**Interview Question**

**Question 1**

**Hashmap mechanism**

The **HashMap** class in Java provides an efficient implementation of the **Map** interface, using a hashtable data structure. Here's how the **HashMap** mechanism works:

1. **Hashing**: When you put a key-value pair into a **HashMap**, the key's **hashCode()** method is used to compute a hash code, which is an integer value. The hash code determines which bucket in the underlying array (**table**) the key-value pair will be stored in.
2. **Bucket Array**: The **HashMap** contains an array of **Node** objects, where each **Node** represents a bucket. Each bucket can hold multiple key-value pairs. The size of the array (**table**) determines the capacity of the **HashMap**.
3. **Index Calculation**: The hash code is then transformed into an index within the **table** array using a hashing algorithm. The **indexFor()** method typically performs this transformation by using the modulo operator (**%**) to ensure that the index falls within the bounds of the array.
4. **Collision Handling**: Since multiple keys can hash to the same index (collision), **HashMap** handles collisions by chaining. If multiple key-value pairs hash to the same index, they are stored in a linked list within the bucket.
5. **Node Structure**: Each **Node** in the linked list contains the key, value, and a reference to the next **Node**. In case of collisions, new key-value pairs are added to the end of the linked list.
6. **Load Factor and Rehashing**: **HashMap** uses a load factor to determine when to resize the **table** array. The load factor is a measure of how full the **HashMap** is, and when it exceeds a certain threshold (typically 0.75), the **HashMap** is resized to accommodate more elements. Rehashing involves creating a new, larger **table** array and reinserting all key-value pairs into the new array, redistributing them based on their new hash codes.
7. **Retrieval**: When you call **get(key)** to retrieve a value from the **HashMap**, the same hash code calculation is performed to determine the index in the **table** array. Then, the linked list in the corresponding bucket is traversed to find the key-value pair with the matching key.
8. **Insertion and Deletion**: When you call **put(key, value)** to insert a key-value pair into the **HashMap**, the key's hash code is calculated to determine the bucket. If there is no collision, the key-value pair is added to the bucket. If there is a collision, the pair is added to the end of the linked list in the bucket. Deletion involves removing the key-value pair from the linked list in the appropriate bucket.

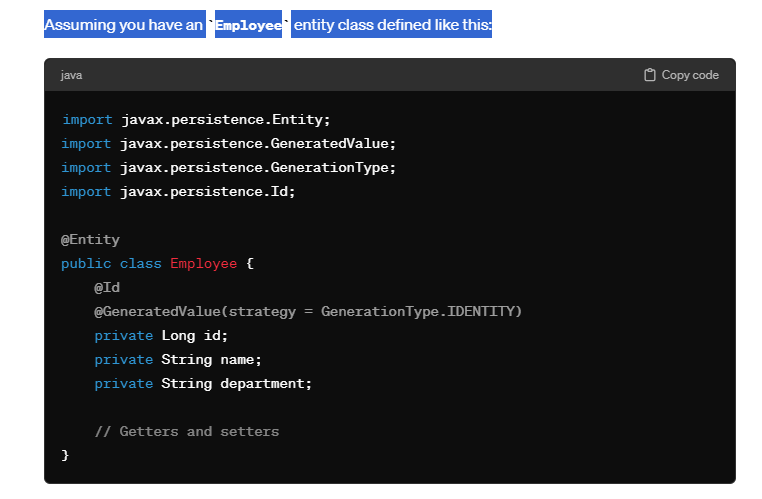
Overall, the **HashMap** mechanism provides efficient storage and retrieval of key-value pairs by leveraging hashing and collision resolution techniques. It offers constant-time performance for most operations (such as **get** and **put**) on average, making it one of the most commonly used data structures in Java for storing and managing collections of key-value pairs.

**Question 2**

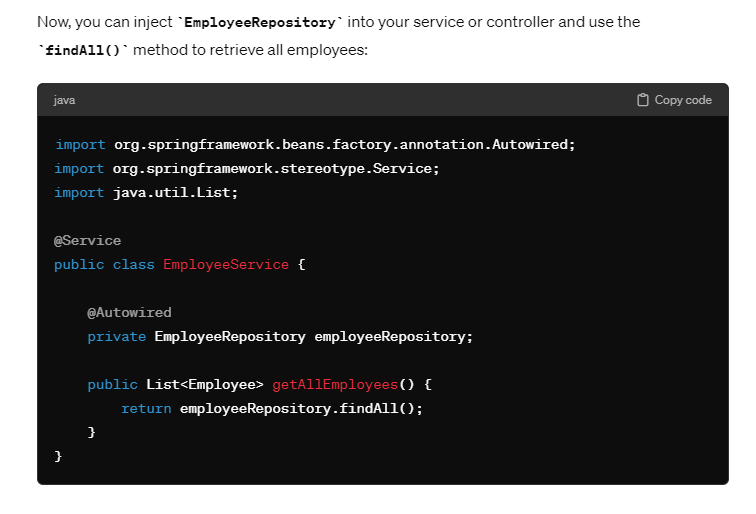
**JPA get all employees**

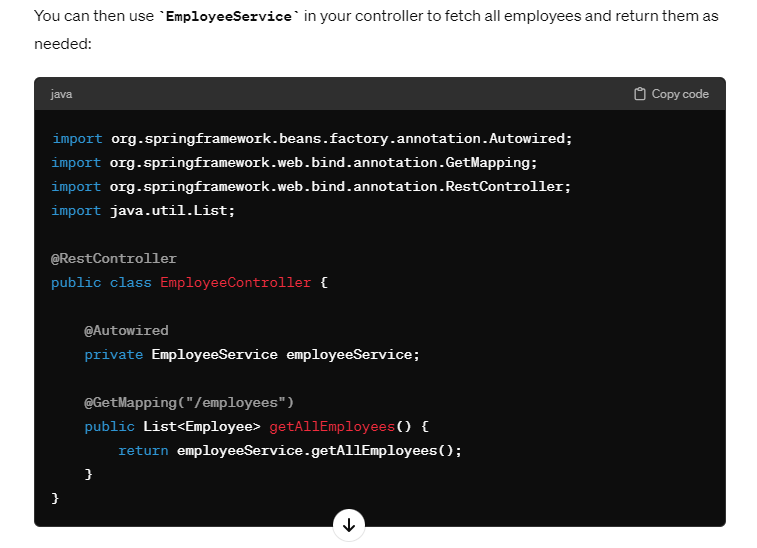
To retrieve all employees using JPA (Java Persistence API), you typically create a repository interface that extends **JpaRepository** or **CrudRepository** provided by Spring Data JPA. Then, you define a method in the repository interface to fetch all employees. Here's an example:

Assuming you have an **Employee** entity class defined like this:

****

****

****

****

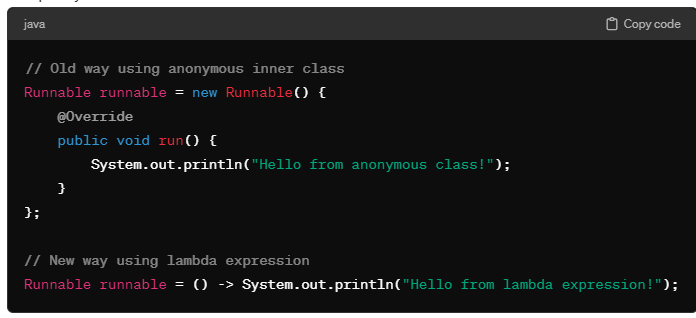
This setup allows you to retrieve all employees from the database using JPA and Spring Data JPA's repository interface. The **findAll()** method provided by **JpaRepository** will fetch all records from the **Employee** table.

