**Collections in Java**

**What Are Collections in Java?**

**Ans:** In Java,

* Collections refer to data structures and classes provided by the Java Collections Framework (JCF) that allow you to store, manipulate, and manage groups of objects.
* Collections provide a wide range of interfaces and classes for working with data in various formats such as lists, sets, maps, queues, and more.
* The Java Collections Framework is part of the java.util package and provides a unified architecture for handling collections of objects.
* Collections in Java offer functionalities such as adding, removing, updating, and accessing elements in the collection.
* They support operations like sorting, searching, filtering, iterating, and converting collections to arrays or streams.

Here are some key points about collections in Java:

**Interfaces:**

The Java Collections Framework includes several core interfaces such as Collection, List, Set, Map, Queue, and Deque.

**Classes:**

The framework provides concrete implementations of collection interfaces, including classes like ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap, PriorityQueue, and ArrayDeque, among others.

**Que: What are the advantages of Collection framework in Java?**

**Ans:** The advantages of the Java Collections Framework (JCF) can be summarized as follows:

* **Standardization:** Provides a consistent and unified way to work with collections.
* **Reusable Data Structures:** Offers ready-to-use data structures and algorithms.
* **Type Safety:** Ensures type safety and catches errors at compile time with generics.
* **Performance Optimization:** Provides optimized collection implementations for different use cases.
* **Concurrency Support**: Includes synchronized and concurrent collections for thread-safe operations.
* **Flexibility:** Allows easy extension and customization of collections.
* **Enhanced Functionality**: Offers rich APIs for complex operations like sorting, filtering, and mapping.
* **Interoperability:** Integrates well with other Java features and APIs.

**Que: What is the need of Collections in Java?**

**Ans**: Prior to the introduction of Collection Framework (or JDK 1.2), the standard techniques for aggregating Java objects (or collections) were Arrays, Vectors, or Hash Tables. There was no common interface for all of these collections. As a result, while the basic goal of all the collections is the same, their implementation was specified independently and there was no correlation between them. Furthermore, users find it challenging to remember all of the various methods, syntax, and constructors included in each collection class.

Collections are needed in programming for the following reasons:

1. Grouping related data together.

2. Allowing dynamic sizing (grow or shrink as needed).

3. Providing a variety of data structures for different purposes (lists, sets, maps, etc.).

4. Offering built-in methods for easy data manipulation (addition, removal, sorting, searching).

5. Efficient storage and retrieval of elements.

6. Basis for implementing algorithms and data processing techniques.

7. Support for concurrency in multi-threaded environments.

8. Enhancing functionality with serialization, iteration, conversion, and integration with other Java APIs.

9. Promoting code reusability and modularity.

**Que: What is the difference between Array and Collection in java?**

**Ans:** Array and Collection are equivalent in terms of storing object references and manipulating data, but they differ in a number of ways.

The main differences between arrays and collections in Java are related to their data structures, flexibility, and functionality. Here's a breakdown of the key differences:

**Data Structure:**

**Array**: An array in Java is a fixed-size data structure that can hold elements of the same type. Once created, the size of an array cannot be changed.

**Collection:** Collections in Java are dynamic data structures that can grow or shrink in size as needed. They can hold elements of different types and provide various data structures like lists, sets, maps, queues, etc.

**Flexibility:**

**Array:** Arrays have fixed sizes and cannot be resized dynamically. Adding or removing elements from an array requires creating a new array with the desired size and copying elements.

**Collection:** Collections are flexible and can grow or shrink dynamically. Elements can be added, removed, or modified easily using built-in methods without worrying about resizing.

**Type of Elements:**

**Array:** Arrays can only hold elements of the same type. For example, an array of integers (int[]) can only store integers.

**Collection:** Collections can hold elements of different types. For instance, a List collection can store a mix of integers, strings, objects, etc.

**Methods and Functionality:**

**Array:** Arrays provide basic operations like accessing elements by index, iterating over elements using loops, and sorting elements using array-specific methods like Arrays.sort().

**Collection:** Collections offer a wide range of methods and functionalities for adding, removing, searching, sorting, iterating, and processing elements. They also support enhanced for-loops (for-each loops) and provide additional features like sorting, filtering, mapping, and streaming operations.

**Usage:**

**Array**: Arrays are suitable for situations where the size is known and fixed, such as storing a fixed number of elements or working with low-level data structures.

**Collection:** Collections are preferred when dealing with dynamic data, handling collections of different sizes, or utilizing advanced data manipulation features provided by the Java Collections Framework.

**Que: Differentiate between Collection and collections in the context of Java?**

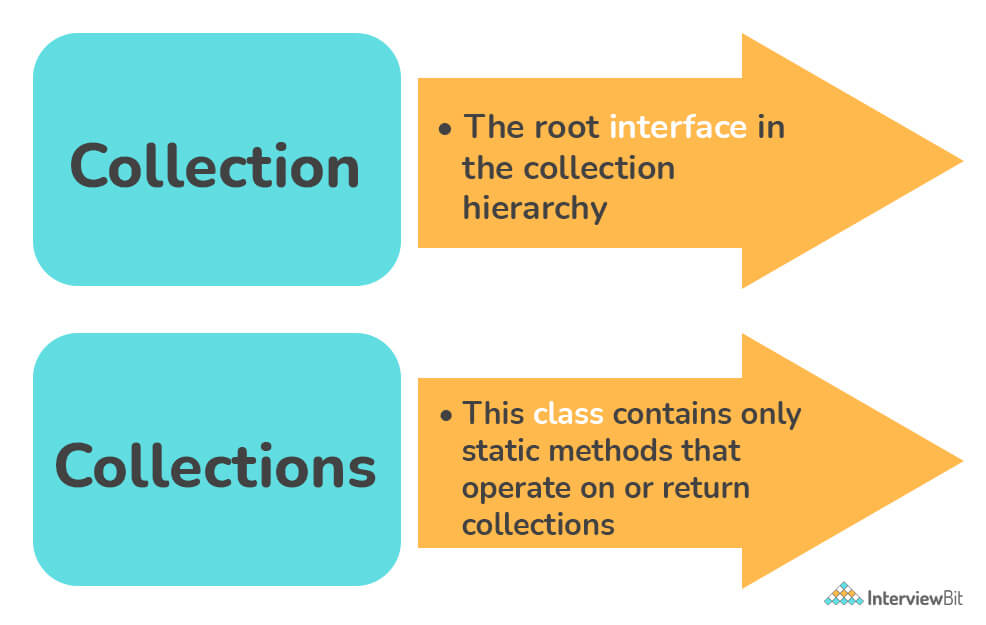
**Ans:**

**Collection (singular):**

* Collection is an interface in the Java Collections Framework (java.util.Collection).
* It represents a group of objects known as elements and defines basic operations for working with collections, such as adding, removing, iterating, and checking for the presence of elements.
* Collection is the root interface of various collection types like lists, sets, queues, and deques.

**Collections (plural):**

* Collections is a utility class in Java (java.util.Collections) that provides static methods for working with collections.
* It contains various methods for performing common operations on collections, such as sorting, searching, synchronizing, reversing, copying, and creating immutable collections.
* The Collections class is used for manipulating and managing collections more efficiently by providing ready-to-use methods for common tasks.



**Que: Explain the hierarchy of the Collection framework in Java?**

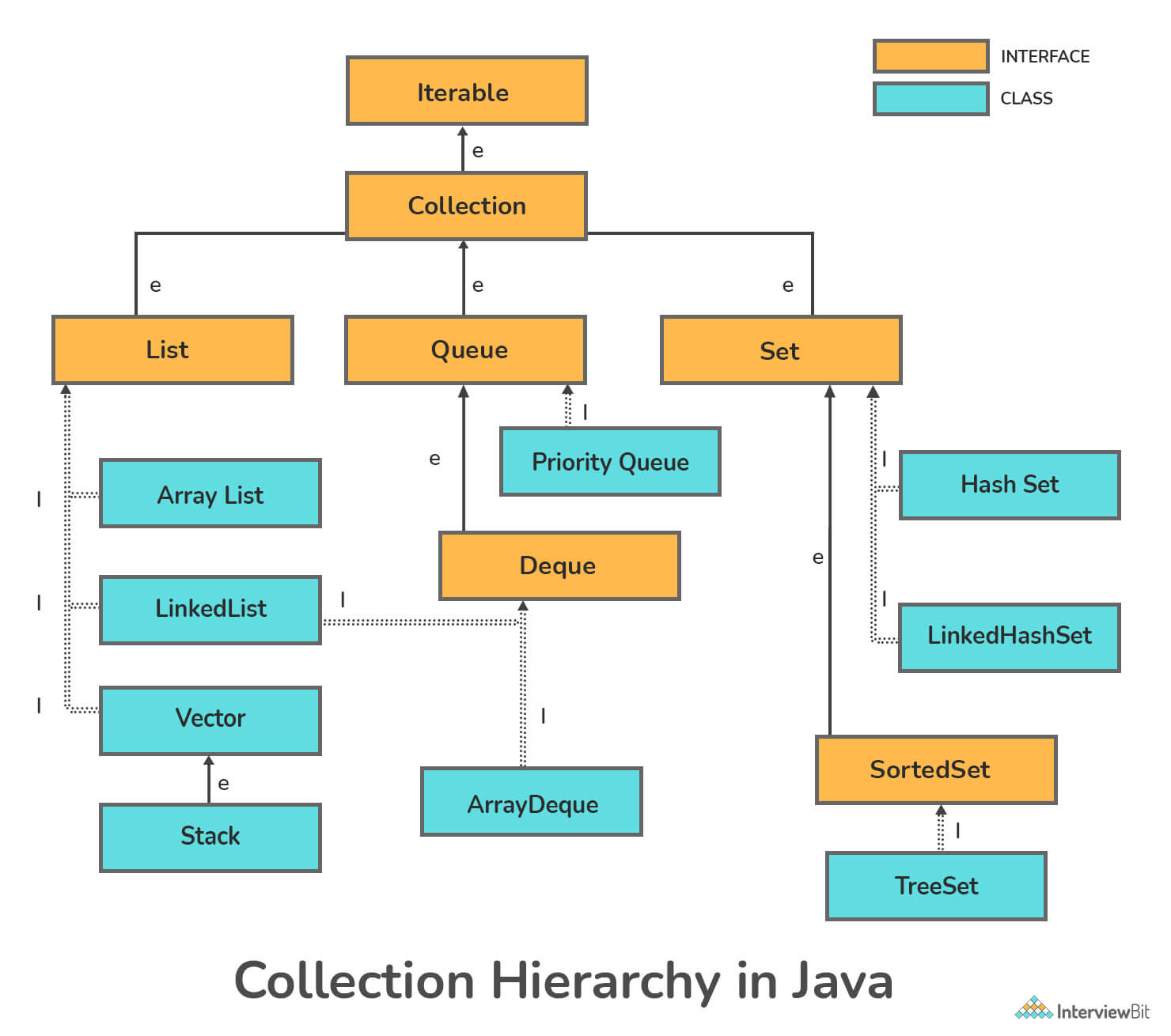
**Ans:** The entire collection framework hierarchy is made up of four fundamental interfaces: Collection, List, Set, Map, and two specific interfaces for sorting called SortedSet and SortedMap. The java.util package contains all of the collection framework's interfaces and classes. The following diagram depicts the Java collection structure.

Here, e denotes extends, i denotes implements.

* **Extends**: The keyword extends is used to create inheritance between two classes and two interfaces.
* **Implements**: The keyword implements are used to create inheritance across classes and interfaces.
* A class can implement an interface--> @override.
* A interface can implement an interface.
* A interface can extend a interface.
* But a class cannot extend an interface.

**Que: Explain the various interfaces used in the Collection framework.**

**Ans:** The Java Collections Framework (JCF) provides a set of interfaces that define different types of collections and their behavior. These interfaces serve as blueprints for various collection classes and allow developers to work with collections in a uniform and consistent manner. Here are the key interfaces used in the Collection framework:



**Iterable Interface:**

* The Iterable interface is located in the java.lang package.
* This is the collection framework's primary interface.
* The iterable interface is extended by the collection interface.
* As a result, all interfaces and classes implement this interface by default.
* This interface's main purpose is to provide an iterator for the collections.
* As a result, this interface only has one abstract method, the iterator.
* It defines a single method iterator() that returns an iterator over a sequence of elements.
* Iterable serves as the basis for implementing classes that support iteration, such as collections and custom data structures.

**Collection Interface:**

* java.util.Collection
* Represents a group of objects known as elements.
* Defines basic operations such as adding, removing, iterating, and checking for the presence of elements.
* Sub interfaces include List, Set, and Queue.

**List Interface:**

* java.util.List
* Extends Collection.
* Represents an ordered collection of elements where duplicate elements are allowed.
* Supports indexed access and allows elements to be added, removed, or replaced at specific positions.
* Implementations include ArrayList, LinkedList, and Vector.

