# Top Core Java Interview Questions

**Q) Why Java is not 100% Object-oriented?**

Answer: Java is often considered to be an object-oriented programming (OOP) language, but it's true that it's not 100% pure object-oriented. There are a few reasons for this:

1. \*\*Primitive Data Types:\*\* Java includes primitive data types like `int`, `float`, `char`, etc., which are not objects. These data types are used to represent basic values and are not part of the class hierarchy. In contrast, a pure object-oriented language would have everything represented as objects.

2. \*\*Static Methods and Fields:\*\* Java allows the creation of static methods and fields within classes. These are associated with the class itself, not with individual instances of the class. In a pure object-oriented language, all behavior and attributes would be tied to objects, rather than being associated with the class itself.

3. \*\*Procedural Programming:\*\* Java supports procedural programming constructs, such as using methods for structuring code and performing operations. While the focus is on objects and classes, Java still allows for procedural-style programming.

4. \*\*Arrays:\*\* Arrays in Java are not true objects. They are a kind of special construct that predates the development of Java's object-oriented features. Arrays have fixed sizes and specific syntax for accessing elements, which is different from the way objects are manipulated.

5. \*\*Primitive Type Conversion:\*\* In Java, you often need to convert between primitive data types (e.g., from `int` to `double`). These conversions can involve direct casting and are not purely object-oriented.

**Q. Why pointers are not used in Java? Mostly asked in java interview questions and Answers in MNC to check you basic knowledge of java and programming languages and logics behind them.**

The concept of pointers is commonly brought up in interviews as a way to assess a candidate's understanding of programming languages and memory management. However, it's important to clarify that Java does not use explicit pointers like some other programming languages (such as C and C++). Instead, Java uses references, which serve a similar purpose but are managed differently. Here's why pointers are not used in Java:

1. \*\*Safety and Memory Management:\*\*

- Pointers in languages like C and C++ can lead to issues like null pointer dereferences, memory leaks, and undefined behavior.

- Java's references are designed to be safer and more reliable. The Java Virtual Machine (JVM) handles memory management, including automatic garbage collection, to help prevent these problems.

2. \*\*Avoiding Memory Corruption:\*\*

- Pointers can be used to directly manipulate memory addresses, which can lead to memory corruption if used improperly.

- Java's reference model prevents direct manipulation of memory addresses, enhancing security and stability.

3. \*\*Platform Independence:\*\*

- Java's main goal is platform independence. Using explicit pointers could lead to non-portable code that works on one platform but not on another.

- References, as managed by the JVM, allow Java programs to be compiled once and run on any platform with a compatible JVM implementation.

4. \*\*Simpler Syntax:\*\*

- Pointers often involve complex syntax for dereferencing, pointer arithmetic, and memory management.

- Java's reference syntax is simpler and more intuitive, promoting readability and maintainability.

5. \*\*Reducing Programmer Errors:\*\*

- Pointers can introduce a range of programming errors, such as memory leaks, dangling pointers, and data corruption.

- Java's reference management helps reduce such errors by automating memory management and providing safer object access.

In interviews, the question about pointers in Java can gauge whether a candidate understands these fundamental differences and can explain the reasoning behind Java's design choices. It's an opportunity to demonstrate knowledge of memory management, security considerations, and the principles of platform independence that underlie Java's design.

**Q. What is JIT compiler in Java?**

The JIT (Just-In-Time) compiler in Java is a crucial component of the Java Virtual Machine (JVM) that plays a significant role in optimizing the performance of Java programs. It is responsible for translating Java bytecode (compiled Java code) into native machine code that can be executed by the host system's hardware. Here's how the JIT compiler works and its benefits:

1. \*\*Compilation Stages:\*\*

- When you write Java code, it's first compiled into bytecode by the Java compiler.

- When a Java program is executed, the bytecode is interpreted by the JVM's interpreter.

- The JIT compiler comes into play to improve execution speed. It compiles parts of the bytecode into native machine code just before they are executed.

2. \*\*Dynamic Compilation:\*\*

- The term "Just-In-Time" refers to the fact that the compilation occurs at runtime, just before the code is about to be executed.

- The JIT compiler analyzes the bytecode and identifies "hot spots" or frequently executed parts of the code.

3. \*\*Optimization:\*\*

- The JIT compiler applies various optimization techniques to the selected bytecode, transforming it into highly efficient native machine code.

- These optimizations might include inlining method calls, eliminating dead code, reordering instructions for better pipelining, and more.

4. \*\*Caching and Deoptimization:\*\*

- The compiled native code is stored in a cache to avoid recompilation of the same code in the future.

- If the runtime conditions change (e.g., if the JVM detects that the assumptions made during compilation are no longer valid), the code can be deoptimized and recompiled with adjusted optimizations.

5. \*\*Trade-Offs:\*\*

- The JIT compiler introduces an initial overhead as it needs to perform compilation before code execution. However, this overhead is often outweighed by the improved performance during execution.

- JIT compilation benefits long-running applications more than short-lived ones because the overhead of compilation is amortized over time.

6. \*\*Tiered Compilation:\*\*

- Modern JVMs often employ a tiered compilation strategy. They start with quick and lightweight optimizations and gradually apply more aggressive optimizations as the code is executed more frequently.

7. \*\*HotSpot JVM:\*\*

- The HotSpot JVM, developed by Oracle, is a popular Java implementation that includes an advanced JIT compiler.

- The HotSpot JVM performs extensive profiling to determine which parts of the code are worth optimizing, leading to highly efficient compiled code.

Overall, the JIT compiler is a key reason why Java programs can achieve good performance despite being executed in a virtualized environment. It dynamically optimizes the code based on runtime behavior, allowing Java applications to strike a balance between interpreted execution and fully compiled execution.

**Q) Why String is immutable in java?**

Strings are immutable in Java for several important reasons:

1. \*\*Efficiency:\*\* Immutability allows strings to be cached and reused. Since strings are widely used in programs, having a cache of frequently used strings improves memory utilization and performance. For example, if multiple variables reference the same string literal, they all point to the same memory location, reducing memory consumption.

2. \*\*Security:\*\* String immutability contributes to security, particularly in situations where strings are used for passwords or sensitive data. If strings were mutable, a malicious program could modify a string after it has been created, potentially compromising security.

