IT department, and two employees in the Marketing department.

Write SQL Query to find students who are enrolled in courses whose price is above 50000?

```
SELECT s.*
```

```
FROM students s
INNER JOIN enrollments e ON s.student_id = e.student_id
INNER JOIN courses c ON e.course_id = c.course_id
WHERE c.price > 50000;
```

Create Database design for Employee and address?

```
CREATE TABLE employees (
  id INT PRIMARY KEY AUTO_INCREMENT,
  first_name VARCHAR(50) NOT NULL,
  last_name VARCHAR(50) NOT NULL,
  age INT NOT NULL,
  email VARCHAR(100) NOT NULL,
  address_id INT,
  FOREIGN KEY (address_id) REFERENCES addresses(id)
);

CREATE TABLE addresses (
  id INT PRIMARY KEY AUTO_INCREMENT,
  street VARCHAR(100) NOT NULL,
  city VARCHAR(50) NOT NULL,
  state VARCHAR(50) NOT NULL,
  zip VARCHAR(10) NOT NULL
);
```

What does the JDBC forName() method do for you when you connect to any DB?

The forName() method is a static method in the java.lang.Class class that is used to load a JDBC driver at runtime. In the context of JDBC, a driver is a software component that provides the necessary functionality to connect to a specific type of database.

The forName() method takes a string parameter that specifies the fully qualified name of the class that implements the JDBC driver. For example, to load the JDBC driver for MySQL, you would use the following code:

```
Class.forName("com.mysql.jdbc.Driver");
```

This code loads the class com.mysql.jdbc.Driver using the current class loader. The class loader then initializes the driver, which registers itself with the DriverManager class. Once the driver is registered, it can be used to establish a connection to the database.

It's worth noting that the forName() method is used less frequently in modern JDBC code, as many JDBC drivers now include a static initialization block that registers the driver automatically when the class is loaded. In such cases, you can simply include the JDBC driver JAR file in your project's classpath and use the DriverManager.getConnection() method to establish a connection to the database.

Do you know triggers and how do they work?

Yes, I am familiar with database triggers and how they work.

A trigger is a database object that is automatically executed in response to certain events, such as changes to data in a table. Triggers are typically used to enforce business rules, audit changes to data, or synchronize data across different tables or databases.

Triggers are associated with a specific table or view and can be set to execute either before or after an event occurs. The events that can trigger a trigger include inserts, updates, and deletes to the table. When a trigger is fired, it can execute one or more SQL statements or call a stored procedure.

The basic syntax for creating a trigger in SQL is as follows:

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON table_name
[FOR EACH ROW]
```

```
BEGIN
    -- Trigger logic here
END;
```

In this example, trigger_name is the name of the trigger, table_name is the name of the table or view that the trigger is associated with, and BEFORE or AFTER specifies when the trigger should be executed. The FOR EACH ROW clause indicates that the trigger should execute once for each row affected by the triggering event.

Once a trigger is created, it is automatically executed whenever the specified event occurs. Triggers can be a powerful tool for enforcing data integrity and automating database tasks, but it's important to use them judiciously to avoid performance issues and unexpected results.

Explain database Joins?

In a relational database, data is typically stored in multiple tables. Database joins are used to combine data from two or more tables into a single result set based on a common column or set of columns.

There are several types of joins in a relational database, including:

Inner Join: An inner join returns only the rows that have matching values in both tables being joined. The join is performed by comparing values in the columns that are common to both tables.

Left Join: A left join returns all the rows from the left table and the matching rows from the right table. If there is no matching row in the right table, the result will contain null values for the right table columns.

Right Join: A right join is similar to a left join, but it returns all the rows from the right table and the matching rows from the left table. If there is no matching row in the left table, the result will contain null values for the left table columns.

Full Outer Join: A full outer join returns all the rows from both tables, including those that do not have matching values in the other table. If there is no matching row in one of the tables, the result will contain null values for the columns of the missing table.

Cross Join: A cross join returns the Cartesian product of the two tables, which means that every row from one table is combined with every row from the other table.

Here's an example to illustrate how a join works:

Suppose you have two tables: Customers and Orders. The Customers table has columns for CustomerID, Name, and Address, while the Orders table has columns for OrderID, CustomerID, and OrderDate. You can join these two tables to get a result set that contains information about customers and their orders.

To perform an inner join on these tables, you would match the rows in the Customers table with the rows in the Orders table based on the CustomerID column. The resulting joined table would contain only the rows where there is a matching value in both tables.

To perform a left join on these tables, you would return all the rows from the Customers table and the matching rows from the Orders table based on the CustomerID column. If there is no matching row in the Orders table, the result would contain null values for the OrderID and OrderDate columns.

To perform a right join on these tables, you would return all the rows from the Orders table and the matching rows from the Customers table based on the CustomerID column. If there is no matching row in the Customers table, the result would contain null values for the Name and Address columns.

These are just a few examples of how joins can be used to combine data from multiple tables in a relational database.

What is complex join in Hibernate?

Hibernate is a powerful ORM (Object-Relational Mapping) framework that allows you to work with complex join queries.

One way to perform complex join queries in Hibernate is to use the Criteria API, which allows you to build queries programmatically. The Criteria API provides a fluent and type-safe way to construct queries, and it supports a wide variety of operations, including inner and outer joins, subqueries, and projections.

Here's an example of how you can use the Criteria API to perform an inner join:

```
Criteria criteria = session.createCriteria(Order.class, "o")
    .createAlias("o.customer", "c")
    .add(Restrictions.eq("c.name", "John"))
    .setProjection(Projections.property("o.total"))
    .addOrder(Order.desc("o.total"));
List results = criteria.list();
```

In this example, the createAlias() method is used to create an inner join between the Order and Customer entities on the customer property of the Order entity. The Restrictions.eq() method is used to add a filter on the name property of the Customer entity, and the setProjection() method is used to select the total property of the Order entity.

Another way to perform complex join queries in Hibernate is to use the HQL (Hibernate Query Language) which is similar to SQL.

```
Query query = session.createQuery("SELECT o.total FROM Order o
JOIN o.customer c WHERE c.name = :name ORDER BY o.total
DESC");
query.setParameter("name", "John");
List results = query.list();
```

In this example, the JOIN clause is used to create an inner join between the Order and Customer entities on the customer property of the Order entity. The WHERE clause is used to add a filter on the name property of the Customer entity, and the ORDER BY clause is used to sort the results by the total property of the Order entity.

In conclusion, hibernate provides several ways to perform complex join queries, such as the Criteria API and HQL.

How to store and navigate hierarchies?

Storing and navigating hierarchies can be accomplished in several ways, depending on the specific needs of the application. Here are a few common approaches:

Adjacency List Model: In this model, each node in the hierarchy is stored as a record in a database table, with a column that references the parent node. This makes it easy to navigate the hierarchy, as you can simply perform a recursive query to retrieve all the descendants of a given node. However, this approach can be inefficient for very large hierarchies, as it requires multiple database queries.

Nested Set Model: In this model, each node is represented by two columns in the database table, which indicate the range of nodes that fall within its subtree. This approach can be more efficient than the adjacency list model for navigating hierarchies, as it only requires a single query to retrieve all the descendants of a given node. However, it can be more complex to implement, as it requires maintaining the nested set values when nodes are added, deleted, or moved.

Materialized Path Model: In this model, each node is represented by a string that includes the path to the root node, delimited by a separator (e.g. /). This approach can be efficient for querying a specific node or subtree, as it only requires a simple string comparison. However, it can be less efficient for complex queries or for updating the hierarchy, as it requires updating the paths of all the affected nodes.

Closure Table Model: In this model, a separate table is created to represent the relationships between nodes in the hierarchy. Each record in the closure table represents a direct path between two nodes in the hierarchy, allowing for efficient queries and updates. However, this approach can be more complex to implement and can require more storage space than the other models.

Once you've decided on a model for storing hierarchies, navigating the hierarchy can be accomplished by querying the database for the appropriate nodes and relationships, and using recursion or iteration to traverse the tree structure. The specific approach will depend on the details of the model and the requirements of the application.

What is the data type of the index in Database?

In a database, the data type of an index depends on the type of data being indexed.

For example, if the index is created on a column that contains string values, such as a column that stores the names of customers, the index data type will likely be a string type, such as VARCHAR or TEXT. If the index is created on a column that contains numeric values, such as a column that stores prices, the index data type will likely be a numeric type, such as INT or DECIMAL.

In general, the index data type should match the data type of the column being indexed. This ensures that the index can be used efficiently to speed up queries that search for specific values in the column. It's also important to consider the length of the indexed values, as longer values may require more storage space and may affect the performance of the index.

It's worth noting that in addition to the data type of the indexed column, the database may also use a specific data structure for the index, such as a B-tree or hash table, to improve the efficiency of index lookups. The specific data structure used will depend on the database system and the configuration options used when creating the index.

Write a query to find duplicate entries in a table against a column?

To find duplicate entries in a table against a specific column, you can use a GROUP BY clause and a HAVING clause to filter the results based on the count of the entries. Here's an example query that finds duplicate entries in a table called my_table against a column called my_column:

```
SELECT my_column, COUNT(*) as count
FROM my_table
GROUP BY my_column
HAVING count > 1;
```

In this query, we group the results by the values in the my_column column using the GROUP BY clause. We then use the COUNT function to count the number of entries in each group, and we give this count the alias count. Finally, we use the HAVING clause to filter the results to only show groups with a count greater than 1, indicating that there are duplicate entries in the table for that column value.

This query will return a list of values in the my column column that have more than one entry in the table, along with the count of entries for each value. You can use this information to identify the duplicate entries in the table and take appropriate action, such as deleting the duplicates or modifying the schema to prevent future duplicates.

What is the differences between Indexing and Partitioning?

Indexing and partitioning are two common techniques used in database optimization, but they have different purposes and use cases.

Indexing is the process of creating an index on one or more columns in a table, in order to speed up the retrieval of data from that table. An index is a data structure that is used to improve the performance of search operations on a table by allowing the database to find specific rows more quickly. Indexes can be created on single or multiple columns and can be either unique or non-unique. Indexing can help to improve the performance of read operations on the table, but it can also have a negative impact on the performance of write operations, as the index must be updated every time data is inserted, updated or deleted.

Partitioning, on the other hand, is the process of dividing a large table into smaller, more manageable pieces called partitions, based

on some criteria such as date range or geographic location. Partitioning can improve the performance of both read and write operations on the table by allowing the database to process only the necessary partitions, instead of the entire table. Partitioning can also help with data management, as it allows administrators to more easily archive or delete old data.

In summary, the main difference between indexing and partitioning is that indexing is used to improve the performance of search operations on a table, while partitioning is used to improve the performance of both read and write operations on a large table. Both techniques can be used together to optimize the performance of a database, but they should be chosen based on the specific use case and requirements of the application.

Explain the Hibernate-JPA structure.

Hibernate is a powerful ORM (Object-Relational Mapping) framework that allows you to work with databases in a Java-based application. It is based on the JPA (Java Persistence API) standard, which is the Java standard for ORM.

The main components of the Hibernate-JPA structure are:

Entity: An entity represents a table in the database, and it is mapped to a Java class. Each entity has a set of fields, which represent the columns in the table, and it also has a set of methods to interact with the data.

EntityManager: The Entity Manager is the main interface for interacting with the database. It provides methods for performing CRUD (Create, Read, Update, Delete) operations, as well as methods for querying the database.

EntityManagerFactory: The Entity Manager Factory is responsible for creating and managing EntityManager instances. It is typically created once during application initialization and is used to create EntityManager instances for each database transaction.

Persistence Unit: The Persistence Unit defines the configuration settings for the Entity Manager Factory, such as the data source, database dialect, and mapping information.

EntityManager and Session: Entity Manager is the interface defined by JPA, Session is the interface defined by Hibernate. Entity Manager is built on top of the Session, and it provides a simpler, more consistent interface for working with JPA entities.

Transaction: A transaction represents a unit of work that is performed against the database. A transaction can include one or more operations, such as inserting, updating, or deleting data.

Mapping files: Hibernate uses XML or annotations in the entity classes to map the entities to the database tables and columns. It defines the relationship between the entities and how the entities should be persisted and retrieved from the database.

In summary, Hibernate-JPA structure is based on the JPA standard, it allows you to work with databases in a Java-based application. It has several main components such as Entity, EntityManager, EntityManagerFactory,Persistence Unit,Transaction and Mapping files that work together to provide a consistent and powerful way to interact with databases.

Which annotation/configuration is required to enable the native SQL in JPA?

To enable native SQL in JPA, you can use the @SqlResultSetMapping annotation or the <sql-result-set-mapping> element in the persistence.xml file.

The @SqlResultSetMapping annotation is used to map the results of a native SQL query to an entity or a DTO (Data Transfer Object). It is used in conjunction with the EntityManager.createNativeQuery() method to execute the native SQL query and map the results to the specified entity or DTO.

Here's an example of how you can use the @SqlResultSetMapping annotation to map the results of a native SQL query to an entity:

```
@SqlResultSetMapping(
    name="EmployeeResult",
    entities={
        @EntityResult(
            entityClass=Employee.class,
            fields={
                @FieldResult(name="id", column="emp_id"),
                @FieldResult(name="name", column="emp_name"),
                @FieldResult(name="salary", column="emp_salary")
            }
        )
    }
)
```

In this example, the @SqlResultSetMapping annotation is used to map the results of a native SQL query to the Employee entity. The name attribute is used to specify a unique name for the mapping, and the entities attribute is used to define the entity mappings.

Alternatively, you can also use the <sql-result-set-mapping> element in the persistence.xml file to define the result set mapping.

Explain Entity in JPA and all annotations used to create Entity class.

In JPA, an entity is a Java class that represents a table in the database. It is used to map the data in the database to Java objects, and it provides a way to interact with the data using the Entity Manager.

Here are some of the main annotations used to create an entity class in JPA:

@Entity: This annotation is used to mark a class as an entity. It is typically placed on the class definition, and it tells the JPA provider

that this class should be treated as an entity and mapped to a table in the database.

@Table: This annotation is used to specify the name of the table that the entity should be mapped to. It can be used to specify the schema, catalog, and other properties of the table.

@Id: This annotation is used to mark a field as the primary key for the entity. The field that is annotated with @Id will be mapped to the primary key column of the table.

@Column: This annotation is used to specify the properties of a field that should be mapped to a column in the table. It can be used to specify the name of the column, the data type, and other properties of the column.

@GeneratedValue: This annotation is used to specify how the primary key should be generated. It can be used to specify that the primary key should be generated automatically by the database or by the JPA provider.

@ManyToOne and @OneToMany : These annotations are used to define the relationship between entities. @ManyToOne is used to define a many-to-one relationship between two entities, while @OneToMany is used to define a one-to-many relationship between two entities.

@Transient: This annotation is used to mark a field that should not be persisted to the database. The field that is annotated with @Transient will not be mapped to a column in the table.

How can we define a composite key in the Entity class?

In JPA, a composite key is a primary key that is made up of more than one field. To define a composite key in an entity class, you can use the @IdClass or @EmbeddedId annotation.

The @IdClass annotation is used to define a composite key using a separate primary key class. The primary key class must have the

same fields as the composite key in the entity class, and it must implement the Serializable interface. Here's an example of how you can use the @IdClass annotation to define a composite key:

```
@Entity
@IdClass(CompositeKey.class)
public class Employee {
    @Id
    private int id;
    @Id
    private String name;
    // ...
}

public class CompositeKey implements Serializable {
    private int id;
    private String name;
    // ...
}
```

In this example, the Employee entity class has a composite key made up of the id and name fields. The CompositeKey class is used as the primary key class, and it has the same fields as the composite key in the Employee entity class.

The @EmbeddedId annotation is used to define a composite key by embedding an @Embeddable class within the entity class. The @Embeddable class contains the fields that make up the composite key. Here's an example of how you can use the @EmbeddedId annotation to define a composite key:

```
@Entity
public class Employee {
    @EmbeddedId
    private CompositeKey key;
    // ...
}
```

```
@Embeddable
public class CompositeKey implements Serializable {
    private int id;
    private String name;
    // ...
}
```

In this example, the Employee entity class has a composite key made up of the id and name fields, which are embedded within.

What are the JPA Annotation used for a composite attribute?

In JPA, you can use the @Embedded annotation to map a composite attribute, which is a group of simple attributes that together form a single logical attribute.

The @Embedded annotation is used on a field or property of an entity class, and it tells the JPA provider to treat the field as an embedded object. The embedded object should be annotated with @Embeddable.

Here's an example of how you can use the @Embedded annotation to map a composite attribute:

```
@Entity
public class Employee {
    @Embedded
    private Address address;
    // ...
}

@Embeddable
public class Address {
    private String street;
    private String city;
    private String state;
    private String zip;
}
```

In this example, the Employee entity class has a composite attribute called address, which is an instance of the Address class. The Address class is annotated with @Embeddable which tells JPA provider that this class is an embeddable class, and it should be treated as an embedded object.

The @Embedded annotation can also be used with @AttributeOverride and @attributeOverrides to customize the column name and table name of the attribute fields.

In summary, the @Embedded annotation is used to map a composite attribute, which is a group of simple attributes that together form a single logical attribute in JPA. It is used on a field or property of an entity class and tells the JPA provider to treat the field as an embedded object. The embedded object should be annotated with @Embeddable.

Which annotation is used to handle the joins between multiple tables at the Entity class level?

The @OneToMany and @ManyToOne annotations are used to handle the relationship between entities in a one-to-many and many-to-one relationship respectively at the entity class level in JPA (Java Persistence API). These annotations are used to map the foreign key relationships between entities.

How do you handle unidirectional join and bidirectional join at the Entity level?

In JPA, a unidirectional join is when one entity has a relationship to another entity, but the other entity does not have a relationship back to the first entity. This can be handled by using the @OneToMany or @ManyToOne annotation on the entity that has the relationship to the other entity, but not on the other entity.

A bidirectional join, on the other hand, is when both entities have a relationship to each other. This can be handled by using the @OneToMany and @ManyToOne annotations on both entities, and

also using the mappedBy attribute to specify the entity that owns the relationship.

For example, if we have an entity called "Department" and another entity called "Employee", where a department can have multiple employees, but an employee can only belong to one department, we can set up a bidirectional relationship as follows:

```
@Entity
public class Department {
    @OneToMany(mappedBy = "department")
    private List<Employee> employees;
}
@Entity
public class Employee {
    @ManyToOne
    private Department department;
}
```

Here, the "Department" entity is the owner of the relationship and the "Employee" entity is the inverse end of the relationship.

It's important to note that for bidirectional relationship, it's crucial to keep both sides of the relationship in sync.

How to handle relationships in spring data JPA?

Here are some commonly used approaches for handling relationships in Spring Data JPA:

One-to-One Relationship: To define a one-to-one relationship between two entities, you can use the @OneToOne annotation. This annotation can be used on a field or a getter method in the entity class that represents the owning side of the relationship. For example:

```
@Entity
public class Employee {
```

```
    @OneToOne
    @JoinColumn(name = "address_id")
    private Address address;
}
@Entity
public class Address {
    @OneToOne(mappedBy = "address")
    private Employee employee;
}
```

In this example, the Employee entity has a one-to-one relationship with the Address entity. The @OneToOne annotation is used on the address field in the Employee entity to define the relationship. The @JoinColumn annotation is used to specify the foreign key column name in the employee's table.

One-to-Many Relationship: To define a one-to-many relationship between two entities, you can use the @OneToMany and @ManyToOne annotations. The @OneToMany annotation is used on a collection field or a getter method in the entity class that represents the owning side of the relationship, and the @ManyToOne annotation is used on a field or a getter method in the entity class that represents the inverse side of the relationship. For example:

```
@Entity
public class Employee {
    @OneToMany(mappedBy = "employee")
    private List<Address> addresses;
}
@Entity
public class Address {
    @ManyToOne
```

```
    @JoinColumn(name = "employee_id")
    private Employee employee;
}
```

In this example, the Employee entity has a one-to-many relationship with the Address entity. The @OneToMany annotation is used on the addresses field in the Employee entity to define the relationship, and the @ManyToOne annotation is used on the employee field in the Address entity to specify the many-to-one side of the relationship. The @JoinColumn annotation is used to specify the foreign key column name in the addresses table.

How to handle the Parent and child relationship in JPA?

JPA (Java Persistence API) is a specification for managing the mapping of Java objects to relational databases. To handle a parent-child relationship in JPA, you can use the following annotations:

@OneToMany and @ManyToOne: These annotations are used to define a one-to-many relationship between two entities, where one entity is the parent and the other is the child. The @OneToMany annotation is added to the parent entity, and the @ManyToOne annotation is added to the child entity.

@JoinColumn: This annotation is used to define the foreign key column in the child table that references the primary key column in the parent table.

CascadeType: JPA provides several options for cascading operations from the parent to the child entities, such as CascadeType.PERSIST, CascadeType.REMOVE, and CascadeType.ALL.

For example, If you have a parent Entity Department and child Entity

```
@Entity
public class Department {
    @Id
    private Long id;
    private String name;
```

```
    @OneToMany(mappedBy = "department", cascade =
CascadeType.ALL, orphanRemoval = true)
    private List<Employee> employees;
    //getters and setters
}

@Entity
public class Employee {
    @Id
    private Long id;
    private String name;
    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;
    //getters and setters
}
```

It is important to note that you should always test your JPA configurations and use appropriate indexes to improve performance.

CHAPTER 13: CODING

Java 8 Stream API coding questions:

Write a Program to find the duplicates in an array using stream API.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class DuplicateFinder {

  public static void main(String[] args) {
    int[] arr = {1, 2, 3, 1, 2, 4, 5};

    List<Integer> list = Arrays.stream(arr)
        .boxed()
        .collect(Collectors.toList());

    list.stream()
        .filter(i -> Collections.frequency(list, i) > 1)
        .distinct()
        .forEach(System.out::println);
  }
}
```

How to sort the employee list in ascending and descending order using java 8 streams API?

You can use the sorted method of the Stream API in Java 8 to sort a list of employees in ascending or descending order. By default, the sorted method sorts the elements in the stream in natural order, which means that it uses the elements' compareTo method if the elements are Comparable, or it throws a ClassCastException if they are not.

Here's an example of how you can sort a list of employees in ascending order based on their salary:

```
List<Employee> employees = getEmployeeList(); // get the list of employees
employees.stream()
        .sorted(Comparator.comparing(Employee::getSalary))
        .forEach(System.out::println);
```

And here's an example of how you can sort the same list in descending order based on the employee's name:

```
employees.stream()
        .sorted(Comparator.comparing(Employee::getName).reversed())
        .forEach(System.out::println);
```

Note that the reversed method on the comparator returns a comparator that gives the opposite ordering of the original comparator.

Find the highest salary of an employee from the HR department using Java stream API.

You have been given an employee list with EMP<Id, Name, Salary, Deptt>, and

You can use the filter and max methods of the Stream API in Java 8 to find the highest salary of an employee from the HR department. Here's an example:

```
List<Employee> employees = getEmployeeList(); // get the list of employees
Optional<Employee> highestPaidHrEmployee = employees.stream()
                                                .filter(e -> "HR".equals(e.getDeptt()))
```

```
                                        .max(Comparator.comparing(E
mployee::getSalary));

if (highestPaidHrEmployee.isPresent()) {
    System.out.println("The highest paid HR employee is: " +
highestPaidHrEmployee.get().getName());
} else {
    System.out.println("No HR employees found in the list.");
}
```

In this example, the filter method is used to select only the employees from the HR department, and the max method is used to find the employee with the highest salary. The max method returns an Optional object that may or may not contain the maximum value, depending on whether the stream is empty or not. So we use the isPresent method to check if a value was found before accessing it.

Find all employees who live in 'Pune' city, sort them by their name, and print the names of employees using Stream API.

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
public class Employee {
    private String name;
    private String city;

    public Employee(String name, String city) {
        this.name = name;
        this.city = city;
    }
    public String getName() {
        return name;
```

```
    }
    public String getCity() {
        return city;
    }
    public static void main(String[] args) {
        // Creating a list of employees
        List<Employee> employees = new ArrayList<>();
        employees.add(new Employee("John Smith", "New York"));
        employees.add(new Employee("Jane Doe", "Chicago"));
        employees.add(new Employee("Bob Johnson", "Pune"));
        employees.add(new Employee("Sarah Lee", "Pune"));

        // Filtering and sorting employees who live in Pune
        List<Employee> puneEmployees = employees.stream()
                .filter(e -> e.getCity().equals("Pune"))
                .sorted(Comparator.comparing(Employee::getName))
                .toList();
        // Printing the names of employees who live in Pune
        System.out.println("Employees who live in Pune:");
        puneEmployees.stream()
                .map(Employee::getName)
                .forEach(System.out::println);
    }
}
```

The code uses the filter method to filter out employees who live in Pune, then the sorted method to sort them by name, and finally, the map method to extract the names of the employees. The toList method is used to convert the filtered and sorted stream of employees to a list.

The output of the code would be:

Employees who live in Pune:
Bob Johnson
Sarah Lee

Find an average of even numbers using Java 8 stream API?

```
import java.util.Arrays;

public class AverageOfEvenNumbersExample {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        double average = Arrays.stream(numbers)
                .filter(n -> n % 2 == 0)
                .mapToDouble(n -> n)
                .average()
                .orElse(0.0);
        System.out.println("The average of even numbers is " +
average);
    }
}
```

In this example, we define an array number with some integers. We then use the Arrays.stream method to create a stream of integers from the array. We filter out the even numbers using the filter method, which takes a predicate that returns true for even numbers. We then use the mapToDouble method to convert the Stream<Integer> to an DoubleStream. We call the average method to get the average of the even numbers. If there are no even numbers in the array, we use the orElse method to return a default value of 0.0.

The output of this program for the array {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} would be:

The average of even numbers is 6.0

How to use sorting in Java-8?

In Java 8, you can use the sorted method of the Stream API to sort elements in a collection. By default, the sorted method sorts elements in their natural order, but you can also provide a Comparator to specify the sort order.

Here's an example of how you can sort a list of integers in ascending order using the sorted method:

```
List<Integer> numbers = Arrays.asList(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5);
numbers.stream()
        .sorted()
        .forEach(System.out::println);
```

And here's an example of how you can sort a list of strings in descending order based on their length:

```
List<String> words = Arrays.asList("apple", "banana", "cherry",
"date", "elderberry");
words.stream()
     .sorted(Comparator.comparingInt(String::length).reversed())
     .forEach(System.out::println);
```

In this example, we use the Comparator.comparingInt method to create a comparator that compares strings based on their length, and then we use the reversed method to reverse the sort order.

Note that the sorted method returns a new stream with the elements sorted in the specified order, and it does not modify the original stream or collection.

Write a program using stream API - Find the employee count in each department in the employee list?

```
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
```

```
public class Employee {
    private String name;
    private String department;
    public Employee(String name, String department) {
        this.name = name;
        this.department = department;
    }
    public String getName() {
        return name;
    }
    public String getDepartment() {
        return department;
    }
}
public class FindEmployeeCountByDepartment {
    public static void main(String[] args) {
        List<Employee> employees = Arrays.asList(
                new Employee("Alice", "Engineering"),
                new Employee("Bob", "Sales"),
                new Employee("Carol", "Engineering"),
                new Employee("Dave", "Marketing"),
                new Employee("Eve", "Sales")
        );
        // Create a stream of the employee list.
        Stream<Employee> employeeStream = employees.stream();
        // Group the employees by department.
        Map<String, Long> employeeCountByDepartment =
employeeStream
```

```
                .collect(Collectors.groupingBy(Employee::getDepartment
, Collectors.counting()));
        // Print the results.
        System.out.println(employeeCountByDepartment);
    }
}
```

Output : {Engineering=2, Sales=2, Marketing=1}

Find employees based on location or city and sort in alphabetical manner using stream API?

(like a-z and each city employee's salary should be sorted max to min salary)

```
import java.util.*;
import java.util.stream.Collectors;
public class EmployeeFilter {
    public static void main(String[] args) {
        List<Employee> employees = Arrays.asList(
                new Employee("John", "New York", 5000),
                new Employee("Jane", "New York", 6000),
                new Employee("Bob", "Chicago", 4500),
                new Employee("Alice", "Chicago", 5500),
                new Employee("Sam", "San Francisco", 7000),
                new Employee("Emily", "San Francisco", 6500)
        );
        String location = "Chicago";
        List<Employee> filteredEmployees = employees.stream()
                .filter(e -> e.getLocation().equals(location))
                .sorted(Comparator.comparing(Employee::getName))
```

```
                .sorted(Comparator.comparing(Employee::getSalary).rev
ersed())
                .collect(Collectors.toList());

        System.out.println("Filtered employees: " +
filteredEmployees);
    }

    static class Employee {
        private final String name;
        private final String location;
        private final int salary;

        public Employee(String name, String location, int salary) {
            this.name = name;
            this.location = location;
            this.salary = salary;
        }

        public String getName() {
            return name;
        }

        public String getLocation() {
            return location;
        }

        public int getSalary() {
            return salary;
        }
```

```
    @Override
    public String toString() {
        return "Employee{" +
                "name='" + name + '\'' +
                ", location='" + location + '\'' +
                ", salary=" + salary +
                '}';
    }
}
}
```

In this program, we first define a List of Employee objects with some elements, where each employee has a name, a location (city), and a salary.

We then define a location variable to filter the employees based on the given location.

We use a stream to filter the employees based on the given location using the filter() method, and then sort them in alphabetical order by name using the sorted() method with Comparator.comparing(Employee::getName). Finally, we sort each city employee's salary from highest to lowest using the sorted() method with Comparator.comparing(Employee::getSalary).reversed().

We collect the filtered and sorted employees into a list using the collect() method with Collectors.toList(), and print the result using the System.out.println() statement.

Filtered employees: [Employee{name='Alice', location='Chicago', salary=5500}, Employee{name='Bob', location='Chicago', salary=4500}]

Find the occurrence of names of employees from the List<Employee>, and find the frequency of each name.