Question 3

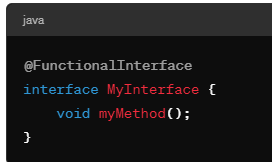
Java 8 features

Java 8 introduced several new features and enhancements to the language and the Java API. Here are some of the key features introduced in Java 8:

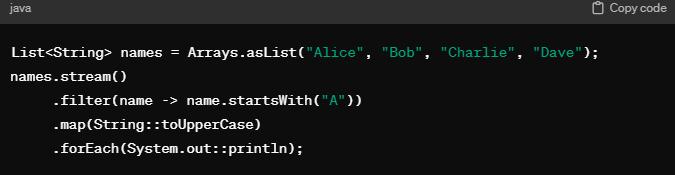
1. **Lambda Expressions**: Lambda expressions enable you to treat functionality as a method argument or create concise anonymous functions. They facilitate functional programming by allowing you to express instances of single-method interfaces (functional interfaces) more compactly.

****

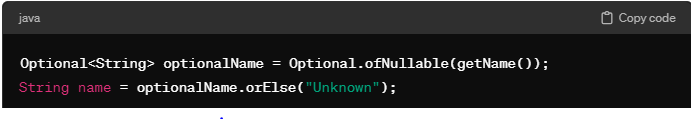
**2- Functional Interfaces**: Functional interfaces are interfaces with a single abstract method. They can be annotated with the **@FunctionalInterface** annotation for clarity. Lambda expressions can be used to provide the implementation of the abstract method of a functional interface.

****

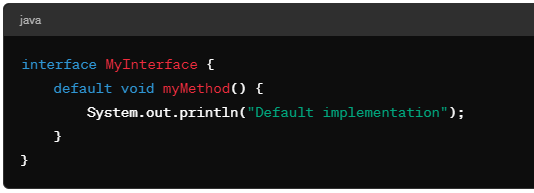
**3- Stream API**: The Stream API provides a fluent and functional way to process collections of objects. Streams allow for declarative operations such as filter, map, reduce, and collect, making it easier to work with collections.

****

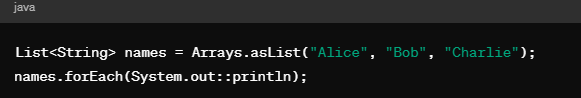
**4- Optional**: The **Optional** class is a container object that may or may not contain a non-null value. It is designed to provide a more expressive way to handle potentially nullable values and to avoid null pointer exceptions.

****

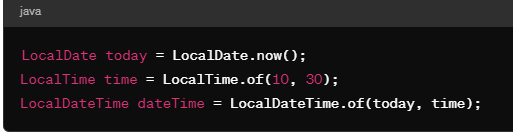
**5-** **Default Methods**: Default methods allow interfaces to have methods with implementations. This feature enables the addition of new methods to existing interfaces without breaking compatibility with implementing classes.

****

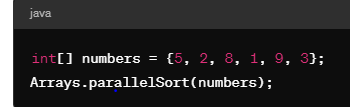
**6-** **Method References**: Method references provide a shorthand syntax for lambda expressions that simply call a method. They can make your code more readable by providing a clear and concise way to reference existing methods.

****

**7-** **Date and Time API**: The new Date and Time API (**java.time**) provides classes for representing dates, times, durations, and intervals. It is more comprehensive, flexible, and immutable compared to the old **java.util.Date** and **java.util.Calendar** classes.

****

**8-** **Parallel Array Sorting**: The **Arrays** class provides parallel sort methods for arrays, allowing you to take advantage of multi-core processors for faster sorting of large arrays.

****

**T** hese are some of the key features introduced in Java 8. They have significantly enhanced the expressiveness, readability, and performance of Java code, making it easier to develop modern, functional, and concurrent applications.

Question 4

Why we serialize object ?

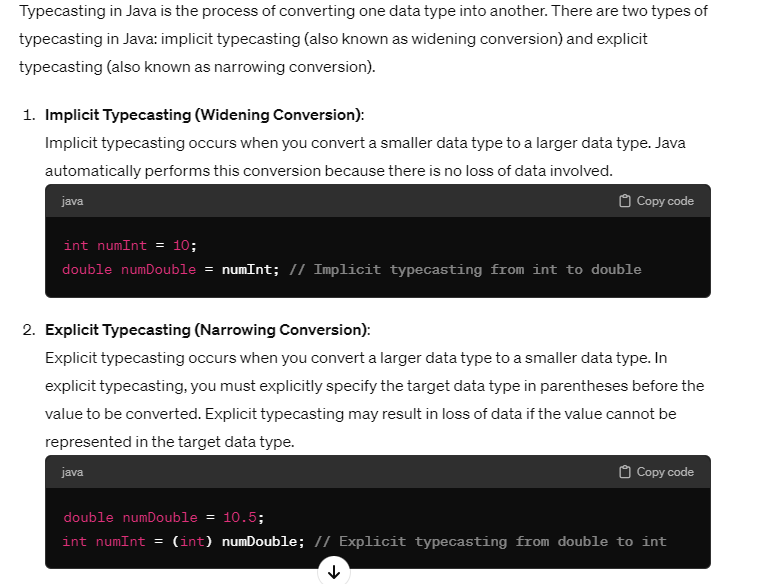
Serialization enables us to save the state of an object and recreate the object in a new location.

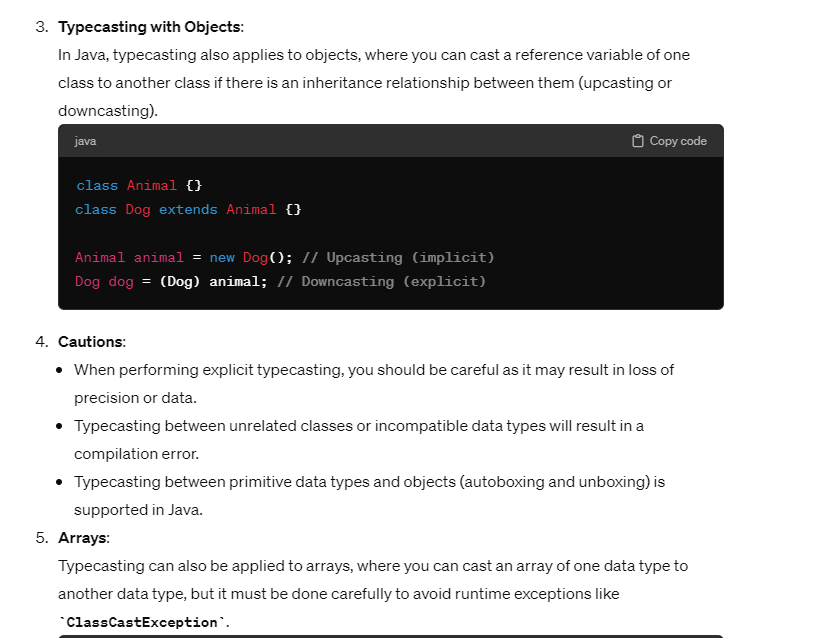
**Serialization in Java refers to the process of converting an object into a stream of bytes so that it can be persisted to a file, sent over a network, or stored in a database. There are several reasons why you might want to serialize objects:**

1. **Persistence**: Serialization allows you to save the state of an object to a file or database. This enables you to store object data persistently and reload it later to reconstruct the object.
2. **Communication**: Serialization is commonly used in network communication to transmit objects between different systems or processes. By serializing objects into a byte stream, you can send them over the network and deserialize them on the receiving end.
3. **Caching**: Serialized objects can be cached in memory or stored in a distributed cache to improve performance and reduce latency. This is particularly useful for frequently accessed or expensive-to-create objects.
4. **Cloning**: Serialization and deserialization can be used to create deep copies of objects by serializing an object and then deserializing it back into a new object. This is useful when you need to create independent copies of objects.
5. **State Transfer**: Serialization is often used in web applications and distributed systems to transfer the state of an object between different layers of the application, such as from the server to the client in a web application.
6. **Backup and Restore**: Serialized objects can be used for backup and restore operations, allowing you to save the state of an application or system at a particular point in time and restore it later if necessary.