3. \*\*Synchronization:\*\* Immutable strings are inherently thread-safe. In a multi-threaded environment, there is no need to synchronize access to strings because they cannot change after creation. This simplifies concurrent programming and reduces the risk of data races.

4. \*\*Hashing and Caching:\*\* Strings are widely used as keys in hash-based data structures like HashMaps. If strings were mutable, changing a string could lead to inconsistent hashing behavior and errors in these data structures. Immutable strings guarantee that the hash code remains consistent throughout the object's lifetime.

5. \*\*Method Caching and Optimization:\*\* String literals are often used in method calls. Since strings are immutable, the Java compiler can optimize method calls by caching the results of string operations for reuse.

6. \*\*Predictable Behavior:\*\* Immutability ensures that the value of a string remains constant over time. This predictability is crucial for maintaining a consistent state in applications.

7. \*\*String Pool:\*\* Java maintains a pool of string literals in memory called the "string pool." When you create a new string, the JVM first checks if a string with the same value already exists in the pool. If it does, the existing string is returned, and a new object is not created. This reduces memory consumption.

8. \*\*Safe Sharing:\*\* Immutable strings can be safely shared among different parts of a program without worrying about unintended modifications.

Overall, the decision to make strings immutable in Java was based on a combination of performance, security, and reliability considerations. It has become an essential characteristic of the language and has numerous benefits in terms of memory usage, concurrent programming, and consistent behavior.

**Q) What is a marker interface?**

A marker interface in Java is an interface that doesn't declare any method or member. Its sole purpose is to indicate a certain capability or characteristic to the implementing classes. Marker interfaces are used as a form of metadata to add specific information to classes that implement them. They are also known as "tag interfaces" because they "tag" a class with a certain property.

Here are a few key points about marker interfaces:

1. \*\*No Methods:\*\* Marker interfaces do not define any methods. They are empty interfaces with no method declarations.

2. \*\*Indicative Purpose:\*\* The main purpose of a marker interface is to provide a way to categorize or flag classes without adding any behavior to them.

3. \*\*Examples:\*\*

- `Serializable`: The `Serializable` interface is a classic example of a marker interface. It indicates that a class is serializable and can be converted into a stream of bytes for storage or transmission.

- `Cloneable`: The `Cloneable` interface indicates that a class can be cloned using the `Object.clone()` method.

- `Remote` (used in early versions of Java RMI): The `Remote` interface indicated that a class could be accessed remotely using Java Remote Method Invocation (RMI).

4. \*\*Framework Use:\*\* Marker interfaces are often used in frameworks or libraries to trigger certain behaviors or indicate that a class should be treated differently.

5. \*\*Advantages:\*\*

- Simplicity: Marker interfaces are simple and easy to use.

- Clear Intent: They clearly express the intended purpose or capability of a class.

6. \*\*Drawbacks:\*\*

- Limited Information: Marker interfaces don't provide any additional information about the specifics of the capability or behavior. Developers often need to consult documentation to understand the purpose.

7. \*\*Alternatives:\*\*

- In modern Java, annotations are commonly used to provide metadata to classes. Annotations can carry additional information and are more flexible than marker interfaces.

It's worth noting that while marker interfaces are a concept in Java, their usage has somewhat diminished in favor of annotations due to the limitations of marker interfaces and the flexibility offered by annotations in conveying metadata and behaviors to classes.

**Q) Can you override a private or static method in Java**

In Java, you cannot override a private or static method. Here's why:

1. \*\*Private Methods:\*\*

- Private methods are declared with the `private` access modifier, which restricts their visibility to the defining class only.

- Subclasses cannot directly access private methods of their parent class. They don't inherit them.

- As a result, there's no concept of overriding private methods because they are not accessible to subclasses.

2. \*\*Static Methods:\*\*

- Static methods belong to the class itself rather than instances of the class.

- When you declare a method as `static`, it's associated with the class and not with instances. Subclasses can define methods with the same signature, but they are not considered overrides.

- Instead, they are "hidden" by the static method of the subclass. The method that is called is determined by the reference type (compile-time type), not the actual object type (runtime type).

Here's an example to illustrate this:

```java

class Parent {

private void privateMethod() {

System.out.println("Parent's private method");

}

static void staticMethod() {

System.out.println("Parent's static method");

}

}

class Child extends Parent {

// This is not an override, but a new method in Child class.

private void privateMethod() {

System.out.println("Child's private method");

}

// This is also not an override; it's a new static method in Child class.

static void staticMethod() {

System.out.println("Child's static method");

}

}

public class Main {

public static void main(String[] args) {

Parent parent = new Child();

parent.privateMethod(); // This won't compile or execute.

parent.staticMethod(); // This will execute "Parent's static method".

}

}

```

A screenshot of a computer program

Description automatically generated

In the example above, the `privateMethod()` in the `Child` class is not an override of the `privateMethod()` in the `Parent` class. Similarly, the `staticMethod()` in the `Child` class is not an override of the `staticMethod()` in the `Parent` class.

In summary, private and static methods do not participate in traditional method overriding. Subclasses cannot access private methods from the parent class, and static methods are associated with the class itself, not instances.

**Q) Does “finally” always execute in Java?**

Yes, the `finally` block in Java always executes, regardless of whether an exception is thrown or not, as long as the corresponding `try` block was entered.

Here's how the `finally` block works:

1. \*\*Normal Execution:\*\* If no exception is thrown inside the `try` block, the `finally` block still executes after the `try` block completes its execution.

2. \*\*Exception Thrown:\*\* If an exception is thrown within the `try` block and it's caught by a matching `catch` block, the `finally` block is executed after the `catch` block.

3. \*\*Exception Uncaught:\*\* Even if an exception is thrown within the `try` block and is not caught by a matching `catch` block within the same `try-catch-finally` structure, the `finally` block still executes before the exception propagates up the call stack.

The primary use of the `finally` block is to ensure that certain cleanup or resource release operations are performed regardless of whether an exception occurred or not. For example, closing files, releasing database connections, or freeing up system resources can be placed in the `finally` block to ensure they are executed regardless of exceptions.

