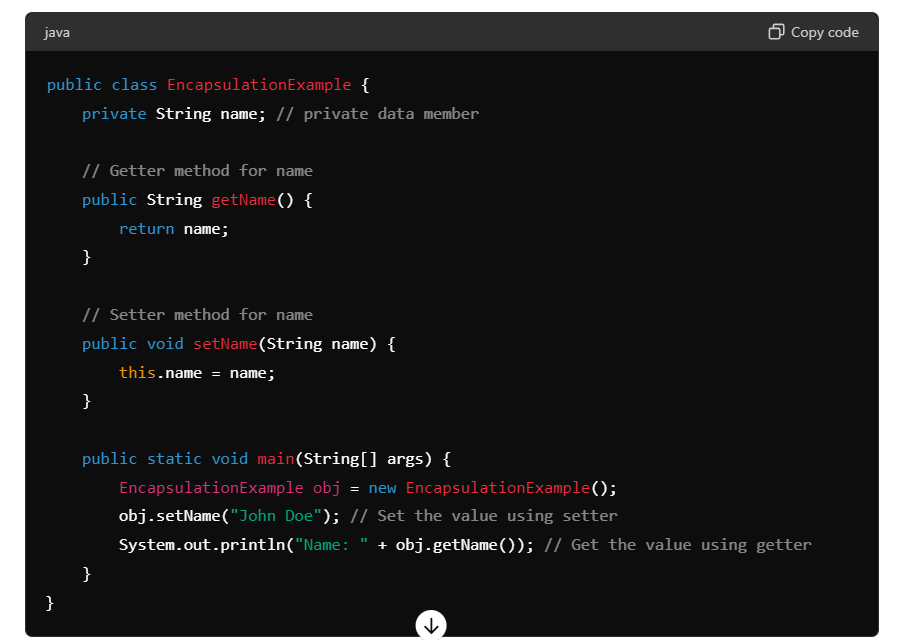
Q 1 :- oops concept in java

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of objects, which contain data and methods. Java is a widely-used programming language that supports OOP principles. Here are the four main OOP concepts in Java:

1. **Encapsulation**
2. **Inheritance**
3. **Polymorphism**
4. **Abstraction**

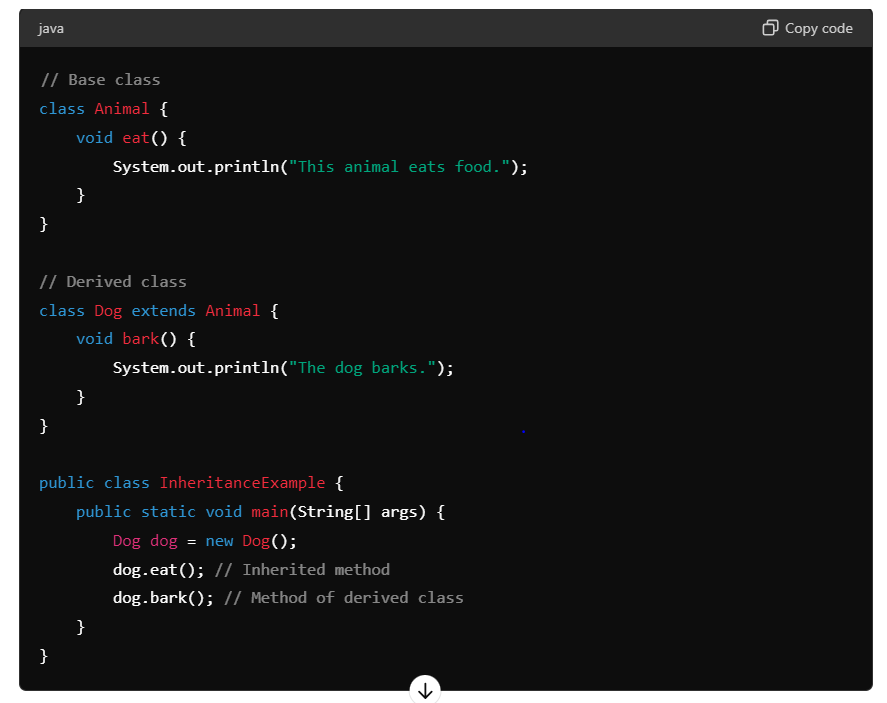
**1. Encapsulation**

Encapsulation is the concept of wrapping data (variables) and methods (functions) together as a single unit. In Java, this is achieved using classes, and access to the data can be restricted using access modifiers (private, protected, public).



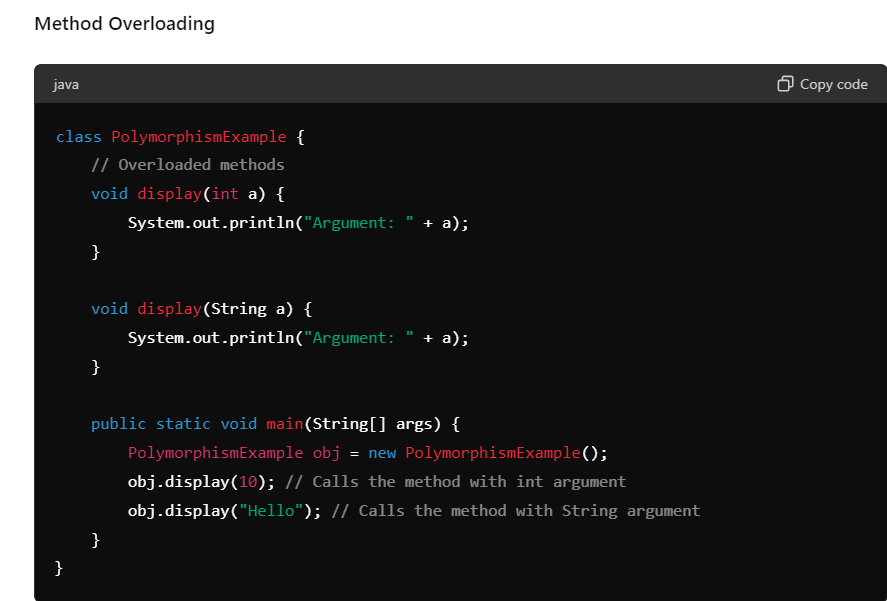
### 2. Inheritance

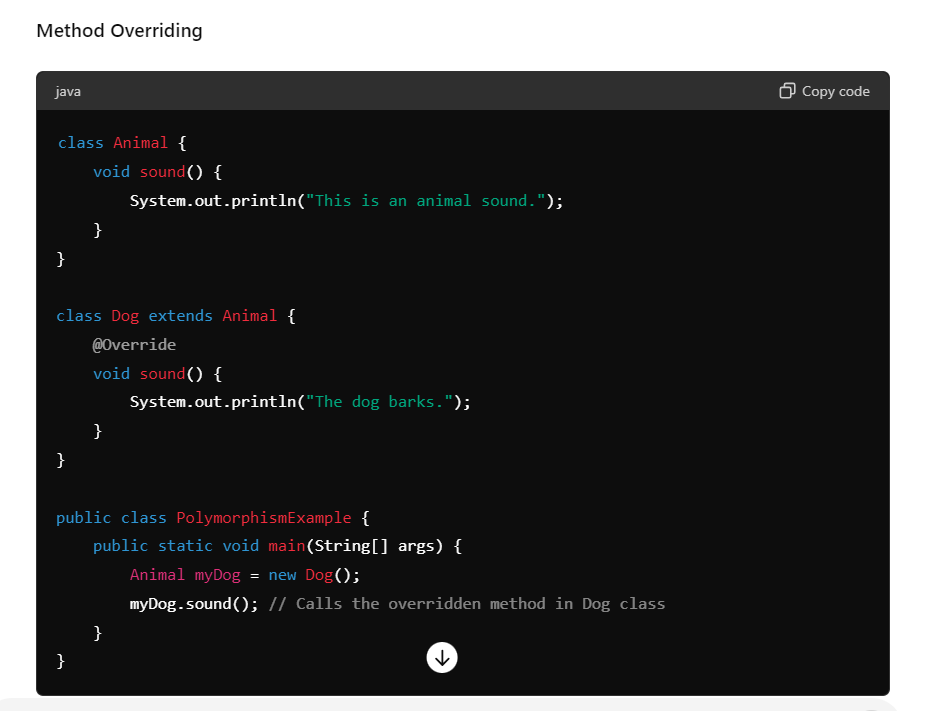
Inheritance is a mechanism wherein a new class is derived from an existing class. The derived class inherits all the properties and behaviors (methods) of the base class, allowing for reusability and hierarchical classification.