Question 5

Typecasting in java





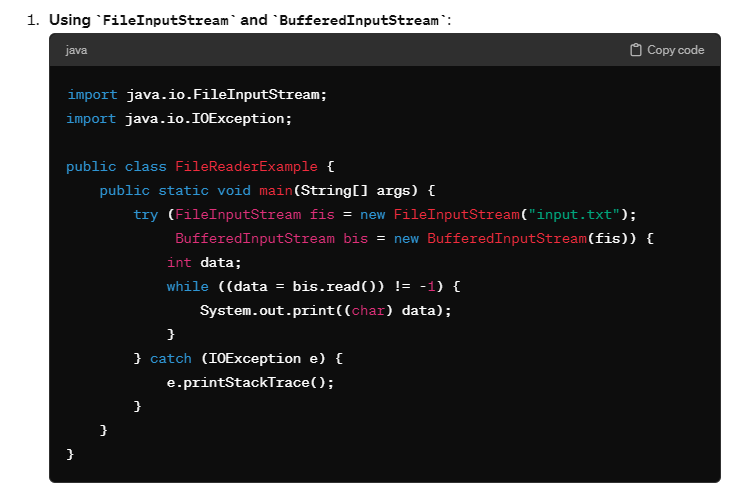


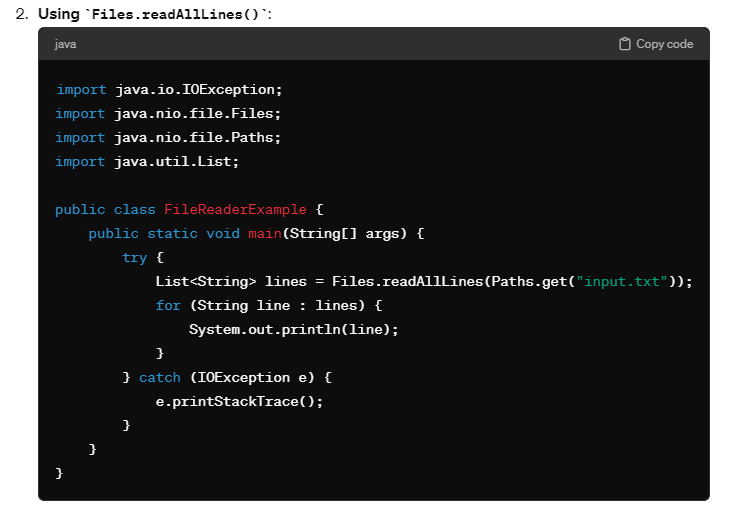
Question 6 :

Files read and write operation :-

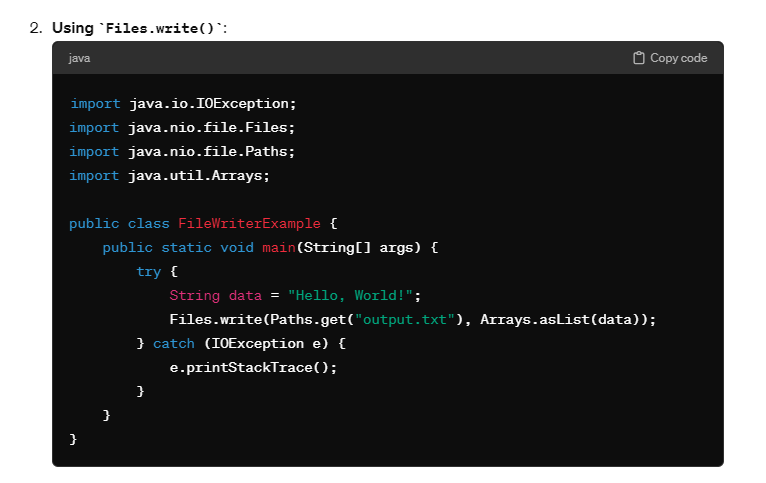
In Java, you can perform file read and write operations using various classes provided by the **java.io** package (or **java.nio.file** package for NIO-based operations). Here's a basic overview of how to perform file read and write operations in Java:

### File Reading:









These are some basic examples of how to perform file read and write operations in Java. Remember to handle exceptions appropriately and close resources properly to avoid resource leaks. Additionally, always specify the full path or relative path of the file you want to read from or write to.

Question 7

When to use interface and abstract ?

In Java, both interfaces and abstract classes provide mechanisms for abstraction and defining contracts, but they serve different purposes and are used in different scenarios. Here's a guideline on when to use interfaces and abstract classes:

### Use Interfaces When:

1. **Defining Contracts**: Use interfaces when you want to define a contract for classes to implement. Interfaces declare abstract methods that concrete classes must implement, allowing you to specify a set of behaviors without providing implementation details.
2. **Implementing Multiple Behaviors**: Java allows classes to implement multiple interfaces, enabling them to inherit behavior from multiple sources. If a class needs to exhibit multiple behaviors, interfaces provide a way to achieve this.
3. **Enforcing Decoupling**: Interfaces help in achieving loose coupling between components by allowing classes to interact based on contracts rather than concrete implementations. This promotes flexibility, extensibility, and easier maintenance of the codebase.
4. **Creating API Contracts**: Interfaces are commonly used in API design to define contracts that other developers must adhere to when implementing their own classes. This ensures consistency and interoperability across different implementations.

### Use Abstract Classes When:

1. **Sharing Code Among Related Classes**: Abstract classes provide a way to share code and common functionality among related classes. They can contain both abstract methods and concrete methods, providing partial implementation that subclasses can override or extend.
2. **Defining Common Behavior**: Abstract classes can define common behavior and characteristics for a group of subclasses. Subclasses can inherit these characteristics and override methods as needed, while still retaining the common behavior.
3. **Partial Implementation**: Abstract classes can contain both abstract methods (methods without implementation) and concrete methods (methods with implementation). This allows you to provide default implementations for some methods, leaving others to be implemented by subclasses.
4. **Extending Existing Functionality**: If you need to extend an existing class hierarchy or provide a base class for multiple subclasses, abstract classes are a good choice. They allow you to define a common base for subclasses while providing flexibility for customization.

In summary, use interfaces to define contracts, enforce decoupling, and create API contracts, especially when dealing with multiple behaviors or implementing interoperability. Use abstract classes to share code among related classes, define common behavior, provide partial implementation, and extend existing functionality within a class hierarchy. Each has its strengths, and the choice between interfaces and abstract classes depends on the specific requirements and design goals of your application.

Question 8 :

Unique employee object as key in hash map ?

If you want to use a unique **Employee** object as a key in a **HashMap**, you need to ensure that the **Employee** class properly overrides the **hashCode()** and **equals()** methods. This is necessary for the **HashMap** to determine whether two **Employee** objects are equal and to correctly handle collisions.





Question 9:

Exception order in catch ?

In Java, when you have multiple catch blocks in a try-catch statement, they are evaluated in the order in which they appear. The catch blocks are evaluated from top to bottom, and the first catch block whose exception type matches the thrown exception type will be executed. Subsequent catch blocks are not evaluated once a matching catch block is found.