Here's an example to illustrate this:

```java

public class FinallyExample {

public static void main(String[] args) {

try {

int result = divide(10, 2);

System.out.println("Result: " + result);

} catch (ArithmeticException e) {

System.out.println("Exception caught: " + e.getMessage());

} finally {

System.out.println("Finally block always executes.");

}

}

public static int divide(int a, int b) {

return a / b;

}

}

```

A screen shot of a computer program

Description automatically generated

In this example, the `finally` block is executed no matter whether an exception is thrown or not. This guarantees that the cleanup or resource release code within the `finally` block is executed regardless of the program flow.

**Q) What Methods Does the Object Class Have?**

The `Object` class is a fundamental class in Java that serves as the root class for all other classes. It provides some essential methods that are inherited by all classes in Java. Here are some of the key methods that the `Object` class has:

1. \*\*`toString()` Method:\*\*

- Returns a string representation of the object.

- The default implementation in `Object` returns the class name followed by the hash code.

2. \*\*`equals(Object obj)` Method:\*\*

- Compares two objects for equality.

- The default implementation compares object references for identity (not content). Subclasses usually override this method to provide custom equality comparison.

3. \*\*`hashCode()` Method:\*\*

- Returns a hash code value for the object.

- The default implementation returns the memory address of the object, which can be overridden for better hash distribution.

4. \*\*`getClass()` Method:\*\*

- Returns the runtime class of an object.

- Useful for getting information about the actual class of an object.

5. \*\*`notify()` and `notifyAll()` Methods:\*\*

- Used for inter-thread communication using the wait-notify mechanism.

- Allows threads to wait for signals from other threads.

6. \*\*`wait()` Method:\*\*

- Causes the current thread to wait until another thread notifies it.

- Used for synchronization and coordination between threads.

7. \*\*`finalize()` Method:\*\*

- Called by the garbage collector before an object is removed from memory.

- Provides an opportunity to release resources or perform cleanup operations.

8. \*\*`clone()` Method:\*\*

- Creates a shallow copy of the object.

- Requires implementing the `Cloneable` interface and overriding the method for proper deep copying.

These methods are inherited by all classes in Java, as every class implicitly extends the `Object` class if no other superclass is explicitly specified. While the default implementations provided by the `Object` class are often not sufficient for specialized classes, they can be overridden to provide more meaningful behavior.

Developers commonly override the `toString()`, `equals()`, and `hashCode()` methods to provide appropriate implementations for their classes.

**Q) What is singleton class in Java and how can we make a class singleton?**

A singleton class in Java is a class that is designed to have only one instance throughout the lifetime of an application. The primary purpose of a singleton is to ensure that a class has a single point of access to its instance, and that instance can be easily shared across multiple parts of the program.

To make a class singleton, you generally follow a specific pattern that restricts the instantiation of the class to a single instance. Here's one common way to create a singleton class:

```java

public class Singleton {

// Private static instance variable

private static Singleton instance;

// Private constructor to prevent direct instantiation

private Singleton() {

// Initialization code, if needed

}

// Public static method to provide access to the instance

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

// Other methods and fields

}

```

A screenshot of a computer program

Description automatically generated

Explanation of the key components:

- The class has a `private static` instance variable that holds the single instance of the class.

- The constructor is `private`, preventing external classes from directly creating new instances.

- A `public static` method named `getInstance()` is provided to access the single instance. If the instance doesn't exist, it's created using the private constructor.

This approach ensures that the class has only one instance, lazily creating it when needed. However, it's worth noting that the above implementation is not thread-safe. In a multithreaded environment, multiple threads might attempt to create separate instances concurrently.

To make the singleton thread-safe, you can use techniques like synchronized methods or the double-checked locking pattern. Alternatively, in Java 5 and later, you can use the "Bill Pugh Singleton" or the "Initialization-on-demand holder idiom" which leverages the Java class-loading mechanism to ensure lazy instantiation and thread safety without explicit synchronization.

Here's an example of the "Bill Pugh Singleton":

```java

public class Singleton {

private Singleton() {

// Initialization code

}

private static class SingletonHolder {

private static final Singleton INSTANCE = new Singleton();

}

public static Singleton getInstance() {

return SingletonHolder.INSTANCE;

}

}

```

A screen shot of a computer program

Description automatically generated

This approach guarantees thread safety during instance creation and maintains a simple and clean code structure.

**Java collections framework interview questions**

**Q. Explain Collection Hierarchy ?**

In Java, the Collection Hierarchy refers to the hierarchy of classes and interfaces provided by the Java Collections Framework for working with collections of objects. The Collections Framework is part of the Java Standard Library and provides a standardized way to store, manipulate, and process groups of objects.

The Collection Hierarchy is organized around two main interfaces: `Collection` and `Map`.

1. \*\*Collection Interface:\*\*

- The `Collection` interface is the root interface for all collection classes. It represents a group of objects (elements) and provides basic operations for manipulating collections.

- Subinterfaces of `Collection` include:

- `List`: An ordered collection that allows duplicate elements. Examples: `ArrayList`, `LinkedList`.

- `Set`: A collection that does not allow duplicate elements. Examples: `HashSet`, `TreeSet`.

- `Queue`: A collection designed for holding elements before processing. Examples: `LinkedList`, `PriorityQueue`.

2. \*\*Map Interface:\*\*

- The `Map` interface represents a collection of key-value pairs, where each key is associated with a value. It doesn't extend the `Collection` interface, but it's an important part of the Collections Framework.

- Subinterfaces of `Map` include:

- `HashMap`: A hash-based implementation of the `Map` interface.

- `TreeMap`: A sorted map implementation that uses a Red-Black Tree.

- `LinkedHashMap`: A map that maintains insertion order.

- `Hashtable`: A synchronized map similar to `HashMap`, but less efficient.

3. \*\*General Collections Framework Structure:\*\*

- `Collection` (Interface)

- `List` (Interface)

- `ArrayList` (Class)

- `LinkedList` (Class)

- `Set` (Interface)

- `HashSet` (Class)

- `TreeSet` (Class)

- `Queue` (Interface)

- `LinkedList` (Class)

- `PriorityQueue` (Class)

- `Map` (Interface)

- `HashMap` (Class)

- `TreeMap` (Class)

- `LinkedHashMap` (Class)

- `Hashtable` (Class)

4. \*\*Important Concepts:\*\*

- Iterators: Used to traverse through elements in collections.

- Generics: Type-safe way to work with collections.

- Comparable and Comparator: Interfaces for sorting elements in collections.

- Collections Utility Class: Provides various utility methods for manipulating collections.