### 3. Polymorphism

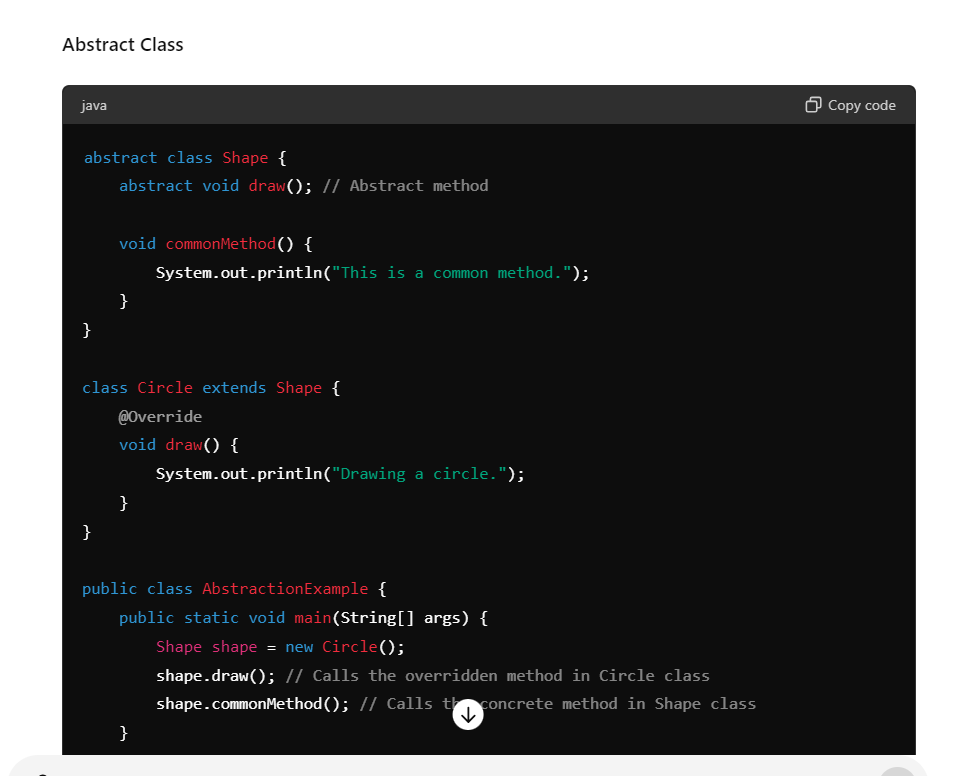
Polymorphism means "many shapes" and allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation. In Java, this can be achieved through method overloading (compile-time polymorphism) and method overriding (runtime polymorphism).

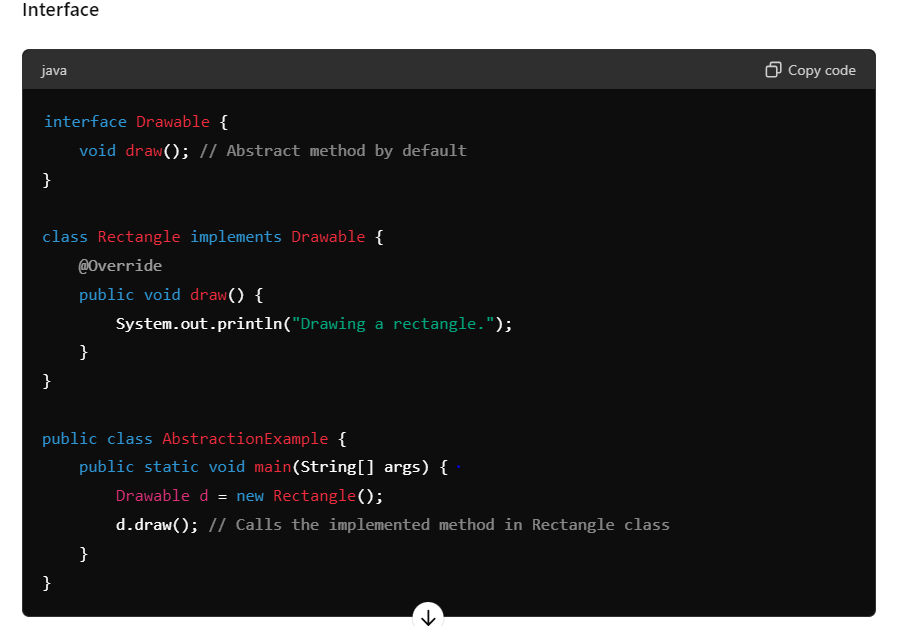




### 4. Abstraction

Abstraction is the concept of hiding the complex implementation details and showing only the necessary features of an object. In Java, abstraction can be achieved using abstract classes and interfaces.





These are the key concepts of OOP in Java, which help in creating modular, reusable, and maintainable code.

Q2 : why string is immutable ?

In Java, the String class is immutable, meaning that once a String object is created, its value cannot be changed. Here are the primary reasons why String is designed to be immutable:

### 1. ****Security****

Strings are widely used in Java applications, especially in contexts involving sensitive data like passwords, usernames, database connection URLs, and network connections. Making strings immutable ensures that such data cannot be altered once created, which helps to prevent security vulnerabilities. For example, if a String representing a database connection URL were mutable, any malicious code could alter it and redirect database operations to a different server.

### 2. ****Thread Safety****

Immutability makes strings inherently thread-safe, as multiple threads can access the same String instance without any synchronization. This eliminates the need for synchronization in concurrent applications, reducing the complexity and potential performance bottlenecks associated with thread synchronization.

### 3. ****Caching and Performance****

String immutability allows for caching and sharing string instances. The Java String Pool takes advantage of this property to store common strings. When a new string is created, the JVM first checks the pool to see if an identical string already exists. If it does, the reference to the existing string is returned, instead of creating a new object. This reduces memory usage and improves performance.

### 4. ****Hash Code Consistency****

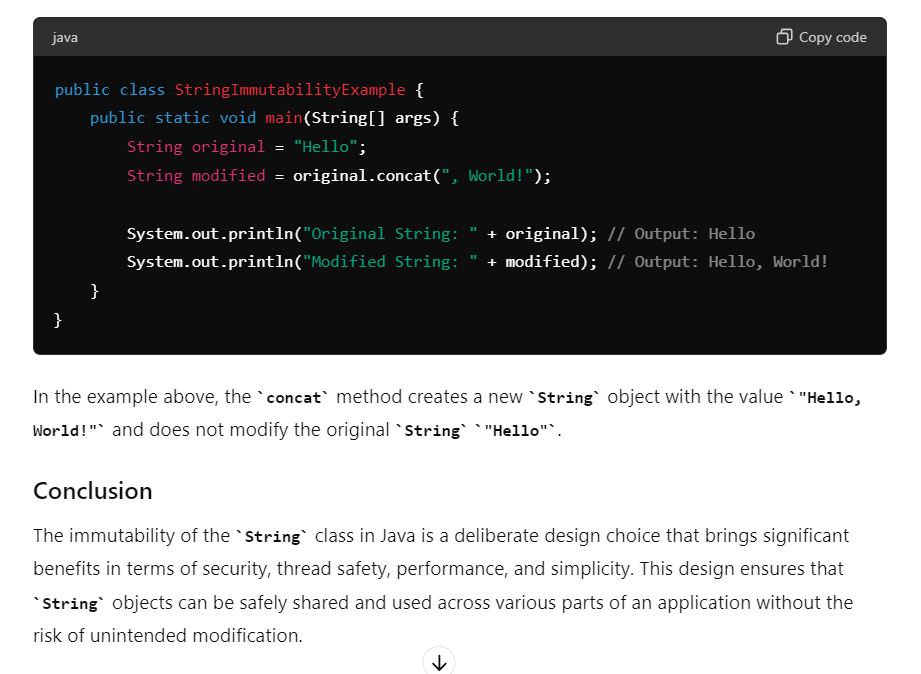
Strings are frequently used as keys in hash-based collections like HashMap, HashSet, and Hashtable. Immutability guarantees that the hash code of a String remains constant, ensuring the reliability and consistency of such collections. If strings were mutable, their hash codes could change, leading to data corruption and unpredictable behavior in hash-based collections.

### 5. ****Design Simplicity****

Immutable objects are simpler to design, implement, and use. They do not require defensive copying and do not lead to issues related to mutable states. This simplicity translates to fewer bugs and more readable code.

### Example Illustrating Immutability

When you modify a string, you actually create a new String object rather than altering the existing one:



Question 3:- Immutable class example :---

An immutable class in Java is a class whose instances cannot be modified after they are created. Once an object of an immutable class is instantiated, its state (i.e., its fields) cannot be changed. Immutable objects are inherently thread-safe and do not require synchronization.