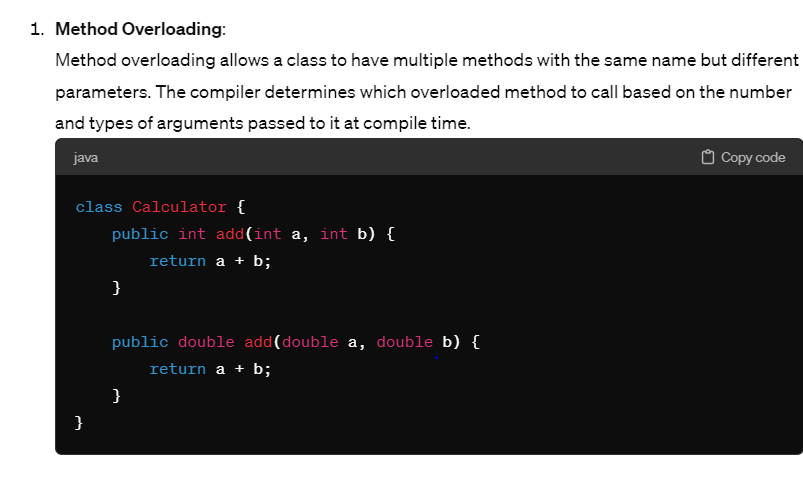
Question 10 :)

Static and dynamic polymorphism ?

Polymorphism in Java refers to the ability of a single method or interface to take different forms depending on the context in which it is used. There are two types of polymorphism in Java: static polymorphism (compile-time polymorphism) and dynamic polymorphism (runtime polymorphism).

***Note:****Run time polymorphism is implemented through****Method overriding.****Whereas Compile Time polymorphism is implemented through****Method overloading****and****Operator overloading****.*

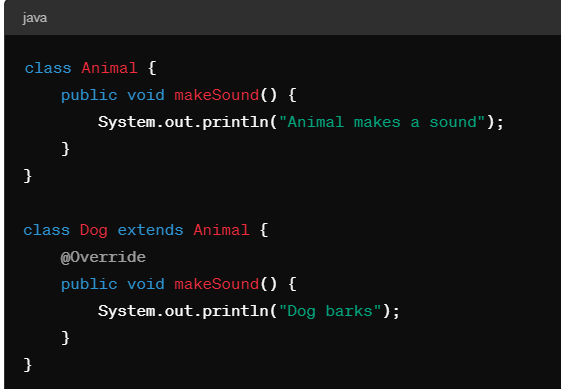
**Static polymorphism** occurs when the compiler determines which method or operation to execute at compile time based on the method signature and arguments provided. It is achieved through method overloading

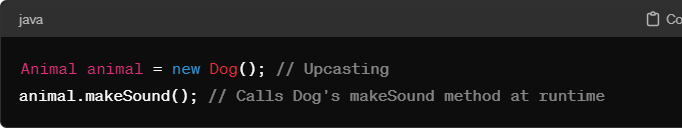
**

### Dynamic Polymorphism (Runtime Polymorphism):

Dynamic polymorphism occurs when the method or operation to execute is determined at runtime based on the actual type of the object. It is achieved through method overriding and interface implementation.

1. **Method Overriding** (Runtime Polymorphism): Dynamic polymorphism also applies to method overriding. When a method is invoked on a subclass object, the JVM determines which method to call based on the actual object type at runtime.

**



In summary, static polymorphism is determined at compile time based on method signatures and arguments, while dynamic polymorphism is determined at runtime based on the actual object type

Question 11:

Thread life cycle

The life cycle of a thread in Java refers to the different states that a thread can be in during its lifetime, from its creation to its termination. The Java language provides several methods and states to manage and control the life cycle of threads. Here are the various states of the thread life cycle:

1. **New**:
   * When a thread is created using the **new** keyword or by instantiating a class that extends **Thread** or implements **Runnable**, it is in the new state.
   * The thread has been created but has not yet started execution.
2. **Runnable**:
   * After calling the **start()** method on a newly created thread, it moves to the runnable state.
   * In this state, the thread is ready to run, but the scheduler has not yet selected it to be the running thread.
3. **Running**:
   * When the scheduler selects a thread from the runnable pool to execute, it enters the running state.
   * The thread's instructions are being executed by the CPU.
4. **Blocked (or Waiting)**:
   * A thread can enter the blocked state if it is waiting for a resource that is currently unavailable.
   * This could occur if the thread is waiting for I/O operations to complete, waiting for a lock, or waiting for a condition to be satisfied.
5. **Timed Waiting**:
   * A thread can enter the timed waiting state when it calls certain methods that require it to wait for a specified period of time.
   * Examples include calling **Thread.sleep()**, **Object.wait(timeout)**, or waiting on a **CountDownLatch**.
6. **Waiting**:
   * Similar to timed waiting, a thread can enter the waiting state when it calls certain methods that require it to wait indefinitely until another thread performs a specific action.
   * Examples include calling **Object.wait()** without a timeout or waiting on a condition variable.
7. **Terminated**:
   * A thread enters the terminated state when it completes its execution or when an uncaught exception terminates its execution abruptly.
   * Once a thread is terminated, it cannot be restarted or resumed.

Question 12

Cohesion ,coupling, aggregation, composition, association

Cohesion

All the functionalities which is related to that service sit inside that service that’s why we want our service to be highly **cohesive**

* Cohesion refers to the degree to which elements within a module (such as a class or a method) are related to each other and work together to achieve a common purpose or functionality.
* High cohesion indicates that the elements within the module are closely related and focused on a single task or responsibility.
* Low cohesion suggests that the elements within the module are loosely related and may perform multiple unrelated tasks.
* High cohesion is generally desirable as it leads to more maintainable and understandable code.

**Coupling**

**Coupling** means if the services are too much depend on each other , know too much about each other share a common source or database that means services are coupled with each other

1. **Coupling**:
   * Coupling refers to the degree of interdependence between modules or components in a system.
   * Tight coupling means that modules are highly dependent on each other, and changes in one module may require changes in many other modules.
   * Loose coupling means that modules are relatively independent of each other, and changes in one module have minimal impact on other modules.
   * Loose coupling is preferred as it promotes modular design, reusability, and maintainability.

**Aggregation**

**aggregation represents a "has-a" relationship**, where one class contains another class as part of its structure, but they can exist independently.

Example: A university "has" departments.

Aggregation is also a relationship between two classes where one class contains another class, but the contained class has an independent existence

**Composition :**

Composition is a stronger relationship, where one class owns another class, and the owned class cannot exist independently

****

1. **Association**:
   * Association represents a relationship between two classes where objects of one class are related to objects of another class.
   * It's a general term and doesn't imply ownership or containment like composition or aggregation.
   * Associations can be one-to-one, one-to-many, or many-to-many, and they can be bidirectional or unidirectional.

In summary, cohesion and coupling are principles that guide the design of individual modules and the relationships between modules. Composition and aggregation are specific forms of relationships between classes, with composition implying strong ownership and lifecycle management, while aggregation implies a weaker relationship and independent existence of the contained object. Association represents a more general relationship between classes without implying ownership or lifecycle management.