**Set Interface:**

* java.util.Set
* Extends Collection.
* Represents a collection of unique elements where duplicate elements are not allowed.
* Does not guarantee the order of elements and typically uses hashing or sorting for efficient element retrieval and storage.
* Implementations include HashSet, TreeSet, and LinkedHashSet.

***Start Remarks: Set in mathematics means a collection of distinct, well-defined objects forming a group.***

***List Allows duplicacy.***

**Queue Interface:**

* java.util.Queue
* Extends Collection.
* Represents a collection designed for holding elements prior to processing.
* Follows the FIFO (First-In-First-Out) order for element processing.
* Provides additional methods for queue-specific operations such as enqueue, dequeue, peek, and poll.
* Implementations include PriorityQueue and LinkedList.

**Deque Interface:**

* java.util.Deque
* Extends Queue.
* Represents a double-ended queue that supports operations at both ends.
* Can be used as a queue (FIFO) or as a stack (LIFO - Last-In-First-Out) depending on the operations performed.
* Implementations include ArrayDeque and LinkedList.

**Map Interface:**

* java.util.Map
* Represents a collection of key-value pairs where each key is unique.
* Allows you to associate keys with values and perform operations such as adding, removing, and retrieving elements based on keys.
* Implementations include HashMap, TreeMap, LinkedHashMap, and Hashtable.

**SortedSet and SortedMap Interfaces:**

* java.util.SortedSet and java.util.SortedMap
* Extend Set and Map interfaces, respectively.
* Represent sets and maps that maintain elements in a sorted order defined by a comparator or natural ordering of elements.

**Que: What are List? What are the classes present in the list Interface?**

Ans: In Java, a list is an ordered collection of elements that allows duplicate elements and maintains the insertion order of elements.

Lists are typically used when you need to store a sequence of elements where the position of each element matters.

The Java Collections Framework provides the List interface to represent lists, and several classes implement this interface to provide different implementations of lists. Some of the commonly used classes that implement the List interface are:

**ArrayList (java.util.ArrayList):**

* Implements List using a resizable array.
* Provides fast random access and efficient element manipulation.
* Suitable for scenarios where elements are frequently accessed by index and the list size may vary dynamically.

**LinkedList (java.util.LinkedList):**

* Implements List and Deque using a doubly linked list.
* Provides efficient insertion and deletion operations, especially for operations near the beginning or end of the list.
* Suitable for scenarios with frequent modifications or when traversal from both ends is required.

**Vector (java.util.Vector):**

* A legacy class that implements List.
* Similar to ArrayList but is synchronized, making it thread-safe for concurrent access.
* Less commonly used compared to ArrayList due to synchronization overhead unless thread safety is explicitly required.

**CopyOnWriteArrayList (java.util.concurrent.CopyOnWriteArrayList):**

* A thread-safe variant of ArrayList that implements List.
* Provides thread-safe iteration without the need for external synchronization.
* Suitable for scenarios where the list is frequently read but rarely modified.

**Que: What are Arraylist?**

**Ans:** ArrayList is a class in Java that implements the List interface from the Java Collections Framework. It provides a resizable array implementation of the List interface, which means that the size of an ArrayList can dynamically grow or shrink based on the number of elements added or removed. Here are the key features and characteristics of ArrayList:

**Resizable Array:**

* Internally, ArrayList uses an array to store elements.
* The array's size grows dynamically as elements are added to the list.
* When the capacity of the internal array is exceeded, ArrayList automatically reallocates and copies elements to a new larger array.

**Ordered Collection:**

* ArrayList maintains the insertion order of elements, meaning the order in which elements are added to the list is preserved.
* Elements in an ArrayList can be accessed and retrieved based on their index in the list.

**Duplicates Allowed:**

* ArrayList allows duplicate elements, meaning the same element can appear multiple times in the list.

**Random Access:**

* ArrayList supports fast random access to elements using index-based retrieval (get(index)).
* Accessing elements by index in an ArrayList has constant-time complexity (O(1)).

**Dynamic Size:**

* The size of an ArrayList can change dynamically as elements are added or removed.
* The size() method returns the current number of elements in the list.

**Array Manipulation Methods:**

* add(E element): Adds the specified element to the end of the list.
* add(int index, E element): Inserts the specified element at the specified position in the list.
* remove(Object o): Removes the first occurrence of the specified element from the list.
* remove(int index): Removes the element at the specified position in the list.
* set(int index, E element): Replaces the element at the specified position in the list with the specified element.

**Performance:**

* ArrayList provides efficient performance for element retrieval by index (constant-time complexity).
* Adding or removing elements from the end of the list is also efficient (amortized constant-time complexity).
* Insertions or removals at arbitrary positions may require shifting elements and have linear-time complexity (O(n)).

**Que: Difference between Array and ArrayList?**

**Ans:** Here are the key differences between arrays and ArrayLists in Java:

**Data Structure:**

* **Array**: Arrays are fixed-size data structures that can hold elements of the same type. Once created, the size of an array cannot be changed.
* **ArrayList**: ArrayLists are dynamic data structures that can grow or shrink in size as needed. They can hold elements of different types and provide resizable capacities.

**Size Flexibility:**

* **Array**: The size of an array is fixed and determined at the time of creation. Adding or removing elements requires creating a new array with the desired size and copying elements.
* **ArrayList**: ArrayLists can dynamically resize themselves based on the number of elements added or removed. They handle resizing automatically, making it easier to manage collections with varying sizes.

**Type of Elements:**

* **Array**: Arrays can hold elements of the same type. For example, an array of integers (int[]) can only store integers.
* **ArrayList**: ArrayLists can hold elements of different types, allowing for more flexibility in storing heterogeneous data.

**Performance:**

* **Array**: Arrays offer better performance for random access to elements using index-based retrieval (arr[index]), as it has constant-time complexity (O(1)).
* **ArrayList**: ArrayLists also provide efficient random access, but they may incur overhead for resizing and copying elements when the internal array capacity changes.

**Resizable Capacity:**

* **Array**: Arrays have a fixed capacity that cannot be changed after creation.
* **ArrayList**: ArrayLists can dynamically resize their capacity based on the number of elements added or removed. They handle resizing automatically, avoiding the need for manual management.

**Memory Management:**

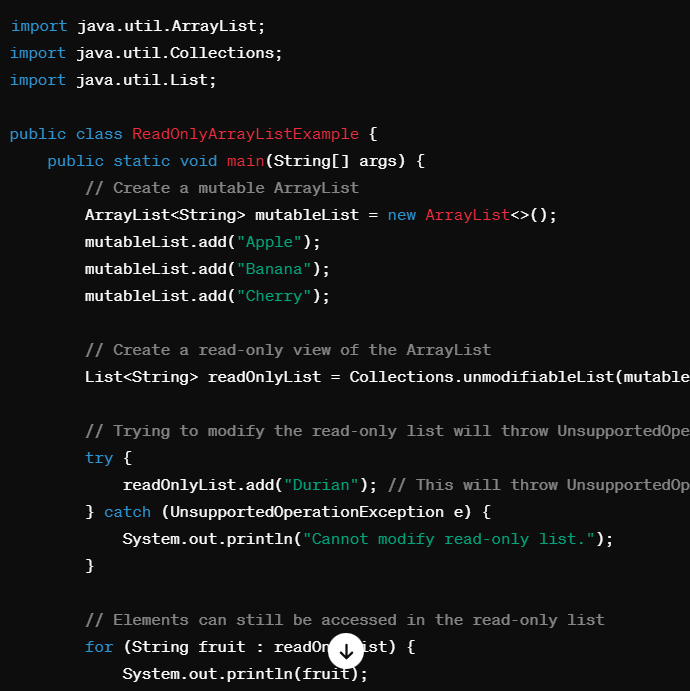
* **Array**: Arrays use contiguous memory allocation, making them efficient in terms of memory usage.
* **ArrayList**: ArrayLists may have extra memory overhead due to their dynamic resizing capabilities and internal array management.

**Usage**:

* **Array**: Arrays are suitable for scenarios where the size is known and fixed, such as storing a fixed number of elements or working with low-level data structures.
* **ArrayList**: ArrayLists are preferred when dealing with dynamic data, handling collections of different sizes, or needing resizable capacities without manual resizing.

**Que: How can you make an ArrayList read-only in Java?**

**Ans:** With the help of Collections.unmodifiableList() method, we can easily make an ArrayList read-only. This function takes a changeable ArrayList as an input and returns the ArrayList's read-only, unmodified view.



**Que: What are LinkedList?**

**Ans:** In Java, LinkedList is a class that implements the List interface from the Java Collections Framework. It provides a doubly linked list implementation of the List interface, offering flexibility in insertion, deletion, and traversal operations.

**What is a doubly linked list?**

A doubly linked list is a type of data structure that allows for the storage of data in a linear fashion, much like a singly linked list. However, unlike a singly linked list, a doubly linked list allows for both forward and backward traversal of the data stored within it.



Here are the key features and characteristics of LinkedList:

**Doubly Linked List:**

* Internally, LinkedList uses a doubly linked list data structure to store elements.
* Each element in the list is represented by a node that contains references to the previous and next nodes in the list.

**Ordered Collection:**

* LinkedList maintains the insertion order of elements, meaning the order in which elements are added to the list is preserved.
* Elements in a LinkedList can be accessed and retrieved based on their index in the list.

**Duplicates Allowed:**

* LinkedList allows duplicate elements, meaning the same element can appear multiple times in the list.
* Efficient Insertion and Deletion.
* LinkedList provides efficient insertion and deletion operations, especially for operations near the beginning or end of the list.
* Adding or removing elements at the beginning or end of a LinkedList has constant-time complexity (O(1)).

**Traversal:**

* LinkedList supports efficient traversal of elements in both forward and backward directions due to its doubly linked structure.
* Traversal from the middle of the list may involve traversing a smaller portion of the list compared to arrays or ArrayList.

**Dynamic Size:**

* The size of a LinkedList can change dynamically as elements are added or removed.
* The size() method returns the current number of elements in the list.

**Array Manipulation Methods:**

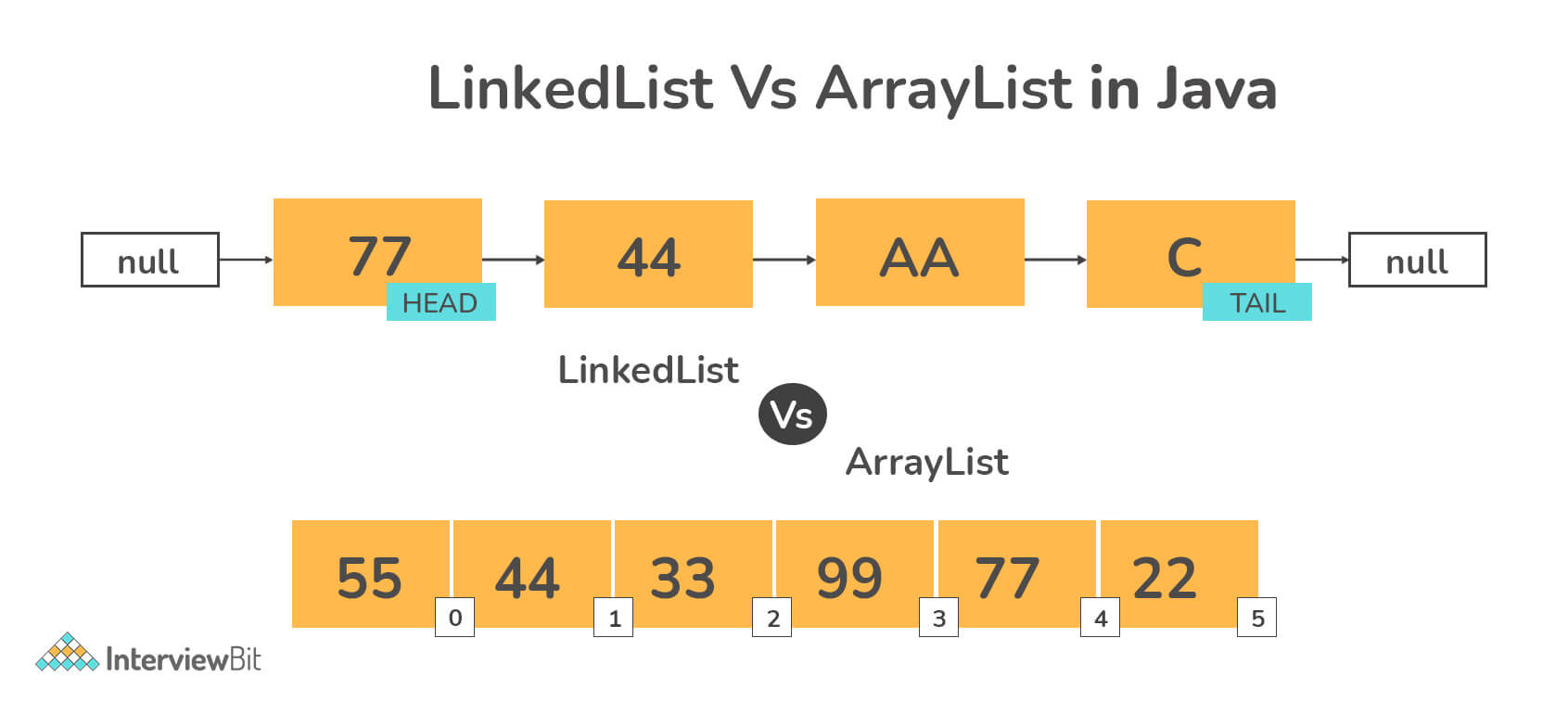
* add(E element): Adds the specified element to the end of the list.
* add(int index, E element): Inserts the specified element at the specified position in the list.
* remove(Object o): Removes the first occurrence of the specified element from the list.
* remove(int index): Removes the element at the specified position in the list.
* set(int index, E element): Replaces the element at the specified position in the list with the specified element.