The Java Collections Framework provides a wide range of collection classes and interfaces, each designed for specific use cases. Understanding this hierarchy and the capabilities of each class/interface helps in choosing the appropriate data structure for your application's needs.

Q. Why Map doesn’t extend the Collection Interface ?  
The decision to not have the `Map` interface extend the `Collection` interface in Java's Collections Framework is based on the fundamental differences in their behavior and purpose. The `Collection` interface represents a group of individual elements, while the `Map` interface represents a collection of key-value pairs. Here are some reasons why they are separate:

1. \*\*Key-Value Pair Semantics:\*\*

- A `Map` represents a mapping between keys and values, where each key is associated with a value. This key-value pair semantics is fundamentally different from the single-element semantics of a `Collection`.

2. \*\*Duplication Handling:\*\*

- Collections like `List` and `Set` deal with individual elements and handle issues like duplicates differently.

- In contrast, `Map` collections need to manage uniqueness based on keys, not on the values. Keys in a map must be unique.

3. \*\*Methods and Contracts:\*\*

- The methods and contracts for collections and maps are significantly different due to their distinct purposes.

- Collections provide methods like `add()`, `remove()`, `contains()`, which do not make sense in the context of a `Map`.

- Maps provide methods for key-based operations such as `put()`, `get()`, `remove()`.

4. \*\*Different Usage Patterns:\*\*

- Collections are typically used for managing and iterating through elements, whereas maps are used for key-based lookups and associations.

5. \*\*Complexity and Specialization:\*\*

- Combining `Map` and `Collection` would create an overly complex interface that might be confusing to implement and use. Keeping them separate allows for clearer design and specialization.

While the `Collection` and `Map` interfaces have different purposes, they do share commonality in that they are part of the broader Collections Framework and follow similar design patterns. For example, they both support generics, iterators, and common utility methods for handling collections. This separation of concerns makes the framework more versatile and easier to understand, as it follows the principle of designing interfaces for specific use cases and behaviors.

Q. Difference between fail-fast and fail-safe Iterators

Fail-Fast and Fail-Safe are two different strategies for handling concurrent modifications in collections while using iterators. These strategies determine how iterators react when the underlying collection is modified during iteration.

\*\*Fail-Fast Iterators:\*\*

- A fail-fast iterator detects if a collection has been modified by other threads while it's being iterated and throws a `ConcurrentModificationException` to signal that the collection's state has changed unexpectedly.

- Fail-fast iterators are typically associated with collections that provide their own built-in synchronization mechanisms (e.g., `ArrayList`, `HashSet`).

- They offer fast detection of potential issues but might terminate the iteration abruptly.

\*\*Fail-Safe Iterators:\*\*

- A fail-safe iterator operates on a snapshot of the collection that was taken when the iterator was created. It doesn't throw exceptions if the collection is modified during iteration, as it doesn't operate on the live collection.

- Fail-safe iterators are typically implemented by creating a copy of the collection's elements before starting iteration or using mechanisms like copy-on-write collections (e.g., `CopyOnWriteArrayList`, `ConcurrentHashMap`).

- They allow safe iteration without the risk of encountering exceptions, but they might iterate over outdated or inconsistent data if the collection is frequently modified.

Here's a simple comparison between fail-fast and fail-safe iterators:

A screenshot of a computer

Description automatically generated

In summary, the choice between fail-fast and fail-safe iterators depends on the specific use case, thread safety requirements, and the potential impact of concurrent modifications on the iteration process.

Q. what is BlockingQueue in java ?

`BlockingQueue` is an interface in Java's Concurrency Utilities that represents a thread-safe queue with blocking operations. It's used for communication and synchronization between producer and consumer threads. A `BlockingQueue` supports operations that allow elements to be added or removed from the queue, and it provides blocking behavior when certain conditions are met.

The main feature of a `BlockingQueue` is its ability to block the calling thread under specific circumstances, which is useful in scenarios where producers and consumers need to coordinate their actions. There are several implementations of the `BlockingQueue` interface that provide different blocking behaviors:

1. \*\*`ArrayBlockingQueue`:\*\* A bounded blocking queue backed by an array. It has a fixed capacity and blocks when the queue is full or empty.

2. \*\*`LinkedBlockingQueue`:\*\* A bounded or unbounded blocking queue backed by a linked list. It can be configured to block when the queue is full or empty.

3. \*\*`PriorityBlockingQueue`:\*\* An unbounded blocking queue that orders elements based on their natural order or according to a specified comparator.

4. \*\*`DelayQueue`:\*\* A blocking queue of elements with an associated delay time. Elements are ordered based on their delay.

5. \*\*`SynchronousQueue`:\*\* A blocking queue with a capacity of 0. Each insert operation blocks until a corresponding remove operation is executed.

6. \*\*`LinkedTransferQueue`:\*\* An unbounded queue that provides additional transfer methods for enhanced consumer-producer interaction.

`BlockingQueue` provides several methods for adding, removing, and inspecting elements, including:

- `put(E e)`: Inserts the specified element into the queue, blocking if the queue is full.

- `take()`: Retrieves and removes an element from the queue, blocking if the queue is empty.

- `offer(E e)`: Inserts the specified element into the queue if possible, returning `false` if the queue is full.

- `poll()`: Retrieves and removes an element from the queue if available, returning `null` if the queue is empty.

The blocking behavior of these methods is the key feature of a `BlockingQueue`, making it an essential tool for managing concurrent scenarios where threads need to wait for certain conditions to be met before proceeding.

Q. Difference between Synchronized Collection and Concurrent Collection with pictorial understanding ?

A diagram of a guitar

Description automatically generated

Synchronized Collections and Concurrent Collections are both approaches to managing thread safety in the context of multi-threaded programming. However, they differ in terms of performance, granularity of synchronization, and use cases. Let's understand the differences with a simple pictorial comparison:

![Synchronized vs Concurrent Collections](https://i.imgur.com/IMYgO9X.png)

\*\*Synchronized Collections:\*\*

- \*\*Granularity:\*\* The synchronization is at the collection level. It locks the entire collection for each operation.

- \*\*Performance:\*\* Because the entire collection is locked, only one thread can access the collection at a time, leading to potential contention and reduced parallelism.

- \*\*Use Case:\*\* Synchronized collections are suitable when you have a low level of concurrent access or when you need to provide synchronized access to non-thread-safe collections.