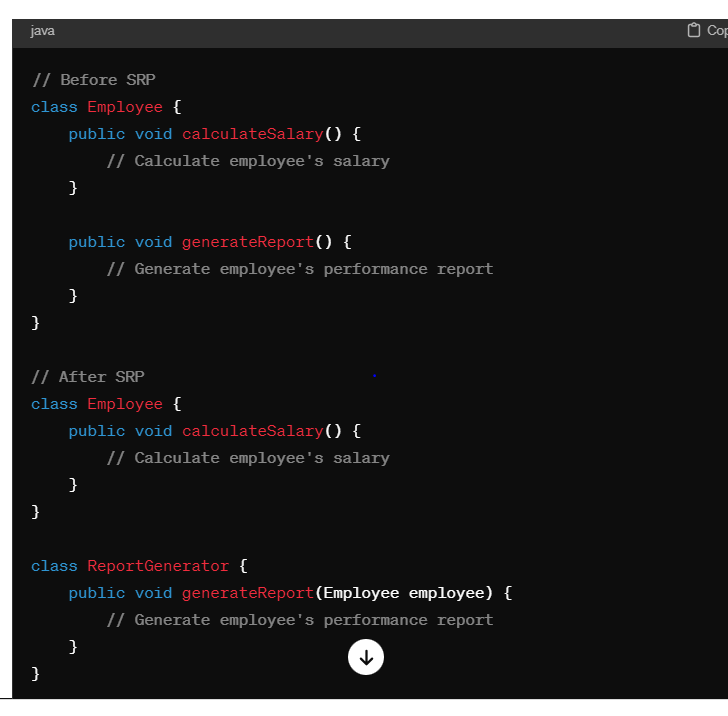
Question 12

Solid principles

The SOLID principles are a set of five design principles for writing maintainable and understandable object-oriented software. Here's a brief overview of each principle along with code examples:

### 1. Single Responsibility Principle (SRP):

* A class should have only one reason to change, meaning that it should have only one responsibility or concern.
* This principle encourages high cohesion and reduces coupling between classes.



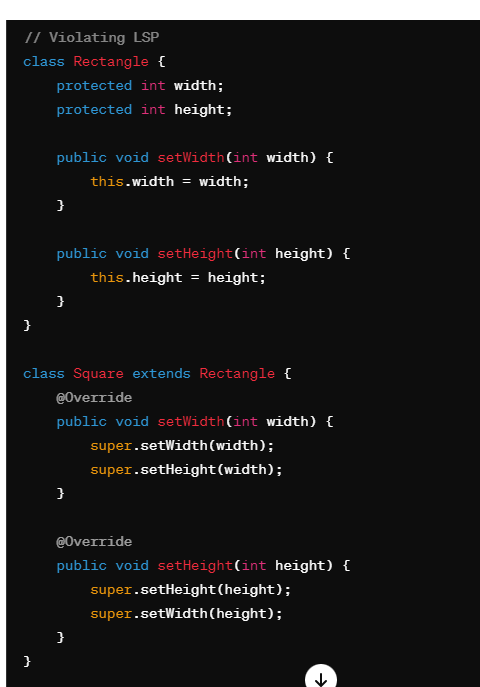
* Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.
* This principle encourages the use of inheritance, polymorphism, and interfaces to allow for easy extension without modifying existing code.

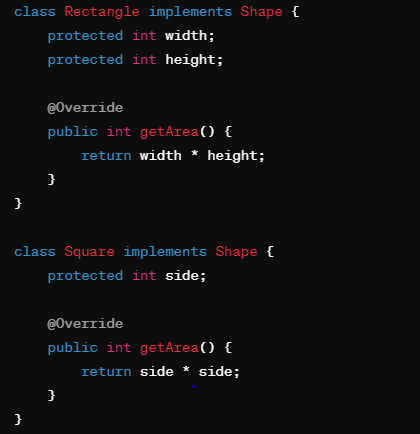
**Example**:



### Liskov Substitution Principle (LSP):

* Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.
* This principle ensures that inheritance hierarchies are well-designed and that subclasses adhere to the contract defined by the superclass.





### 4. Interface Segregation Principle (ISP):

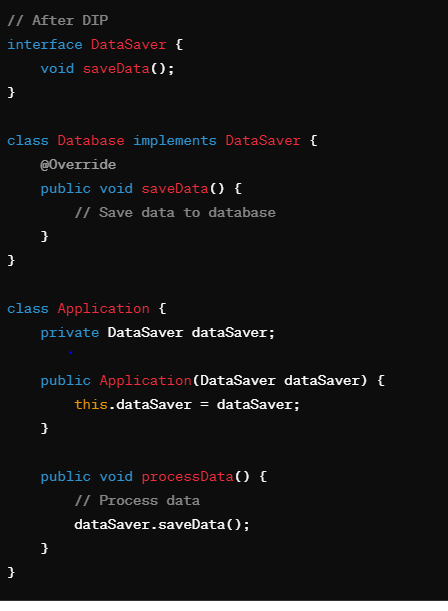
* Clients should not be forced to depend on interfaces they do not use.
* This principle encourages the creation of smaller, cohesive interfaces tailored to specific client needs rather than large, monolithic interfaces.



### 5. Dependency Inversion Principle (DIP):

* High-level modules should not depend on low-level modules. Both should depend on abstractions.
* Abstractions should not depend on details. Details should depend on abstractions.
* This principle promotes loose coupling between modules and allows for easier changes and substitutions.





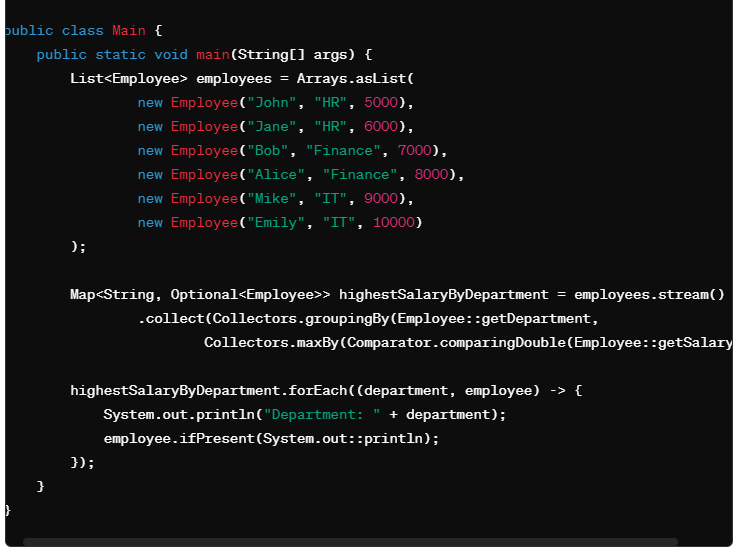
These code examples demonstrate how to apply each SOLID principle in Java to create more maintainable, flexible, and understandable software.

Question 13

Get highest salary from each department?

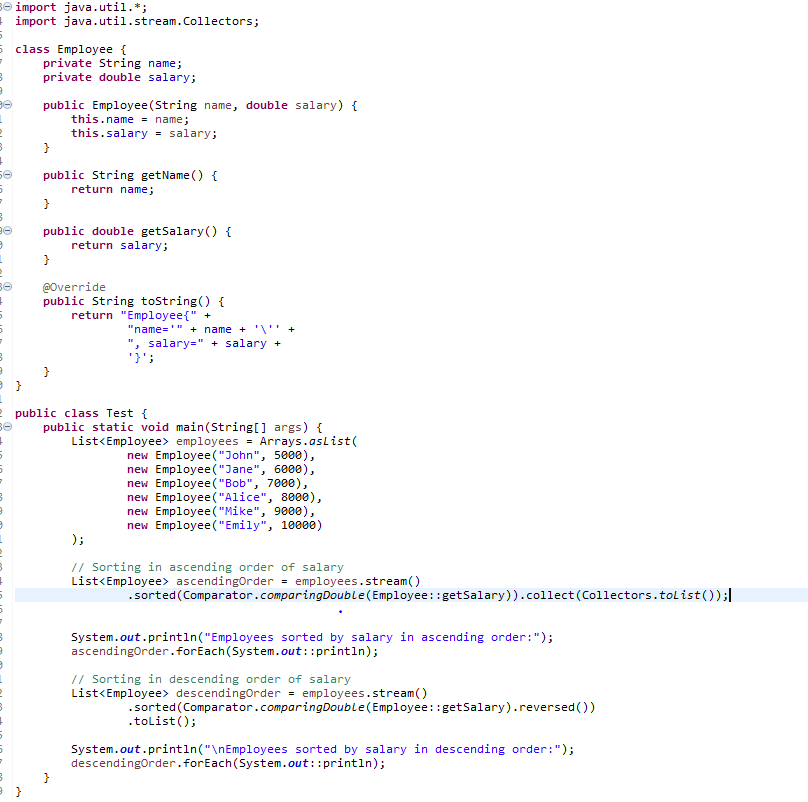
1. Group the list of employees by department.
2. For each department group, find the employee with the highest salary.
3. Collect the result into a map where the key is the department and the value is the employee with the highest salary.