**Performance Considerations:**

* LinkedList performs well for frequent insertions and deletions, especially near the beginning or end of the list.
* Random access to elements (e.g., accessing elements by index) in LinkedList may have linear-time complexity (O(n)), as it requires traversing the list from the beginning or end to reach the desired index.

**Que: Difference between ArrayList and LinkedList?**

**Ans:** Both ArrayList and LinkedList are implementations of the List interface in Java, but they have different characteristics and performance trade-offs. Here's a comparison between ArrayList and LinkedList along with guidance on when to use each:



**Data Structure:**

* **ArrayList**: Uses a resizable array internally to store elements. Provides fast random access but slower insertion and deletion operations in the middle of the list.
* **LinkedList**: Uses a doubly linked list internally. Provides efficient insertion and deletion operations, especially near the beginning or end of the list, but slower random access.

**Random Access:**

* **ArrayList**: Supports fast random access to elements using index-based retrieval (get(index)). Accessing elements by index has constant-time complexity (O(1)).
* **LinkedList**: Random access in a LinkedList may have linear-time complexity (O(n)) as it requires traversing the list from the beginning or end to reach the desired index.

**Insertion and Deletion:**

* **ArrayList**: Adding or removing elements at the end of the list is efficient (amortized constant-time complexity). Insertions or removals at arbitrary positions may require shifting elements and have linear-time complexity (O(n)).
* **LinkedList**: Provides efficient insertion and deletion operations, especially near the beginning or end of the list (constant-time complexity). Insertions or removals in the middle of the list are faster compared to ArrayList.

**Memory Usage:**

* **ArrayList**: Generally uses less memory compared to LinkedList as it only needs to store elements and an internal array.
* **LinkedList**: May have slightly higher memory overhead due to each element requiring additional pointers for linking nodes in the list.

**Traversal:**

* **ArrayList**: Traversal using indexes (for loops) is efficient due to constant-time random access.
* **LinkedList**: Traversal is efficient for forward and backward directions due to its doubly linked structure, but accessing elements by index may be slower.

**Que: When to use ArrayList and When to use Linkedlist?**

**Ans:** Usage Guidance:

**Use ArrayList:**

* When frequent random access or access by index is required.
* When the list size is known or relatively stable, as ArrayList is more efficient for fixed-size lists.
* When memory efficiency is a concern, as ArrayList generally uses less memory.

**Use LinkedList:**

* When frequent insertion or deletion operations are required, especially near the beginning or end of the list.
* When the list size may vary dynamically and resizing is common, as LinkedList can resize efficiently.
* When memory overhead is not a major concern, as LinkedList may have slightly higher memory usage.

*In summary, ArrayList is suitable for scenarios requiring fast random access and stable list sizes, while LinkedList is preferable for frequent insertion/deletion operations, dynamic list sizes, and efficient operations near the beginning or end of the list. Choosing between them depends on your specific requirements and performance considerations.*

**Que: What are Vector?**

**Ans:** In Java, Vector is a class that implements the List interface and provides a resizable array-like data structure. It is similar to ArrayList but differs in that Vector is synchronized, meaning it is thread-safe for concurrent access. Here are the key features and characteristics of Vector:

**Synchronized Operations:**

* One of the main distinctions of Vector is its synchronization. All methods in Vector are synchronized, making it thread-safe for concurrent access by multiple threads.
* Synchronization ensures that multiple threads can safely modify and access a Vector instance without causing data corruption or race conditions.

**Resizable Array:**

* Like ArrayList, Vector internally uses a resizable array to store elements.
* The array's size grows dynamically as elements are added to the Vector.

**Ordered Collection:**

* Vector maintains the insertion order of elements, meaning the order in which elements are added to the Vector is preserved.
* Elements in a Vector can be accessed and retrieved based on their index in the list.

**Random Access:**

* Vector supports fast random access to elements using index-based retrieval (get(index)).
* Accessing elements by index in a Vector has constant-time complexity (O(1)).

**Dynamic Size:**

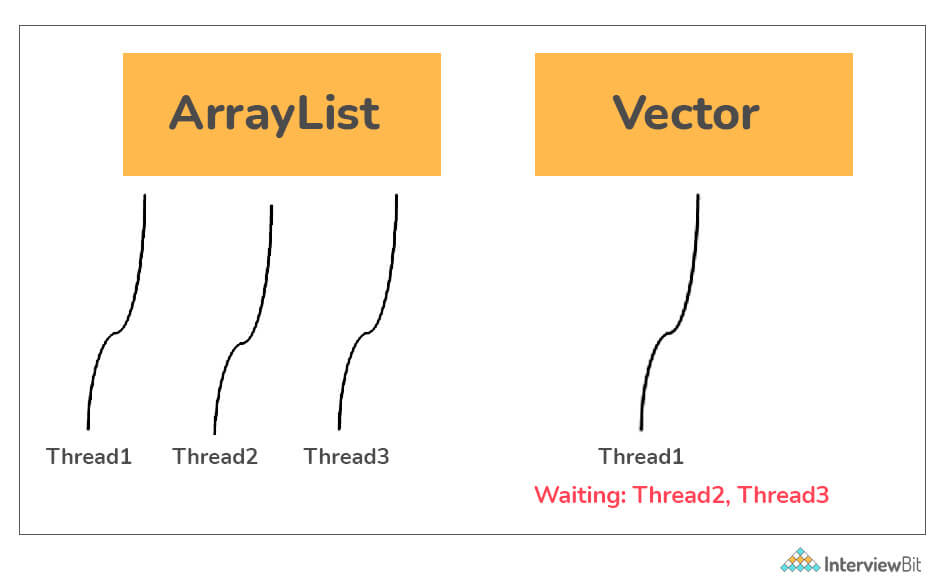
* The size of a Vector can change dynamically as elements are added or removed.
* The size() method returns the current number of elements in the Vector.

**Performance Considerations:**

* Due to synchronization overhead, Vector may have slightly lower performance compared to ArrayList for single-threaded applications.
* However, Vector is suitable for multi-threaded scenarios where thread safety is critical and concurrent access to the collection is required.

**Que: Difference between ArrayList and Vector?**

**Ans:**



**Synchronization:**

* Vector: All methods in Vector are synchronized, making it thread-safe for concurrent access by multiple threads. Synchronization ensures that multiple threads can safely modify and access a Vector instance without causing data corruption or race conditions.
* ArrayList: ArrayList is not synchronized by default. It is not thread-safe for concurrent access unless synchronized externally using synchronization blocks or using Collections.synchronizedList() method to obtain a synchronized wrapper.

**Performance:**

* Vector: Due to synchronization overhead, Vector may have slightly lower performance compared to ArrayList, especially in single-threaded applications. Synchronization can introduce contention and reduce throughput.
* ArrayList: ArrayList does not have inherent synchronization overhead, making it more efficient in single-threaded scenarios. However, it requires external synchronization for thread safety in concurrent environments.

**Resizable Array:**

* Vector: Internally uses a resizable array to store elements, similar to ArrayList.
* ArrayList: Also uses a resizable array to store elements. Both Vector and ArrayList can dynamically resize their internal arrays as elements are added or removed.

**Thread Safety:**

* Vector: Designed to be thread-safe, making it suitable for multi-threaded environments where concurrent access to the list needs synchronization.
* ArrayList: Not inherently thread-safe. It requires explicit synchronization measures when used in concurrent environments to avoid data inconsistencies or race conditions.

**Usage:**

* Vector: Use Vector when thread safety is a primary concern, and concurrent access to the list by multiple threads is required. It provides built-in synchronization.
* ArrayList: Use ArrayList when single-threaded performance is critical or when thread safety can be ensured externally using synchronization mechanisms. It is more efficient in single-threaded scenarios.

**Synchronization Overhead:**

* Vector: The overhead of synchronization can impact performance, especially in scenarios where concurrent access is not heavily utilized or necessary.
* ArrayList: Lacks synchronization overhead, making it more efficient for single-threaded operations but requiring explicit synchronization for multi-threaded safety.

**Que: What are set?**

**Ans:** In Java, the Set interface is part of the Java Collections Framework and represents a collection that does not allow duplicate elements. It models the mathematical set abstraction and provides methods for adding, removing, and checking the presence of elements in the set. Here are the key points about the Set interface in Java:

**No Duplicate Elements:**

* Sets do not allow duplicate elements. If an element is already present in the set, adding it again will have no effect.

**Unordered Collection:**

* Unlike lists, sets do not guarantee the order of elements. The order in which elements are stored or retrieved from a set may vary.

**Unique Element Checking:**

* Sets use the equals() method to check for duplicate elements. If an element is equal to an existing element in the set, it will not be added again.

**Set Operations:**

* Adding Elements:
* add(E element): Adds the specified element to the set if it is not already present.
* Removing Elements:
* remove(Object o): Removes the specified element from the set if it is present.
* Checking Containment:
* contains(Object o): Returns true if the set contains the specified element, false otherwise.
* Set Operations (Union, Intersection, Difference, etc.):
* addAll(Collection<? extends E> c): Adds all elements from the specified collection to the set.
* retainAll(Collection<?> c): Retains only the elements in the set that are also in the specified collection.
* removeAll(Collection<?> c): Removes all elements from the set that are also in the specified collection.
* clear(): Removes all elements from the set.

**Set Implementations:**

* HashSet: Implements the Set interface using a hash table. Provides constant-time performance for basic operations but does not guarantee ordering of elements.
* TreeSet: Implements the SortedSet interface using a sorted tree structure (Red-Black tree). Maintains elements in sorted order and provides efficient operations for range queries and sorted iteration.
* LinkedHashSet: Extends HashSet and maintains insertion order or access order (based on configuration). Provides predictable iteration order.
* Custom Set Implementations: Developers can also create custom set implementations based on specific requirements.

**Usage:**

* Sets are useful when you need to store a collection of unique elements and do not require ordering.
* Common use cases include removing duplicates from a collection, checking membership of elements, performing set operations, etc.

**Que: Difference between Set and the List?**

Ans: The main differences between a Set and a List in Java are in terms of the following aspects:

**Duplicate Elements:**

* **Set**: Does not allow duplicate elements. If you try to add an element that is already present in the set, it won't be added again.
* **List**: Allows duplicate elements. You can add the same element multiple times to a list, and it will maintain the order of insertion.

**Ordering**:

* **Set**: Does not guarantee the order of elements. The order in which elements are stored or retrieved from a set may not be the same as the order in which they were added.
* **List**: Maintains the order of elements based on their insertion sequence. The order of elements in a list remains consistent unless explicitly modified.

**Index-based Access:**

* **Set**: Does not support direct index-based access to elements. You cannot retrieve elements from a set using an index.
* **List**: Supports direct index-based access to elements. You can retrieve elements from a list using their index position, such as list.get(index).

**Performance:**

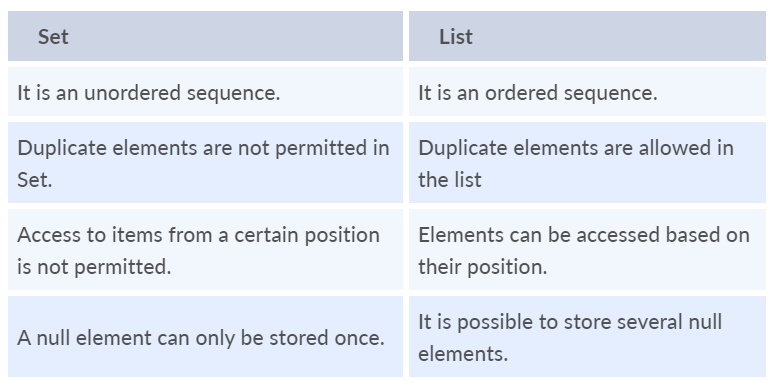
* **Set**: Depending on the implementation, sets may offer faster lookup times for checking element existence (contains() operation) compared to lists.
* **List**: Lists provide efficient access to elements based on their index (random access). However, searching for elements in an unsorted list may require linear time complexity (O(n)).

**Usage:**

* **Set:** Typically used when you need to store a collection of unique elements and do not require duplicates. Sets are useful for checking membership, removing duplicates, and performing set operations like union, intersection, etc.
* **List:** Used when you need to maintain the order of elements and allow duplicates. Lists are suitable for scenarios where you require indexed access to elements, maintaining insertion order, and allowing duplicates.