**Key Characteristics of Immutable Classes:**

1. **Final Class:**
   * The class is declared as final to prevent subclassing, which could potentially alter the immutable nature.
2. **Final Fields:**
   * All fields are declared final so that their values can be assigned only once.
3. **Private Fields:**
   * Fields are declared private to prevent direct access and modification from outside the class.
4. **No Setter Methods:**
   * The class does not provide setter methods for its fields.
5. **Constructor:**
   * A constructor is provided to initialize all the fields of the class.
6. **Defensive Copies:**
   * If the class has fields that are mutable objects, it should return a copy of the object rather than the object itself. Similarly, if the class accepts mutable objects, it should store a copy rather than the original object.

**Example of an Immutable Class:**



**Explanation:**

1. **Final Class:**
   * public final class ImmutableClass prevents the class from being subclassed.
2. **Final Fields:**
   * The fields name, age, and birthDate are declared as final.
3. **Private Fields:**
   * All fields are private, ensuring they cannot be accessed directly.
4. **No Setter Methods:**
   * No setter methods are provided, so the fields cannot be changed after object creation.
5. **Constructor:**
   * The constructor initializes all the fields. For the Date field, a new Date object is created to avoid the mutable reference being shared.
6. **Defensive Copies:**
   * The getBirthDate method returns a new Date object to prevent the internal birthDate from being modified.

**Advantages of Immutable Classes:**

* **Thread Safety:** Since the state of immutable objects cannot be changed after creation, they are naturally thread-safe and do not require synchronization.
* **Simplicity:** Immutable objects are simple to construct, test, and use.
* **Cache-Friendly:** Immutable objects can be safely shared and cached.

Immutable classes provide a robust way to create objects whose state is guaranteed not to change, making the code more predictable and less error-prone.

Question 4: advantages of runnable over thread

**Differences between Extending Thread and Implementing Runnable:**

1. **Separation of Concerns**:
   * **Extending Thread**: When you extend the Thread class, your custom class is directly tied to the thread of execution. This means that each instance of your subclass represents a unique thread, and the task logic (run() method override) is embedded within the thread instance.
   * **Implementing Runnable**: When you implement the Runnable interface, your class defines the task (run() method) but is not directly tied to any thread. You then create instances of Thread and pass an instance of your Runnable implementation to each Thread instance. This separation allows you to reuse the same Runnable instance with multiple threads if needed.
2. **Flexibility**:
   * **Extending Thread**: Java does not support multiple inheritance, so if you extend the Thread class, your subclass cannot extend any other class. This limits flexibility in your class hierarchy.
   * **Implementing Runnable**: Since implementing Runnable does not affect your class hierarchy (you can extend any other class as needed), it provides more flexibility. You can also pass the same Runnable implementation to different types of Thread instances, allowing for more versatile thread management.
3. **Resource Management**:
   * **Extending Thread**: Each instance of a subclass of Thread consumes system resources, such as memory for thread-specific data structures. Creating many instances of Thread directly may be less efficient and harder to manage.
   * **Implementing Runnable**: By separating the task (Runnable) from the thread (Thread), you can manage threads more efficiently. For example, you can use thread pools (ExecutorService) to manage a fixed set of threads that execute different Runnable tasks.

**Key Differences Recap:**

* **Extending Thread**: Directly associates the task (run() method) with the thread instance. Each instance of the subclass represents a unique thread.
* **Implementing Runnable**: Separates the task (run() method) from the thread. You create instances of Thread and pass instances of Runnable to them, allowing for more flexible and efficient thread management.

In practice, using Runnable is often preferred for better code organization, flexibility, and efficient resource management, especially in larger applications or systems where concurrent programming is crucial.

Question 5 :- find first non repeated character in string

**public** **class** TestNew {

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

String str = "saurabhkes";

Map<Character, Long> freqMap = str.chars().mapToObj(ch -> (**char**) ch)

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

Character character = freqMap.entrySet().stream().filter(entry -> entry.getValue() == 1L)

.map(entry -> entry.getKey()).findFirst().get();

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

}

}

Question 5:- kafka flow

Kafka is a distributed streaming platform designed to handle large volumes of real-time data feeds efficiently. Its architecture is built around several key components that work together to ensure scalability, fault tolerance, and high throughput. Here's an overview of Kafka's architecture along with a simplified diagram:

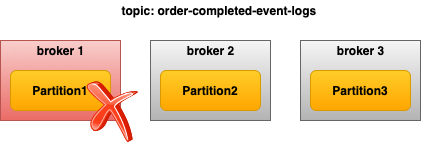
**Kafka Architecture Components:**

1. **Producers**:
   * Applications that publish data to Kafka topics.
   * Producers decide which topic and partition to send the message to.
2. **Topics**:
   * A category or feed name to which messages are published by producers.
   * Each topic can have one or more partitions.
3. **Partitions**:
   * Each topic is divided into partitions to enable parallelism and scalability.
   * Each partition is an ordered and immutable sequence of messages.
4. **Brokers**:
   * Kafka nodes that store and manage the partitions.
   * Each broker can handle multiple partitions across different topics.
   * Brokers facilitate data replication and fault tolerance.
5. **Consumers**:
   * Applications that subscribe to topics and process the feed of published messages.
   * Consumers read messages in the order they are stored in the partitions.
6. **Consumer Groups**:
   * Consumers are organized into consumer groups for load balancing and parallel processing.
   * Each consumer group can have multiple consumer instances.
7. **ZooKeeper**:
   * Used for managing and coordinating Kafka brokers.
   * ZooKeeper maintains broker metadata, leader election, and cluster coordination.

Question 6: kafka fault tolerance:

In the [**Introduction to Kafka blog**](https://medium.com/@anchan.ashwithabg95/introduction-to-apache-kafka-b1386c29da66)**,**we saw what Kafka & Zookeeper is and how the topic is divided into partitions for scalability.

What happens when the broker goes down? Is all the data lost?

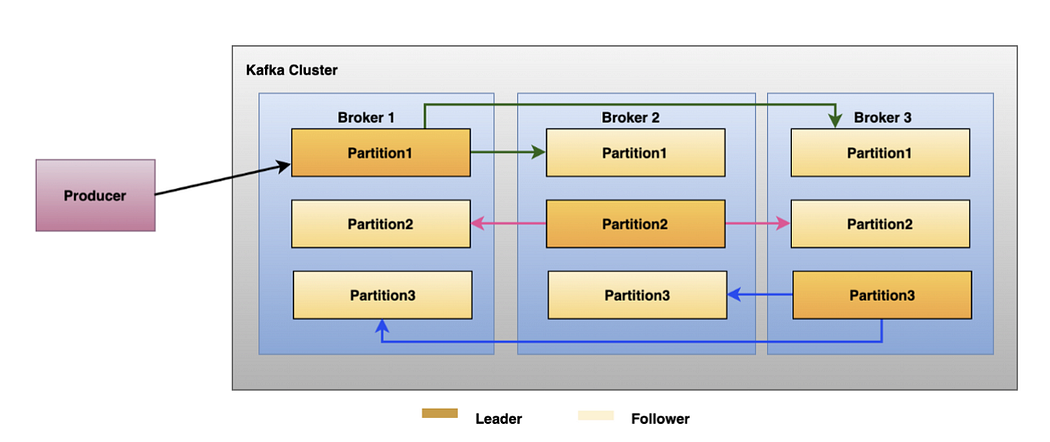


Broker failure

# Partition Replication

Fault tolerance in Kafka is done by copying the partition data to other brokers which are known as **replicas**. There is a configuration that specifies how many copies of the partition you need. Its called a [**replication factor.**](https://www.confluent.io/blog/hands-free-kafka-replication-a-lesson-in-operational-simplicity/#:~:text=KAFKA%20REPLICATION:%200%20TO%2060%20IN%201%20MINUTE&text=Every%20topic%20partition%20in%20Kafka,in%20the%20presence%20of%20failures.)

Each broker will hold one or more partitions. And each of these partitions can either be a replica or leader for the topic. All the writes and reads to a topic go through the leader and the leader coordinates to update replicas with new data.



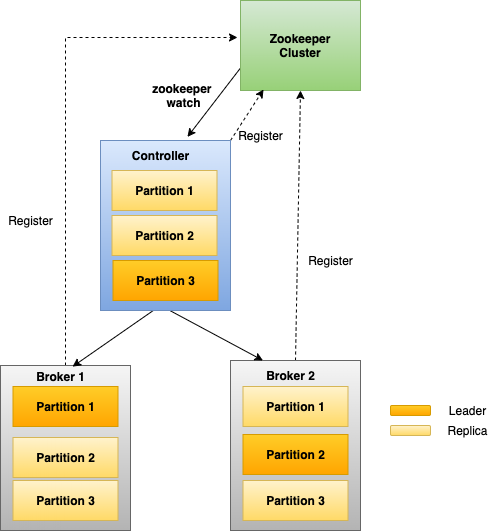
Replication

In the above example, there are three brokers. For **partition-1**, **Broker-1** is a leader. **Broker-2** and **Broker-3** are the replica brokers. The leader partitions and replica brokers are kept in separate brokers because if a leader partition goes down, one of the replica partition brokers can serve as the leader.