```
import java.util.*;
public class EmployeeNameFrequency {
    public static void main(String[] args) {
        List<Employee> employees = Arrays.asList(
                new Employee("John", "New York", 5000),
                new Employee("Jane", "New York", 6000),
                new Employee("Bob", "Chicago", 4500),
                new Employee("Alice", "Chicago", 5500),
                new Employee("Sam", "San Francisco", 7000),
                new Employee("Emily", "San Francisco", 6500),
                new Employee("John", "Chicago", 5500),
                new Employee("Jane", "San Francisco", 6500),
                new Employee("Bob", "San Francisco", 7000)
        );

        Map<String, Integer> nameFrequencyMap = new
HashMap<>();
        for (Employee employee : employees) {
            String name = employee.getName();
            nameFrequencyMap.put(name,
nameFrequencyMap.getOrDefault(name, 0) + 1);
        }

        System.out.println("Name frequency: " +
nameFrequencyMap);
    }

    static class Employee {
```

```
        private final String name;
        private final String location;
        private final int salary;
        public Employee(String name, String location, int salary) {
            this.name = name;
            this.location = location;
            this.salary = salary;
        }

        public String getName() {
            return name;
        }
        public String getLocation() {
            return location;
        }
        public int getSalary() {
            return salary;
        }
        @Override
        public String toString() {
            return "Employee{" +
                    "name='" + name + '\'' +
                    ", location='" + location + '\'' +
                    ", salary=" + salary +
                    '}';
        }
    }
}
```

We then define a nameFrequencyMap map to store the frequency of each name.

We iterate over the employees using a for-each loop, and for each employee, we extract the name using the getName() method. We then put the name in the nameFrequencyMap map and increment its frequency using nameFrequencyMap.getOrDefault(name, 0) + 1.

Finally, we print the nameFrequencyMap map using the System.out.println() statement.

The output of the above program would be:

Name frequency: {Bob=2, Emily=1, Alice=1, Sam=1, Jane=2, John=2}

Write a Program to print only numbers from an alphanumeric char array using stream API in java-8.

```
import java.util.Arrays;
public class AlphanumericFilterExample {
    public static void main(String[] args) {
        String str = "a1b2c3d4e5f6g7h8i9j0";
        char[] arr = str.toCharArray();
        System.out.println("Original array: " + Arrays.toString(arr));
        int[] nums = new String(arr)
                .chars()
                .filter(Character::isDigit)
                .map(Character::getNumericValue)
                .toArray();
        System.out.println("Numbers only: " +
Arrays.toString(nums));
    }
}
```

character array. Then we create a stream from the characters of the string, filter only the numeric characters using the isDigit method of the Character class, convert the character to a numeric value using the getNumericValue method, and finally convert the result to an int array using the toArray method.

Original array: [a, 1, b, 2, c, 3, d, 4, e, 5, f, 6, g, 7, h, 8, i, 9, j, 0]

Numbers only: [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]

Write a program to find the sum of the entire array result using java 8 streams?

```
import java.util.Arrays;

public class ArraySum {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int sum = Arrays.stream(arr).sum();
        System.out.println("Sum of array elements: " + sum);
    }
}
```

In this program, we first define an integer array arr with some elements. Then we use the Arrays.stream() method to create a stream of integers from the array, and then we call the sum() method on the stream to find the sum of all the elements in the array.

Finally, we print the sum using the System.out.println() statement.

Write a program to find even numbers from a list of integers and multiply by 2 using stream java 8?

```
import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class EvenNumbers {
```

```
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9,
10);
        List<Integer> evenNumbersDoubled = numbers.stream()
                .filter(n -> n % 2 == 0)
                .map(n -> n * 2)
                .collect(Collectors.toList());
        System.out.println("Even numbers doubled: " +
evenNumbersDoubled);
    }
}
```

In this program, we first define a list of integers numbers with some elements. Then we create a stream of integers from the list using the stream() method on the list.

We then use the filter() method on the stream to filter out all the odd numbers, and then use the map() method to multiply each even number by 2.

Finally, we collect the result of the stream into a list using the collect() method with Collectors.toList() and print the result using the System.out.println() statement.

The output of the above program would be:

Even numbers doubled: [4, 8, 12, 16, 20]

Write a program to find the occurrence of each word in a given string in java?

```
import java.util.HashMap;
import java.util.Map;
public class WordCounter {
    public static void main(String[] args) {
        String str = "Hello world hello java world";
```

```
        Map<String, Integer> wordCounts = new HashMap<>();
        // Split the string into words
        String[] words = str.split(" ");
        // Count the occurrence of each word
        for (String word : words) {
            if (!wordCounts.containsKey(word)) {
                wordCounts.put(word, 1);
            } else {
                int count = wordCounts.get(word);
                wordCounts.put(word, count + 1);
            }
        }
        // Print the occurrence of each word
        for (String word : wordCounts.keySet()) {
            System.out.println(word + ": " + wordCounts.get(word));
        }
    }
}
```

The program takes a string as input and uses a HashMap to store the count of each word. It splits the string into words using the split method, and then iterates through each word, incrementing its count in the HashMap. Finally, it prints the occurrence of each word.

The output of the program for the input string "Hello world hello java world" would be:

world: 2

java: 1

Hello: 1

hello: 1

Write a Program to find a common element from three integer ArrayList. eg. arr1, arr2, and arr3.

```
import java.util.ArrayList;
import java.util.Arrays;
public class CommonElement {
    public static void main(String[] args) {
        ArrayList<Integer> arr1 = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));
        ArrayList<Integer> arr2 = new ArrayList<>(Arrays.asList(2, 4, 6, 8, 10));
        ArrayList<Integer> arr3 = new ArrayList<>(Arrays.asList(3, 5, 7, 9, 11));

        for (int num : arr1) {
            if (arr2.contains(num) && arr3.contains(num)) {
                System.out.println("Common element found: " + num);
            }
        }
    }
}
```

In this program, we first define three ArrayList of integers arr1, arr2, and arr3 with some elements.

We then iterate over the elements of the first list arr1 using a for-each loop, and for each element, we check if it is present in the other two lists arr2 and arr3 using the contains() method. If the element is present in both lists, we print it as a common element.

The output of the above program would be:

Common element found: 2

Common element found: 4

Common element found: 5

Write a program to convert string to integer in java without any API?

```
public static int stringToInt(String str) throws
NumberFormatException {
    int num = 0;
    int len = str.length();
    boolean negative = false;
    if (len == 0) {
        throw new NumberFormatException("Empty string");
    }
    int i = 0;
    char firstChar = str.charAt(0);
    if (firstChar == '-') {
        negative = true;
        i++;
    } else if (firstChar == '+') {
        i++;
    }
    for (; i < len; i++) {
        char ch = str.charAt(i);
        if (ch >= '0' && ch <= '9') {
            num = num * 10 + (ch - '0');
        } else {
            throw new NumberFormatException("Invalid character: " +
ch);
        }
    }
    return negative ? -num : num;
```

```
}
```

In this implementation, we first check if the input string is empty and throw a NumberFormatException if it is. Then we check if the first character is a sign indicator (+ or -) and set a flag negative accordingly. We iterate over the remaining characters of the string and compute the integer value by multiplying the previous value by 10 and adding the current digit. If we encounter an invalid character, we throw a NumberFormatException.

Here's an example usage of the method:

```
String str = "12345";
int num = stringToInt(str);
System.out.println(num);
```

Write a program to find the first occurrence of a character in a string in java?

```
public class FirstOccurrenceExample {
    public static void main(String[] args) {
        String str = "Hello World";
        char ch = 'o';
        int index = str.indexOf(ch);
        if (index == -1) {
            System.out.println("Character not found");
        } else {
            System.out.println("First occurrence of '" + ch + "' is at index " + index);
        }
    }
}
```

In this example, we define a string str and a character ch. We then use the indexOf method of the String class to find the first occurrence of the character in the string. If the character is not

found, the indexOf method returns -1. Otherwise, it returns the index of the first occurrence of the character.

The output of this program for the string "Hello World" and the character 'o' would be:

Write a program to find the missing number in an Array in java.

```
public class MissingNumberExample {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 6, 7, 8, 9, 10};
        int n = arr.length + 1;

        int expectedSum = (n * (n + 1)) / 2;
        int actualSum = 0;

        for (int i = 0; i < arr.length; i++) {
            actualSum += arr[i];
        }

        int missingNumber = expectedSum - actualSum;

        System.out.println("The missing number is " +
missingNumber);
    }
}
```

In this example, we define an array arr that has a missing number. We then calculate the length of the array plus 1, which gives us the number of elements in the sequence including the missing number. We calculate the sum of the first n natural numbers using the formula (n * (n + 1)) / 2, and store it in expectedSum. We calculate the sum of the elements in the array by iterating over the elements and adding them up, and store it in actualSum. We then subtract actualSum from expectedSum to get the missing number.

The output of this program for the array {1, 2, 3, 4, 6, 7, 8, 9, 10} would be:

The missing number is 5

Write a Program to Find a possible combination of the given string “GOD”?

```
public class StringCombinationExample {
    public static void main(String[] args) {
        String str = "GOD";
        int n = str.length();

        System.out.println("All possible combinations of the string \""
+ str + "\":");
        combinations("", str);
    }
    private static void combinations(String prefix, String str) {
        int n = str.length();
        if (n == 0) {
            System.out.println(prefix);
        } else {
            for (int i = 0; i < n; i++) {
                combinations(prefix + str.charAt(i), str.substring(0, i) +
str.substring(i + 1, n));
            }
        }
    }
}
```

In this example, we define a string str with the value "GOD". We then call the combinations method with an empty prefix and the string str. The combinations method takes a prefix and a string as

arguments. If the length of the string is 0, it prints the prefix. Otherwise, it recursively calls itself with each character of the string added to the prefix, and the character removed from the string.

The output of this program for the string "GOD" would be:

All possible combinations of the string "GOD":

GOD

GDO

OGD

ODG

DOG

DGO

Write a program for valid parenthesis in java?

keep track of opening and closing parentheses. The idea is to iterate through the string, push an opening parenthesis onto the stack, and pop a closing parenthesis off the stack whenever we encounter one. If the stack is empty when we encounter a closing parenthesis or there are leftover opening parentheses in the stack at the end of the iteration, then the string is not valid.

Here's an example program that checks for valid parentheses in Java using a stack:

```
import java.util.*;
public class ValidParentheses {
    public static boolean isValid(String s) {
        Stack<Character> stack = new Stack<>();
        for (char c : s.toCharArray()) {
            if (c == '(' || c == '{' || c == '[') {
                stack.push(c);
            } else if (c == ')' && !stack.isEmpty() && stack.peek() ==
'(') {
```

```
                stack.pop();
            } else if (c == '}' && !stack.isEmpty() && stack.peek() ==
'{') {
                stack.pop();
            } else if (c == ']' && !stack.isEmpty() && stack.peek() ==
'[') {
                stack.pop();
            } else {
                return false;
            }
        }
        return stack.isEmpty();
    }
    public static void main(String[] args) {
        String s1 = "()";
        String s2 = "()[]{}";
        String s3 = "(]";
        String s4 = "([)]";
        String s5 = "{[]}";
        System.out.println(s1 + " is valid: " + isValid(s1));
        System.out.println(s2 + " is valid: " + isValid(s2));
        System.out.println(s3 + " is valid: " + isValid(s3));
        System.out.println(s4 + " is valid: " + isValid(s4));
        System.out.println(s5 + " is valid: " + isValid(s5));
    }
}
```

indicating whether s contains a valid set of parentheses. We initialize a stack stack to keep track of opening parentheses. We then iterate

through each character c in the string s. If c is an opening parenthesis, we push it onto the stack. If c is a closing parenthesis, we check if the stack is not empty and the top of the stack contains the corresponding opening parenthesis. If both of these conditions are true, we pop the opening parenthesis from the stack. If c is not a valid parenthesis, we return false. After iterating through the entire string, we return true if the stack is empty (meaning we matched all opening parentheses with closing parentheses) or false otherwise.

The main method in this example demonstrates how to use the isValid method with some sample inputs. The output of running this program would be:

() is valid: true

()[]{} is valid: true

(] is valid: false

([)] is valid: false

{[]} is valid: true

Write a program to find duplicates in an ArrayList.

To find duplicates in an ArrayList in Java, you can create a HashSet to keep track of unique elements and a separate ArrayList to store the duplicates. The idea is to iterate through the elements of the ArrayList and add them to the HashSet. If an element is already in the HashSet, it is a duplicate and we add it to the duplicate ArrayList.

Here's an example program that finds duplicates in an ArrayList in Java:

```
import java.util.*;

public class FindDuplicates {
    public static List<Integer> findDuplicates(List<Integer> list) {
        Set<Integer> set = new HashSet<>();
        List<Integer> duplicates = new ArrayList<>();
```

```
        for (Integer i : list) {
            if (!set.add(i)) {
                duplicates.add(i);
            }
        }
        return duplicates;
    }
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(1, 2, 3, 1, 4, 2, 5);
        List<Integer> duplicates = findDuplicates(list);
        System.out.println("Duplicates: " + duplicates);
    }
}
```

Write a program for the Quick sort algorithm.

```
public class QuickSort {
    public static void quickSort(int[] array, int left, int right) {
        if (left < right) {
            int pivotIndex = partition(array, left, right);
            quickSort(array, left, pivotIndex - 1);
            quickSort(array, pivotIndex + 1, right);
        }
    }
    private static int partition(int[] array, int left, int right) {
        int pivot = array[right];
        int i = left - 1;
        for (int j = left; j < right; j++) {
            if (array[j] < pivot) {
```

```
                i++;
                swap(array, i, j);
            }
        }
        swap(array, i + 1, right);
        return i + 1;
    }
    private static void swap(int[] array, int i, int j) {
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

    public static void main(String[] args) {
        int[] array = {5, 2, 9, 1, 5, 6, 7};
        quickSort(array, 0, array.length - 1);
        for (int i : array) {
            System.out.print(i + " ");
        }
    }
}
```

Write a program to check the minimum number of occurrences of a character in a given string in java.