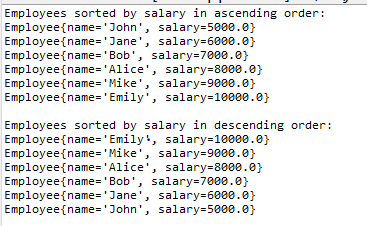


Question 14

**sort list of employee using salary ascending and descending in java 8 with code ?**

****

**//output**

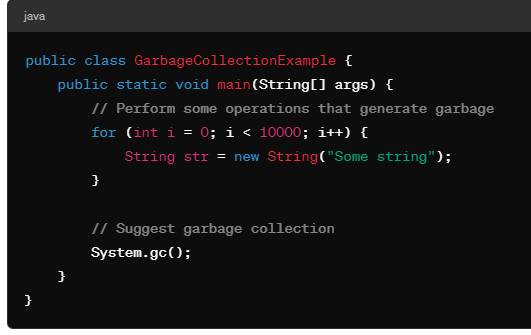
****

**Question 15**

**is there any manual way to do garbage collector with code ?**

In Java, you can manually suggest the JVM to run the garbage collector, but you cannot force it to perform garbage collection at a specific time. The decision to run the garbage collector is ultimately determined by the JVM based on various factors such as memory pressure, allocation rate, and heap size.

However, you can suggest garbage collection using the **System.gc()** method. This method is a hint to the JVM that it should perform garbage collection, but it does not guarantee that garbage collection will actually occur. Here's how you can use it:

****

Keep in mind the following points when using **System.gc()**:

1. Invoking **System.gc()** doesn't ensure immediate garbage collection. The JVM may choose to ignore the suggestion based on its internal logic.
2. Explicitly calling **System.gc()** should generally be avoided because it can disrupt the JVM's optimization and may not improve performance.
3. Modern JVM implementations are sophisticated and usually perform garbage collection efficiently without manual intervention.

In most cases, it's better to rely on the JVM's automatic garbage collection mechanism rather than attempting to trigger it manually. Trust the JVM to manage memory effectively, and focus on writing efficient and memory-conscious code.

**Question 16**

**Stack memory vs heap memory ?**

Stack memory and heap memory are two distinct regions in a computer's memory allocation system, and they are used for different purposes:

### Stack Memory:

1. **Purpose**:
   * Stack memory is used for storing method calls, local variables, and reference variables.
   * It is primarily used for supporting method calls and managing function execution.
2. **Allocation**:
   * Memory allocation on the stack is done in a sequential manner, known as Last-In-First-Out (LIFO) order.
   * When a method is called, a new stack frame is created, which contains space for local variables, method arguments, return address, and other bookkeeping information.
   * As methods return, their stack frames are deallocated, and the memory is reclaimed.
3. **Size**:
   * Stack memory is limited in size and typically smaller than heap memory.
   * The size of the stack is determined at compile time or runtime and depends on factors such as the operating system and platform settings.
4. **Access**:
   * Access to stack memory is faster than heap memory because memory allocation and deallocation are simple and predictable.
5. **Usage**:
   * Stack memory is used for managing the execution flow of a program, handling method calls, and storing local variables with short lifetimes.

### Heap Memory:

1. **Purpose**:
   * Heap memory is used for dynamic memory allocation, which means objects created at runtime are stored in the heap.
   * It is used for storing objects, arrays, and other data structures that require memory allocation during program execution.
2. **Allocation**:
   * Memory allocation on the heap is more flexible than on the stack and can be done at any time during program execution.
   * Objects created on the heap persist until they are explicitly deallocated by the garbage collector or until the program terminates.
3. **Size**:
   * Heap memory is typically larger than stack memory and can grow dynamically as needed.
   * The size of heap memory is limited by factors such as the available physical memory and system settings.
4. **Access**:
   * Access to heap memory is slower than stack memory because memory allocation and deallocation may involve more complex operations.
   * However, accessing objects in heap memory is efficient once they are allocated.
5. **Usage**:
   * Heap memory is used for storing objects with longer lifetimes, such as objects created dynamically during program execution, large data structures, and objects shared across multiple methods or threads.

In summary, stack memory is used for managing method calls and local variables with short lifetimes, while heap memory is used for dynamic memory allocation and storing objects with longer lifetimes. Understanding the differences between stack and heap memory is important for efficient memory management and designing scalable and robust software systems.

**Question 17**

**What is immutable class and example ?**

In Java, an **immutable class** is a class whose instances cannot be modified after they are created. Once an instance of an immutable class is created, its state (the values of its fields) cannot be changed. This is achieved by making the fields of the class final and private, so they cannot be modified directly, and by not providing any setter methods that would allow modification of the fields.

In Java, you can create an immutable class by following these steps:

1. Declare the class as **final** to prevent inheritance.
2. Make all fields **private** and **final**, so they cannot be modified once initialized.
3. Provide only getters methods to access the fields.
4. Ensure any mutable objects within the class are not directly exposed or modified.

****

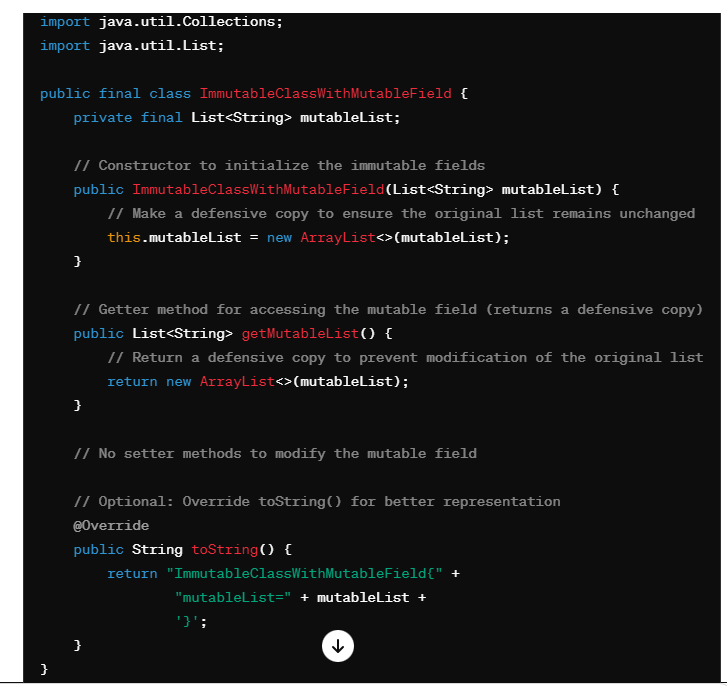
**Question 18 :**

**Having mutuable property inside immutable class.how to stop to break immutability ?**

**Defensive copying** is a technique where an identical, but disconnected, copy of an object (including arrays) is returned instead of the original one (similar to a deep copy - Wiki). Thus any modification to the returned object will not affect the original object.