But how are leaders elected? There should be somebody who is responsible for electing leaders in the Kafka cluster. That’s where Controller brokers come into the picture.

# Controller broker

**The controller Broker**takes care of electing the leader broker for the partitions. It’s just a normal broker with extra responsibility.[There will be only one controller for the Kafka cluster.](https://stackoverflow.com/questions/49525141/how-many-kafka-controllers-are-there-in-a-cluster-and-what-is-the-purpose-of-a-c) The controller broker keeps track of brokers joining and leaving the cluster with the help of Zookeeper.



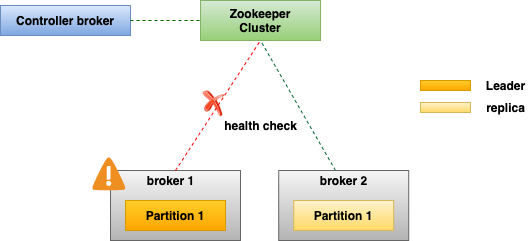
controller

As we saw earlier, [**Zookeeper**](https://stackoverflow.com/questions/23751708/is-zookeeper-a-must-for-kafka) is the centralized service for storing metadata of topic, partition, and broker. Every time a broker starts up, it registers itself to the zookeeper. And the zookeeper keeps track of each broker by calling a health check on it. Just like Kafka brokers, even Zookeepers run as a cluster, known as an **ensemble**.

# Leader partition election

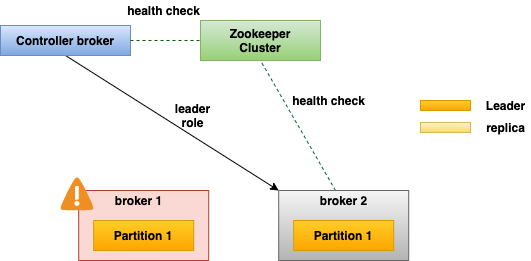
When the leader broker goes down,

1. The Zookeeper informs the Controller.
2. The controller selects one of the [**in-sync replicas**](https://www.cloudkarafka.com/blog/2019-09-28-what-does-in-sync-in-apache-kafka-really-mean.html) (ISR) as the leader. ISR is a replica broker that is fully caught up with the changes of the leader broker. The leader is responsible for keeping track of ISR and sending this information to the zookeeper.
3. When the broker comes back up, then it will be assigned again as the leader.



Leader failure

In the above example, there is broker-1 which is down due to some issue. And the zookeeper keeps track of it because of health checks. Zookeeper sends a notification to the controller regarding the broker-1 unavailability.

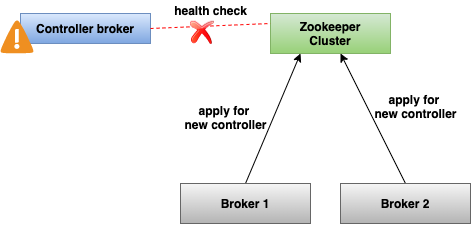


leader election

The controller chooses one of the replica partitions as the leader for this partition. In the above example, it is broker-2. When the broker-1 comes back up, it will be assigned again as the leader.

But what if the controller itself goes down?

# Controller election



Controller failure

When the controller goes down,

* Zookeeper informs all the brokers that the controller failed.
* All the brokers will apply to be the controller.
* The first broker who applies for this position will become the controller.

# Summary

Kafka is a distributed system. The topic is divided into partitions and kept in different brokers. If any broker fails, data should not be lost. For fault-tolerance purposes, the partition is replicated and stored in different brokers. If leader brokers fail, then the controller will elects one of the replicas as the leader. Even controller brokers can fail, in this case, Zookeeper will help in electing the broker as the controller.

Question 7 what is api gateway :-

In a microservices architecture, an API Gateway is a central component that plays a crucial role in managing the interactions between clients and the microservices themselves. Here’s how an API Gateway fits into a microservices environment:

1. **Single Entry Point**: Each microservice typically exposes its own set of APIs. An API Gateway provides a unified entry point for clients to access these APIs without needing to know the specific details of each microservice.
2. **Routing and Aggregation**: The API Gateway routes client requests to the appropriate microservice based on the request URL, headers, or other parameters. It can also aggregate multiple requests into a single request to reduce the number of round trips between clients and microservices.
3. **Protocol Translation**: Microservices may use different communication protocols or data formats (e.g., REST, GraphQL, JSON, XML). The API Gateway can handle protocol translation and data transformation to ensure that clients receive responses in a format they understand.
4. **Load Balancing**: To improve performance and reliability, the API Gateway can distribute incoming requests across multiple instances of a microservice. This helps to optimize resource usage and ensure high availability.
5. **Authentication and Authorization**: Security is a critical concern in microservices. The API Gateway handles authentication (verifying the identity of clients) and authorization (determining whether clients have the necessary permissions to access specific resources).
6. **Rate Limiting and Throttling**: To prevent abuse and ensure fair usage of resources, the API Gateway can enforce rate limits and throttling policies on incoming requests.
7. **Monitoring and Analytics**: The API Gateway provides monitoring capabilities to track metrics such as request throughput, latency, error rates, and traffic patterns. This data is essential for optimizing performance and diagnosing issues.
8. **Service Discovery**: In dynamic microservices environments where services may come and go, the API Gateway can integrate with service discovery mechanisms to dynamically locate and route requests to available instances of microservices.
9. **API Lifecycle Management**: Managing the lifecycle of APIs, including versioning, deprecation, and retirement, is simplified with an API Gateway. It can help ensure smooth transitions and backward compatibility for clients consuming microservices.

Overall, an API Gateway in a microservices architecture acts as a facilitator, enhancing the scalability, security, and manageability of interactions between clients and the distributed set of microservices that make up the application.

Question 8 : -what is service discovery in microservice and implementation

Service discovery in a microservices architecture is the process of dynamically locating and identifying available instances of services (microservices) within a network. In a traditional monolithic application, services are typically statically configured or known beforehand. However, in a microservices environment where services are distributed, elastic, and may scale up or down dynamically, service discovery becomes essential for enabling communication between services.

**Importance of Service Discovery:**

1. **Dynamic Nature**: Microservices can be deployed, scaled, or replaced independently and frequently. Service discovery allows other services to dynamically locate these instances without manual intervention.
2. **Load Balancing**: Service discovery enables load balancers to distribute incoming requests across multiple instances of a service, improving performance and fault tolerance.
3. **Fault Tolerance**: If an instance of a service fails, service discovery helps redirect traffic to healthy instances automatically.
4. **Simplifies Configuration**: Services can be deployed with minimal configuration since they register themselves with the service discovery mechanism.

**Implementation of Service Discovery:**

There are several approaches to implementing service discovery in a microservices architecture:

1. **Client-Side Discovery**:
   * **Description**: Each client (typically a service or a client application) is responsible for determining the network locations of available service instances from the service registry.
   * **Implementation**: Clients periodically fetch an updated list of service instances from the service registry (e.g., Consul, Eureka, ZooKeeper) and use this information to make requests to the appropriate instance.
2. **Server-Side Discovery**:
   * **Description**: A load balancer or a proxy (like an API Gateway) is responsible for routing requests to appropriate service instances based on the information stored in the service registry.
   * **Implementation**: The load balancer or proxy queries the service registry to obtain the current list of service instances and dynamically adjusts its routing accordingly.
3. **Service Registry**:
   * **Description**: Centralized database or registry that keeps track of information about available service instances (e.g., host, port, health status).
   * **Implementation**: Services register themselves with the registry upon startup and deregister when they shut down or become unhealthy. The registry provides an API for querying and updating service information.

**Popular Service Discovery Tools:**

* **Consul**: Provides service discovery, health checking, and key-value storage features.
* **Eureka**: Netflix's service registry and discovery tool designed for cloud environments.
* **ZooKeeper**: Distributed coordination service for maintaining configuration information, naming, providing distributed synchronization, and group services.

**Example Workflow:**

1. **Service Registration**:
   * When a microservice instance starts up, it registers itself with the service registry, providing its network location (host, port), health status, and other metadata.
2. **Service Querying**:
   * When a client needs to communicate with a particular service, it queries the service registry to obtain the network location of available instances.
3. **Dynamic Routing**:
   * Based on the information obtained from the service registry, requests are dynamically routed to one of the available instances of the service.
4. **Health Checking**:
   * The service registry performs health checks on registered instances to ensure they are responsive and healthy. Unhealthy instances are automatically removed from the registry, preventing them from receiving traffic.