```
import java.util.Scanner;
public class MinimumOccurrence {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a string: ");
```

```
        String string = scanner.nextLine();
        System.out.print("Enter a character: ");
        char ch = scanner.nextLine().charAt(0);
        int minCount = Integer.MAX_VALUE;
        for (int i = 0; i < string.length(); i++) {
            if (string.charAt(i) == ch) {
                int count = 0;
                for (int j = i; j < string.length(); j++) {
                    if (string.charAt(j) == ch) {
                        count++;
                    }
                }
                if (count < minCount) {
                    minCount = count;
                }
            }
        }

        if (minCount == Integer.MAX_VALUE) {
            System.out.println("The character " + ch + " is not present in the string");
        } else {
            System.out.println("The minimum number of occurrences of " + ch + " in the string is " + minCount);
        }
    }
}
```

Here's how the program works:

First, we prompt the user to enter a string and a character using a Scanner.

We initialize an integer variable minCount to the maximum possible value, Integer.MAX_VALUE, to ensure that any occurrence of the character in the string will be smaller than this initial value.

We then loop through each character in the string, and if we encounter the specified character, we count the number of times it occurs in the string by looping through the string again starting from the current index. If the count is less than the current minimum count, we update minCount to the new count.

Finally, we check if minCount has been updated from its initial value. If it hasn't, we know that the specified character was not present in the string.

Example usage and output of the program:

Enter a string: hello world

Enter a character: l

The minimum number of occurrences of l in the string is 2

Enter a string: java programming

Enter a character: z

The character z is not present in the string

Write a program of an array, it must multiply the array, leaving itself aside, and that multiplication should be kept in that array position in Java.

```
public class ArrayMultiplication {
    public static void main(String[] args) {
        int[] arr = {2, 3, 4, 5, 6};
        int n = arr.length;
        // Initialize a new array to store the result
        int[] result = new int[n];
```

```
        // Compute the product of elements to the left of each
element
        int left_product = 1;
        for (int i = 0; i < n; i++) {
            result[i] = left_product;
            left_product *= arr[i];
        }
        // Compute the product of elements to the right of each
element
        int right_product = 1;
        for (int i = n-1; i >= 0; i--) {
            result[i] *= right_product;
            right_product *= arr[i];
        }
        // Print the result
        for (int i = 0; i < n; i++) {
            System.out.print(result[i] + " ");
        }
    }
}
```

Here's how the program works:

We start by initializing the input array and its length.

We then initialize a new array called result with the same length as the input array. This array will store the final result.

We then compute the product of all elements to the left of each element in the input array. We do this by maintaining a variable called left_product that keeps track of the product of all elements seen so far. We iterate through the input array from left to right,

storing the current value of left_product in the corresponding index of result, and updating left_product to include the current element.

Next, we compute the product of all elements to the right of each element in the input array. We do this in a similar way to the previous step, but iterating through the input array from right to left. We maintain a variable called right_product that keeps track of the product of all elements seen so far from the right side of the array. We multiply each element in result by right_product and update right_product to include the current element.

Finally, we print the result array.

Example output for the input array {2, 3, 4, 5, 6}:

Output - 180 120 90 72 60

The result shows that the product of all elements in the array except the first one is 3 x 4 x 5 x 6 = 360. Similarly, the product of all elements except the second one is 2 x 4 x 5 x 6 = 240, and so on. The output matches these products with the corresponding element removed.

Can you write down a Spring boot rest API for addition of two integers?

```
@RestController
public class AdditionController {
    @GetMapping("/addition/{a}/{b}")
    public int addition(@PathVariable int a, @PathVariable int b) {
        return a + b;
    }
}
```

This REST API creates a simple Spring MVC @RestController that listens for GET requests to the URL path /addition/{a}/{b} where {a} and {b} are placeholders for integer values. The @PathVariable annotation maps these values to the int parameters a and b, and the addition() method simply returns the sum of a and b. This API can

be easily extended to handle other HTTP methods or to perform more complex operations.

How to count every character in string using java 8?

To count every character in a string using Java 8, you can use the chars() method of the String class to create an IntStream of Unicode code points, and then use the boxed() method to convert it to a Stream<Integer>. Finally, you can use the collect() method to group the characters by their Unicode code points and count the occurrences of each character using the Collectors.groupingBy() and Collectors.counting() methods respectively. Here's an example code snippet:

```
String str = "Hello, world!";
Map<Integer, Long> charCount = str.chars()
        .boxed()
        .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));
charCount.forEach((k, v) -> System.out.println("Character '" + (char) k.intValue() + "' occurs " + v + " times."));
```

output:

```
Character ' ' occurs 1 times.
Character ',' occurs 1 times.
Character 'H' occurs 1 times.
Character 'e' occurs 1 times.
Character 'd' occurs 1 times.
Character 'l' occurs 3 times.
Character 'o' occurs 2 times.
Character 'r' occurs 1 times.
Character 'w' occurs 1 times.
Character '!' occurs 1 times.
```

In this example, the groupingBy() method groups the characters by their Unicode code points, and the counting() method counts the number of occurrences of each character. The resulting map charCount contains the counts for each character, which we can print out using a forEach() loop.

Check the unique String program?

I have a string "india" i have to check all the characters are unique or not, if they have unique characters print true if not then false using java

```
public static boolean hasUniqueChars(String str) {
    if (str == null || str.length() > 128) {
        return false;
    }
    boolean[] charSet = new boolean[128];
    for (int i = 0; i < str.length(); i++) {
        int val = str.charAt(i);

        if (charSet[val]) {
            return false;
        }
        charSet[val] = true;
    }
    return true;
}
```

Explanation:

We first check if the input string is null or if its length is greater than 128, which is the maximum number of unique ASCII characters. If either of these conditions is true, we return false.

We create a boolean array of size 128, which will be used to keep track of whether a character has been seen before or not.

We iterate through each character of the input string and convert it to its ASCII value. We check if the corresponding index in the boolean array is already true. If it is, it means that the character has been seen before and we return false. Otherwise, we mark the corresponding index in the boolean array as true.

If we have iterated through the entire string without finding any repeated characters, we return true.

Here's how you can call this function with the input string "india" and print the result:

```
String input = "india";

boolean hasUnique = hasUniqueChars(input);

System.out.println(hasUnique); // This will print false, since 'i' appears twice in the input string
```

Write a program to Find spikes in a stock of integer array?

```
import java.util.ArrayList;

import java.util.List;

public class FindSpikes {
  public static List<Integer> findSpikes(int[] stockPrices) {
    List<Integer> spikes = new ArrayList<>();
    for (int i = 1; i < stockPrices.length; i++) {
      if (stockPrices[i] - stockPrices[i - 1] > 10) {
        spikes.add(i);
      }
    }
    return spikes;
  }
  public static void main(String[] args) {
    int[] stockPrices = {100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200};
```

```
    List<Integer> spikes = findSpikes(stockPrices);
    System.out.println(spikes);
  }
}
```

Find the output of below program?

```
import java.util.HashMap;
import java.util.Map;
public class FindTheOutput {
    public static void main(String[] args) {
        Map<String,String> map =new HashMap<>();
        String e1 = new String("AJAY");
        String e2 = new String("AJAY");
        String e3 = new String("AJAY")
        map.put(e1,"I");
        map.put(e2,"M2");
        System.out.println(map.get(e1));
        System.out.println(map.get(e2));
    }
}
```

The output of the program is:

M2

M2

Explanation:

In Java, String literals are interned, meaning all String objects with the same content share the same memory space.

Therefore, e1, e2, and e3 refer to the same String object "AJAY" in the memory.

When adding entries to the map, the keys are compared based on their object identity, not just their content.

Since e1 and e2 are the same object, they map to the same value ("I" and "M2" respectively).

Therefore, the output is "M2" and "M2", even though they have the same content "AJAY".

CHAPTER 14: SCENARIO-BASED

Thread-Pool Based Scenario:

Scenario, you must create a cached thread pool using the executor framework, but you don't know the capacity or number of threads needed to achieve that task. how will you determine how many threads you will need based on your requirement? is there any mechanism in the executor's framework to know its capacity? (if you know the answers, please comment below so we can discuss a lot)?

Determining the number of threads needed for a cached thread pool can be a challenging task, as it depends on various factors such as the nature of the tasks, the resources they require, and the overall system performance.

Here are some general guidelines that can help you determine the number of threads needed for a cached thread pool:

Monitor system performance: Observing the CPU and memory usage of your system while running the tasks can give you an idea of the optimal number of threads needed to achieve maximum performance.

Analyse task requirements: If the tasks are CPU-bound, having a large number of threads can increase the CPU utilization, while if they are I/O-bound, having fewer threads can result in better performance.

Experiment: You can experiment with different numbers of threads and observe the performance to determine the optimal number for your system.

Start with a small number of threads: You can start with a small number of threads and increase it gradually until you find the optimal number.

Keep in mind that having too many threads can lead to thread context switching overhead and degrade performance, while having too few threads can limit the utilization of available resources.

In summary, finding the optimal number of threads for a cached thread pool is not a straightforward task and requires monitoring and experimentation to determine the best configuration for your system.

Rest-API & Database-based-scenario

How to build a restful web service that can fetch 1GB of data from the database and send it back without using Pagination, so the question is if you have a big size response how would you send it back from a rest web service?

Sending a large amount of data, such as 1GB, in a single request can have significant performance and scalability issues, such as increased memory usage, longer response times, and increased network bandwidth utilization.

One common solution to handle large data sets is to use pagination, where the data is split into smaller chunks and sent in multiple requests. However, if you are not able to use pagination, there are a few alternative solutions you can consider:

Compression: You can compress the data before sending it back in the response. This can reduce the amount of data sent over the network and improve the response time. You can use algorithms such as GZIP to compress the data.

Streaming: You can stream the data from the database directly to the response without storing it in memory. This can reduce the memory usage and allow you to handle large data sets more efficiently. You can use the Java API for Streaming XML (StAX) to stream the data.

Asynchronous Processing: You can implement asynchronous processing to fetch and send the data in the background. This can free up the main thread to handle other requests and improve the

overall performance and scalability of the system. You can use frameworks such as Spring's DeferredResult or Java's CompletableFuture to implement asynchronous processing.

Keep in mind that these solutions may not be suitable for all use cases and that the best solution depends on your specific requirements and constraints.

Database-based-scenario:1

How would you design a binary tree kind of data structure in database design? Basically, the interviewer wants to know how you would design a database in a hierarchical way.

A binary tree is a hierarchical data structure that can be easily modelled in a relational database. To design a binary tree in a database, you need to create a table to represent the nodes of the tree and a self-referencing foreign key to represent the parent-child relationships.

Here's an example of how you can design a binary tree in a relational database:

```
CREATE TABLE node (
  id INT PRIMARY KEY,
  data VARCHAR(255),
  parent_id INT,
  FOREIGN KEY (parent_id) REFERENCES node (id)
);
```

In this example, the node table has four columns: id, data, parent_id, and a self-referencing foreign key parent_id that refers to the id of the parent node. The root node of the tree will have a NULL value in the parent_id column, and the other nodes will have a reference to their parent node.

You can use SQL queries to traverse the tree and perform various operations, such as inserting, updating, and deleting nodes. You can also use recursion to traverse the tree and retrieve the data for all nodes in a specific order, such as pre-order, in-order, or post-order.

Database-based-scenario:2

How would you store millions of records in a table? How many tables does it require, any database pattern can you use here?

Storing millions of records in a single table can lead to performance and scalability issues, as the table size increases and the query response time becomes longer. To handle such a large number of records, you can use the following strategies:

Partitioning: Partitioning the table into smaller, more manageable chunks, based on a specific criterion, can improve the performance and scalability of the system. For example, you can partition the data by date, so that each partition contains data for a specific time range. You can use either horizontal partitioning, where the data is split across multiple tables, or vertical partitioning, where the data is split across multiple columns in the same table.

Sharding: Sharding is a method of distributing the data across multiple databases to increase the scalability and performance of the system. You can shard the data based on specific criteria, such as geographic location or user ID, so that each shard contains a subset of the data.

Denormalization: You can denormalize the data by duplicating data across multiple tables to reduce the number of joins required to retrieve the data. This can improve the performance of the queries and reduce the response time.

Indexing: Indexing the columns used in the queries can improve the query performance and reduce the response time. You can use either clustered or non-clustered indexes, depending on the specific requirements.

In addition, you can use database patterns, such as the Star Schema, to design the database and improve the performance and scalability of the system. The Star Schema is a data warehousing pattern that uses a central fact table to store the data and dimension tables to store the metadata. This pattern can improve the query performance and reduce the response time for large data sets.

In summary, storing millions of records in a single table requires careful planning and design, and you can use a combination of partitioning, sharding, denormalization, indexing, and database patterns to improve the performance and scalability of the system.

If one of the microservice is having high latency, how can you handle that, and in which direction can you think of to resolve this problem?

High latency in a microservice can lead to poor application performance and decreased user satisfaction. To handle high latency in a microservice, you can follow these steps:

Monitor the Latency: You need to monitor the latency of the microservice to identify the root cause of the problem. You can use tools like Application Performance Management (APM) software or log analysis tools to monitor the microservice.

Identify the Root Cause: You need to identify the root cause of the high latency. The root cause can be anything from slow database queries to resource constraints on the server. Once you identify the root cause, you can take the necessary steps to resolve it.

Optimize the Code: You can optimize the code to reduce the latency. For example, you can improve the algorithms used in the microservice, reduce the number of database queries, or improve the caching mechanism.

Scale the Microservice: You can scale the microservice to handle the increased load. You can scale the microservice horizontally, by adding more instances, or vertically, by increasing the resources of the existing instances.

Use Caching: You can use caching to reduce the latency. Caching can help reduce the load on the database and improve the response time of the microservice.

Load Balancing: You can use load balancing to distribute the load across multiple instances of the microservice. Load balancing can help improve the performance and availability of the microservice.

In conclusion, handling high latency in a microservice requires a combination of monitoring, optimization, scaling, caching, and load balancing. The specific solution depends on the root cause of the high latency, and you need to identify the root cause and implement the appropriate solution to resolve the issue.

Microservice-based-scenario

Service A is calling Service B, C, D. I want to log or handle specific conditions before calling B, C, and D but in a generic way. How can you handle this situation?

One way to handle this situation is to use the Aspect-Oriented Programming (AOP) technique. In AOP, you can define "aspects" that represent a set of cross-cutting concerns, such as logging or error handling, that can be applied to multiple points in your code in a modular and reusable way.

In your case, you can define an aspect to handle specific conditions before calling Service B, C, and D. The aspect can include code to log the conditions or perform error handling, and you can apply the aspect to the methods in Service A that call Service B, C, and D.

To implement this in Java, you can use a framework like Spring AOP. In Spring AOP, you can define aspects using annotations and apply them to your code using pointcuts. For example:

```
@Aspect
@Component
public class ServiceALogger {
  @Before("execution(* com.example.ServiceA.*(..))")
  public void logBefore(JoinPoint joinPoint) {
    // Log or handle specific conditions before calling B, C, and D
  }
}
```

In this example, the @Before annotation is used to define an aspect that will be executed before each method in the

com.example.ServiceA class. The logBefore method contains the code to log or handle specific conditions before calling Service B, C, and D.

By using AOP, you can centralize the handling of specific conditions in a single aspect and apply it to multiple points in your code in a generic way. This helps to make your code more maintainable and scalable.

Inheritance Scenario-Based

If a child class overrides the parent where the singleton pattern is implemented, then will it break the same? If Yes/No, why?

The Singleton pattern is used to ensure that a class has only one instance, and provides a global point of access to that instance. If a child class overrides the parent where the Singleton pattern is implemented, then it may break the Singleton pattern, depending on how the overriding is done.

If the child class simply inherits the parent's Singleton instance, and does not override any Singleton-specific methods or properties, then the Singleton pattern will still be maintained. The child class will have access to the same instance as the parent, and any modifications made to that instance in the parent or child class will be visible to both classes.

However, if the child class overrides the Singleton-specific methods or properties of the parent class, it may break the Singleton pattern. This is because the child class may create its own instance of the Singleton, or modify the existing Singleton instance in ways that are not compatible with the Singleton pattern. In this case, the child class may have its own Singleton instance, which is different from the Singleton instance used by the parent class and any other classes.

In order to ensure that the Singleton pattern is maintained in the child class, it is recommended to follow the same pattern as the parent class, and not override any Singleton-specific methods or

properties. If the child class needs to modify the Singleton instance, it should do so in a way that is compatible with the Singleton pattern, such as through a static method or property in the parent or child class.

How to create a custom HashMap of size 2GB?

It's not recommended to create a HashMap of size 2GB, as it would consume a large amount of memory and may cause the program to crash due to lack of memory. A HashMap is implemented using an array, and the size of the array is determined by the number of key-value pairs that need to be stored. When the number of key-value pairs exceeds the capacity of the array, the array is resized to prevent collisions.

If you really need a HashMap with a large size, there are a few things you can do to optimize the performance:

Increase the initial capacity of the HashMap: By setting the initial capacity to a higher value, you can reduce the number of times the array needs to be resized. However, this will increase the memory consumption.

HashMap<String, String> map = new HashMap<>(2000000000);
Increase the load factor: The load factor is a metric that determines when the HashMap should resize the array. By increasing the load factor, you can reduce the number of times the array needs to be resized. However, this will increase the risk of collisions.

HashMap<String, String> map = new HashMap<>(2000000000, 2);

Use a ConcurrentHashMap: A ConcurrentHashMap is a thread-safe alternative to HashMap that can handle a high number of concurrent operations.

ConcurrentHashMap<String, String> map = new ConcurrentHashMap<>(2000000000);

It's important to note that even with these optimizations, creating a HashMap with a size of 2GB may still cause the program to crash

due to lack of memory, and it is not recommended. Instead, you should consider using a more efficient data structure like a trie, a B-tree, or a Bloom filter, or using a database to store the data, they are more efficient and suitable to handle large data sets.

Design an application where you are getting millions of requests how will you design it.

(How will you design the application in which layer you will write what.)

Designing an application to handle millions of requests requires careful consideration of various factors such as scalability, performance, reliability, and security. Here are some steps to design such an application:

Use a load balancer: A load balancer distributes incoming requests across multiple servers to ensure that no single server gets overwhelmed. This distributes the load and provides redundancy and fault tolerance.

Choose the right architecture: Microservice, serverless, or monolithic architecture can all work depending on the specific use case. Microservices allow for more flexibility and can handle scale better but are more complex to manage. Serverless architecture can automatically scale to handle load but may have limitations on customization.

Use a distributed cache: A distributed cache can help reduce the load on the database by storing frequently accessed data in memory. This can speed up the application's response time and reduce the number of database queries.

Optimize database performance: Databases are often a bottleneck in highly scalable applications. To optimize database performance, use techniques like indexing, caching, and partitioning.

Implement asynchronous processing: By using asynchronous processing, the application can handle multiple requests at the same time, improving performance and scalability.

Implement caching: Implementing caching can help reduce the load on servers by serving frequently accessed content from a cache instead of generating it dynamically each time.

Use Content Delivery Network (CDN): CDN is a network of servers that can distribute content globally, delivering it from the closest server to the user, thus reducing the latency and improving the application's performance.

Use containers and orchestration tools: Containers like Docker can help to package applications and their dependencies, allowing them to be deployed and scaled quickly. Orchestration tools like Kubernetes or Docker Swarm can automate the deployment and management of containers.

Use a distributed file system: A distributed file system can help with scalability and redundancy by distributing files across multiple servers.

Monitor and optimize: Monitor the application's performance and usage patterns and optimize the infrastructure accordingly. Implementing logging, monitoring, and alerting can help detect issues and optimize performance in real-time.

Overall, designing an application to handle millions of requests requires a combination of techniques and tools to ensure scalability, performance, reliability, and security.

Suppose you have an application where the user wants the order history to be generated, and that history pdf generation take almost 15 minutes how will you optimise this solution. How this can be reduced.

Generating a PDF of order history that takes 15 minutes can be a frustrating experience for the user. Here are some ways to optimize this process and reduce the time it takes to generate the PDF:

Optimize the database queries: The slow generation of a PDF could be due to slow database queries. You can optimize the database queries by using indexing, caching, and partitioning. This will help

the queries execute faster, and the PDF generation time will be reduced.

Generate the PDF asynchronously: You can generate the PDF in the background while the user continues to use the application. This way, the user will not have to wait for the PDF to be generated. You can also notify the user when the PDF is ready to be downloaded.

Use a queueing system: Instead of generating the PDF immediately, you can put the request in a queue and generate it later. This way, the user will not have to wait, and the server can generate the PDF when it is free. You can use a queueing system like RabbitMQ or Apache Kafka for this purpose.

Use a caching system: You can cache the generated PDF and serve it to subsequent requests. This way, if the same user requests the same PDF, you can serve it from the cache, and the user will not have to wait for the PDF to be generated.

Optimize the PDF generation code: You can optimize the PDF generation code to make it more efficient. This may involve changing the libraries or tools you are using or optimizing the code itself.

Use a distributed system: You can distribute the PDF generation task across multiple servers to reduce the time it takes to generate the PDF. This is especially useful if you have a large number of users requesting the PDF.

Optimize the server: You can optimize the server to handle the load better. This may involve increasing the server's processing power, memory, or storage.

In summary, to reduce the time it takes to generate a PDF of order history, you can optimize the database queries, generate the PDF asynchronously, use a queueing system, use a caching system, optimize the PDF generation code, use a distributed system, and optimize the server. By implementing one or more of these solutions, you can significantly reduce the time it takes to generate the PDF and improve the user experience.

CHAPTER 15: JAVA FEATURES FROM JAVA 8 TILL JAVA 21 WITH EXAMPLE

In this chapter, we're going to look at how Java has changed and improved from Java 8 to the latest version, Java 21.

We'll cover what happened in Java since its update in 2014 to the most recent developments. Instead of just sticking to Java 8 topics, we'll explore the significant improvements and new tools introduced in the later versions.

This will help us answer interview questions more effectively. I've focused on the important features that matter in interviews, so let's jump in and see what's new in Java!

Java 8 Features

Features:

1.Lambda Expressions:

Enables functional programming by allowing the use of anonymous functions.

Concise syntax for writing functional interfaces.

2.Functional Interfaces:

An interface with a single abstract method, facilitating the use of lambda expressions.

@FunctionalInterface annotation to mark such interfaces.

3.Stream API:

Introduces a new abstraction called Stream for processing sequences of elements.

Supports functional-style operations on streams like filter, map, reduce, etc.

4.Default Methods:

Allows interfaces to have method implementations.

Helps in evolving interfaces without breaking existing implementations.

5.Method References:

Provides a shorthand notation for lambda expressions.

Allows referring to methods or constructors using the :: operator.

6.Optional class:

A container object that may or may not contain a non-null value.

Helps to handle null checks more effectively and avoids NullPointerExceptions.

7.New Date and Time API:

java.time package introduced for a more comprehensive and flexible API to handle dates and times.

Solves various issues with the old java.util.Date and java.util.Calendar classes.

8.Default Methods:

Allows interfaces to have method implementations.

Helps in evolving interfaces without breaking existing implementations.

9.Nashorn JavaScript Engine:

Replaces the old Rhino JavaScript engine.

Provides better performance and improved compatibility with modern JavaScript standards.

10.Parallel Streams:

Allows parallel processing of streams using the parallel() method.

Enhances performance on multi-core systems for certain types of operations.

11.Collectors:

Introduces a set of utility methods in the Collectors class for common reduction operations, such as toList(), toSet(), joining(), etc.

12.Functional Interfaces in java.util.function package:

New functional interfaces like Predicate, Function, Consumer, and Supplier to support lambda expressions.

Java 9 Features

1.Improved Process API:

Java 9 introduced enhancements to the Process API, providing better control over native processes. The new ProcessHandle class allows developers to interact with processes and obtain information about them.

```
// Using the ProcessHandle API to get information about the current process
public class ProcessHandleExample {
    public static void main(String[] args) {
        ProcessHandle currentProcess = ProcessHandle.current();
        System.out.println("Process ID: " + currentProcess.pid());
        System.out.println("Is alive? " + currentProcess.isAlive());
    }
}
```

2.Collection Factory Methods:

Java 9 added new static factory methods to the collection interfaces (List, Set, Map, etc.), making it more convenient to create immutable instances of these collections.

```
public class CollectionFactoryMethodsExample {
    public static void main(String[] args) {
        // Creating an immutable list using List.of() factory method
        List<String> colors = List.of("Red", "Green", "Blue");
```

```
        System.out.println(colors);
    }
}
```

3.Improved Stream API:

The Stream API was enhanced with several new methods, such as takeWhile, dropWhile, and ofNullable, which improve the flexibility and functionality of working with streams.

```
public class StreamAPIImprovementsExample {
    public static void main(String[] args) {
        // Example 1: takeWhile
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        List<Integer> lessThanFive = numbers.stream()
                .takeWhile(n -> n < 5)
                .collect(Collectors.toList());
        System.out.println("Numbers less than 5: " + lessThanFive);
        // Example 2: dropWhile
        List<Integer> greaterThanThree = numbers.stream()
                .dropWhile(n -> n <= 3)
                .collect(Collectors.toList());
        System.out.println("Numbers greater than 3: " + greaterThanThree);
        // Example 3: ofNullable
        List<String> names = Arrays.asList("Alice", "Bob", null, "Charlie", null, "David");
        List<String> nonNullNames = names.stream()
                .flatMap(name -> StreamAPIImprovementsExample.nullSafeStream(name))
```

```
                .collect(Collectors.toList());
        System.out.println("Non-null names: " + nonNullNames);
    }
    // Helper method to create a stream from a potentially null value
    private static <T> java.util.stream.Stream<T> nullSafeStream(T value) {
        return value == null ? java.util.stream.Stream.empty() : java.util.stream.Stream.of(value);
    }
}
```

In this example:

takeWhile is used to take elements from the stream until a certain condition is met (in this case, numbers less than 5).

dropWhile is used to drop elements from the stream while a certain condition is met (in this case, numbers less than or equal to 3).

ofNullable is used to create a stream from a potentially null value, filtering out null values (in this case, filtering out null names from the list).

4.Private Methods in Interfaces:

Interfaces in Java 9 can have private methods, allowing developers to encapsulate common functionality within an interface without exposing it to external classes.

```
// Interface with private method
public interface PrivateMethodInterface {
    default void publicMethod() {
        // Public method can call private method
        privateMethod();
    }

    private void privateMethod() {
        System.out.println("Private method in interface");
```

```
    }
}
```

5.HTTP/2 Client:

Java 9 introduced a new lightweight HTTP client that supports HTTP/2 and WebSocket. This client is designed to be more efficient and flexible than the old HttpURLConnection API.

```
public class HttpClientExample {
    public static void main(String[] args) throws Exception {
        HttpClient httpClient = HttpClient.newHttpClient();
        HttpRequest httpRequest = HttpRequest.newBuilder()
                .uri(new URI("https://www.example.com"))
                .GET()
                .build();

        HttpResponse<String> response =
httpClient.send(httpRequest,
HttpResponse.BodyHandlers.ofString());
        System.out.println("Response Code: " +
response.statusCode());
        System.out.println("Response Body: " + response.body());
    }
}
```

Java 10 Features

1.Local-Variable Type Inference (var):

Java 10 introduced the ability to use the var keyword for local variable type inference. This allows developers to declare local variables without explicitly specifying the type, letting the compiler infer it based on the assigned value.