**Implementations:**

* **Set:** Implementations include HashSet, TreeSet, LinkedHashSet, etc., each providing different characteristics such as hashing, sorting, and maintaining insertion order.
* **List:** Implementations include ArrayList, LinkedList, Vector, etc., offering different performance trade-offs such as random access, efficient insertions/deletions, and synchronization.



*In summary, the key differences between Set and List revolve around handling duplicate elements, maintaining order, supporting index-based access, performance characteristics, and their typical usage scenarios. Choose between Set and List based on your specific requirements regarding uniqueness, ordering, access patterns, and performance considerations.*

**Que: What are HashSet?**

**Ans:** HashSet is a class in Java that implements the Set interface from the Java Collections Framework. It represents a collection that does not allow duplicate elements and uses a hash table-based implementation for storing elements. Here are the key features and characteristics of HashSet:

**No Duplicate Elements:**

* HashSet does not allow duplicate elements. If an element is already present in the set, attempting to add it again will have no effect.

**Hash Table Implementation:**

* Internally, HashSet uses a hash table (specifically, a HashMap instance) to store elements.
* Hashing is used to efficiently determine the bucket (index) where an element should be placed or retrieved from.

**Unordered Collection:**

* The order of elements in a HashSet is not guaranteed. The iteration order of elements may vary and is not based on the order of insertion.

**Unique Element Checking:**

* HashSet uses the equals() and hashCode() methods to check for duplicate elements.
* If an element's hashCode matches an existing element's hash code and equals() returns true, the element is considered a duplicate and will not be added.

**Performance:**

* HashSet provides constant-time performance (O(1)) for basic operations such as adding, removing, and checking for element existence (assuming a good hash function).
* Efficient hashing reduces the time complexity for these operations, making HashSet suitable for scenarios requiring fast lookups and uniqueness checks.

**Set Operations:**

**Adding Elements:**

* add(E element): Adds the specified element to the set if it is not already present.

**Removing Elements:**

* remove(Object o): Removes the specified element from the set if it is present.

**Checking Containment:**

* contains(Object o): Returns true if the set contains the specified element, false otherwise.

**Set Operations (Union, Intersection, Difference, etc.):**

* addAll(Collection<? extends E> c): Adds all elements from the specified collection to the set.
* retainAll(Collection<?> c): Retains only the elements in the set that are also in the specified collection.
* removeAll(Collection<?> c): Removes all elements from the set that are also in the specified collection.
* clear(): Removes all elements from the set.

**Usage:**

* HashSet is commonly used when you need to store a collection of unique elements and do not require the elements to be ordered.
* It is useful for removing duplicates from a collection, checking membership of elements, and performing set operations like union, intersection, etc.

**Que: What is Hashing and Hash Table?**

**Ans:** Hashing and Hashtable are related concepts in computer science and are often used in data structures and algorithms.

**Hashing:**

* **Definition:** Hashing is the process of converting input data (such as a key or value) into a fixed-size value, typically using a hash function.

The hash code is typically used as an index or key to access data quickly in hash-based data structures.

* **Hash Function:** A hash function takes input data and computes a hash code, which is a unique numeric value representing the input data. Ideally, a hash function should produce different hash codes for different inputs to minimize collisions.
* **Purpose:** Hashing is used for efficient data retrieval and storage. It allows quick lookup of data by transforming the input into an index or address in a data structure like a hashtable or hash set.

It is commonly used in hash tables, hash sets, hash maps, and other hash-based data structures for fast lookup operations.

**Hashtable:**

* **Definition:** Hashtable is a data structure that uses hashing to store key-value pairs. It maps keys to values using hash codes and provides efficient lookup and retrieval of values based on keys.
* **Implementation**: Internally, a hashtable typically uses an array (called a hash table) to store elements. Each element is stored at an index computed from its hash code using a hash function.
* **Collision** **Handling**: Collisions occur when two different inputs produce the same hash code. Hashtables use collision resolution techniques (such as chaining or open addressing) to handle collisions and store multiple elements at the same hash code index.

**Key Characteristics:**

* **Fast** **Lookup**: Hashtables provide constant-time complexity (O(1)) for average-case lookup, insertion, and deletion operations.
* **Key**-**Value** **Mapping**: Each key in a hashtable maps to a unique value, allowing efficient retrieval of values based on keys.
* **Dynamic** **Resizing**: Hashtables can dynamically resize their internal arrays (rehashing) to maintain efficiency as the number of elements grows or shrinks.
* **Non**-**Sorted**: Hashtables do not maintain the order of elements based on keys. The order of iteration may not match the insertion order.

**Que: What is Collision Handling and How it is achieve in Hash Table and what are its type?**

**Ans:** Collision handling is an important aspect of hash tables and hash-based data structures to deal with situations where multiple keys hash to the same index. Here are the common types of collision handling techniques used in hash tables:

**1. Chaining:**

* Uses linked lists at each hash table index to store multiple key-value pairs that hash to the same index.
* Simple to implement and effective but can lead to increased memory usage.

**2. Open Addressing:**

* Resolves collisions by finding alternative slots within the hash table itself.
* Includes techniques like linear probing, quadratic probing, and double hashing.
* **Linear probing** is a method of resolving collisions in open addressing by searching for the next available slot in a linear manner.
* **Quadratic** **probing** is another collision resolution technique in open addressing that uses a quadratic function to search for the next available slot.
* Can have lower memory overhead but may suffer from clustering.
* **Clustering** refers to the phenomenon where elements in a hash table tend to cluster or group together, causing consecutive slots to become occupied and potentially leading to longer probe sequences.

**3. Robin Hood Hashing:**

* A variant of open addressing that aims to balance probe lengths by redistributing elements efficiently.
* Minimizes variance in probe lengths but requires additional calculations during insertion.

**Que: What is LinkedHashSet?**

**Ans:** LinkedHashSet is a class in Java that extends HashSet and provides an ordered collection of elements with no duplicates. It combines the features of a hash table and a linked list to maintain the insertion order of elements while offering constant-time performance for basic operations such as adding, removing, and checking for element existence. Here are the key features and characteristics of LinkedHashSet:

**Ordered Collection:**

* LinkedHashSet maintains the order in which elements were inserted into the set. When iterating through the set, elements are returned in the order they were added.
* This ordered behavior is achieved by using a linked list internally along with the hash table structure for fast access.

**No Duplicate Elements:**

* Like HashSet, LinkedHashSet does not allow duplicate elements. If an element is added to the set that is already present, it won't be added again.

**Hash Table with Linked List:**

* Internally, LinkedHashSet uses a hash table (implemented by a HashMap) to store elements and a linked list to maintain the insertion order.
* Each element in the set is stored as a key in the hash table, and the linked list maintains the order of insertion.

**Performance:**

* LinkedHashSet provides constant-time performance (O(1)) for basic operations such as adding, removing, and checking for element existence, similar to HashSet.
* Iterating through the set in insertion order also has linear time complexity (O(n)) proportional to the size of the set.

**Usage:**

* LinkedHashSet is useful when you need an ordered collection of unique elements and want to maintain the order in which elements were added.
* It is suitable for scenarios where you need predictable iteration order based on insertion sequence and also want fast access and uniqueness checks.

**Que: Difference between hashset and LinkedHashSet?**

**Ans:** The key differences between HashSet and LinkedHashSet :

**1. Ordering:**

* **HashSet:** Does not maintain any specific order of elements. Iteration order may vary.
* **LinkedHashSet**: Maintains the insertion order of elements. Iterates in the order they were added.

**2. Internal Structure:**

* **HashSet:** Uses a hash table for storage and efficient element lookup.
* **LinkedHashSet**: Combines a hash table with a linked list to preserve insertion order.

**3. Performance:**

* **HashSet:** Offers faster performance for basic operations like add, remove, and contains due to its hash table implementation.
* **LinkedHashSet**: Slightly slower than HashSet for basic operations due to maintaining the linked list for ordering.

4. Usage:

* **HashSet**: Suitable for scenarios where ordering of elements is not important and fast element lookup is desired.
* **LinkedHashSet**: Used when you need to preserve the order in which elements were added, such as maintaining a history or processing elements in a specific sequence.

**Que: What are SortedSet or TreeSet?**

**Ans:** SortedSet interface is provided as part of the Java Collections Framework to represent sorted sets. The most commonly used implementation of SortedSet is TreeSet, which internally uses a self-balancing binary search tree (specifically, a Red-Black Tree) to store elements in sorted order based on their natural ordering or a custom comparator.

**Sorted Order:**

* Elements in a sorted set are automatically sorted based on their natural ordering (if they implement the Comparable interface) or a custom comparator provided during set creation.
* The sorting is maintained automatically, ensuring that elements are always in sorted order as new elements are added or existing elements are removed.

**No Duplicates:**

* Like other sets, sorted sets do not allow duplicate elements. If an attempt is made to add a duplicate element, it will not be added again.

**Efficient Retrieval:**

* Sorted sets provide efficient retrieval of elements in sorted order. Operations like finding the first or last element, finding elements greater than or less than a given value, and iterating in sorted order are performed with good time complexity.

**Implementation:**

* The TreeSet class in Java implements the SortedSet interface using a Red-Black Tree, which ensures balanced storage and efficient retrieval of sorted elements.

**Usage:**

* Sorted sets are useful in scenarios where you need to maintain a collection of elements in a specific sorted order, such as maintaining a leaderboard, managing sorted lists of items, or implementing data structures like priority queues.
* TreeSet components are sorted in ascending order.

**Red-Black Tree**

* A Red-Black Tree is a self-balancing binary search tree data structure that maintains balance during insertions and deletions.

**Binary Search Tree (BST) Property:**

* Like other binary search trees, a Red-Black Tree follows the BST property:
* All nodes in the left subtree of a node have values less than the node's value.
* All nodes in the right subtree of a node have values greater than the node's value.

**So, can we Conclude that SortedSet and TreeSet are same.**

No, TreeSet and SortedSet are not the same, although they are related concepts in Java's Collections Framework.

* **SortedSet**: SortedSet is an interface in Java that defines the behavior of a set (a collection that does not allow duplicates) where elements are stored in sorted order. It extends the Set interface and provides additional methods related to sorting, such as retrieving the first or last element, finding elements greater than or less than a given value, etc.
* **TreeSet**: TreeSet is a specific implementation of the SortedSet interface in Java. It uses a self-balancing binary search tree (usually a Red-Black Tree) to store elements in sorted order based on their natural ordering (if they implement Comparable) or a custom comparator provided during set creation.

**Que: Difference between HashSet and TreeSet?**

**Ans:** Here are the key differences between HashSet and TreeSet in Java:

**Internal Data Structure:**

* **HashSet**: Internally uses a hash table (specifically, a HashMap instance) to store elements. It provides constant-time performance (O(1)) for basic operations like add, remove, and contains.
* **TreeSet**: Internally uses a Red-Black Tree to store elements in sorted order. It provides logarithmic-time performance (O(log n)) for basic operations due to the balanced nature of Red-Black Trees.

**Ordering**:

* **HashSet**: Does not maintain any specific order of elements. Iteration order may vary and is not predictable.
* **TreeSet**: Maintains elements in sorted (ascending) order based on their natural ordering (if they implement Comparable) or a custom comparator provided during set creation.

**Duplicates**:

* Both HashSet and TreeSet do not allow duplicate elements. If you attempt to add a duplicate element, it will not be added again.

**Allowing Null values :**

* Null objects are allowed in HashSet.
* TreeSet does not allow null objects and throws a NullPointerException.

**Performance**:

* **HashSet**: Offers faster performance for basic operations like add, remove, and contains due to its hash table implementation.
* **TreeSet**: Provides efficient performance for operations requiring sorted order, such as finding the first or last element, finding elements greater than or less than a given value, etc.

**Usage**:

* **HashSet**: Suitable for scenarios where ordering of elements is not important, and fast element lookup is desired. It's commonly used when you need to store a unique collection of elements without caring about their order.
* **TreeSet**: Used when you need to maintain elements in sorted order and perform operations based on sorted order, such as range queries, finding closest elements, or maintaining a sorted collection of data.

**Que: Can you add a null element into a TreeSet or HashSet?**

**Ans:** We can add null elements in a HashSet but we cannot add null elements in a TreeSet. The reason is that TreeSet uses the compareTo() method for comparing and it throws a NullPointerException when it encounters a null element.

**Que: What is a priority queue in Java?**

**Ans:** In Java's Collections Framework, the PriorityQueue class represents a priority queue data structure. It is a specialized queue implementation where elements are ordered based on their priority, and the highest priority element is dequeued first. The PriorityQueue class provides efficient priority-based operations and is commonly used in algorithms and applications that require prioritized processing of elements.

Here are the key characteristics and features of PriorityQueue in Java:

**Priority-Based Ordering:**

* Elements in a PriorityQueue are stored and dequeued based on their priority, which is determined by either the natural ordering of elements (if they implement the Comparable interface) or a custom comparator provided during queue creation.