\*\*Concurrent Collections:\*\*

- \*\*Granularity:\*\* The synchronization is more fine-grained. Different parts of the collection can be accessed concurrently by multiple threads.

- \*\*Performance:\*\* Concurrent collections are optimized for high concurrency. Multiple threads can access different segments of the collection simultaneously, improving parallelism.

- \*\*Use Case:\*\* Concurrent collections are designed for scenarios where high concurrency is required, such as scenarios with a large number of threads accessing the collection simultaneously.

In the pictorial representation:

1. \*\*Synchronized Collections:\*\*

- A single lock (red) is used to control access to the entire collection.

- Threads must wait in line to access the collection, causing potential contention and reduced concurrency.

2. \*\*Concurrent Collections:\*\*

- The collection is divided into segments, each with its own lock (green).

- Threads can access different segments concurrently, leading to improved parallelism and reduced contention.

In general, if you have scenarios where multiple threads need to access different parts of the collection concurrently, concurrent collections are more suitable. On the other hand, if you have scenarios with lower levels of concurrency or the need to provide synchronized access to non-thread-safe collections, synchronized collections might be sufficient.

Q. Internal Working of hash map with pictorial explanation

A screenshot of a computer

Description automatically generated

A `HashMap` is a widely used data structure in Java that stores key-value pairs and provides fast access to values based on their associated keys. It uses a hash table to store and retrieve elements efficiently. Let's break down the internal working of a `HashMap` with a pictorial explanation:

![HashMap Internal Working](https://i.imgur.com/0oPgl99.png)

\*\*1. Hashing:\*\*

- When a key-value pair is added to the `HashMap`, the hash code of the key is computed using the `hashCode()` method.

- The hash code is used to determine the index (bucket) in which the key-value pair will be stored.

\*\*2. Bucket Array:\*\*

- The `HashMap` maintains an array of "buckets" (also known as "slots"), where each bucket can store one or more key-value pairs.

- The array size is determined by the capacity of the `HashMap`.

\*\*3. Index Calculation:\*\*

- The computed hash code is subjected to further calculations (often using bitwise operations) to derive the index of the bucket where the key-value pair will be stored.

\*\*4. Collisions:\*\*

- Since multiple keys can have the same hash code or hash code-derived index, collisions can occur when multiple key-value pairs are mapped to the same bucket.

\*\*5. Handling Collisions:\*\*

- In case of collisions, the `HashMap` uses a linked list (or a tree in Java 8+ for larger buckets) to store multiple key-value pairs within the same bucket.

- The linked list/tree stores key-value pairs with the same hash code or index.

\*\*6. Get Operation:\*\*

- To retrieve a value associated with a key, the `HashMap` calculates the hash code of the key and finds the corresponding bucket.

- If there's only one key-value pair in the bucket, the value is directly returned.

- If there are multiple key-value pairs in the bucket, the actual key comparison is performed to find the correct key-value pair.

\*\*7. Put Operation:\*\*

- When adding a key-value pair, the `HashMap` calculates the hash code and determines the bucket.

- If the bucket is empty, the key-value pair is added directly.

- If the bucket already contains key-value pairs, the `HashMap` checks for collisions and appends the new key-value pair to the linked list or tree within the bucket.

\*\*8. Resize and Rehash:\*\*

- As the number of key-value pairs grows, the `HashMap` periodically checks if the number of elements in the map exceeds a threshold (load factor).

- If the threshold is exceeded, the `HashMap` is resized and all key-value pairs are rehashed (recalculating new indices and redistributing them into the new buckets).

The internal workings of a `HashMap` involve a combination of hashing, indexing, handling collisions, and resizing to ensure efficient storage and retrieval of key-value pairs. Keep in mind that the actual implementation might vary between different Java versions, but the general principles remain consistent.

# Java is pass/call by value or pass/call by reference [MOST IMP. JAVA INTERVIEW QUESTION]|

Q. is java A pass by value or pass by reference?

In Java, method arguments are passed by value. This means that when you pass an argument to a method, you are passing a copy of the value of the argument, not a reference to the original object. However, this distinction can sometimes be a bit confusing due to the way it works with objects.

Let's break it down with a simple explanation:

\*\*Passing by Value:\*\*

When you pass a primitive data type (e.g., `int`, `char`, `double`, etc.) to a method, you are passing the actual value of that primitive.

A screen shot of a computer

Description automatically generated

\*\*Example:\*\*

```java

void modifyValue(int x) {

x = x + 1;

}

public static void main(String[] args) {

int num = 5;

modifyValue(num);

System.out.println(num); // Output: 5 (unchanged)

}

```

\*\*Passing Object References by Value:\*\*

When you pass an object (including arrays) to a method, you are passing a copy of the reference to that object, not the object itself. This means the method can access and modify the object's state.

\*\*Example:\*\*

```java

class Person {

String name;

Person(String name) {

this.name = name;

}

}

void modifyPersonName(Person person) {

person.name = "Alice";

}

public static void main(String[] args) {

Person person = new Person("Bob");

modifyPersonName(person);

System.out.println(person.name); // Output: Alice (modified)

}

```

A computer screen shot of a computer program

Description automatically generated

Even though objects are passed by value, changes made to the object's internal state (like modifying fields) within the method will be reflected outside the method, since both the original reference and the copy of the reference point to the same object in memory.

In summary, Java is strictly pass-by-value, but for objects, it's more accurately "passing the value of the reference." This distinction becomes clearer when considering primitive data types and object references.

# Why comparable and comparator is needed || Need of Comparable and Comparator in Java

Q. why are Comparable and Comparator Interfaces required in Java?

The `Comparable` and `Comparator` interfaces in Java provide mechanisms for comparing objects and sorting collections. They are required because they allow you to define custom comparison logic for objects, which is essential for various sorting and ordering operations. These interfaces provide flexibility in how objects are compared and sorted based on different criteria.

\*\*1. `Comparable` Interface:\*\*

The `Comparable` interface is used to define the natural ordering of objects. When a class implements the `Comparable` interface, it specifies how its instances should be compared to each other. This natural ordering is used by sorting methods like `Arrays.sort()` and `Collections.sort()`.

The main advantages of using `Comparable` are:

- It provides a default comparison method for objects of a specific class.