In summary, service discovery simplifies the complexity of managing and scaling microservices by enabling dynamic, automated discovery of service instances within a distributed architecture. This approach enhances flexibility, scalability, and fault tolerance in modern cloud-native applications.

Question 9: expression balanced

**package** com;

**import** java.util.Stack;

**public** **class** TestNew {

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

String str = "[{(}]";

**boolean** checkBalanced = *checkBalanced*(str);

**if** (checkBalanced) {

System.***out***.println("expression is balanced");

} **else** {

System.***out***.println("expression is not balanced");

}

}

**private** **static** **boolean** checkBalanced(String str) {

Stack<Character> st = **new** Stack();

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

**char** ch = str.charAt(i);

**if** (ch == '(' || ch == '{' || ch == '[') {

st.push(ch);

} **else** **if** (ch == ')' || ch == '}' || ch == ']') {

**if** (st.isEmpty()) {

**return** **false**;

}

**char** top = st.pop();

**if** ((ch == ')' && top != '(') || (ch == '{' && top != '}') || (ch == '[' && top != ']')) {

**return** **false**;

}

}

}

**return** st.isEmpty();

}

}

Question 10 solid principles

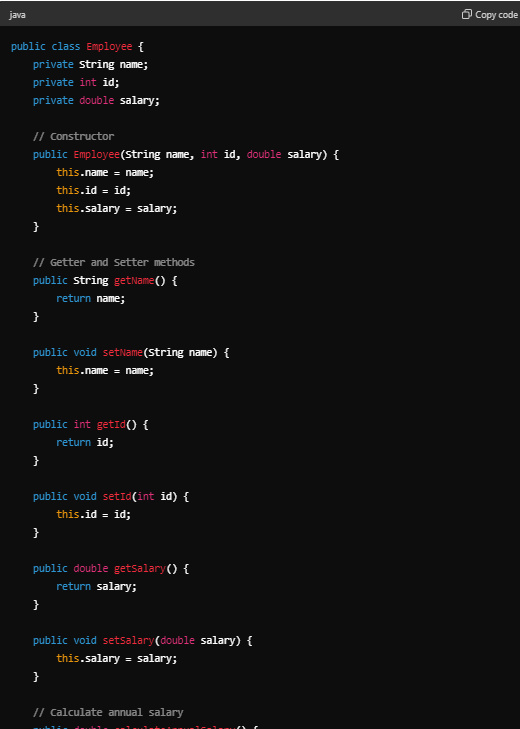
**Single Responsibility Principle (SRP)**

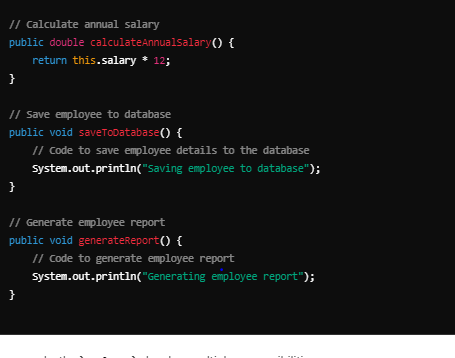
The **Single Responsibility Principle (SRP**) is one of the five SOLID principles of object-oriented design. It states that a class should have only one reason to change, meaning that it should have only one job or responsibility. When a class adheres to SRP, it is easier to understand, maintain, and modify.

**Example of SRP Violation in Java**

Violates

Here's an example of a class that violates the Single Responsibility Principle:





In this example, the Employee class has multiple responsibilities:

1. It handles employee data (fields and their getters/setters).
2. It calculates the annual salary.
3. It saves the employee to a database.
4. It generates a report for the employee.

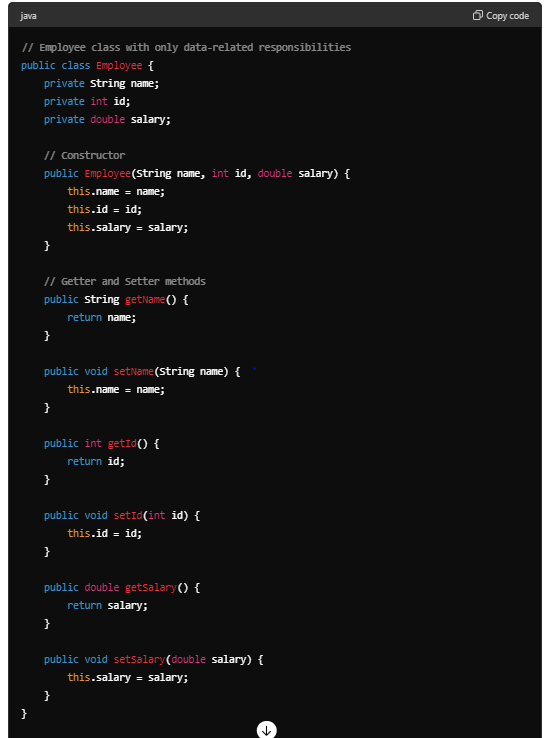
**Refactored Example to Adhere to SRP**

To adhere to the Single Responsibility Principle, we can refactor the code by creating separate classes for each responsibility:

1. A class for employee data.
2. A class for salary calculation.
3. A class for database operations.
4. A class for report generation.

Here is how we can do it:

Solution

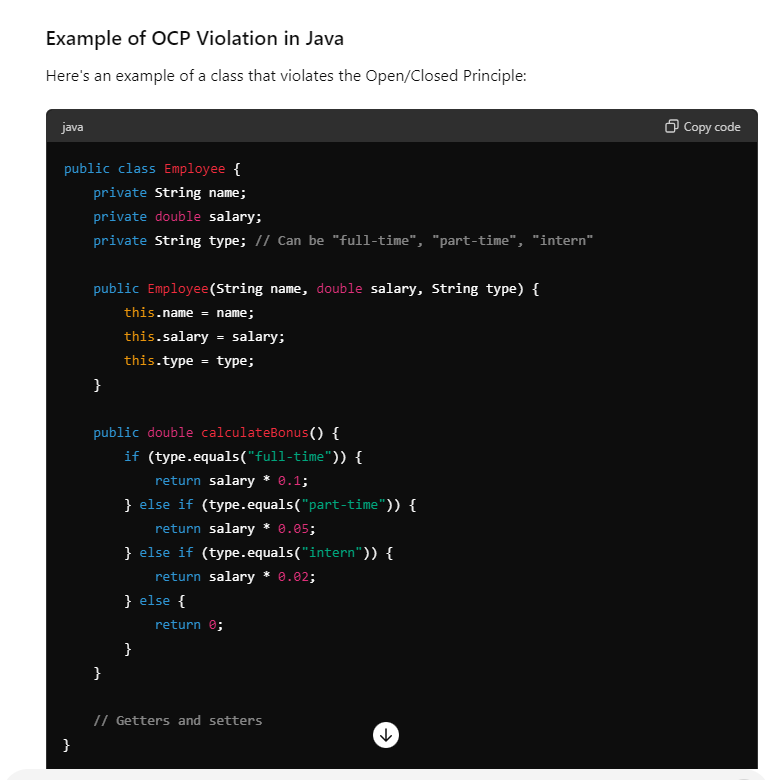




Open/Closed Principle (OCP)

The Open/Closed Principle (OCP) is another one of the SOLID principles of object-oriented design. It states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This means you should be able to add new functionality to an entity without changing its existing code.

Violates

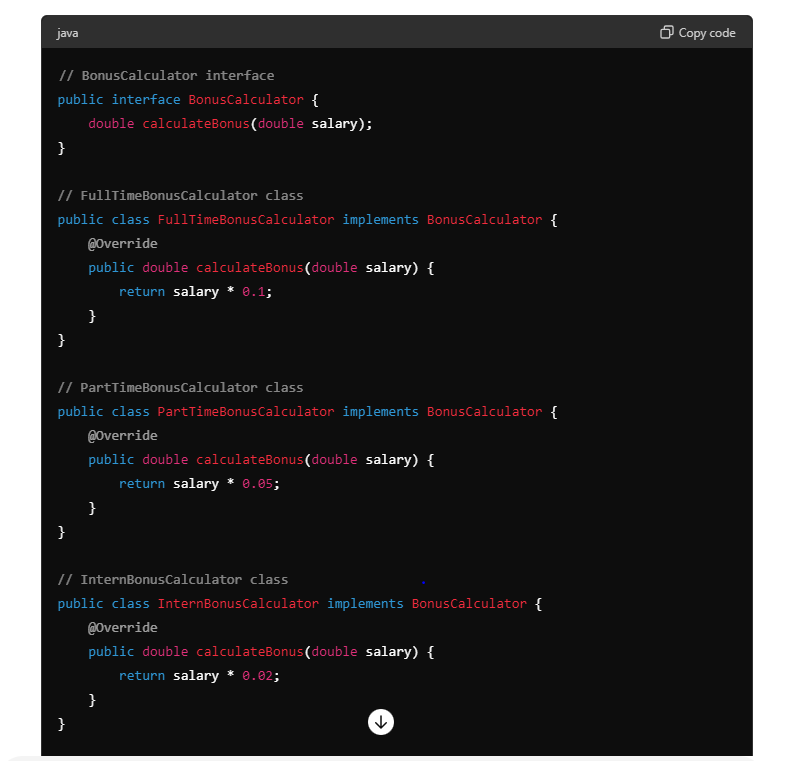


In this example, every time we need to add a new employee type, we have to modify the calculateBonus method, which violates the Open/Closed Principle.