**Underlying Data Structure:**

* By default, PriorityQueue uses a binary heap (specifically, a min-heap) to maintain the priority order of elements.
* The binary heap structure ensures efficient insertion, removal of the highest priority element, and access to the highest priority element.
* Operations:
* Enqueue (Insertion): Adds an element to the priority queue with its associated priority. The element is placed in the appropriate position based on its priority.
* Dequeue (Removal): Removes and returns the highest priority element from the priority queue. Subsequent elements are adjusted to maintain the priority order.
* Peek: Retrieves the highest priority element without removing it from the queue.
* Size: Returns the number of elements in the priority queue.
* Is Empty: Checks if the priority queue is empty.

**Custom Comparators:**

* You can provide a custom comparator to PriorityQueue during initialization to define the priority ordering based on specific criteria.
* If no comparator is provided, elements are expected to implement the Comparable interface, and their natural ordering is used for priority comparison.

**Applications:**

* PriorityQueue is used in various applications and algorithms where elements need to be processed based on their importance, urgency, or priority.
* Common applications include task scheduling, event handling systems, network packet scheduling, Dijkstra's algorithm for shortest paths, and more.

**Insertion and Removal Operations:**

* **add**(E element): Adds the specified element to the priority queue. Equivalent to the offer method.
* **offer**(E element): Adds the specified element to the priority queue. Returns true if the element was added successfully, false otherwise.
* **remove**(): Removes and returns the highest priority element from the priority queue. Throws **NoSuchElementException** if the queue is empty.
* **poll**(): Removes and returns the highest priority element from the priority queue, or returns null if the queue is empty.
* **clear**(): Removes all elements from the priority queue.

**Access and Retrieval Operations:**

* **peek**(): Retrieves the highest priority element from the priority queue without removing it. Returns null if the queue is empty.
* **element**(): Retrieves the highest priority element from the priority queue without removing it. Throws NoSuchElementException if the queue is empty.
* **iterator**(): Returns an iterator over the elements in the priority queue. The iterator does not guarantee any specific order.

**Que: What are some of the best practices while using Java Collections?**

**Ans:** Using Java Collections effectively involves following best practices to ensure code readability, performance, and maintainability. Here are some best practices to consider when working with Java Collections:

**1. Choose the Right Collection Type:**

* Choose the appropriate collection type (e.g., List, Set, Map) based on your specific requirements (e.g., order, uniqueness, key-value pairs).
* Use interfaces (e.g., List, Set, Map) as reference types to allow flexibility in switching implementations if needed.

**2. Use Generics:**

* Always use generics (<>) to specify the type of elements stored in collections to ensure type safety and avoid casting issues.
* Prefer using diamond operator (<>) for type inference when initializing collections.

**3. Prefer Interfaces Over Concrete Implementations:**

* Use interfaces such as List, Set, and Map as variable types instead of concrete implementations (e.g., ArrayList, HashSet, HashMap) to decouple code from specific implementations.
* This allows easier swapping of implementations and promotes code extensibility.

**4. Initialize Collections Appropriately:**

* Initialize collections with an initial capacity (if known) to avoid frequent resizing and improve performance (e.g., ArrayList<Integer> list = new ArrayList<>(initialCapacity)).
* Consider using Collections.emptyList(), Collections.singletonList(), and Collections.singletonMap() for immutable empty, single-element, and single-entry collections, respectively.

**5. Use Enhanced For-Loop (for-each):**

* Prefer using enhanced for-loop (for-each) for iterating over collections to improve code readability and avoid index-based iteration.
* Example: for (Integer num : list) { /\* Process num \*/ }.

**6. Avoid Raw Types:**

* Avoid using raw types (e.g., ArrayList, HashSet) without specifying the type parameter (ArrayList<Integer>, HashSet<String>).
* Raw types can lead to type safety issues and are discouraged in modern Java programming.

**7. Consider Performance Implications:**

* Understand the performance characteristics of different collection types (e.g., ArrayList vs. LinkedList, HashSet vs. TreeSet) and choose the appropriate type based on access patterns, insertion/deletion operations, and memory usage requirements.
* Use StringBuilder for string concatenation instead of repeatedly appending strings to improve performance.

**8. Handle Null Values Appropriately:**

* Be mindful of null values in collections and handle them appropriately to avoid NullPointerExceptions.
* Consider using specialized collections like Optional, Collections.emptyXXX(), or custom null-handling strategies.

**9. Use Immutable Collections When Appropriate:**

* Use immutable collections (e.g., Collections.unmodifiableList, Collections.unmodifiableSet) to create read-only views of collections when modification is not required.
* Immutable collections help in thread-safety and prevent accidental modifications.

**10. Implement Proper Equals and HashCode:**

* If you create custom classes used as keys in HashMap or elements in HashSet, override equals() and hashCode() methods properly for correct behavior.
* Ensure consistency between equals() and hashCode() implementations to avoid unexpected behavior in hash-based collections.

**11. Specifying the initial capacity of the Collection.**

**12. Using isEmpty() instead of size():**

* To check if a collection is empty or not we should use the isEmpty() method rather than finding the size of the collection and comparing it with zero. This enhances the readability of the code.

**In short:**

* Choose the right collection type (List, Set, Map) based on your needs.
* Always use generics (<>) for type safety.
* Prefer interfaces (List, Set, Map) over concrete implementations.
* Initialize collections with an initial capacity when possible.
* Use enhanced for-loop (for-each) for iteration.
* Avoid raw types and handle null values appropriately.
* Consider performance implications of different collection types.
* Use immutable collections when modification is not needed.
* Implement proper equals() and hashCode() for custom classes used in collections.
* Specifying the initial capacity of the Collection.

**Que: What is Map in Java?**

**Ans**: A Map is an interface that represents a collection of key-value pairs. It allows you to associate keys with values, where each key is unique within the Map. The Map interface provides methods for adding, removing, and accessing elements based on their keys.

Here are some key points about Maps in Java:

**1. Key-Value Pair:**

* Each element in a Map is a key-value pair.
* Keys are used to retrieve corresponding values, and each key is unique within the Map.
* Values can be accessed and modified using their corresponding keys.

**2. No Duplicate Keys:**

* Maps do not allow duplicate keys. If you attempt to add a key that already exists, the existing value associated with that key will be replaced.

**3. Common Implementations:**

* Java provides several implementations of the Map interface, such as HashMap, TreeMap, LinkedHashMap, and ConcurrentHashMap.
* Each implementation has its characteristics in terms of performance, ordering, concurrency support, etc.

Java Map Hierarchy

**4.HashMap:**

* HashMap is the most commonly used implementation of the Map interface.
* It uses a hash table to store key-value pairs, providing fast access (constant-time complexity O(1)) for most operations (e.g., get, put, remove).

**5. TreeMap:**

* TreeMap is an implementation of the SortedMap interface, which extends Map.
* It uses a Red-Black Tree to store key-value pairs in sorted order based on the keys' natural ordering or a custom comparator.
* TreeMap provides ordered iteration over its elements (sorted by keys).

**6. LinkedHashMap:**

* LinkedHashMap is a subclass of HashMap that maintains the insertion order of key-value pairs.
* It adds a doubly linked list to the hash table to preserve the order in which elements were added.

**7. ConcurrentHashMap:**

* ConcurrentHashMap is designed for concurrent (multi-threaded) usage, providing thread-safe operations without the need for external synchronization.
* It uses a segmented structure to achieve higher concurrency compared to synchronized maps.

**8. Methods:**

* Common methods in the Map interface include put(key, value), get(key), containsKey(key), containsValue(value), remove(key), size(), isEmpty(), keySet(), values(), entrySet(), etc.

**Use maps in Java when you need:**

* Unique identification with keys.
* Fast lookup based on keys (O(1) complexity).
* Associative data storage.
* Storing configuration/settings.
* Caching frequently accessed data.
* Counting occurrences or maintaining frequency.
* Representing object relationships.
* Sorting/ordering based on keys.
* Ensuring concurrency safety in multi-threaded applications.

**Que: What are SortedMap?**

Ans: SortedMap is an interface in Java's Collections Framework that extends the Map interface to provide a collection of key-value pairs where the keys are sorted in a specific order. Unlike a regular HashMap, which does not guarantee any particular order of keys, a SortedMap maintains the keys in sorted order according to their natural ordering or a custom comparator.

Here are some key points about SortedMap:

**1. Ordered Keys:**

* SortedMap maintains its keys in a sorted order, which can be either the natural ordering of keys (if they implement Comparable) or a custom ordering defined by a comparator.
* The sorting order can be ascending (from lowest to highest) or descending (from highest to lowest), depending on the implementation.

**2. Common Implementations:**

* Java provides several implementations of the SortedMap interface, such as TreeMap and ConcurrentSkipListMap.
* TreeMap is the most commonly used implementation, which uses a Red-Black Tree to store key-value pairs in sorted order.
* ConcurrentSkipListMap is a concurrent implementation that uses a skip list data structure for concurrency support.

**3. Methods:**

* SortedMap includes methods such as firstKey(), lastKey(), headMap(), tailMap(), and subMap() to perform operations based on key ranges and subsets.
* It also provides methods for accessing the keys and values in sorted order, such as keySet(), values(), and entrySet().

**4. Use Cases:**

* SortedMap is useful when you need to maintain keys in a specific order, such as alphabetical order, numerical order, or a custom order based on object properties.
* It is commonly used for tasks that require sorted key-value pairs, like maintaining a sorted dictionary, implementing ordered data structures, or performing range-based operations on keys.

**Que: What is Tree Map?**

**Ans:** In Java, TreeMap is an implementation of the SortedMap interface, which itself extends the Map interface in the Java Collections Framework. TreeMap is specifically designed to store key-value pairs in a sorted order based on the natural ordering of keys (if they implement Comparable) or a custom comparator provided during TreeMap creation.

Here are key points about TreeMap:

**1. Sorted Map Implementation:**

* TreeMap is a sorted map implementation, meaning it maintains its entries in sorted order based on keys.
* The sorting order can be natural (if keys implement Comparable) or custom (using a comparator).

**2. Balanced Binary Search Tree:**

* Internally, TreeMap uses a self-balancing binary search tree (commonly a Red-Black Tree) to store key-value pairs.
* The tree structure ensures logarithmic-time complexity (O(log n)) for most operations, including search, insertion, and deletion.

**3. Key-Value Pairs:**

* Each element in a TreeMap is a key-value pair, where keys are unique and keys are sorted in ascending order by default.
* Duplicate keys are not allowed; if you attempt to add a duplicate key, the existing value associated with that key is updated.

**4 .Sorted Order Operations:**

* TreeMap provides methods to perform operations based on sorted order, such as firstKey(), lastKey(), headMap(), tailMap(), and subMap() for range-based operations.

**5. Insertion and Retrieval:**

* Inserting elements into a TreeMap follows the binary search tree insertion algorithm, ensuring keys are inserted in their sorted position.
* Searching for a key in a TreeMap involves traversing the binary search tree based on key comparisons, providing efficient search operations.

**6. Custom Comparator:**

* You can provide a custom comparator to TreeMap during initialization to define a custom sorting order based on specific criteria.
* This allows flexibility in sorting elements according to different rules.

**7. Efficiency:**

* TreeMap provides efficient operations with logarithmic complexity for most operations, making it suitable for scenarios where sorted order and efficient key-based access are required.

**Que: What are HashMap?**

**Ans:** HashMap is a class in Java that implements the Map interface, allowing you to store key-value pairs in a hash table data structure. It provides efficient performance for basic operations such as insertion, deletion, and retrieval of elements based on keys.

Here are key points about HashMap:

**1. Hash Table Implementation:**

* HashMap internally uses a hash table to store key-value pairs.
* The hash table is an array of buckets, and each bucket can store multiple entries (key-value pairs).
* Hashing is used to determine the index (bucket) where an entry will be stored based on its key.

**2. Key-Value Pairs:**

* Each element in a HashMap is a key-value pair, where keys are unique within the map.
* Duplicate keys are not allowed; if you attempt to add a duplicate key, the existing value associated with that key is updated.

**3. Efficient Operations:**

* HashMap provides efficient performance for basic operations:
* Insertion (put(key, value)): Adds a key-value pair to the map.
* Deletion (remove(key)): Removes the key-value pair associated with the specified key.
* Retrieval (get(key)): Retrieves the value associated with the specified key.
* These operations have an average time complexity of O(1) (constant-time), making HashMap suitable for high-performance scenarios.

**4. Hashing and Buckets:**

* Hashing is used to compute the hash code of keys, which determines the index (bucket) where the key-value pair will be stored.
* Collisions may occur when multiple keys hash to the same index. HashMap uses separate chaining (linked list) or tree-based storage (for JDK 8 and later) to handle collisions in buckets efficiently.

**5. Null Keys and Values:**

* HashMap allows one null key and multiple null values.
* A null key is hashed to index 0 in the hash table.

**6. Iterating over Elements:**

* You can iterate over HashMap elements using iterators (Iterator, forEach) or enhanced for-loop (for-each).
* Iteration order is not guaranteed and may not be predictable (unless using LinkedHashMap for insertion order).