- It simplifies the sorting process by allowing the use of standard sorting methods.

\*\*2. `Comparator` Interface:\*\*

The `Comparator` interface allows you to define custom comparison logic for objects without modifying their class. It's particularly useful when you want to sort objects based on different criteria, or when you're working with classes that you can't modify (e.g., classes from libraries).

The main advantages of using `Comparator` are:

- It allows you to sort objects based on various criteria, not just their natural ordering.

- It provides flexibility to sort objects in different ways without modifying their original class.

- It can be used to sort objects from different classes using a consistent comparison logic.

\*\*Use Cases:\*\*

- \*\*Comparable:\*\* Use it when you want to define a default ordering for objects within their class. For example, sorting a list of `String` objects alphabetically or sorting a list of `Date` objects chronologically.

- \*\*Comparator:\*\* Use it when you want to sort objects in ways other than their natural ordering, or when you want to sort objects from different classes based on a common criterion. For example, sorting a list of `Person` objects by age, then by name.

In summary, the `Comparable` and `Comparator` interfaces provide ways to compare and sort objects based on different criteria. They offer flexibility and customization in sorting operations, making them essential for working with collections of objects in a wide range of scenarios.

Q. Difference between Comparator and Comparable?

`Comparator` and `Comparable` are both interfaces in Java that are used for comparing objects and sorting collections. However, they serve different purposes and are used in different scenarios. Here are the key differences between `Comparator` and `Comparable`:

1. \*\*Purpose:\*\*

- \*\*`Comparator`:\*\* The `Comparator` interface is used when you want to define custom comparison logic for objects without modifying their class. It provides a way to compare objects based on criteria other than their natural ordering.

- \*\*`Comparable`:\*\* The `Comparable` interface is used to define the natural ordering of objects within their own class. It specifies how objects of a class should be compared to each other by implementing the `compareTo` method.

2. \*\*Implementation:\*\*

- \*\*`Comparator`:\*\* You create a separate class that implements the `Comparator` interface and overrides the `compare` method to define the comparison logic. This allows you to have multiple different comparison strategies for the same class.

- \*\*`Comparable`:\*\* You implement the `Comparable` interface directly in the class whose objects you want to compare. You override the `compareTo` method in that class to define the natural ordering.

3. \*\*Usage:\*\*

- \*\*`Comparator`:\*\* You can use a `Comparator` to sort objects of a class in various ways without modifying the class itself. This is particularly useful when working with classes from libraries or third-party code that you can't modify.

- \*\*`Comparable`:\*\* The natural ordering defined by `Comparable` is used by default sorting methods like `Arrays.sort()` and `Collections.sort()`.

4. \*\*Flexibility:\*\*

- \*\*`Comparator`:\*\* Offers greater flexibility in terms of defining different comparison strategies for the same class. You can have multiple `Comparator` implementations to sort objects in different ways.

- \*\*`Comparable`:\*\* Defines a single natural ordering for objects within the class, so it's less flexible when you need different sorting criteria.

5. \*\*Sorting Objects from Different Classes:\*\*

- \*\*`Comparator`:\*\* Allows you to sort objects from different classes using a common comparison logic.

- \*\*`Comparable`:\*\* Can't be used to sort objects from different classes using a common comparison logic.

6. \*\*Usage Example:\*\*

- \*\*`Comparator`:\*\* Sorting a list of `Person` objects based on age using a custom `AgeComparator` class.

- \*\*`Comparable`:\*\* Sorting a list of `String` objects alphabetically or a list of `Date` objects chronologically.

In summary, the key distinction is that `Comparator` allows you to define custom comparison logic externally, whereas `Comparable` defines the comparison logic within the object's own class. The choice between them depends on whether you need a single natural ordering (use `Comparable`) or multiple comparison strategies (use `Comparator`).

Q. Example of Comparator and Comparable ?

Sure, here are examples of both `Comparator` and `Comparable` implementations to illustrate their usage:

\*\*Example of `Comparator`: Sorting `Person` Objects by Age\*\*

Suppose you have a `Person` class with attributes `name` and `age`, and you want to sort a list of `Person` objects based on their age using a custom `AgeComparator`:

A screen shot of a computer program

Description automatically generated

```java

import java.util.\*;

class Person {

String name;

int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

}

class AgeComparator implements Comparator<Person> {

@Override

public int compare(Person p1, Person p2) {

return Integer.compare(p1.age, p2.age);

}

}

public class ComparatorExample {

public static void main(String[] args) {

List<Person> people = new ArrayList<>();

people.add(new Person("Alice", 25));

people.add(new Person("Bob", 30));

people.add(new Person("Carol", 22));

Collections.sort(people, new AgeComparator());

for (Person person : people) {

System.out.println(person.name + " - " + person.age);

}

}

}

```

\*\*Example of `Comparable`: Sorting `Employee` Objects by ID\*\*

Suppose you have an `Employee` class with attributes `id` and `name`, and you want to sort a list of `Employee` objects based on their ID using the `Comparable` interface:

A computer screen shot of a program

Description automatically generated

```java

import java.util.\*;

class Employee implements Comparable<Employee> {

int id;

String name;

public Employee(int id, String name) {

this.id = id;

this.name = name;

}

@Override

public int compareTo(Employee other) {

return Integer.compare(this.id, other.id);

}

}

public class ComparableExample {

public static void main(String[] args) {

List<Employee> employees = new ArrayList<>();

employees.add(new Employee(101, "Alice"));

employees.add(new Employee(103, "Bob"));

employees.add(new Employee(102, "Carol"));

Collections.sort(employees);

for (Employee employee : employees) {

System.out.println(employee.id + " - " + employee.name);

}

}

}

```

In the first example (`Comparator`), we create a separate class `AgeComparator` that implements the `Comparator` interface to define the sorting logic based on age. In the second example (`Comparable`), we implement the `Comparable` interface directly in the `Employee` class to define the natural ordering based on the employee's ID.

Both examples showcase how you can control the sorting behavior of objects using either a custom `Comparator` or the `Comparable` interface.

Q. Explain getters and setters ?

Getters and setters are methods used to access and modify the private fields (attributes) of a class. They are a fundamental concept in object-oriented programming and are used to encapsulate the internal state of an object while providing controlled access to that state from outside the class.