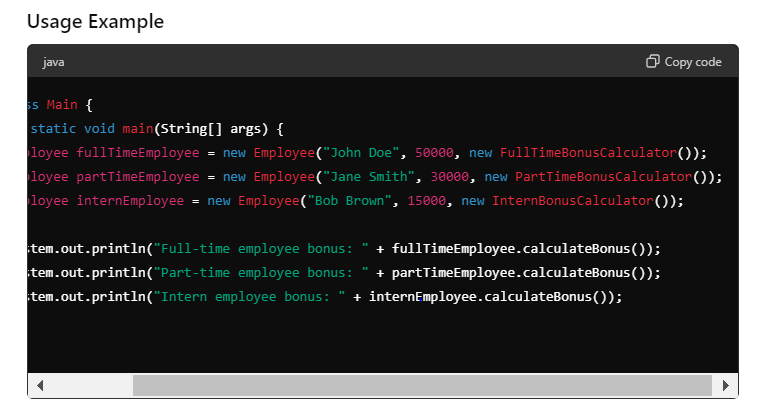
**solution**

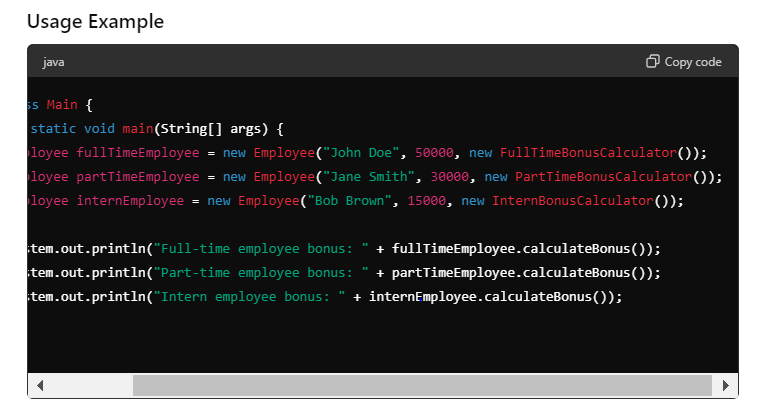
**Refactored Example to Adhere to OCP**

To adhere to the Open/Closed Principle, we can use polymorphism. We'll create an interface for calculating bonuses and implement this interface for each employee type.









Liskov Substitution Principle (LSP)

**subclass does not change the expected behavior of the base class**

The **Liskov Substitution Principle (LSP)** is another SOLID principle that states that objects of a superclass should be replaceable with objects of a subclass without affecting the functionality of the program. In other words, subclasses should be substitutable for their base classes.

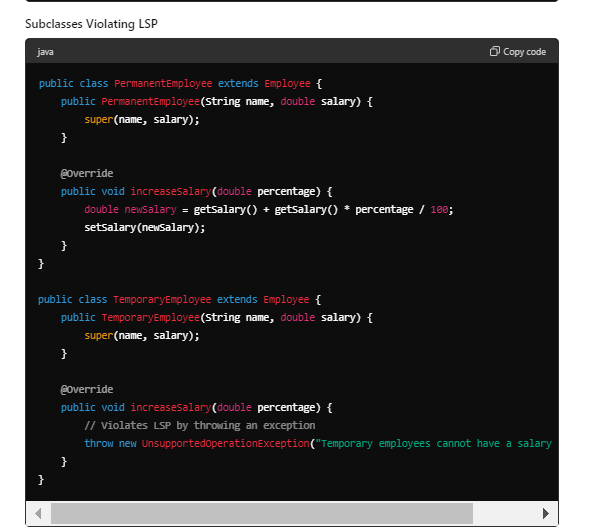
Violation

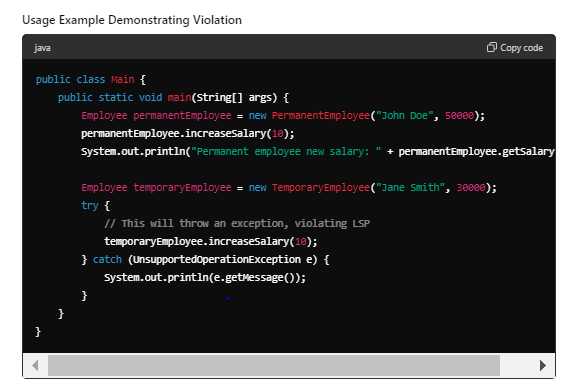
Certainly! Let's start by showing how a violation of the Liskov Substitution Principle (LSP) can occur even when using an abstract class, followed by a corrected version that adheres to LSP.

**Violation of LSP with Abstract Class**

In this example, the TemporaryEmployee class overrides the increaseSalary method in a way that breaks the contract expected by clients of the Employee class.







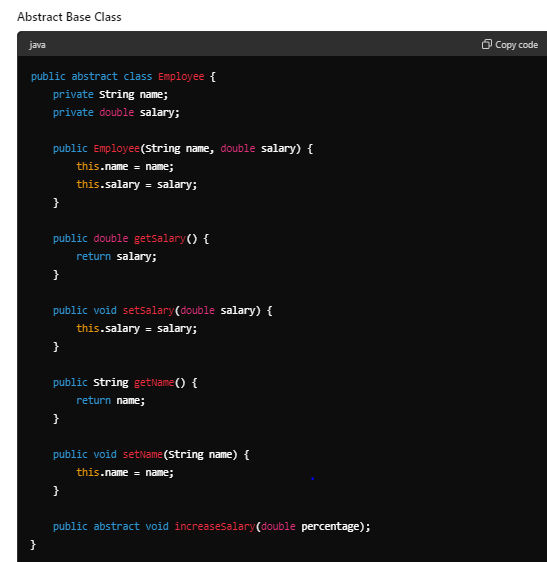
In this violation example, the TemporaryEmployee class throws an UnsupportedOperationException in the increaseSalary method, which breaks the LSP. Clients expecting to use an Employee object should not encounter such exceptions when calling increaseSalary.

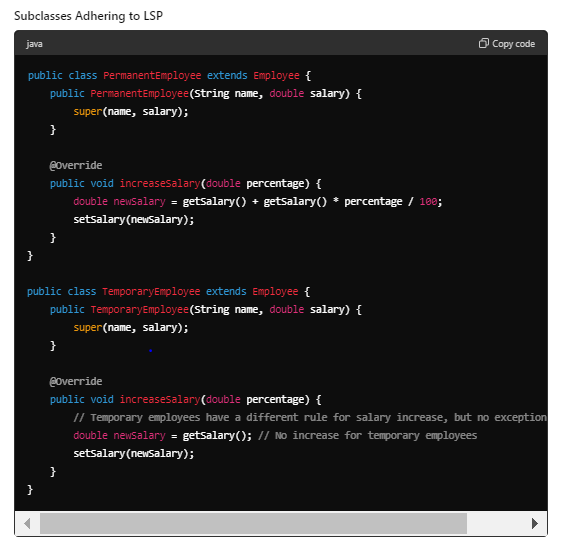
solution

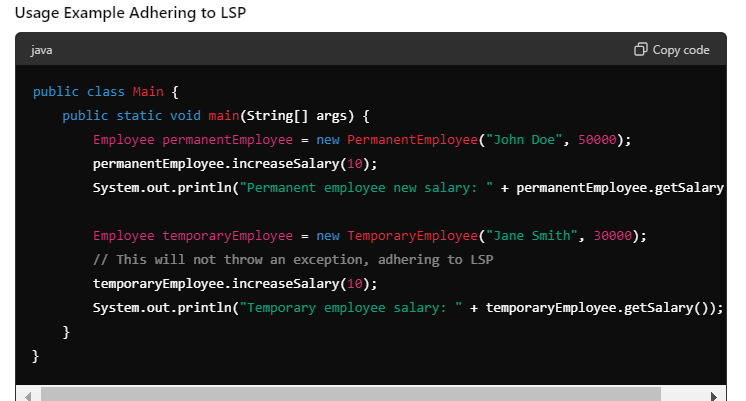
### Adhering to LSP -------

To adhere to LSP, we can redesign the classes to ensure that the behavior expected by the Employee class is preserved in its subclasses.