**7. Capacity and Load Factor:**

* HashMap has initial capacity and load factor parameters.
* Capacity is the number of buckets in the hash table, and load factor determines when the hash table should be resized (rehashed) based on the ratio of elements to buckets.

**8. Concurrency:**

* HashMap is not synchronized and is not thread-safe for concurrent access.
* For concurrent access, consider using ConcurrentHashMap or synchronizing access externally.

**Internal Working of HashMap**

The internal working of a HashMap in Java involves several key concepts, including hashing, buckets, and collision resolution mechanisms. Here's an overview of how HashMap works internally:

**1. Hashing:**

* When you add a key-value pair to a HashMap, the HashMap uses the key's hash code to determine the index (or bucket) where the entry will be stored.
* The hash code is computed using the key's hashCode() method.

**2. Bucket Structure:**

* Internally, a HashMap is organized as an array of buckets, where each bucket can hold multiple entries.
* The index of each bucket is determined by applying a hash function to the key's hash code.

**3. Hash Collision:**

* Hash collisions occur when two different keys hash to the same index (bucket) in the HashMap.
* To handle collisions, HashMap uses a chaining mechanism. Each bucket in the array can hold a linked list of entries that hashed to the same index.

**4. Entry Structure:**

* Each entry in the HashMap is represented by an instance of the HashMap. Entry class (or Node in older versions).
* The entry contains the key, value, hash code of the key, and a reference to the next entry in case of a collision (linked list structure).

**5. Insertion:**

* When you insert a key-value pair into a HashMap, the hash code of the key is used to determine the bucket index.
* If there is no collision, the entry is added to the bucket. If a collision occurs, the new entry is appended to the linked list in the corresponding bucket.

**6. Retrieval:**

* When you retrieve a value from a HashMap using a key, the hash code of the key is used to locate the bucket.
* The linked list in the bucket is then traversed (if necessary) to find the entry with the matching key.

**7. Load Factor and Rehashing:**

* HashMap maintains a load factor threshold (default is 0.75) to control the density of entries in the buckets.
* When the number of entries exceeds the load factor multiplied by the capacity, the HashMap is resized (rehashed), and the entries are redistributed to new buckets.
* This resizing and redistribution process helps maintain efficient performance by balancing the number of entries per bucket.

**8. Performance:**

* HashMap provides constant-time complexity (average case) for insertion, retrieval, and removal operations, assuming a good hash function and minimal collisions.
* However, excessive collisions can degrade performance, so choosing an appropriate initial capacity and load factor is important.

**Load Factor:**

The load factor is a concept used in hash-based data structures like HashMaps to determine when the structure should be resized (rehashed) to maintain efficiency. In simple terms, the load factor represents the ratio of the number of entries (elements) in the data structure to the number of buckets available. It is calculated as:

Load Factor=Number of Buckets / Number of Entries​.

**Que: What is the default size of the load factor in hashing based collection?**

**Ans:** The default load factor size is 0.75. The default capacity is calculated by multiplying the initial capacity by the load factor.

**Que: Differentiate between Set and Map in Java?**

**Ans:** In Java, both Set and Map are part of the Java Collections Framework and serve different purposes:

**Set:**

* A Set is a collection that stores unique elements.
* It does not allow duplicate elements, meaning if you try to add an element that already exists in the Set, the operation will be ignored.
* Sets do not maintain an order of elements, although some implementations like TreeSet provide sorted ordering based on natural or custom ordering of elements.
* Common implementations of Set in Java include HashSet, TreeSet, and LinkedHashSet.

**Map:**

* A Map is a collection that stores key-value pairs.
* Each key in a Map is associated with a corresponding value, allowing you to retrieve values using their keys.
* Keys in a Map must be unique, but values can be duplicated.
* Maps do not maintain a specific order of key-value pairs, although some implementations like TreeMap provide sorted ordering based on the keys.
* Common implementations of Map in Java include HashMap, TreeMap, LinkedHashMap, and ConcurrentHashMap.

**Que: Differentiate between HashSet and HashMap?**

**Ans:** Sure, here's a brief differentiation between HashSet and HashMap:

* HashSet stores unique elements (no duplicates), while HashMap stores key-value pairs.
* HashSet uses HashMap internally to store elements as keys with a dummy value.
* Internally, HashSet uses HashMap to add entries. The key K in a HashSet is the argument supplied in the add(Object) method. For each value supplied in the add(Object) method, Java assigns a dummy value.
* HashSet ensures uniqueness of elements, while HashMap requires unique keys but allows duplicate values.
* HashSet does not maintain any order of elements, while HashMap's order is not predictable (except for LinkedHashMap).
* HashSet is used for storing unique elements, while HashMap is used for key-value associations and efficient value retrieval based on keys.

**Que: What is LinkedHashMap?**

**Ans:** LinkedHashMap is an implementation of the Map interface in Java that maintains the order of key-value pairs based on the insertion order. It combines the features of a HashMap and a LinkedList, providing efficient key-value mapping along with predictable iteration order.

**--> Doubly linked list me order of insertion maintain karat Hai**

Here are the key points about LinkedHashMap:

**Order Maintenance:**

* LinkedHashMap maintains the insertion order of key-value pairs.
* The order is the same as the order in which elements were added to the map.

**Hashing Mechanism:**

* Like HashMap, LinkedHashMap uses hashing to store key-value pairs in buckets for efficient retrieval.
* Each bucket in the map contains a doubly-linked list of entries, preserving the insertion order within each bucket.

**Performance**:

* LinkedHashMap's performance is similar to HashMap for most operations, with a slight overhead for maintaining the linked list structure.
* Access order mode incurs additional overhead for maintaining access patterns but can be useful for certain scenarios like cache implementations maintaining the linked list for order.

**Usage:**

* LinkedHashMap is useful when you need predictable iteration order based on insertion or access patterns.
* It's commonly used in scenarios where maintaining order is important, such as maintaining a history of operations, LRU (Least Recently Used) caches, and ordered mappings.

**Que: What is the difference between HashMap and LinkedHashMap?**

**Ans:** The key differences between HashMap and LinkedHashMap in Java:

**HashMap:**

* Does not maintain any specific order of key-value pairs.
* Uses an array of buckets and chaining for collision resolution.
* Provides better performance for basic operations due to simpler internal structure.
* Iteration order is not predictable and may vary between iterations.
* Suitable for efficient key-value mapping without order guarantees.

**LinkedHashMap:**

* Maintains insertion order of key-value pairs.
* Combines hash table with a doubly-linked list for order maintenance.
* Has slightly lower performance compared to HashMap due to linked list overhead.
* Iteration order is predictable and remains consistent.
* Useful when predictable insertion order or access-based ordering is required.
* Such as maintaining order or implementing LRU caches.

**Que: What is the difference between HashMap and HashTable?**

Ans: Here are the key differences between HashMap and HashTable in Java:

**HashMap:**

**1. Synchronization:**

* HashMap is not synchronized, meaning it is not thread-safe.
* Multiple threads can access and modify a HashMap concurrently, but external synchronization is needed to ensure thread safety.

**2. Null Values:**

* HashMap allows null values for both keys and values.
* You can have one null key and multiple null values in a HashMap.

**3. Performance:**

* HashMap generally offers better performance than HashTable due to its lack of synchronization.
* It is preferred in scenarios where thread safety is not a concern.

**4. Iterating:**

* Iterators in HashMap are fail-fast, meaning they throw ConcurrentModificationException if the HashMap is structurally modified while iterating.

**HashTable:**

**1. Synchronization:**

* HashTable is synchronized, making it thread-safe.
* All methods of HashTable are synchronized, ensuring that multiple threads can access and modify it safely without external synchronization.

**2. Null Values:**

* HashTable does not allow null keys or null values.
* If you try to store null as a key or value, HashTable throws a NullPointerException.

**3. Performance:**

* HashTable may have slightly degraded performance compared to HashMap due to the overhead of synchronization.
* It is suitable for scenarios where concurrent access and modification by multiple threads are required.

**4. Iterating:**

* Iterators in HashTable are fail-safe, meaning they do not throw ConcurrentModificationException if the HashTable is modified while iterating.
* Instead, they work on the original copy of the HashTable's elements and do not reflect subsequent modifications made after the iterator is created.

**Que: Why does HashMap allow null whereas HashTable does not allow null?**

**Ans:** HashMap allows null values because it is designed to be more flexible and efficient, catering to scenarios where developers may need to store and handle null values as part of their data structure.

HashTable does not allow null keys or values because it is designed with a focus on thread safety and robustness in concurrent environments.

**Que: What is ConcurrentHashMap?**

Ans: ConcurrentHashMap is a class in Java that provides a thread-safe implementation of the Map interface. It is designed to support concurrent access by multiple threads without causing data corruption or inconsistencies. Here are the key points about ConcurrentHashMap:

**1. Thread-Safety:**

* ConcurrentHashMap is designed to be thread-safe, allowing multiple threads to read and write to the map concurrently without external synchronization.
* It uses various locking mechanisms internally to ensure thread safety during concurrent access.

**2. Partitioning**:

* ConcurrentHashMap divides the underlying data structure into segments (default is 16 segments).
* Each segment acts as an independent hash table, and threads can operate on different segments concurrently without blocking each other.

**3. Concurrency Level:**

* The concurrency level of a ConcurrentHashMap determines the number of threads that can concurrently modify the map.
* Higher concurrency levels can improve concurrency performance but may also increase memory overhead.

**4. Performance:**

* ConcurrentHashMap provides high-performance concurrent access for both read and write operations.
* It optimizes performance by minimizing contention between threads operating on different segments.

**5. Iterators:**

* ConcurrentHashMap supports weakly consistent iterators, meaning they reflect the state of the map at the time of iterator creation and may not show the most recent updates made by other threads.

**6. Scalability:**

* ConcurrentHashMap is scalable for concurrent access, allowing multiple threads to access different segments concurrently without blocking each other.
* It is suitable for scenarios where high concurrency and thread safety are required, such as concurrent data structures in multi-threaded applications.

**Que: What is Enumeration in Java?**

**Ans:** In Java, Enumeration is an interface that defines methods to enumerate (or iterate over) elements in a collection of objects. It's part of the legacy collections framework and was commonly used before the introduction of iterators in Java 2.

Here are the key points about Enumeration in Java collections:

**1. Interface Structure:**

* Enumeration interface is defined in the java.util package.
* It declares three methods:
* boolean hasMoreElements(): Returns true if the enumeration contains more elements, false otherwise.
* E nextElement(): Returns the next element in the enumeration as type E.
* default void forEachRemaining(Consumer<? super E> action): Performs the given action for each remaining element until all elements have been processed or the action throws an exception.

**2. Usage:**

* Enumeration was commonly used with legacy collections like Vector, Hashtable, and Properties to iterate over their elements.
* It doesn't support modification operations like adding or removing elements during iteration, unlike iterators.

**3. Limitations**:

* Enumeration is read-only and does not support modification operations like adding or removing elements from the collection during iteration.
* It lacks the functionalities and flexibility provided by iterators and enhanced for loops introduced in later versions of Java.

**4. Replacement:**

* In modern Java programming, iterators (like Iterator and ListIterator) and enhanced for loops (for-each loop) are preferred over Enumeration due to their enhanced functionalities, type safety, and support for modification operations.

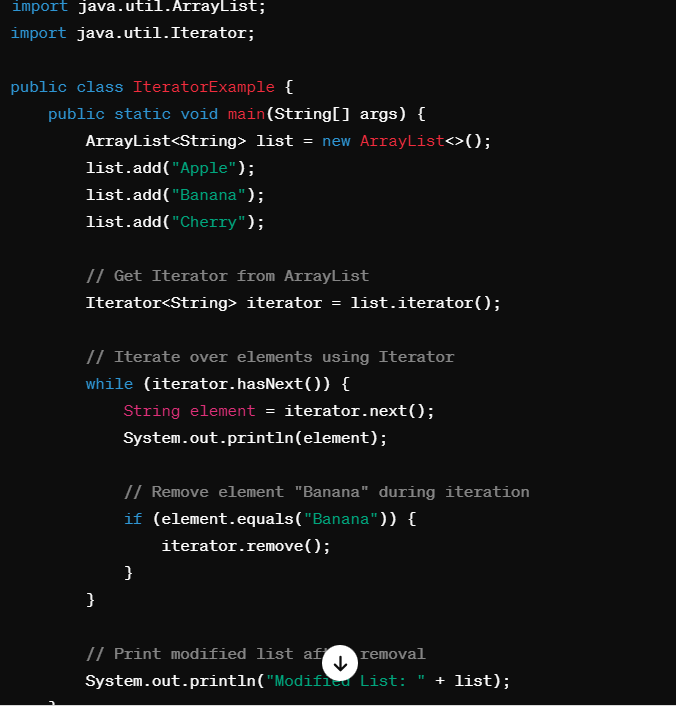
**Que: What are Iterators in Java?**

**Ans:** Iterators in Java collections are objects that provide a way to traverse and access elements in a collection sequentially. They are part of the Java Collections Framework and offer several advantages over older interfaces like Enumeration, including support for safe removal of elements and enhanced functionality for iterating over collections.