\*\*Getters:\*\*

A getter method is used to retrieve the value of a private field. It follows the naming convention `get<FieldName>` and returns the value of the field.

A black background with white circles and red dots

Description automatically generated

In this example, the `getName()` method allows external code to retrieve the value of the private `name` field.

\*\*Setters:\*\*

A setter method is used to modify the value of a private field. It follows the naming convention `set<FieldName>` and takes a parameter to set the new value of the field.

A screen shot of a computer code

Description automatically generated

In this example, the `setAge(int age)` method allows external code to update the value of the private `age` field.

\*\*Benefits of Getters and Setters:\*\*

1. \*\*Encapsulation:\*\* Getters and setters provide controlled access to the internal state of an object. This encapsulation helps maintain data integrity and ensures that the object's state can be modified in a controlled manner.

2. \*\*Data Validation:\*\* Setters can include validation logic to ensure that only valid values are set for a field. For example, you can check if an age value is non-negative before setting it.

3. \*\*Flexibility:\*\* By using getters and setters, you can change the implementation of your class's attributes without affecting the external code that uses the class. This provides flexibility to refactor and improve your code over time.

4. \*\*Readability:\*\* Getters and setters can have meaningful names that make the code more readable and self-explanatory.

5. \*\*Access Control:\*\* Getters and setters allow you to control the visibility of your fields. You can keep fields private and only provide access through these methods, maintaining control over who can modify or read the field.

In modern Java development, many IDEs offer automated generation of getters and setters, making it easy to create and manage these methods without writing boilerplate code.

Q. Explain toString() method ?

The `toString()` method is a method defined in the `Object` class in Java. It's used to provide a textual representation of an object's state, which is useful for debugging, logging, and displaying information to users. By default, all classes in Java inherit this method from the `Object` class, but you can override it in your own classes to provide a more meaningful string representation of the object.

Here's the basic syntax of the `toString()` method:

A black rectangular object with white text

Description automatically generated

When you call `toString()` on an object, it returns a string that typically includes important information about the object's attributes and values. This makes it easier to understand the object's content without having to inspect its attributes individually.

\*\*Example:\*\*

Suppose you have a `Person` class:

A computer screen with text and numbers

Description automatically generated

In this example, the `toString()` method is overridden to provide a custom string representation for the `Person` object. When you call `toString()` on a `Person` object, it will return a string like `"Person{name='Alice', age=25}"`.

\*\*Usage:\*\*

The `toString()` method is automatically called by various parts of Java's standard library, such as when you print an object, concatenate it with a string, or use it in logging or debugging contexts. For example:

A screenshot of a computer

Description automatically generated

By providing a meaningful implementation of `toString()`, you make it easier to understand the content of your objects during development and debugging. It's a good practice to override `toString()` in your classes to provide clear and concise information about the object's state.

**Q. Creating List and adding values in List ?**

In Java, you can create a list using the `List` interface, which is a part of the Java Collections Framework. You can choose from various implementations of the `List` interface, such as `ArrayList`, `LinkedList`, or `Vector`, based on your specific needs. Here's how you can create a list and add values to it using the `ArrayList` implementation as an example:

A screen shot of a computer program

Description automatically generated

In this example, we create an `ArrayList` named `namesList`, add several values to it using the `add()` method, insert a value at a specific position using the `add(index, element)` method, print the list, access elements using the `get()` method, modify elements using the `set()` method, remove an element using the `remove()` method, and finally, iterate through the list using a enhanced for loop.

You can choose a different `List` implementation depending on your requirements. The basic operations for adding, modifying, accessing, and removing elements are generally the same across different `List` implementations in the Java Collections Framework.

**Q. explain Collection.sort method to sort the List ?**

The `Collections.sort()` method in Java is used to sort a list of elements in their natural order or using a custom comparator. It's part of the `java.util.Collections` class and is commonly used to sort lists, including lists of objects.

Here's the syntax of the `Collections.sort()` method:

A black rectangular object with text

Description automatically generated

- `list`: The list to be sorted. The list's elements must implement the `Comparable` interface to define their natural ordering.

\*\*Example: Sorting a List of Integers\*\*

A computer screen with white and red text

Description automatically generated

\*\*Example: Sorting a List of Custom Objects\*\*

Suppose you have a `Person` class and you want to sort a list of `Person` objects by their names:

A screen shot of a computer program

Description automatically generated

In this example, the `compareTo()` method is overridden in the `Person` class to define the comparison logic based on the names of the people. The `Collections.sort()` method uses this comparison logic to sort the list of `Person` objects.

In summary, the `Collections.sort()` method is a convenient way to sort lists of elements, whether they are primitive types or objects, using either their natural ordering (for elements implementing `Comparable`) or a custom comparator.

Q. When to use Comparable and when to use Comparator ?

The decision to use `Comparable` or `Comparator` depends on the control and flexibility you need over sorting behavior and whether you can modify the class whose objects you want to sort. Here's when to use each approach:

\*\*Use `Comparable` when:\*\*

1. \*\*Natural Ordering:\*\* You want to define a default natural ordering for objects within their own class. For example, sorting numbers in ascending order or strings alphabetically.

2. \*\*Single Sorting Criteria:\*\* You have a single, well-defined way of comparing objects that makes sense for all scenarios. The natural ordering is inherent to the class.

3. \*\*Inherent Comparison:\*\* The comparison logic is tightly related to the nature of the objects. It's how you would naturally expect the objects to be ordered.

4. \*\*Modification of Class:\*\* You have control over the class's implementation and can modify it to implement the `Comparable` interface. You want to provide a consistent ordering for all users of the class.

\*\*Use `Comparator` when:\*\*

1. \*\*Custom Sorting Criteria:\*\* You need to sort objects based on multiple criteria or in different ways depending on the context. You want to provide flexible and customizable sorting.

2. \*\*External Comparison:\*\* The comparison logic is external to the class, or you don't have control over the class's implementation. This is common when working with third-party or library classes.

3. \*\*Changing Sorting Criteria:\*\* You want to sort the same objects using different criteria at different times without modifying the class itself.

4. \*\*Decoupling Sorting Logic:\*\* You want to keep the comparison logic separate from the class itself, making your code more modular and easier to maintain.

5. \*\*Multi-Criteria Sorting:\*\* You need to sort objects using multiple criteria simultaneously, such as sorting employees by department and then by age.