#### Abstract Base Class







In this corrected example, the TemporaryEmployee class overrides the increaseSalary method in a way that doesn't break the expected behavior of the Employee class. Instead of throwing an exception, it simply doesn't change the salary. This adheres to the Liskov Substitution Principle, ensuring that clients can rely on the consistent behavior of Employee objects regardless of their specific type.

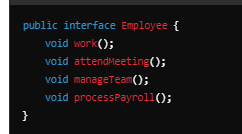
**Interface Segregation Principle (ISP)**

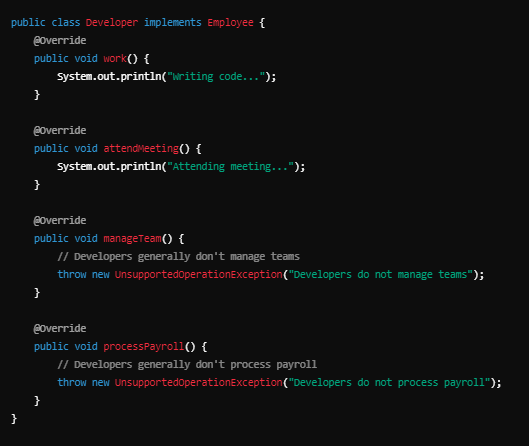
The **Interface Segregation Principle (ISP)** states that no client should be forced to depend on methods it does not use. Essentially, it's better to have multiple, smaller, and more specific interfaces than one large, general-purpose interface.

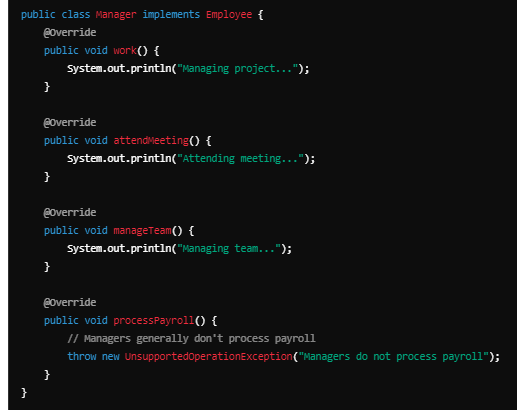
### Example of ISP Violation in Java

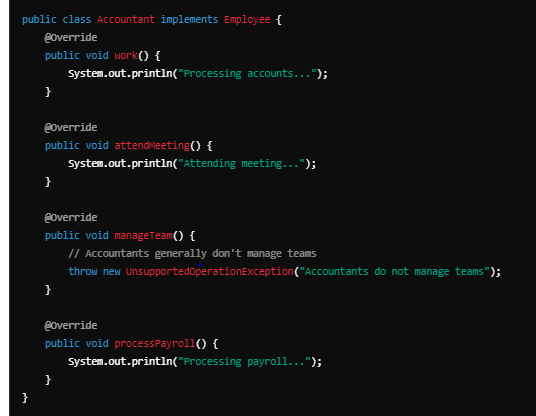
Let's consider an example where we have a general Employee interface that is too broad and forces all implementing classes to provide implementations for methods they don't need.

#### Interface that Violates ISP ----- violation







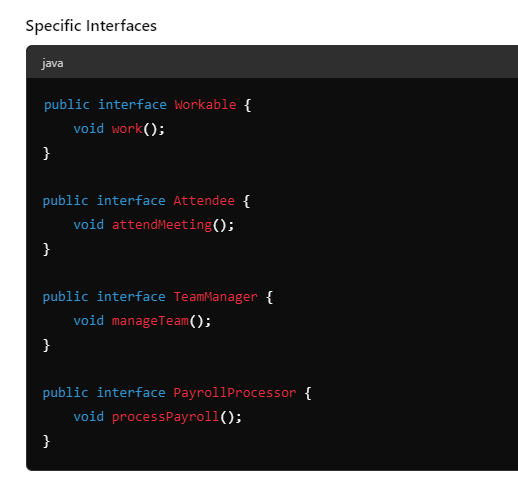


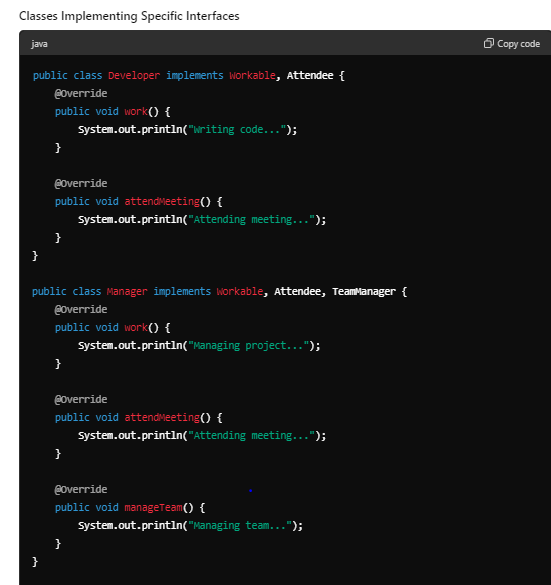
In this example, the Employee interface forces Developer, Manager, and Accountant classes to implement methods they do not need, leading to UnsupportedOperationException in many cases.

solution

**Adhering to ISP by Creating Specific Interfaces**

To adhere to the Interface Segregation Principle, we can create more specific interfaces that capture the responsibilities of different types of employees.







**Explanation**

* **Specific Interfaces**: Instead of one large Employee interface, we create smaller interfaces like Workable, Attendee, TeamManager, and PayrollProcessor to capture specific responsibilities.
* **Implementing Classes**: Each class implements only the interfaces relevant to its responsibilities. For example, Developer implements Workable and Attendee, while Manager implements Workable, Attendee, and TeamManager.
* **Usage**: The classes use only the methods they need without any unsupported operations, adhering to the Interface Segregation Principle.

By following ISP, we ensure that classes are not forced to implement methods they do not use, resulting in cleaner and more maintainable code.

**Dependency Inversion Principle (DIP)**

The **Dependency Inversion Principle (DIP)** states that:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g., interfaces).
2. Abstractions should not depend on details. Details should depend on abstractions.

Violation

### DIP Violation Example

Imagine a scenario where an OrderService depends directly on a concrete PayPalPaymentProcessor class. This setup violates DIP because the high-level OrderService is dependent on the low-level PayPalPaymentProcessor.

#### Concrete Classes and DIP Violation



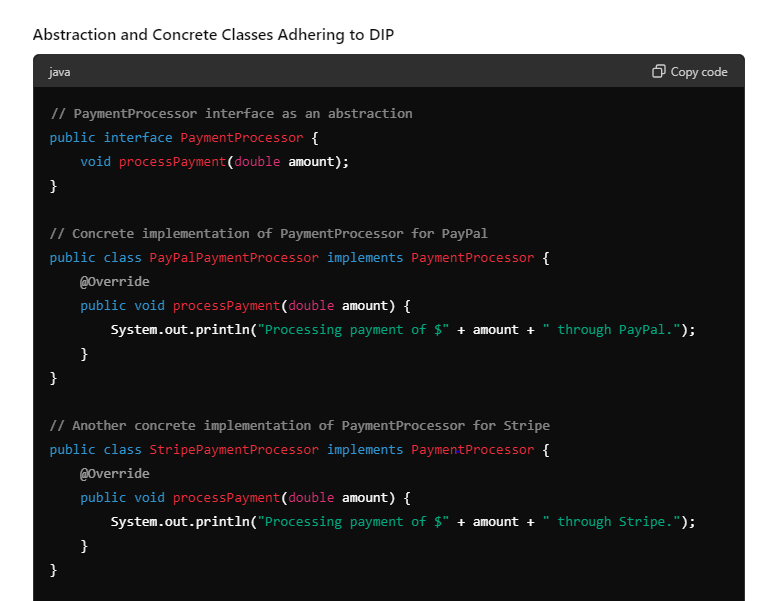
In this example, OrderService is tightly coupled to PayPalPaymentProcessor, making it difficult to switch to a different payment processor or to test the OrderService in isolation.

### Adhering to DIP Using Interfaces

To adhere to the Dependency Inversion Principle, we can introduce an interface PaymentProcessor that both the OrderService and PayPalPaymentProcessor depend on. This way, OrderService depends on an abstraction rather than a concrete implementation.