Here are the key points about iterators in Java collections:

**1. Interface:**

* Iterators are represented by the Iterator interface in Java, which is part of the java.util package.
* The Iterator interface defines three main methods:
* boolean hasNext(): Returns true if there are more elements in the collection to iterate over.
* E next(): Returns the next element in the iteration as type E.
* void remove(): Removes the last element returned by the iterator from the underlying collection (optional operation).



**2**. **Usage**:

* Iterators are commonly used with collection classes like ArrayList, LinkedList, HashSet, HashMap, etc., to traverse and access elements.
* They provide a safe way to iterate over collections, as they support the removal of elements during iteration without causing concurrent modification exceptions.

**3. Advantages**:

* Iterators provide a safe way to traverse and modify collections, including support for removal operations during iteration.
* They offer more functionalities and flexibility compared to older interfaces like Enumeration.

**4. Limitations**:

* Iterators are unidirectional and only support forward traversal of elements.
* They cannot be used to traverse elements in reverse order, which requires a separate interface like ListIterator for lists.

**Que: What are ListIterators?**

**Ans:** ListIterators in Java are iterators specifically designed for traversing elements in lists, such as ArrayList, LinkedList, and other list-based collections. Unlike regular iterators (which are unidirectional and can only move forward in the collection), ListIterators provide bidirectional traversal, allowing movement in both forward and backward directions within the list.

Here are the key points about ListIterators in Java:

**1. Interface:**

* ListIterators are represented by the ListIterator interface in Java, which extends the Iterator interface.
* The ListIterator interface provides additional methods beyond the basic iterator operations for bidirectional traversal and modification of lists.

**2. Bidirectional** **Traversal**:

* ListIterators allow traversal in both forward and backward directions within the list.
* Methods like hasNext() and next() move the iterator forward, while methods like hasPrevious() and previous() move it backward.



**3. Additional Operations:**

* ListIterators support additional operations like add(), remove(), and set() to modify the list during iteration.
* The add() method adds an element to the list at the iterator's current position.
* The remove() method removes the last element returned by the iterator from the list.
* The set() method replaces the last element returned by the iterator with a specified element.

**4. Usage:**

* ListIterators are commonly used when you need to traverse elements in lists bidirectionally and perform modifications during traversal.
* They are useful for tasks like iterating over a list, removing specific elements, adding new elements, and updating existing elements.

**Que: What are the difference between Iterators and ListIterators?**

Ans: Certainly, here are the differences between Iterator and ListIterator in points:

**Iterator:**

1. Supports forward traversal only.

2. Provides methods like `hasNext()` and `next()` for moving forward in the collection.

3. Does not have methods for backward traversal or accessing previous elements.

4. Supports basic modification operations like `remove()` during iteration.

5. Used for iterating over collections like ArrayList, HashSet, etc.

6. Iterators cannot be used to obtain Indexes.

7. It work on traversal of Maps, List and Set.

8. Throws a Concurrent Modification Exception since it can’t add element.

**ListIterator**:

1. Supports bidirectional (forward and backward) traversal.

2. Provides methods like `hasNext()`, `next()`, `hasPrevious()`, and `previous()` for bidirectional movement.

3. Allows adding, removing, and updating elements during iteration using methods like `add()` and `set()`.

4. Specifically designed for traversing elements in lists like ArrayList, LinkedList, etc.

5. Ideal for tasks requiring bidirectional iteration and list modifications.

6. It offers methods to get element indexes at any time while traversing List, such as next Index() and previous Index().

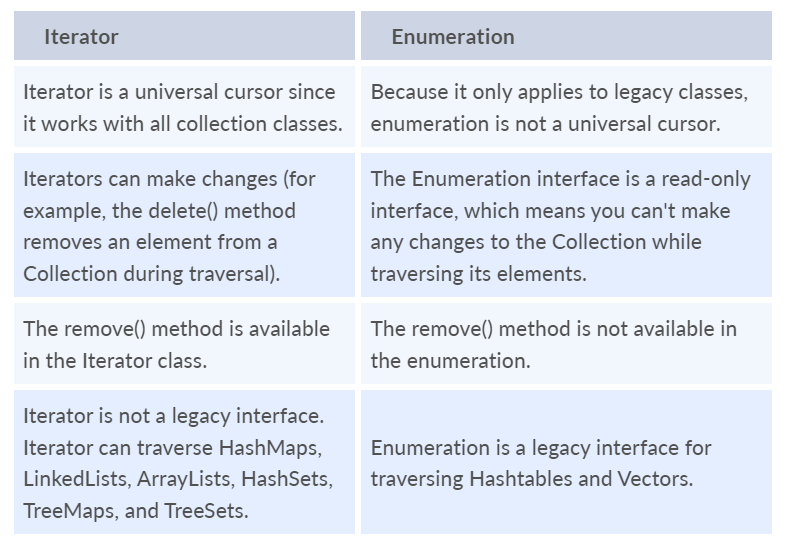
7. Only List may be traversed, not set and maps.

8. At any time, you can quickly add elements to a collection.

In summary, Iterator is for forward traversal and basic modifications, while ListIterator offers bidirectional traversal and advanced modifications, especially for lists.

**Que: Difference between Iterators and Enumeration?**

**Ans:**

****

**Que: What is fail-fast and fail-safe iterators in java collection?**

**Ans:** Fail-Fast and Fail-Safe are two different approaches used in handling concurrent modification exceptions in Java collections, especially when multiple threads are involved in modifying a collection concurrently.

**Fail-Fast Iterators:**

* Fail-Fast iterators detect and throw a ConcurrentModificationException if the underlying collection is modified structurally (adding or removing elements) during the iteration.
* These iterators operate on the original copy of the collection's elements and do not tolerate concurrent modifications while iterating.
* Fail-Fast iterators are commonly used in classes like ArrayList, HashMap, HashSet, etc.
* **Example: When an ArrayList is modified structurally (e.g., an element is added or removed) while an iterator is traversing it, the iterator will throw a ConcurrentModificationException.**

**Fail-Safe Iterators:**

* Fail-Safe iterators do not throw ConcurrentModificationException even if the underlying collection is modified structurally during iteration.
* They operate on a cloned copy of the collection or use a copy-on-write strategy, ensuring that the iterator continues to work on the original collection's snapshot.
* Fail-Safe iterators are commonly used in classes like CopyOnWriteArrayList, ConcurrentHashMap, and CopyOnWriteArraySet.
* **Example: In a CopyOnWriteArrayList, modifications made to the list during iteration will not affect the iterator's behavior, as it works on a snapshot of the list taken at the time of iterator creation.**

**Comparison:**

**Fail-Fast Iterators:**

* Detect and throw ConcurrentModificationException if the collection is modified during iteration.
* Operate on the original collection, making them efficient but not suitable for concurrent modifications.
* Used in non-thread-safe collections like ArrayList, HashMap, etc.

**Fail-Safe Iterators:**

* Do not throw ConcurrentModificationException even if the collection is modified during iteration.
* Operate on a copy of the collection or use copy-on-write strategy, ensuring thread safety.
* Used in thread-safe collections like CopyOnWriteArrayList, ConcurrentHashMap, etc.

**Que: What is Properties Class in Java?**

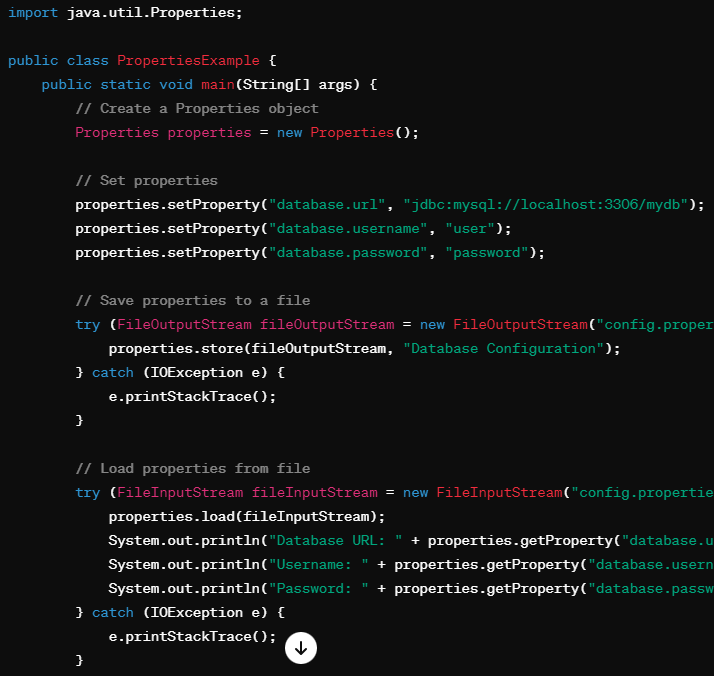
**Ans:** In Java, Properties refers to a class that represents a persistent set of properties or configuration settings. It's part of the Java.util package and is commonly used for handling application configurations, settings, and other key-value pairs. Here are the key points about Properties in Java:

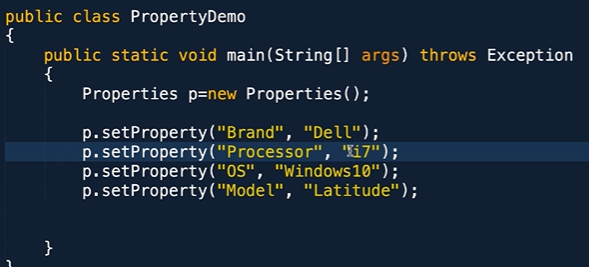
**Key-Value Pairs:**

* Properties is a subclass of Hashtable and stores key-value pairs where both keys and values are strings.
* Each key and its corresponding value in Properties is a string.

**Usage:**

* Properties are often used for storing and managing configuration settings in Java applications.
* They can be used to store properties like database connection information, application settings, localization data, etc.



 Property and the Value.

**Reading Properties:**

* Properties can be loaded from various sources such as files, input streams, or directly added programmatically.
* Common methods for loading properties include load(InputStream in) to load from a file or input stream, and setProperty(String key, String value) to add properties programmatically.

**Saving** Properties:

* Once properties are loaded or set, they can be saved back to a file or output stream using the store(OutputStream out, String comments) method.
* The store method writes the properties to the specified output stream with optional comments.

**Advantages of the Properties file:**

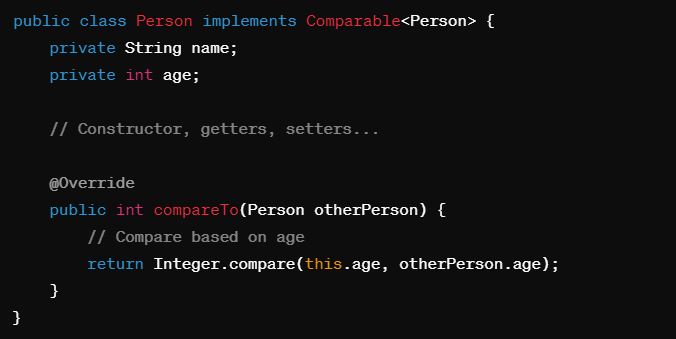
* If the information in a properties file is modified, no recompilation is required:
* You don't need to recompile the java class if any information in the properties file changes.
* It is used to keep track of information that needs to be updated frequently.

**Que: What are Comparable in Java?**

**Ans:** In Java, Comparable are interfaces used for sorting objects in collections like lists, sets, and arrays. Here's an explanation of each:

**Comparable Interface:**

* Comparable is an interface in Java that allows a class to implement natural ordering for its objects.
* The Comparable interface has a single method compareTo(Object obj) that compares the current object (this) with the specified object (obj) and returns an integer value indicating the comparison result.
* The compareTo method should return a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than the specified object obj, respectively.
* If a class implements the Comparable interface, it can be sorted using methods like Collections.sort() or Arrays.sort(). These sorting methods internally use the compareTo method to compare objects and arrange them in the correct order.
* This is particularly useful for sorting objects in collections like lists or arrays.
* The Comparable interface is part of the java.lang package.



**Que: What are Comparators in Java?**

**Ans:** In Java, Comparator is an interface that allows you to define custom sorting rules for objects of a particular class, regardless of whether that class implements the Comparable interface or not. Comparators are useful when you need to sort objects based on criteria other than their natural ordering or when you want to have multiple sorting options for the same class.