```
public class LocalVarInference {
```

```
    /**
     * Allowed: only as a local variable
     * Not allowed: anywhere else (class field, method param, etc.)
     * User var responsibly!
     *
     * Use:
     *  - when it's clear what the type is (string, int)
     *  - to shorten very long ugly types
     *
     *  Don't use:
     *      - returned value is unclear (var data = service.getData();)
     */
    public static void main(String[] args) {
        // allowed, but brings little benefit
        var b = "b";
        var c = 5; // int
        var d = 5.0; // double
        var httpClient = HttpClient.newHttpClient();
        // one hell of an inference :)
        var list = List.of(1, 2.0, "3");
        // the benefit becomes more evident with types with long
names
        var reader = new BufferedReader(null);
        // vs.
        BufferedReader reader2 = new BufferedReader(null);
    }
}
```

Optional API — new methods introduced

```
public class OptionalApi {
    /**
     * new .orElseThrow()
     */
    public static void main(String[] args) {

        Optional<Flight> earliestFlight = FlightSchedule.getFlights()
                .stream()
                .filter(f -> "Boston".equals(f.from()))
                .filter(f -> "San Francisco".equals(f.to()))
                .min(comparing(Flight::date));

        earliestFlight.orElseThrow(FlightNotFoundException::new);
    }
}
```

Java 11 Features

1.HTTP client

Here is the program to explain the HTTP client changes,

```
public class HttpClientBasicExample {
    /**
     * Create a client, send a GET Request, print Response info
     */
    public static void main(String... args) throws Exception {
        HttpClient client = HttpClient.newHttpClient();

        HttpRequest request =
                HttpRequest.newBuilder(URI.create("https://github.com/
"))
                        .GET()  // default, may be omitted
```

```
                .build();
        HttpResponse<String> response =
                client.send(request,
HttpResponse.BodyHandlers.ofString());

        print("Status code was: " + response.statusCode());

        print(response.headers().map());
    }
}
```

2.New File Methods: Java 11 introduced several new methods in the java.nio.file package, providing additional functionality for working with files and directories. Some of the notable methods include:

Files.readString(Path path) and Files.writeString(Path path, CharSequence content, OpenOption... options):

These methods simplify reading and writing the contents of a file as a string. The readString method reads the entire content of a file into a string, and the writeString method writes a string to a file.

2. Files.readAllLines(Path path) and Files.write(Path path, Iterable<? extends CharSequence> lines, OpenOption... options):

These methods simplify reading and writing the contents of a file as a list of strings. The readAllLines method reads all lines from a file into a list, and the write method writes a collection of strings to a file.

3. Files.newBufferedReader(Path path) and Files.newBufferedWriter(Path path, OpenOption... options):

These methods create buffered readers and writers for efficient reading and writing of files. They simplify the process of working with character streams.

4.files.mismatch(Path path1, Path path2):

This method compares the content of two files and returns the position of the first mismatched byte. If the files are identical, it returns -1.

Here is the example,

```
public class NewFilesMethods {
    static String filePath = System.getProperty("user.dir") +
"/src/main/resources/";
    static String file_1 = filePath + "file_1.txt";
    /**
     * Files.readString() and .writeString()
     */
    public static void main(String[] args) throws IOException {
        // reading files is much easier now
        // not to be used with huge files
        Path path = Paths.get(file_1);
        String content = Files.readString(path);
        print(content);
        Path newFile = Paths.get(filePath + "newFile.txt");
        if(!Files.exists(newFile)) {
            Files.writeString(newFile, "some str",
StandardOpenOption.CREATE);
        } else {
            Files.writeString(newFile, "some str",
StandardOpenOption.TRUNCATE_EXISTING);
        }
    }
}
```

Java 12 Features

1.Compact Number Formatting:

Java 12 introduced a new feature called “Compact Number Formatting” as part of JEP 357. This enhancement provides a more concise way to format large numbers in a locale-specific manner.

The NumberFormat class in the java.text package was enhanced to support the new Style enum, including the Style.SHORT and Style.LONG constants. These styles can be used to format large numbers in a compact form based on the specified locale.

```
public class CompactNumberFormattingExample {
    public static void main(String[] args) {
        // Creating a number formatter with compact style
        NumberFormat compactFormatter = NumberFormat.getCompactNumberInstance(Locale.US, NumberFormat.Style.SHORT);
        // Formatting large numbers
        System.out.println("Short Format: " + compactFormatter.format(1000));  // Output: 1K
        System.out.println("Short Format: " + compactFormatter.format(1000000));  // Output: 1M

        // Creating a number formatter with compact style (long)
        NumberFormat compactLongFormatter = NumberFormat.getCompactNumberInstance(Locale.US, NumberFormat.Style.LONG);
        // Formatting large numbers in long style
        System.out.println("Long Format: " + compactLongFormatter.format(10000000));  // Output: 10 million
        System.out.println("Long Format: " + compactLongFormatter.format(1000000000));  // Output: 1 billion
    }
}
```

2.String::indent (JEP 326):

The String class in Java 12 introduced a new method called indent(int n). This method is used to adjust the indentation of each line in a string by a specified number of spaces.

```
String indentedString = "Hello\nWorld".indent(3);

// indentedString is now "   Hello\n   World"
```

3.New Methods in java.util.Arrays (JEP 326):

Java 12 added several new methods to the java.util.Arrays class, including copyOfRange and equals variants that take a Comparator.

4.Improvements in java.util.stream.Collectors (JEP 325):

The Collectors utility class in Java 12 introduced new collectors like teeing, which allows combining two collectors into a single collector.

5.New File Methods:

```
public class NewFilesMethod {
    static String filePath = System.getProperty("user.dir") +
"/src/main/resources/";
    static String file_1 = filePath + "file_1.txt";
    static String file_2 = filePath + "file_2.txt";
    public static void main(String[] args) throws IOException {

        // Finds and returns the position of the first mismatched byte
in the content of two files,
        // or -1L if there is no mismatch
        long result = Files.mismatch(Paths.get(file_1),
Paths.get(file_2));
        print(result);      // -1
    }
}
```

Java-13 Features

Nothing much interesting happend: — API update to ByteBuffer — Update to localization (support for new chars and emojis) — GC

updates

Java-14 Features

1.“Switch Expressions” (SE) instead of “Switch Statements” (SS):

Enhanced Switch Expressions:

Switch expressions, introduced as a preview feature in Java 12 and finalized in Java 13, allow developers to use switch statements as expressions, providing a more concise and expressive syntax.

```
int dayOfWeek = 2;
String dayType = switch (dayOfWeek) {
    case 1, 2, 3, 4, 5 -> "Weekday";
    case 6, 7 -> "Weekend";
    default -> throw new IllegalArgumentException("Invalid day of the week: " + dayOfWeek);
};
```

“Yield” Statement:

The “yield” statement was introduced in Java 14 to complement switch expressions. It allows you to specify a value to be returned from a switch arm, providing more flexibility in combining both imperative and functional styles.

```
String dayType = switch (dayOfWeek) {
    case 1, 2, 3, 4, 5 -> {
        System.out.println("Working day");
        yield "Weekday";
    }
    case 6, 7 -> {
        System.out.println("Weekend");
        yield "Weekend";
    }
```

```
    default -> throw new IllegalArgumentException("Invalid day of the week: " + dayOfWeek);
};
```

One More example,

```
/**
* "Switch Expressions" (SE) instead of "Switch Statements" (SS)
* (Both can be used, but SE is better than SS)
*/
public class SwitchExpressions {
    public static void main(String[] args) {
        oldStyleWithBreak(FruitType.APPLE);
        withSwitchExpression(FruitType.PEAR);
        switchExpressionWithReturn(FruitType.KIWI);
        switchWithYield(FruitType.PINEAPPLE);
    }
    // Old style is more verbose and error-prone (forgotten "break;" causes the switch to fall through)
    private static void oldStyleWithBreak(FruitType fruit) {
        print("==== Old style with break ====");
        switch (fruit) {
            case APPLE, PEAR:
                print("Common fruit");
                break;
            case PINEAPPLE, KIWI:
                print("Exotic fruit");
                break;
            default:
                print("Undefined fruit");
```

```
        }
    }
    private static void withSwitchExpression(FruitType fruit) {
        print("==== With switch expression ====");
        switch (fruit) {
            case APPLE, PEAR -> print("Common fruit");
            case PINEAPPLE -> print("Exotic fruit");
            default -> print("Undefined fruit");
        }
    }
    private static void switchExpressionWithReturn(FruitType fruit) {
        print("==== With return value ====");
        // or just "return switch" right away
        String text = switch (fruit) {
            case APPLE, PEAR -> "Common fruit";
            case PINEAPPLE -> "Exotic fruit";
            default -> "Undefined fruit";
        };
        print(text);
    }
    /**
     * "Yield" is like "return" but with an important difference:
     * "yield" returns a value and exits the switch statement.
Execution stays within the enclosing method
     * "return" exits the switch and the enclosing method
     */
    // https://stackoverflow.com/questions/58049131/what-does-the-
new-keyword-yield-mean-in-java-13
```

```
    private static void switchWithYield(FruitType fruit) {
        print("==== With yield ====");
        String text = switch (fruit) {
            case APPLE, PEAR -> {
                print("the given fruit was: " + fruit);
                yield "Common fruit";
            }
            case PINEAPPLE -> "Exotic fruit";
            default -> "Undefined fruit";
        };
        print(text);
    }
    public enum FruitType {APPLE, PEAR, PINEAPPLE, KIWI}
}
```

Java 15 Features:

1.Text-block:

Text blocks are a new kind of string literals that span multiple lines. They aim to simplify the task of writing and maintaining strings that span several lines of source code while avoiding escape sequences.

Example without text blocks:

```
String html = "<html>\n" +
              "    <body>\n" +
              "        <p>Hello, world</p>\n" +
              "    </body>\n" +
              "</html>";
```

Example with text blocks:

```
String html = """
```

```
<html>
    <body>
        <p>Hello, world</p>
    </body>
</html>
""";
```

Key features of text blocks include:

Multiline Strings: Text blocks allow you to represent multiline strings more naturally, improving code readability.

Whitespace Control: Leading and trailing whitespaces on each line are removed, providing better control over the indentation.

Escape Sequences: Escape sequences are still valid within text blocks, allowing the inclusion of special characters.

Text blocks were designed to make it easier to express strings that include multiple lines of content, such as HTML, XML, JSON, or SQL queries. If there have been any updates or new features related to text blocks in Java 15 or subsequent releases, it's advisable to check the official documentation or release notes for the specific version.

```
/**
* Use cases for TextBlocks (What's New in Java 15 > Text Blocks in Practice)
* - Blocks of text using markdown
* - Testing, defining hard-coded JSON strings
* - Simple templating
*/
public class TextBlocks {
    public static void main(String[] args) {
        oldStyle();
        emptyBlock();
```

```
    jsonBlock();
    jsonMovedEndQuoteBlock();
    jsonMovedBracketsBlock();
}
private static void oldStyle() {
    print("******** Old style ********");
    String text = "{\n" +
            "  \"name\": \"John Doe\",\n" +
            "  \"age\": 45,\n" +
            "  \"address\": \"Doe Street, 23, Java Town\"\n" +
            "}";
    print(text);
}
private static void emptyBlock() {
    print("******** Empty Block ********");
    String text = """
            """;
    print("|" + text + "|");
}
private static void jsonBlock() {
    print("******** JSON Block ********");
    String text = """
            {
              "name": "John Doe",
              "age": 45,
              "address": "Doe Street, 23, Java Town"
            }
```

```
            """; // <-- no indentation if char is aligned with first "
        print(text);
    }
    private static void jsonMovedEndQuoteBlock() {
        print("******** Json Moved End Quote Block ********");
        String text = """
                {
                  "name": "John Doe",
                  "age": 45,
                  "address": "Doe Street, 23, Java Town"
                }
                    """;
        print(text);
    }
    private static void jsonMovedBracketsBlock() {
        print("******** Json Moved Brackets Block ********");
        String text = """
                {
                  "name": "John Doe",
                  "age": 45,
                  "address": "Doe Street, 23, Java Town"
                }
               """; // <-- indented by 2 spaces as it is aligned with
third "
        print(text);
    }
}
```

Java 16 Features

1.Pattern matching for instanceof:

Java 16’s pattern matching for instanceof is a nifty feature that improves type checking and extraction. Here's a rundown of its key aspects:

What it does:

Introduces type patterns instead of just checking against a single type.

Allows declaring a variable within the instanceof check to hold the extracted object.

Combines type checking and casting into a single, more concise and readable expression.

Benefits:

Reduced boilerplate: Eliminates the need for separate instanceof checks, casts, and variable declarations.

Improved readability: Makes code clearer and easier to understand, especially for complex type hierarchies.

Reduced errors: Less chance of casting exceptions due to mistaken types.

Syntax:

```
if (obj instanceof String s) {
  // Use "s" directly as a String here
} else if (obj instanceof List<Integer> list) {
  // Use "list" directly as a List<Integer> here
} else {
  // Handle other cases
}
```

Example

```
public class PatternMatchingForInstanceof {
```

```
    public static void main(String[] args) {
        Object o = new Book("Harry Potter", Set.of("Jon Doe"));
        // old way
        if (o instanceof Book) {
            Book book = (Book) o;
            print("The book's author(s) are " + book.getAuthors());
        }

        // new way
        if (o instanceof Book book) {
            print("The book's author(s) are " + book.getAuthors());
        }
    }
}
```

Record:

Records in Java are a special type of class specifically designed for holding immutable data. They help reduce boilerplate code and improve readability and maintainability when dealing with simple data structures.

Here’s a breakdown of their key characteristics:

1. Conciseness:

Unlike traditional classes, records require minimal code to define. You just specify the data fields (components) in the record declaration, and the compiler automatically generates essential methods like:

Constructor with parameters for each component.

Getters for each component.

equals and hashCode methods based on component values.

toString method representing the record's state.

2. Immutability:

Record fields are declared as final, making the data stored within them unmodifiable after the record is created. This ensures data consistency and simplifies thread safety concerns.

3. Readability:

The auto-generated methods and predictable behavior of records enhance code clarity and make it easier to understand what the record represents and how it interacts with other parts of your program.

4. Reduced Errors:

By minimizing boilerplate, records reduce the risk of common mistakes like forgetting getters or implementing equals incorrectly. This leads to more robust and reliable code.

Overall, records are a valuable tool for Java developers to create concise, immutable, and readable data structures, leading to cleaner, more maintainable code.

```
/**
* Record are data-only immutable classes (thus have specific use cases)
* They are a restricted (specialized) form of a class (like enums)
* Not suitable for objects that are meant to change state, etc.
* <p>
* Records are NOT:
* - Boilerplate reduction mechanism
* <p>
* Records generate constructors, getters, fields; equals, hashCode, toString
* <p>
* Use cases:
* - to model immutable data
```

```
* - to hold read-only data in memory
* - DTOs - Data Transfer Objects
*/
public class RecordsDemo {
    public static void main(String[] args) {
        Product p1 = new Product("milk", 50);
        Product p2 = new Product("milk", 50);
        print(p1.price()); // without "get" prefix
        print(p1);         // auto-generated toString() - Product[name=milk, price=50]
        print(p1 == p2);       // false    - different objects
        print(p1.equals(p2));  // true     - values of fields (milk, 50) are compared by the auto-generated equals()/hashCode()
    }
}

/**
* params are called "Components"
* want more fields - must add into the signature
* Extending not allowed, implementing interfaces IS allowed
*/
public record Product(String name, int price) {
    // static fields allowed, but not non-static
    static String country = "US";
    // constructor with all fields is generated
    // can add validation
    public Product {
        if(price < 0) {
```

```
            throw new IllegalArgumentException();
        }
    }
    // possible to override auto-generated methods like toString()
}
```

2.Date Time Formatter API:

General usage and features of the DateTimeFormatter API in Java 16: This includes understanding format patterns, creating custom formats, parsing dates and times, and available formatting options.

New features introduced in Java 16 for date formatting: Specifically, the day period support using the "B" symbol and its various styles.

Comparison of DateTimeFormatter with older formatters like SimpleDateFormat: Exploring the advantages and disadvantages of each approach.

Examples of using DateTimeFormatter for specific formatting tasks: Like formatting dates in different locales, handling time zones, or generating human-readable representations.