To ensure immutability in a class containing mutable fields, you need to take precautions to prevent modification of those mutable fields after object creation. Here's how you can achieve that:

1. **Make the mutable fields private:** Encapsulate the mutable fields by declaring them as private.
2. **Do not provide setter methods:** Remove any methods that allow modification of the mutable fields.
3. **If the mutable field is a collection (e.g., List, Map), return a defensive copy:** When returning the mutable field from a getter method, return a copy of the field instead of the field itself. This ensures that the original field remains unchanged.



In this example:

* The **mutableList** field is declared as **private final**, ensuring immutability.
* The constructor takes a **List<String>** as a parameter and makes a defensive copy of it to initialize the **mutableList** field.
* The **getMutableList()** method **returns a defensive copy of the** **mutableList** field to prevent modification of the original list.
* There are no setter methods, preventing modification of the mutable field after object creation.

By following these practices, you can ensure immutability even when dealing with classes containing mutable fields.

**Question 19 :**

**What we did in hashmap for performance improvement using java 8 ?**

In Java 8, several improvements were made to the **HashMap** class to enhance its performance and usability. Some of the key improvements include:

1. **Tree-based structure for collision resolution in large buckets**: In earlier versions of Java, when multiple entries hashed to the same bucket, they were stored as a linked list. This could lead to performance degradation if many entries hashed to the same bucket, as the lookup time would be linear in the number of elements in the bucket. In Java 8, once a bucket exceeds a certain threshold (usually 8), it switches from using a linked list to a balanced binary tree for collision resolution. This helps improve the worst-case performance of operations like **get()**, **put()**, and **remove()** from O(n) to O(log n) in large buckets.
2. **Reduction of hash collisions**: The implementation of the **hashCode()** method for certain key types (such as **String**, **Integer**, **Long**, etc.) was enhanced to produce better distribution of hash codes, reducing the likelihood of hash collisions. This helps improve the overall performance of **HashMap** by reducing the number of collisions and subsequent treeification of buckets.
3. **Parallel operations**: Java 8 introduced support for parallel bulk operations on **HashMap** using the Stream API. Operations such as **forEach()**, **removeIf()**, and **replaceAll()** can now be performed in parallel, potentially improving the performance for large maps by utilizing multiple CPU cores.
4. **Performance improvements in resizing**: The resizing strategy of **HashMap** was improved to reduce the number of elements moved when resizing. This reduces the overhead associated with resizing the hash table, leading to better overall performance.

These improvements in Java 8's **HashMap** implementation help enhance its performance and scalability, making it more efficient for applications that heavily rely on hash maps for storing and retrieving key-value pairs.

**Question 20**

**What is race condition ?**

Race conditions are most commonly associated with computer science and programming. They occur when two computer program processes, or threads, attempt to access the same resource at the same time and cause problems in the system. Race conditions are considered a common issue for multithreaded applications

Question 21.

**which collection data structure use for multithreaded environment ?**

In a multithreaded environment, where multiple threads may concurrently access and modify shared data structures, it's essential to use thread-safe data structures to prevent race conditions and ensure data integrity. Java provides several thread-safe collections in the **java.util.concurrent** package, which are designed specifically for multithreaded environments. Some commonly used thread-safe data structures include:

1. **ConcurrentHashMap**: ConcurrentHashMap is a thread-safe implementation of the Map interface. It allows multiple threads to read and write concurrently without the need for external synchronization. It achieves concurrency by dividing the map into segments, each of which is independently synchronized.
2. **ConcurrentLinkedQueue**: ConcurrentLinkedQueue is a thread-safe implementation of the Queue interface. It provides a non-blocking, lock-free, and scalable queue implementation suitable for high-concurrency applications. It uses techniques such as lock-free linked nodes to enable concurrent access without the need for explicit synchronization.
3. **CopyOnWriteArrayList**: CopyOnWriteArrayList is a thread-safe implementation of the List interface. It maintains immutability by creating a new copy of the underlying array whenever a modification operation (such as add, set, or remove) is performed. While it provides excellent thread safety for read-heavy scenarios, it may not be suitable for scenarios with frequent modifications due to the overhead of copying the array.
4. **ConcurrentSkipListMap and ConcurrentSkipListSet**: ConcurrentSkipListMap and ConcurrentSkipListSet are thread-safe implementations of the SortedMap and SortedSet interfaces, respectively. They provide sorted collections with logarithmic time complexity for most operations, making them suitable for concurrent access in multithreaded environments.
5. **BlockingQueue**: BlockingQueue is an interface that extends the Queue interface to support blocking operations such as put and take, which block the calling thread if the queue is empty or full, respectively. Implementations like LinkedBlockingQueue and ArrayBlockingQueue provide thread-safe blocking queues suitable for producer-consumer scenarios.

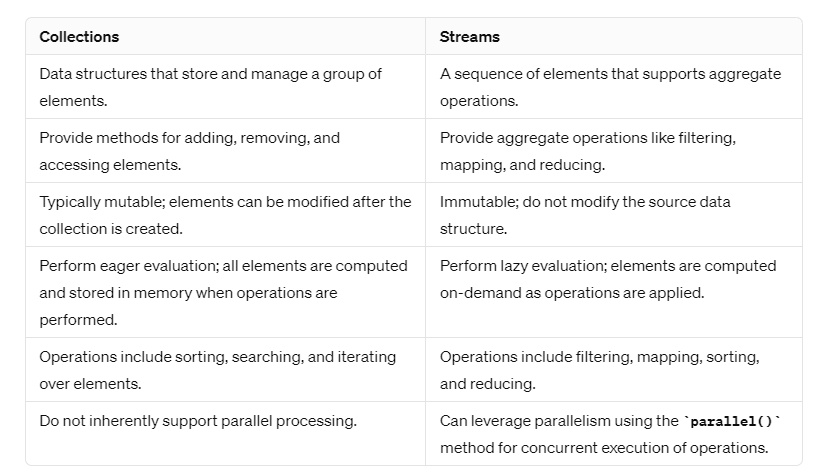
**Question 22**

**What is functional programming ?**

Basically, functional programming is a style of writing computer programs that treat computations as evaluating mathematical functions. **In mathematics,** a function is an expression that relates an input set to an output set. Importantly, the output of a function depends only on its input.

Question 23

Difference between collection vs stream ?



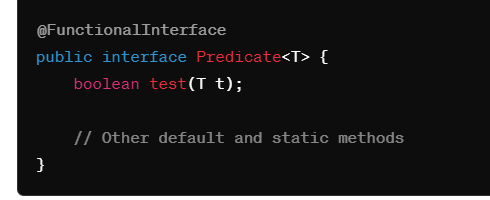
Streams promote **immutability, lazy evaluation, and parallelism**, making them well-suited for functional programming paradigms.

**Question 24**

**Java 8 predicate ?**

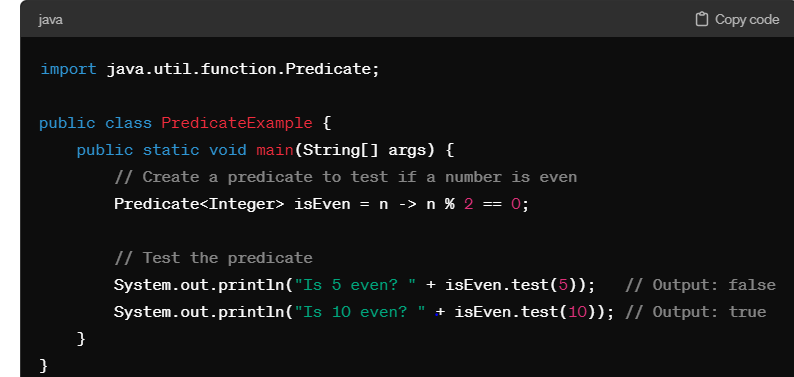
In Java 8, **Predicate** is a functional interface that represents a predicate (boolean-valued function) of one argument. It is commonly used in functional-style programming and provides a way to test a condition and return a boolean result.