\*\*Guidelines:\*\*

- If you're working with your own classes and you have a natural ordering for the objects that won't change often, implement `Comparable` to provide a default sorting behavior.

- Use `Comparator` when you need to sort objects based on multiple criteria, need to sort objects from different classes using a common logic, or want to keep the sorting logic separate from the class.

- In scenarios where you have both `Comparable` and `Comparator` implementations available, the `Comparator` takes precedence over the natural ordering defined by `Comparable`.

In summary, use `Comparable` for default and inherent sorting behavior within a class, and use `Comparator` for custom and flexible sorting behavior that can change or when working with external classes.

**Q. What is the `compareTo()` method?**

The `compareTo()` method is a method defined in the `Comparable` interface in Java. It's used to establish the natural ordering of objects and is implemented by classes to provide a way for objects to be compared to each other. The `compareTo()` method returns a negative integer, zero, or a positive integer based on whether the object is less than, equal to, or greater than the specified object being compared.

Here's the basic syntax of the `compareTo()` method:

A black and white rectangular object with red text

Description automatically generated

- `other`: The object to be compared with.

\*\*Example: `compareTo()` in `String`\*\*

A computer screen shot of a program code

Description automatically generated

In this example, `str1.compareTo(str2)` returns a negative value, indicating that "apple" comes before "banana" in lexicographical order.

**Q. How to use `Comparable`?**

To use `Comparable`, you need to follow these steps:

1. Implement the `Comparable` interface in your class.

2. Override the `compareTo()` method to provide the comparison logic based on the natural ordering of your class.

\*\*Example: Using `Comparable` to Sort `Person` Objects\*\*

A screenshot of a computer program

Description automatically generated

\*\*Q. How to use `Comparator`?\*\*

Using `Comparator` involves these steps:

1. Create a separate class that implements the `Comparator` interface.

2. Implement the `compare()` method to define the custom comparison logic.

3. Use the custom comparator in sorting methods like `Collections.sort()`.

\*\*Example: Using `Comparator` to Sort `Person` Objects by Name\*\*

A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

In this example, we create a `NameComparator` class that implements the `Comparator` interface and provides the comparison logic based on the names of the `Person` objects. We then use this custom comparator to sort the list of `Person` objects.

To summarize:

- Use `Comparable` when you want to define the natural ordering of objects within their own class.

- Use `Comparator` when you need custom sorting criteria or when working with external classes.

- Implement the appropriate interface (`Comparable` or `Comparator`) and override the necessary methods to provide the comparison logic.

# Equals and Hashcode Contract in Java [Important Java Interview Question]

Q. what is Equals and Hashcode Contract in Java ?

The "equals and hashCode contract" refers to the relationship between the `equals()` and `hashCode()` methods in Java. These methods are used to compare objects for equality and generate hash codes for objects, respectively. There are certain expectations and rules that must be followed to maintain consistency when dealing with these methods. The contract ensures that if two objects are considered equal based on the `equals()` method, their hash codes must be the same according to the `hashCode()` method.

Here are the key points of the contract:

1. \*\*Equal Objects Must Have Equal Hash Codes:\*\*

If `obj1.equals(obj2)` returns `true` (i.e., two objects are considered equal), then `obj1.hashCode()` must be equal to `obj2.hashCode()`.

2. \*\*Unequal Objects Might Have Equal Hash Codes:\*\*

It's possible for two different objects to have the same hash code, but this should be relatively rare to maintain efficient hash-based data structures.

3. \*\*Consistency:\*\*

If the object's state changes in a way that affects its equality comparison, the `hashCode()` method must be updated to reflect the new state.

4. \*\*hashCode() Should Use the Same Fields as equals():\*\*

The fields used for equality comparison in the `equals()` method should also be used to calculate the hash code in the `hashCode()` method. This ensures that two equal objects have the same hash code.

5. \*\*hashCode() Should Distribute Hash Codes Evenly:\*\*

The `hashCode()` method should strive to generate hash codes that distribute objects across different buckets of hash-based data structures to minimize collisions.

\*\*Example:\*\*

Consider a `Person` class with two attributes: `name` and `age`. Here's an example of how to implement the `equals()` and `hashCode()` methods following the contract:

A screenshot of a computer program

Description automatically generated

In this example, the `equals()` method compares both `name` and `age` fields for equality, and the `hashCode()` method generates a hash code based on these fields.

By adhering to the equals and hashCode contract, you ensure that objects behave consistently when used in hash-based data structures like `HashMap` or `HashSet`, maintaining the integrity and reliability of these data structures.

Q. How equals and hashcode contract works in java ?

The `equals` and `hashCode` contract in Java ensures that the behavior of these two methods is consistent and follows certain rules, especially when objects are used in hash-based data structures like `HashMap`, `HashSet`, etc. This contract helps maintain the integrity of these data structures and ensures that equal objects have equal hash codes. Here's how the contract works:

\*\*1. Equal Objects Must Have Equal Hash Codes:\*\*

If two objects are considered equal according to the `equals` method, their `hashCode` values must also be equal. This means that if you override the `equals` method in your class to compare objects based on certain attributes, you should also ensure that the `hashCode` method uses those same attributes to calculate the hash code.

\*\*2. Consistency:\*\*

The `hashCode` value of an object should remain consistent as long as its attributes relevant to equality are not modified. If an object's attributes change in a way that affects its equality comparison, you should also update the `hashCode` calculation to reflect the changes.

\*\*3. Distributing Hash Codes:\*\*

While it's not required that each unique object has a unique hash code, it's important that the `hashCode` method attempts to distribute hash codes evenly across different buckets in hash-based data structures. This helps minimize collisions and ensures efficient retrieval of objects from these data structures.

\*\*Example Scenario:\*\*

Consider a `Person` class with `name` and `age` attributes. Let's assume we want to compare two `Person` objects based on their names and ages. If two `Person` objects have the same name and age, they are considered equal.

A screenshot of a computer program

Description automatically generated

In this example, the `equals` method compares `name` and `age`, and the `hashCode` method calculates the hash code based on the same attributes. By following this contract, two `Person` objects with the same name and age will have equal hash codes, and they will be considered equal in hash-based data structures.

Adhering to the `equals` and `hashCode` contract ensures that objects behave predictably and consistently when used in hash-based collections, maintaining the correctness and efficiency of these collections.