```
public class DateTimeFormatterApi {
    static Map<TextStyle, Locale> map = Map.of(
            TextStyle.FULL, Locale.US,
            TextStyle.SHORT, Locale.FRENCH,
            TextStyle.NARROW, Locale.GERMAN
    );
    public static void main(String[] args) {
        for (var entry : map.entrySet()) {
            LocalDateTime now = LocalDateTime.now();
            DateTimeFormatter formatter = new
DateTimeFormatterBuilder()
                    .appendPattern("yyyy-MM-dd hh:mm ")
```

```
                    .appendDayPeriodText(entry.getKey())    // at night,
du soir, abends, etc.
                    .toFormatter(entry.getValue());
            String formattedDateTime = now.format(formatter);
            print(formattedDateTime);
        }
    }
}
```

3.Changes in Stream API:

Java 16 brought some exciting changes to the Stream API, making it even more powerful and convenient to use. Here are the key highlights:

1. Stream.toList() method: This new method provides a concise way to collect the elements of a stream into a List. Previously, you had to use collect(Collectors.toList()), which is now slightly redundant.

2. Stream.mapMulti() method: This method allows you to map each element of a stream to zero or more elements, creating a new stream of the resulting elements. It's handy for splitting or flattening complex data structures.

3. Enhanced line terminator handling: Java 16 clarifies the definition of line terminators in the java.io.LineNumberReader class. This eliminates inconsistencies and ensures consistent behavior when reading line-based data.

4. Other minor changes:

String streams now support the limit and skip methods directly, removing the need for intermediate operations.

The peek method can now be used with parallel streams, allowing side effects without impacting parallelism.

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

```
        List<Integer> ints = Stream.of(1, 2, 3)
                .filter(n -> n < 3)
                .toList();  // new, instead of the verbose
.collect(Collectors.toList())

        ints.forEach(System.out::println);
    }
}
```

Java 17 Features

1.Sealed classes(Subclassing):

Sealed classes are a brand-new feature introduced in Java 17 (JEP 409) that gives you more control over inheritance hierarchies. They essentially let you restrict which classes can extend or implement your class or interface. This can be incredibly useful for a variety of reasons, including:

1. Enhanced Type Safety: By specifying allowed subclasses, you prevent unexpected or unwanted extensions that could break your code or introduce security vulnerabilities.

2. Improved Library Design: You can create closed ecosystems within your libraries, ensuring users only work with approved extensions and don't create incompatible implementations.

3. Easier Code Maintenance: Knowing the exact set of possible subclasses simplifies reasoning about your code and makes it easier to understand and maintain.

How do sealed classes work?

You declare a class or interface as sealed using the sealed keyword. Then, you use the permits clause to specify a list of classes that are allowed to extend or implement it. Only these permitted classes can directly inherit, while all other classes are prohibited.

```
sealed class Shape {
  permits Circle, Square, Triangle;
```

```
  // ... implementation details
}
class Circle extends Shape {
  // ...
}
// This will cause a compile-time error because Rectangle isn't permitted
class Rectangle extends Shape {
  // ...
}
```

A sealed class or interface can be extended or implemented only by those classes and interfaces permitted to do so.

Benefits:

1) help enforce a well-defined and limited set of possible implementations — communicates INTENTION

2) Better security — help prevent unexpected or unauthorized subclassing and behavior from third-party code

Rules:

1. A sealed class uses “permits” to allow other classes to subclass it.

2. A child class MUST either be final, sealed or non-sealed. (or code won’t compile)

3. A permitted child class MUST extend the parent sealed class. Permitting without using the permit is now allowed.

4. The classes specified by permits must be located near the superclass:

either in the same module (if the superclass is in a named module) (see Java 9 modularity)

or in the same package (if the superclass is in the unnamed module).

More on point 4:

The motivation is that a sealed class and its (direct) subclasses are tightly coupled since they must be compiled and maintained together.

In a modular world, this means "same module"; in a non-modular world, the best approximation for this is "same package".

If you use modules, you get some additional flexibility, because of the safety boundaries modules give you.

Java 18 Features

1.UTF-8 by Default:

Java 18 makes UTF-8 the default character encoding for the platform, aligning with modern standards and simplifying character handling.

```
public class Utf8ByDefault {
    // https://openjdk.org/jeps/400 - Platform Default Encoding
    public static void main(String[] args) throws IOException {
        // Problem:
        // 1) On Windows, use the below FileWriter to write characters outside the ASCII table, e.g. some exotic Unicode chars, without explicitly specifying a char set
        // 2) Copy or transfer the file to a UNIX-based OS, like a Mac, and read the file using the default char encoding of the system
        // 3) Likely result - garbled output
        // Hence the problem - unpredictable behavior.
        FileWriter writer = new FileWriter("out.txt");
        // Solution before Java 18: always specify the charset, (and good luck not forgetting it!)
        FileWriter writer2 = new FileWriter("out.txt", StandardCharsets.UTF_8);
```

```
        // Solution since Java 18: UTF-8 is now default, so no need to specify the Char set


    }
}
```

2.Simple Web Server:

This new API provides a basic web server for serving static files, ideal for quick prototyping and embedded applications.

Ensure you have Java 18 or later installed on your system.

Have your static files (HTML, CSS, JavaScript, images, etc.) ready in a specific directory.

```
public class SimpleWebServer {
    public static void main(String[] args) throws Exception {
        String documentRoot = "/path/to/your/static/files";  // Replace with your actual directory
        int port = 8080;  // You can change the port if needed
        HttpServer server = HttpServer.create(new InetSocketAddress(port), 0);
        SimpleFileServer fileServer = new SimpleFileServer(documentRoot);
        server.setExecutor(null);  // Use single-threaded executor
        server.createContext("/", fileServer);
        server.start();
        System.out.println("Server started on port " + port);
    }
}
<!DOCTYPE html>
<html>
```

<head>

<title>Title of the document</title>

</head>

<body>

This page can be served with Java's Simple Web Server using the "jwebserver" command

</body>

</html>

. Compile and run:

Compile the Java file: javac SimpleWebServer.java

Run the compiled class: java SimpleWebServer

4. Access the page:

Open your web browser and navigate to http://localhost:8080 (or the specified port).

You should see the default file (usually index.html) from your static files directory being served.

5. Stopping the server:

To stop the server, press Ctrl+C in the terminal where it's running.

The simplest way to get started:

1) open the terminal in this package (java18)

2) run "java -version" and make sure it is at least java 18

3) run the "jwebserver" command

It should output:

Binding to loopback by default. For all interfaces use "-b 0.0.0.0" or "-b ::".

Serving path/to/your/subdirectory and subdirectories on 127.0.0.1 port 8000

URL http://127.0.0.1:8000/

The HTML page is now served at:
http://127.0.0.1:8000/java18/doc.html

IP address, port and other parameters may be changed

3.HEAD() convenience method added:

```
public class HttpHeadDemo {
    /**
     * HEAD() convenience method added
     */
    public static void main(String[] args) throws IOException,
InterruptedException {
        HttpRequest head =
HttpRequest.newBuilder(URI.create("https://api.github.com/"))
                .HEAD()
                .build();
        var response = HttpClient.newHttpClient().send(head,
HttpResponse.BodyHandlers.ofString());
        print(response);
    }
}
```

Reimplement Core Reflection with Method Handles: This reimplementation aims to improve performance and stability of reflection functionalities.

Deprecate Finalization for Removal: Finalization, intended for resource cleanup, has inherent drawbacks. Its deprecation paves the way for safer and more reliable alternatives.

Java 19 Features

Either preview or incubator features:

Oracle Releases Java 19

New release delivers seven JDK Enhancement Proposals to increase developer productivity, improve the Java language, and…

www.oracle.com

Java 20 Features

Either preview or incubator features:

Oracle Releases Java 20

New release delivers seven JDK Enhancement Proposals to increase developer productivity, improve the Java language, and…

www.oracle.com

Java 21 Features

1.Virtual Threads:

this feature introduces lightweight threads that run on top of the operating system threads, aiming to simplify concurrent programming and improve performance for certain workloads.

Here's a simple demo showcasing virtual threads in Java 21:

Create Virtual Threads:

```
Thread vThread1 = Thread.ofVirtual().start(() -> {
  for (int i = 0; i < 10; i++) {
    System.out.println("Virtual Thread 1: " + i);
  }
});
Thread vThread2 = Thread.ofVirtual().start(() -> {
  for (int i = 0; i < 10; i++) {
    System.out.println("Virtual Thread 2: " + i);
  }
});
```

2. Wait for Completion:

```
vThread1.join();
```

vThread2.join();

Output:

This will interleave the outputs from both virtual threads, demonstrating concurrent execution without the overhead of full OS threads. You might see something like:

Virtual Thread 1: 0

Virtual Thread 2: 0

Virtual Thread 1: 1

Virtual Thread 2: 1

...

Virtual Thread 1: 9

Virtual Thread 2: 9

Virtual Thread Explanation:

Virtual threads are lightweight units of execution that run on top of a smaller pool of underlying OS threads. They offer several advantages:

Lighter weight: Compared to OS threads, virtual threads have significantly lower creation and context switching costs.

Improved concurrency: More virtual threads can be efficiently managed within a limited number of OS threads, allowing better utilization of resources for certain workloads.

Simpler concurrency programming: Virtual threads eliminate the need for complex thread management and synchronization, making concurrent programming easier for developers.

The following is an example of virtual threads and a good contrast to OS/platform threads. The program uses the ExecutorService to create 10,000 tasks and waits for all of them to be completed. Behind the scenes, the JDK will run this on a limited number of carrier and OS threads, providing you with the durability to write concurrent code with ease.

try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {

```
    IntStream.range(0, 10_000).forEach(i -> {
        executor.submit(() -> {
            Thread.sleep(Duration.ofSeconds(1));
            return i;
        });
    });
}  // executor.close() is called implicitly, and waits
```

Record Patterns (Project Amber):

Records were introduced as a preview in Java 14, which also gave us Java enums. record is another special type in Java, and its purpose is to ease the process of developing classes that act as data carriers only.

In JDK 21, record patterns and type patterns can be nested to enable a declarative and composable form of data navigation and processing.

```
// To create a record:
Public record Todo(String title, boolean completed){}
// To create an Object:
Todo t = new Todo("Learn Java 21", false);
```

Before JDK 21, the entire record would need to be deconstructed to retrieve accessors.. However, now it is much more simplified to get the values. For example:

```
static void printTodo(Object obj) {
    if (obj instanceof Todo(String title, boolean completed)) {
        System.out.print(title);
System.out.print(completed);
    }
}
```

The other advantage of record patterns is also nested records and accessing them. An example from the JEP definition itself shows the ability to get to the Point values, which are part of ColoredPoint, which is nested in a Rectangle. This makes it way more useful than before, when all the records needed to be deconstructed every time.

```
// As of Java 21
static void printColorOfUpperLeftPoint(Rectangle r) {
    if (r instanceof Rectangle(ColoredPoint(Point p, Color c),
                               ColoredPoint lr)) {
        System.out.println(c);
    }
}
```

2.Sequenced collections:

In JDK 21, a new set of collection interfaces are introduced to enhance the experience of using collections. For example, if one needs to get a reverse order of elements from a collection, depending on which collection is in use, it can be tedious. There can be inconsistencies retrieving the encounter order depending on which collection is being used; for example, SortedSet implements one, but HashSet doesn't, making it cumbersome to achieve this on different data sets.

To fix this, the SequencedCollection interface aids the encounter order by adding a reverse method as well as the ability to get the first and the last elements. Furthermore, there are also SequencedMap and SequencedSet interfaces.

```
interface SequencedCollection<E> extends Collection<E> {
    // new method
    SequencedCollection<E> reversed();
    // methods promoted from Deque
    void addFirst(E);
    void addLast(E);
```

```
    E getFirst();
    E getLast();
    E removeFirst();
    E removeLast();
}
```

3.String templates:

String templates are a preview feature in JDK 21. However, it attempts to bring more reliability and better experience to String manipulation to avoid common pitfalls that can sometimes lead to undesirable results, such as injections. Now you can write template expressions and render them out in a String.