The **Predicate** interface in Java 8 is defined as follows:

****

As shown in the definition, the **Predicate** interface has a single abstract method **test(T t)** that takes an argument of type **T** and returns a boolean value. The **test** method is used to evaluate the predicate condition on the input argument.

Here's an example of how to use the **Predicate** interface in Java 8:

****

**Predicate** interfaces are commonly used in various Java APIs, such as the **filter** method in streams, to filter elements based on a condition,

**Question 25:**

we have synchronized hashmap then why concurrent hash map ?

While both **SynchronizedHashMap** and **ConcurrentHashMap** provide thread-safe access to a map, they differ in their implementation and performance characteristics. Here's why you might choose **ConcurrentHashMap** over **SynchronizedHashMap**:

1. **Granularity of Locking**:
   * **SynchronizedHashMap** uses a single lock to synchronize all its methods, which means that only one thread can access the map at a time. This can lead to contention and reduced performance, especially in high-concurrency scenarios.
   * **ConcurrentHashMap**, on the other hand, uses a finer-grained approach by partitioning the map into segments, each of which is independently locked. This allows multiple threads to access different segments concurrently, reducing contention and improving scalability.
2. **Performance**:
   * Due to its finer-grained locking mechanism, **ConcurrentHashMap** typically offers better performance than **SynchronizedHashMap**, especially in scenarios with high levels of concurrency.
   * **ConcurrentHashMap** is optimized for concurrent access and can handle a large number of read and write operations concurrently without incurring significant overhead.
3. **Concurrent Iteration**:
   * **ConcurrentHashMap** supports concurrent iteration, which means that you can iterate over the map while other threads are modifying it. This is not possible with **SynchronizedHashMap**, where iteration requires acquiring the lock, which can block other threads attempting to modify the map.
4. **Scalability**:
   * **ConcurrentHashMap** scales better than **SynchronizedHashMap** as the number of threads accessing the map increases. Its finer-grained locking allows for better parallelism and reduces contention, leading to improved scalability.
5. **Additional Features**:
   * **ConcurrentHashMap** offers additional features such as atomic operations (**putIfAbsent**, **remove**, **replace**) and non-blocking operations (**compute**, **merge**) that are not available in **SynchronizedHashMap**. These features make it more versatile and suitable for a wider range of use cases.

**Question 26 :**

**Predefined functional interface ?**

n Java, functional interfaces are interfaces that have exactly one abstract method. These interfaces are also known as Single Abstract Method (SAM) interfaces. While you can create your own functional interfaces, Java provides several predefined functional interfaces in the **java.util.function** package, which are commonly used in functional programming and lambda expressions. Some of the most commonly used predefined functional interfaces include:

1. **Predicate<T>**:
   * Represents a predicate (boolean-valued function) of one argument.
   * Method: **boolean test(T t)**
2. **Consumer<T>**:
   * Represents an operation that accepts a single input argument and returns no result.
   * Method: **void accept(T t)**
3. **Function<T, R>**:
   * Represents a function that accepts one argument and produces a result.
   * Method: **R apply(T t)**
4. **Supplier<T>**:
   * Represents a supplier of results.
   * Method: **T get()**
5. **UnaryOperator<T>**:
   * Represents an operation on a single operand that produces a result of the same type as its operand.
   * Method: **T apply(T t)**
6. **BinaryOperator<T>**:
   * Represents an operation upon two operands of the same type, producing a result of the same type as the operands.
   * Method: **T apply(T t1, T t2)**
7. **BiFunction<T, U, R>**:
   * Represents a function that accepts two arguments and produces a result.
   * Method: **R apply(T t, U u)**
8. **BiPredicate<T, U>**:
   * Represents a predicate (boolean-valued function) of two arguments.
   * Method: **boolean test(T t, U u)**

**Question 27 :**

**What is method reference ?**

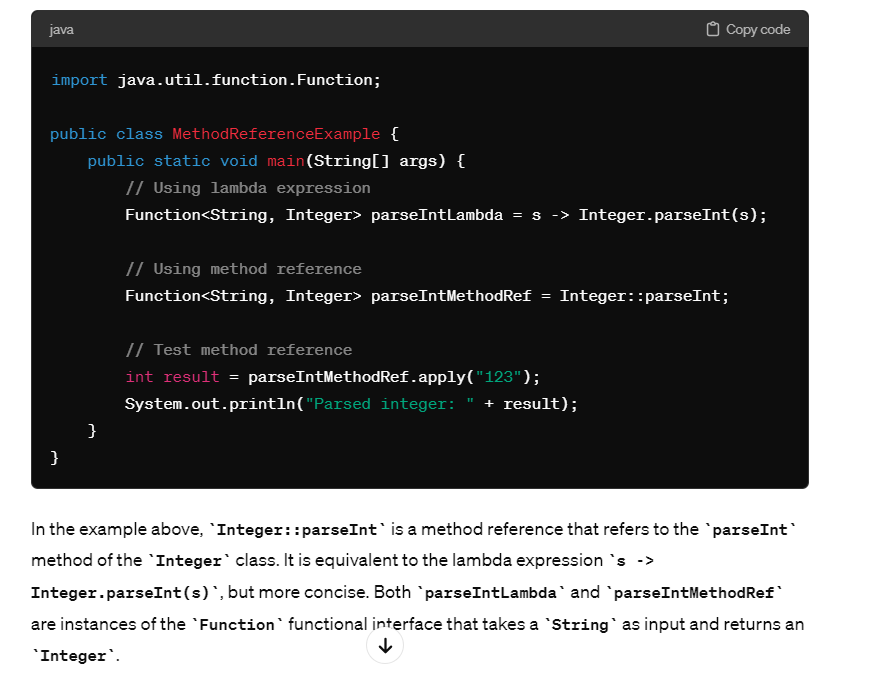
In Java 8, method reference is a shorthand syntax that allows you to refer to methods or constructors of classes or objects without invoking them explicitly. It is a concise way to pass methods as arguments to functional interfaces or lambda expressions.

Method references can be used in situations where the lambda expression simply delegates to an existing method. They provide more readable and concise code by eliminating boilerplate code and improving code readability.

There are several types of method references:

1. **Reference to a static method**: Syntax: **ClassName::staticMethodName**
   * Example: **Integer::parseInt** is a reference to the static method **parseInt** of the **Integer** class.
2. **Reference to an instance method of an object**: Syntax: **object::instanceMethodName**
   * Example: **System.out::println** is a reference to the instance method **println** of the **PrintStream** object **System.out**.
3. **Reference to an instance method of a particular object type**: Syntax: **ClassName::instanceMethodName**
   * Example: **String::length** is a reference to the instance method **length** of the **String** class.
4. **Reference to an instance method of an arbitrary object of a particular type**: Syntax: **ClassName::instanceMethodName**
   * Example: **String::toUpperCase** is a reference to the instance method **toUpperCase** of the **String** class. When used in a context where a **Function** or **Consumer** interface is expected, it would apply to the argument passed to that function or consumer.
5. **Reference to a constructor**: Syntax: **ClassName::new**
   * Example: **ArrayList::new** is a reference to the constructor of the **ArrayList** class.

Here's a simple example demonstrating the use of method references:

****

**Question 28 :**

**What is reflection and why it is useful ?**

**Java Reflection** is a *process of examining or modifying the run time behavior of a class at run time*

In Java, **java.lang.Class** is a special class that represents a class or interface at runtime. It is a fundamental part of Java's reflection API, which provides the ability to inspect and manipulate the structure of classes, interfaces, fields, methods, and constructors at runtime.

Reflection can be used to get information about class, constructors, and methods