#### Abstraction and Concrete Classes Adhering to DIP

Solution





**Explanation**

* **Interface PaymentProcessor**: This interface provides the abstraction that both OrderService and payment processors (e.g., PayPalPaymentProcessor and StripePaymentProcessor) depend on.
* **Concrete Classes PayPalPaymentProcessor and StripePaymentProcessor**: Implement the PaymentProcessor interface.
* **High-level Class OrderService**: Depends on PaymentProcessor, making it independent of the specific payment processor implementations.

**Usage**

In the Main class, we can easily switch between different payment processor implementations (e.g., PayPalPaymentProcessor and StripePaymentProcessor) by injecting the appropriate implementation into OrderService. This makes the OrderService more flexible and easier to test.

By following DIP, the high-level OrderService does not depend on the low-level PayPalPaymentProcessor or StripePaymentProcessor directly. Instead, both the high-level and low-level modules depend on the PaymentProcessor abstraction. This results in a more decoupled, maintainable, and testable codebase.

Question 11 : treeset problem in java

Certainly! The TreeSet in Java is a sorted collection implemented using a tree structure. It stores elements in a sorted order defined either by the natural ordering of its elements or by a specified comparator.

### Problem with TreeSet in Java

One common issue that developers encounter with TreeSet revolves around its behavior when adding custom objects that don't correctly implement the Comparable interface or when the comparator used is inconsistent with the equals method.

#### Example: Incorrect Usage Leading to Unexpected Behavior

Let's consider a scenario where we have a custom class Person and we want to use a TreeSet to store instances of Person sorted by their age.

Q u

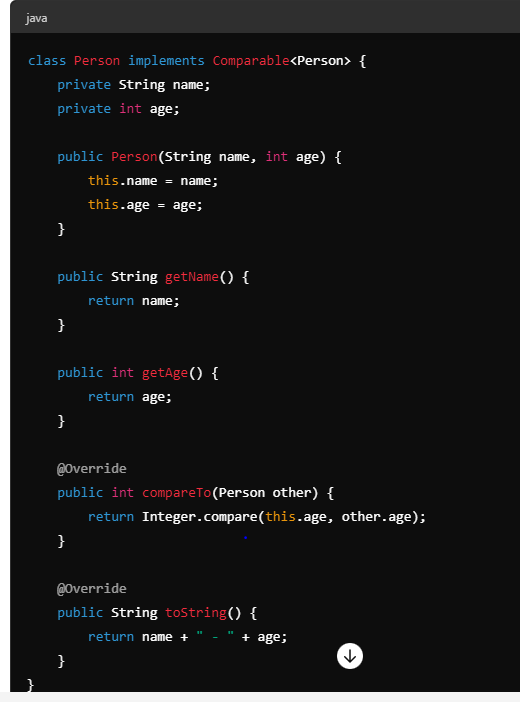
**Explanation**

In this example, we define a Person class with name and age attributes. The TreeSet<Person> is used to store Person objects. However, running this code will result in a ClassCastException because the Person class does not implement the Comparable interface or provide a Comparator to the TreeSet.

**Solution: Implementing Comparable Interface or Providing Comparator**

To fix this issue, you have two options:

1. **Implement the Comparable Interface in Person Class**: This allows TreeSet to sort Person objects based on a natural ordering (e.g., by age).



**Provide a Comparator to the TreeSet**: If you prefer sorting by a different criterion or if Person cannot implement Comparable, you can provide a Comparator during the TreeSet initialization.



**Conclusion**

When using TreeSet in Java, ensure that:

* Elements inserted into the TreeSet either implement the Comparable interface for natural ordering, or
* You provide a Comparator during TreeSet initialization that defines the sorting criteria.

Failure to do so can lead to ClassCastException or unexpected ordering of elements in the TreeSet. This example demonstrates the importance of understanding how TreeSet manages sorting and ordering of elements.

**Question 12 : employee join after 2006 with highest salary.**

Optional<Employee> highestSalaryEmployee = employees.stream() .filter(e -> e.getJoiningYear() > 2006) .max(Comparator.comparingDouble(Employee::getSalary));

highestSalaryEmployee.ifPresent(employee -> { System.out.println("Employee with highest salary joined after 2006:");

System.out.println(employee); });

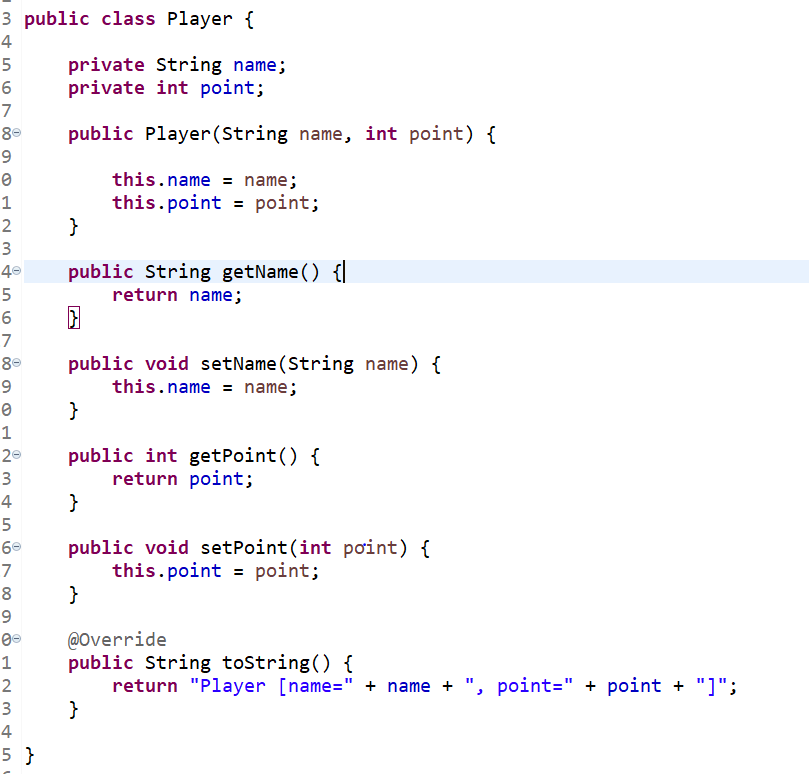
--------------------------------------------or-----------------------------------------------------------------

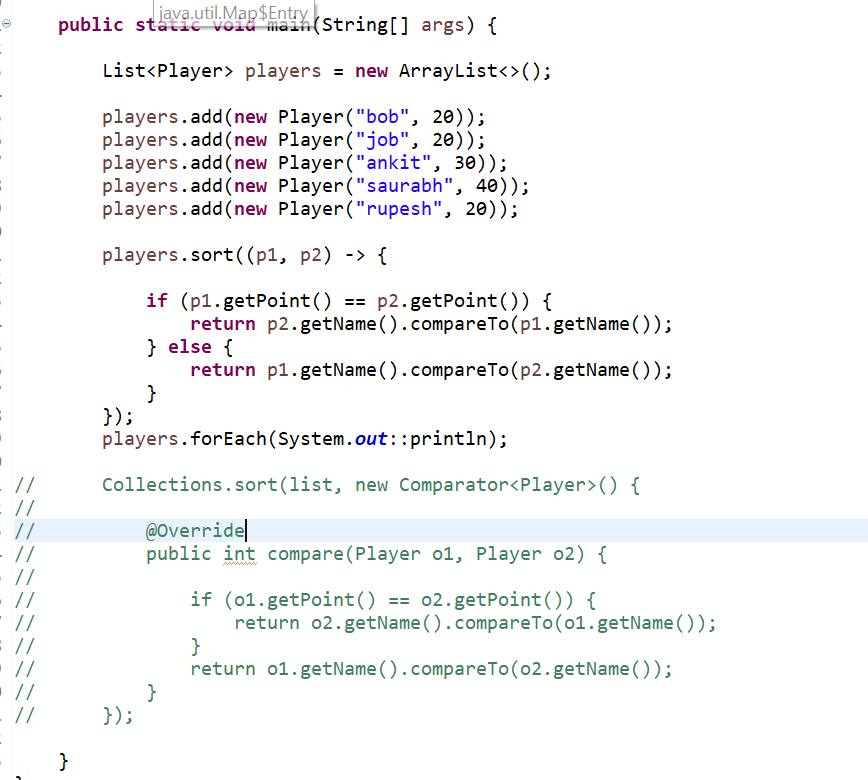
Employee employee = employees.stream().filter(emp -> emp.getJoiningYear() > 2006)

.max((e1, e2) -> Double.*compare*(e1.getSalary(), e2.getSalary())).get();

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

Question 13 : wap to find player by name in descending order if point are same else in ascending order





// output

Player [name=ankit, point=30]

Player [name=rupesh, point=20]

Player [name=job, point=20]

Player [name=bohn, point=20]

Player [name=saurabh, point=40]