```
// As of Java 21
String name = "Ajay"
String greeting = "Hello \{name}";
System.out.println(greeting);
```

In this case, the second line is the expression, and upon invoking, it should render Hello Ajay. Furthermore, in cases where there is a chance of illegal Strings—for example, SQL statements or HTML that can cause security issues—the template rules only allow escaped quotes and no illegal entities in HTML documents.

CHAPTER 16: KAFKA

Explain the producer & consumer-based architecture of Kafka.

Kafka is a distributed streaming platform that is designed to handle high-throughput, real-time data feeds. The architecture of Kafka is based on the producer and consumer model, which allows for efficient data ingestion and consumption.

In the producer and consumer model of Kafka, data is produced by one or more producers and sent to Kafka, which acts as a centralized data hub. The data is then consumed by one or more consumers, which can be part of the same or different applications.

Here's how the producer and consumer-based architecture of Kafka works:

Producers: Producers are applications or components that generate data and send it to Kafka. Producers publish data to Kafka topics, which act as channels or streams that can be subscribed to by consumers. Producers can send data in batches or one record at a time, and can also control the partitioning and keying of the data.

Kafka Broker: Kafka brokers are the core components of the Kafka system. They store and manage the data that is produced by the producers, and make it available for consumption by the consumers. Kafka brokers are distributed and can be scaled horizontally to handle large amounts of data.

Topics: Topics are logical channels or streams of data in Kafka. Each topic is partitioned into one or more partitions, and each partition is replicated across multiple Kafka brokers for fault tolerance. Producers publish data to topics, and consumers subscribe to one or more topics to receive the data.

Consumers: Consumers are applications or components that read and process data from Kafka. Consumers can be part of the same or

different applications as the producers, and can consume data in real-time or in batches. Consumers can also control the offset of the data they consume, allowing them to start from a specific point in a topic.

Consumer Groups: Consumers can be organized into consumer groups, which allows for parallel processing of data. Each consumer group has one or more consumers that consume data from one or more partitions of a topic. The partitions are assigned to the consumers in the group dynamically, based on the workload and availability of the consumers.

Overall, the producer and consumer-based architecture of Kafka provides a scalable and fault-tolerant way of processing large amounts of data in real-time. It allows for decoupling of data producers and consumers, and provides parallel processing and fault-tolerance through partitioning and replication.

How to persist data directly from Kafka topic.is it possible or not?

Yes, it is possible to persist data directly from a Kafka topic. There are a few ways to do this, depending on your use case and requirements.

One way to persist data from a Kafka topic is to use a Kafka consumer to read data from the topic and write it to a database or file system. This can be done using a Kafka consumer application that reads data from the topic and writes it to a file or database.

Another way to persist data from a Kafka topic is to use Kafka Connect, which is a framework for streaming data between Kafka and other data systems. Kafka Connect can be used to move data from a Kafka topic to a database or other storage system, and can also be used to move data from a database or other storage system to a Kafka topic.

Kafka Connect provides pre-built connectors for popular data storage systems like HDFS, Amazon S3, and Elasticsearch, as well as connectors for JDBC-compliant databases. Kafka Connect can be

configured to read data from a Kafka topic and write it to a database or other storage system in real-time, providing a way to persist data from Kafka directly to storage.

Additionally, some databases have their own Kafka connectors that allow you to persist data directly to the database from a Kafka topic. For example, Confluent provides a Kafka connector for PostgreSQL that can be used to write data from Kafka to a PostgreSQL database.

In summary, it is possible to persist data directly from a Kafka topic using a Kafka consumer, Kafka Connect, or a database-specific Kafka connector. The best approach depends on your specific use case and requirements.

What is offset in Kafka?

If any consumer fails or crashed and then comes alive after some time, then can it continue consuming the messages?

In Kafka, an offset is a unique identifier that represents the position of a consumer within a partition of a topic. The offset is used to track the progress of a consumer, allowing it to continue reading messages from where it left off, even if it fails or crashes.

When a consumer reads messages from a Kafka topic, it keeps track of the last offset it has consumed for each partition it is subscribed to. This offset is periodically committed to a special topic called the "consumer_offsets" topic, which is used to persist the offset information for all consumer groups.

If a consumer fails or crashes, and then comes alive after some time, it can continue consuming messages from the last committed offset for each partition it is subscribed to. When the consumer restarts, it retrieves the last committed offset for each partition from the "__consumer_offsets" topic and starts consuming messages from that point.

Kafka provides two types of offset management: automatic and manual. With automatic offset management, the consumer's offset is automatically committed to the "__consumer_offsets" topic at regular intervals, or when a batch of messages has been processed.

With manual offset management, the consumer is responsible for explicitly committing the offset after processing a batch of messages.

In summary, offsets are used in Kafka to track the progress of a consumer and allow it to continue reading messages from where it left off, even if it fails or crashes. This provides fault tolerance and enables real-time data processing in distributed systems.

What is consumer offset?

In Kafka, the "consumer_offsets" topic is a special internal topic that is used to store the committed offsets of a consumer group. The consumer group is a group of one or more consumers that work together to consume messages from one or more partitions of a topic.

Each consumer in a consumer group keeps track of its own offset, which represents the position of the last message it has processed in each partition it is consuming. These offsets are periodically committed to the "consumer_offsets" topic, which serves as a centralized store for the committed offsets of all consumers in the group.

By using the "consumer_offsets" topic, Kafka enables consumers to resume reading from their last known offset, even if they have been restarted, moved to a different machine, or joined/removed from the consumer group. This provides fault tolerance and scalability, allowing consumers to consume messages in parallel and distribute the load across multiple machines.

The "consumer_offsets" topic is managed by the Kafka broker and is replicated across all brokers in the cluster to ensure availability and reliability. It is also subject to retention policies and cleanup rules, which determine how long the offset information should be retained and when it should be deleted.

In summary, the "consumer_offsets" topic in Kafka is a special internal topic that stores the committed offsets of a consumer group. It enables consumers to resume reading from their last known

offset, provides fault tolerance and scalability, and is managed by the Kafka broker.

How to configure the Kafka details?

Configuring Kafka involves setting up various components and parameters, such as broker settings, topic settings, producer settings, and consumer settings. Here are some of the steps to configure Kafka:

Install Kafka: First, you need to download and install Kafka. You can download Kafka from the Apache Kafka website or from a cloud provider like Confluent or Amazon Web Services (AWS).

Start ZooKeeper: Kafka uses Apache ZooKeeper to manage the brokers and the distributed state of the cluster. You need to start ZooKeeper first before starting Kafka. You can start ZooKeeper using the command bin/zookeeper-server-start.sh config/zookeeper.properties.

Configure Kafka broker settings: The broker is the main component of Kafka that stores and manages messages in the topics. You need to configure various settings for the broker, such as the broker ID, port, log directories, and replication factors. The broker settings are defined in the config/server.properties file.

Create a Kafka topic: A Kafka topic is a category or feed name to which messages are published. You need to create a topic before producing and consuming messages. You can create a topic using the bin/kafka-topics.sh script.

Configure Kafka producer settings: The producer is the component that produces messages to Kafka topics. You can configure various settings for the producer, such as the topic name, compression type, and batch size. The producer settings are defined in the producer code or in a configuration file.

Configure Kafka consumer settings: The consumer is the component that consumes messages from Kafka topics. You can configure various settings for the consumer, such as the topic name, group ID,

auto commit interval, and offset reset policy. The consumer settings are defined in the consumer code or in a configuration file.

Start Kafka: Once you have configured the necessary components and parameters, you can start Kafka using the command bin/kafka-server-start.sh config/server.properties.

These are the basic steps to configure Kafka. Depending on your use case and requirements, you may need to configure additional settings or use additional components, such as Kafka Connect, Kafka Streams, or Kafka Schema Registry.

How to determine the replication factor?

The replication factor is an important configuration parameter in Kafka that determines how many copies of each partition should be maintained across the brokers in a Kafka cluster. Choosing the right replication factor is critical for ensuring data availability and fault tolerance in the event of a broker failure or network outage.

Here are some guidelines for determining the appropriate replication factor for your Kafka deployment:

Consider the number of available brokers: The replication factor should not exceed the number of available brokers in your Kafka cluster. For example, if you have three brokers, you can set the replication factor to 2 or 3, but not 4.

Consider the desired level of fault tolerance: A higher replication factor provides greater fault tolerance and availability, but also increases the amount of storage and network traffic required to maintain the replicas. You should balance the desired level of fault tolerance with the cost and complexity of maintaining additional replicas.

Consider the throughput and latency requirements: A higher replication factor can also impact the throughput and latency of your Kafka cluster. Additional replicas can increase the network traffic and introduce additional processing overhead, which can affect the performance of the cluster.

Consider the retention period of data: If you need to retain data for a long period of time, you may want to increase the replication factor to ensure that the data is not lost in the event of a broker failure or network outage.

In general, a replication factor of 2 or 3 is recommended for most Kafka deployments. This provides a good balance between fault tolerance, storage requirements, and performance. However, you should adjust the replication factor based on your specific requirements and the characteristics of your Kafka cluster.

Which annotation is used to enable Kafka?

In Spring Boot, the @EnableKafka annotation is used to enable Kafka support. This annotation is typically placed on a configuration class and it will register a KafkaListenerContainerFactory and a KafkaTemplate bean.

For example, you can create a configuration class like this:

```
@Configuration
@EnableKafka
public class KafkaConfig {
    @Bean
    public ProducerFactory<String, String> producerFactory() {
        Map<String, Object> config = new HashMap<>();
        config.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");
        config.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        config.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFI
G, StringSerializer.class);
        return new DefaultKafkaProducerFactory<>(config);
    }
    @Bean
```

```
    public KafkaTemplate<String, String> kafkaTemplate() {
        return new KafkaTemplate<>(producerFactory());
    }
}
```

This configuration class will enable the Kafka support in your Spring Boot application and it also creates a KafkaTemplate bean that you can use to send messages to a Kafka topic.

Also, you should have kafka and zookeeper running on your local machine or on the server.

CHAPTER 17: MISCELLANEOUS

KUBERNETS/DOCKER:

What is the difference between a container and a virtual machine?

Containers and virtual machines (VMs) are both ways to run applications in isolated environments, but they differ in their approach, level of isolation, and resource usage.

A container is a lightweight, portable environment that packages an application and its dependencies into a single unit. Containers share the same operating system kernel and are isolated from other containers and the host operating system. Containers provide a consistent and reproducible way to deploy and run applications, while also allowing for efficient resource usage and fast start-up times.

In contrast, a virtual machine is a complete operating system and hardware abstraction that runs on top of a hypervisor. VMs have their own virtualized hardware, such as CPU, memory, storage, and network interfaces, and they can run their own operating system and applications. VMs provide strong isolation and security, as each VM runs independently of the host operating system and other VMs. However, this also means that they require more resources and have slower startup times.

Here are some key differences between containers and virtual machines:

Overhead: Containers have lower overhead, as they share the same operating system kernel and can be deployed more efficiently. VMs have higher overhead, as they require a full operating system and virtualized hardware.

Resource usage: Containers are more efficient in their use of resources; as multiple containers can run on the same operating

system kernel. VMs require more resources, as each VM has its own virtualized hardware and operating system.

Portability: Containers are more portable, as they can be moved easily between different environments that support the container runtime. VMs can be more challenging to move between different virtualization platforms or between physical servers.

Isolation: Containers provide weaker isolation, as they share the same operating system kernel and resources. VMs provide stronger isolation, as each VM has its own operating system and virtualized hardware.

Start-up time: Containers start up much faster than VMs, as they don't need to boot up an entire operating system.

In summary, containers and virtual machines are both useful tools for running applications in isolated environments, but they have different strengths and weaknesses. Containers are more lightweight, efficient, and portable, while VMs provide stronger isolation and security but require more resources. The choice between the two will depend on the specific requirements of the application and the environment in which it will be deployed.

Differences between Dockerization and Virtualisation?

Dockerization and virtualization are both ways to run software applications in isolated environments, but they differ in their approach and level of isolation.

Virtualization involves running a complete operating system on top of a virtual machine (VM) hypervisor. This allows multiple VMs to run on a single physical server, each with its own operating system and resources. Each VM can host its own applications and dependencies, and these are isolated from the host operating system and other VMs on the same physical server.

Dockerization, on the other hand, involves running applications in containers, which are lightweight and portable environments that share the host operating system kernel. Containers allow developers to package an application and its dependencies into a single unit

that can run on any machine that has Docker installed. Each container is isolated from other containers and from the host operating system, but they all share the same kernel.

Here are some key differences between Dockerization and Virtualization:

Overhead: Virtualization requires more overhead, as it runs a complete operating system on top of the hypervisor. Dockerization has less overhead, as it only runs the application and its dependencies in the container.

Resource usage: Virtualization typically uses more resources, as each VM has its own operating system and resources. Dockerization is more efficient in its use of resources, as multiple containers can run on the same operating system kernel.

Portability: Dockerization provides greater portability, as containers can be moved easily between different environments that have Docker installed. Virtualization requires more effort to move VMs between different virtualization platforms or between physical servers.

Isolation: Virtualization provides stronger isolation, as each VM has its own operating system and resources. Dockerization provides weaker isolation, as containers share the host operating system kernel and resources.

Startup time: Docker containers start up much faster than virtual machines, as they don't need to boot up an entire operating system.

In summary, Dockerization is a more lightweight and efficient way to run applications in an isolated environment, whereas virtualization provides stronger isolation but requires more overhead and resources. The choice between the two will depend on the specific requirements of the application and the environment in which it will be deployed.

What is a pod in Kubernetes?

In Kubernetes, a pod is the smallest and simplest unit in the Kubernetes object model. It represents a single instance of a

running process in a cluster. A pod can contain one or more containers that share the same network namespace and are scheduled together on the same node.

A pod is a logical host for one or more containers, and it provides a shared environment for those containers to run in. Containers within a pod can communicate with each other using local hostnames and ports, and they can share the same storage volumes.

Pods are designed to be ephemeral, meaning that they can be created, scaled, and destroyed dynamically in response to changes in demand or failure conditions. When a pod is created, Kubernetes assigns it a unique IP address and hostname, and it schedules the pod to run on a specific node in the cluster. The pod remains on that node until it is deleted or rescheduled by Kubernetes.

Pods are usually not deployed directly in Kubernetes, but rather as part of a higher-level deployment or replica set. These higher-level objects define the desired state of the pods and manage their creation, scaling, and termination.

Pods are an important abstraction in Kubernetes, as they enable the deployment and management of containerized applications in a consistent and scalable way. They provide a unit of deployment and scaling that is easy to manage and automate, while also providing isolation and resource constraints for running containers.

Can we write j-units for static methods?

Yes, it is possible to write JUnit tests for static methods.

JUnit is a testing framework that is commonly used in Java applications to write and run automated tests. One of the features of JUnit is the ability to write tests for static methods.

To write a JUnit test for a static method, you can use the @Test annotation and call the static method directly from the test method

JUNIT/ UNIT Testing

How to resolve this Mockito exception “Mockito cannot mock this class”?

The "Mockito cannot mock this class" exception can occur when trying to mock a class that Mockito cannot create a mock for. Mockito can only mock classes that are non-final and have a visible constructor.

Here are a few reasons why you might be getting this exception:

The class is final: Mockito cannot create mocks of final classes. You can either remove the "final" modifier from the class, or use a different mocking library that supports mocking final classes, such as PowerMock or JMockit.

The class or constructor is private: Mockito needs to be able to create an instance of the class using a visible constructor. If the constructor is private, you can use the PowerMockito library to mock the constructor.

The class is a primitive or a final class from the java.lang package: Mockito cannot mock primitives or final classes from the java.lang package, such as String or Integer. You can use a real instance of these classes or a test double like a spy instead of a mock.

The class is loaded by a different class loader: If the class is loaded by a different class loader than the test class, Mockito may not be able to create a mock of the class. You can try adding the class to the test classpath or use a different mocking library that supports mocking classes loaded by different class loaders.

The class is an interface: If the class is an interface, you should use the Mockito.mock() method instead of Mockito.mock(class).

In general, if you are getting the "Mockito cannot mock this class" exception, it is a sign that you may need to refactor your code or your test in order to make it more testable. You may also want to consider using a different mocking library that supports mocking the specific class or scenario you are working with.

What is binary search tree?

A binary search tree is a type of binary tree in which each node has at most two child nodes, and each node's value is greater than or equal to all the values in its left subtree and less than or equal to all the values in its right subtree.

Binary search trees are useful data structures for storing and searching large sets of data efficiently. They are often used in computer science applications such as database indexing, file system organization, and network routing algorithms.

Inserting a new value into a binary search tree involves traversing the tree from the root node to a leaf node, comparing the value to be inserted with the value of each node along the way, and choosing the appropriate child node to continue the traversal. Searching for a value in a binary search tree follows a similar process, starting at the root node and traversing the tree until the desired value is found or it is determined that the value is not in the tree.

The efficiency of binary search trees is determined by their height, or the number of levels in the tree. A well-balanced binary search tree has a height of log(n), where n is the number of nodes in the tree. This allows for efficient searching and insertion operations, as the number of nodes that need to be visited is minimized. However, an unbalanced binary search tree can have a worst-case height of n, making operations much less efficient. Therefore, algorithms for balancing binary search trees, such as AVL trees and Red-Black trees, have been developed to ensure efficient performance.