Here are the key points about comparators in Java:

**1. Interface Definition:**

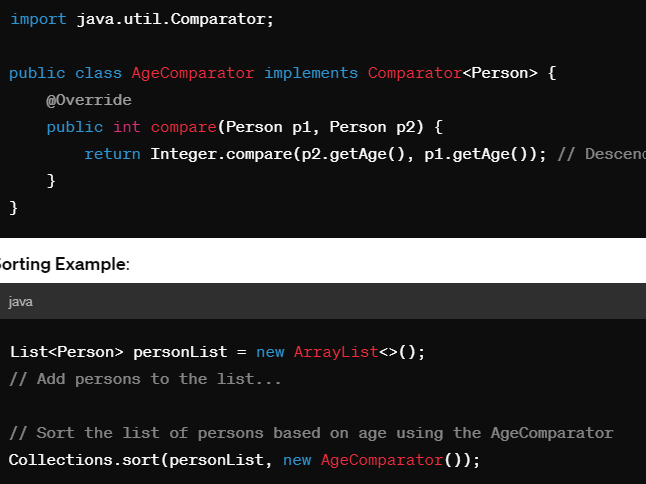
* The Comparator interface is part of the java.util package.
* It declares a single method compare(T o1, T o2) that compares two objects o1 and o2 and returns an integer value indicating the comparison result.
* The compare method should return a negative integer, zero, or a positive integer if o1 is less than, equal to, or greater than o2, respectively.

**2. Usage:**

* Comparators are used to define custom sorting logic for objects based on specific criteria.
* They can be applied to classes that do not implement Comparable or to override the natural ordering defined by Comparable.

**3. Sorting:**

* Collections and arrays of objects can be sorted using comparators by passing the comparator object to sorting methods like Collections.sort(List<T> list, Comparator<? super T> c) or Arrays.sort(T[] array, Comparator<? super T> c).



**Que: Difference between Comparator and Comparable in Java?**

**Ans**: Here are the key differences between Comparable and Comparator in Java in short points:

**Comparable:**

1. Interface in Java located in `java.lang` package.

2. Used to define natural ordering for a class's objects.

3. Implemented by the class whose instances need to be sorted.

4. Contains `compareTo(Object obj)` method for comparison.

5. Objects are sorted based on their natural ordering.

6. Suitable for sorting a class's objects based on a single criteria.

**Comparator:**

1. Interface in Java located in `java.util` package.

2. Used to define custom sorting rules for a class's objects.

3. Implemented separately from the class being sorted.

4. Contains `compare(T o1, T o2)` method for comparison.

5. Objects are sorted based on custom criteria defined by the comparator.

6. Suitable for sorting objects based on multiple criteria or when natural ordering is not desired or available.

In summary, **Comparable is used for defining natural ordering within a class, while Comparator is used for defining custom sorting logic externally, allowing for more flexible and diverse sorting options like multiple sorting criteria.**

**Que: What do you understand about BlockingQueue in Java?**

**Ans:** BlockingQueue is a data structure in Java that represents a queue with the added capability of blocking operations when the queue is either full (in the case of adding elements) or empty (in the case of removing elements). It's part of the java.util.concurrent package and is designed for concurrent programming scenarios where multiple threads are accessing and modifying the queue simultaneously.

Here are the key points about BlockingQueue in Java:

**1. Thread-Safe:**

* BlockingQueue implementations are thread-safe, meaning they can be safely accessed and modified by multiple threads concurrently without external synchronization.

**2. Blocking Operations:**

* BlockingQueue supports blocking operations like put() and take().
* put(E element) blocks the calling thread if the queue is full until space becomes available for adding the element.
* take() blocks the calling thread if the queue is empty until an element is available for removal.

**3. Variants:**

* Java provides several implementations of BlockingQueue, such as LinkedBlockingQueue, ArrayBlockingQueue, PriorityBlockingQueue, etc.
* Each variant has its characteristics regarding capacity, ordering, and fairness in thread scheduling.

**4. Usage:**

* BlockingQueue is commonly used in concurrent applications where producer and consumer threads interact.
* Producers can add elements to the queue using put() without worrying about queue overflow, as the operation blocks when the queue is full.
* Consumers can remove elements from the queue using take() without worrying about empty queues, as the operation blocks when the queue is empty.

**Que: What is the purpose of RandomAccess Interface? Name a collection type which implements this interface?**

**Ans:** The purpose of the `RandomAccess` interface in Java is to indicate that a particular collection supports efficient random access to its elements, typically through index-based retrieval. Implementing the `RandomAccess` interface implies that accessing elements at any position within the collection has constant time complexity (O(1)).

The primary use case for the `RandomAccess` interface is to optimize algorithms and operations that heavily rely on random access, such as iterating over elements using index-based loops or accessing elements by their index position.

An example of a collection type that implements the `RandomAccess` interface is `ArrayList`. `ArrayList` provides constant-time random access to its elements, making it suitable for scenarios where efficient index-based retrieval and manipulation of elements are required. Other collections like `LinkedList` do not implement `RandomAccess` because they may have varying time complexity for random access operations depending on the position of the elements.

**Que: Why does not the Map interface extend the Collection Interface or vice-versa?**

**Ans:** The decision for the Map interface not to extend the Collection interface (or vice versa) in Java is based on fundamental differences in their behaviors and functionalities. Here are the key reasons why they are kept separate:

**Data Structure and Purpose:**

* The Collection interface represents a group of objects (elements) that are typically stored and accessed in a sequential manner (e.g., lists, sets).
* On the other hand, the Map interface represents a key-value pair association, where each key maps to a unique value. Maps are not sequential collections but rather associative collections.

**Method Signatures and Operations:**

* The methods and operations defined in the Collection interface (e.g., add, remove, iterate) are focused on managing individual elements within a collection.
* In contrast, the methods in the Map interface (e.g., put, get, remove) are designed to work with key-value pairs and perform operations based on keys rather than individual elements.

**Que: How can you synchronize an ArrayList in Java?**

**Ans:**

**1. Using Collections.synchronizedList():**

* You can synchronize an ArrayList in Java using the Collections.synchronizedList() method. This method returns a synchronized (thread-safe) wrapper around the given ArrayList, ensuring that access and modification operations on the list are synchronized and safe for concurrent use by multiple threads.

**2. Using CopyOnWriteArrayList:**

* CopyOnWriteArrayList maintains thread safety by creating a new copy of the underlying array whenever modification operations (add, set, remove, etc.) are performed.
* Concurrent read operations (such as iteration) on the CopyOnWriteArrayList do not require synchronization and can proceed concurrently.
* Modifications to the list (adding, removing elements) are thread-safe and automatically synchronized by the class, ensuring consistency across threads.

**Que: Why do we need synchronized ArrayList when we have Vectors (which are synchronized) in Java?**

**Ans:** The main reason for using a synchronized ArrayList over Vectors in Java is flexibility and performance:

**Flexibility:**

* Synchronized ArrayList allows more fine-grained control over synchronization compared to Vectors.
* With synchronized ArrayList, you can choose which specific operations need synchronization by using synchronized blocks or methods, whereas Vectors synchronize all operations by default.

**Performance:**

* Synchronized ArrayList may offer better performance in certain scenarios due to its more granular synchronization approach.
* Vectors synchronize all methods, which can lead to unnecessary synchronization overhead, especially in single-threaded or less concurrent scenarios.

**Que: What happens if I try to send duplicate element to set , is it throw exception or ignores it ?**

**Ans:** When you try to add a duplicate element to a Set in Java, it does not throw an exception; instead, it simply ignores the duplicate element. Sets in Java are designed to store unique elements, meaning that duplicate elements are automatically discarded when added to the Set.

**Que: In hashmap if you send duplicate key at compile time and at runtime ?**

**Ans**: In Java, a HashMap does not allow duplicate keys. If you attempt to add a duplicate key to a HashMap at compile time or runtime, it will not throw a compilation error, but the new value associated with the duplicate key will replace the existing value.

**Que: Can we have null values in key or value in Hashmap ?**

**Ans**: Yes, HashMap in Java allows both null keys and null values.

**1. Null Key:**

HashMap permits a single null key. That means you can add a key-value pair with a null key into a HashMap.

***hashMap.put(null, 1);***

**2. Null Value:**

HashMap allows multiple null values. That means you can have multiple key-value pairs where the value is null.

**3. Handling Null Keys and Values:**

* When you use a null key, it will be mapped to a particular bucket in the HashMap, and you can retrieve the value associated with the null key using the get(null) method.
* Similarly, when you use a null value, it will be stored in the HashMap as part of the key-value pair, and you can retrieve it using the corresponding key.

**Que: What is difference between Collection and Collections in Java?**

**Ans:** The terms "Collections" and "Collection" in Java refer to different concepts:

**1. Collection (Interface):**

* Collection is an interface in Java that represents a group of objects, known as elements. It is part of the Java Collections Framework (JCF) and is found in the java.util package.
* Collections that implement the Collection interface can contain duplicate elements and are not necessarily ordered. Examples of classes that implement Collection include List, Set, and Queue.

**2. Collections (Class):**

* Collections (with an "s") is a utility class in Java that provides static methods for working with collections. It is also part of the Java Collections Framework and is found in the java.util package.
* The Collections class provides methods for operations such as sorting, searching, shuffling, synchronization, and creating read-only collections. These methods are useful for manipulating and managing collections.
* Collections (class) is a utility class that provides static methods for working with collections, such as sorting and synchronization.

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

***// Creating a Collection (List)***

***Collection<String> namesCollection = new ArrayList<>();***

***namesCollection.add("Alice");***

***namesCollection.add("Bob");***

***namesCollection.add("Charlie");***

***// Using the Collections utility class to sort the Collection***

***List<String> sortedNamesList = new ArrayList<>(namesCollection);***

***Collections.sort(sortedNamesList); // Using Collections.sort() to sort the List***

***System.out.println("Original Collection: " + namesCollection);***

***System.out.println("Sorted List: " + sortedNamesList);***

**Que: Mention the used case of each Collection?**

**Ans:** Sure, here are some common use cases for each type of collection in Java:

**1. List:**

* Use a List when you need to store a collection of elements where duplicate elements are allowed and the order of elements is significant.
* Use cases:
* Storing and manipulating a list of items such as to-do tasks, shopping items, or user input history where the order matters.
* Implementing data structures like stacks or queues using ArrayList or LinkedList.

**2. Set:**

* Use a Set when you need to store a collection of unique elements where duplicate elements are not allowed, and you do not require a specific order for elements.
* Use cases:
* Storing a collection of unique identifiers, email addresses, or usernames.
* Performing set operations such as union, intersection, or difference between sets.

**3. Map:**

* Use a Map when you need to store key-value pairs where each key maps to a single value, and each key must be unique.
* Use cases:
* Storing and retrieving data based on keys, such as user preferences (key: setting name, value: setting value).
* Implementing lookup tables or dictionaries for quick access to data based on keys.

**4. Queue:**

* Use a Queue when you need to store and process elements in a FIFO (First-In-First-Out) manner, where elements are added at the end and removed from the beginning of the queue.
* Use cases:
* Implementing task queues, message queues, or job scheduling systems where the order of processing matters.
* Modeling scenarios such as waiting lines or order processing systems.

**5. Deque:**

* Use a Deque (Double-ended queue) when you need to store and process elements in a FIFO (First-In-First-Out) or LIFO (Last-In-First-Out) manner, allowing elements to be added or removed from both ends of the deque.
* Use cases:
* Implementing data structures such as stacks (LIFO) or queues (FIFO) using methods like push, pop, offer, poll, etc.
* Managing sliding window operations or maintaining a history of recent actions.

**6. Other Specialized Collections:**

Java also provides specialized collections like TreeSet and TreeMap for sorted sets and maps, LinkedHashSet and LinkedHashMap for maintaining insertion order, and concurrent collections (ConcurrentHashMap, CopyOnWriteArrayList, etc.) for thread-safe operations in concurrent environments.

**Que: What are Collectors in Java?**

**Ans:** In Java, the Collectors class is part of the Java Streams API (java.util.stream.Collectors) and provides a set of static utility methods for performing reduction operations on streams and collecting elements into various data structures. These methods are commonly used when working with Java streams to transform, filter, and aggregate data. Here are some important methods and their functionalities from the Collectors class:

**1. toList():**

Collects stream elements into a List.

List<String> list = stream.collect(Collectors.toList());

**2. toSet():**

Collects stream elements into a Set.

Set<String> set = stream.collect(Collectors.toSet());

**3. toMap():**

Collects stream elements into a Map based on provided key and value mappers.

Map<Integer, String> map = stream.collect(Collectors.toMap(MyObject::getId, MyObject::getName));

**4.joining():**

Concatenates stream elements into a single String using a delimiter, prefix, and suffix.

String result = stream.collect(Collectors.joining(", ", "[", "]"));

**5. groupingBy():**

Groups stream elements by a classifier function into a Map.

Map<Integer, List<MyObject>> groupedMap = stream.collect(Collectors.groupingBy(MyObject::getCategoryId));

**6. partitioningBy():**

Partitions stream elements into two groups based on a predicate.

Map<Boolean, List<MyObject>> partitionedMap = stream.collect(Collectors.partitioningBy(MyObject::isActive));

**7. counting():**

Counts the number of elements in the stream.

long count = stream.collect(Collectors.counting());

**8. summarizingInt(), summarizingDouble(), summarizingLong():**

Collects statistics (count, sum, min, max, average) for numeric properties of stream elements.

IntSummaryStatistics stats = stream.collect(Collectors.summarizingInt(MyObject::getQuantity));

**9. toCollection():**

Collects stream elements into a specified collection type (e.g., ArrayList, LinkedList).

List<MyObject> customList = stream.collect(Collectors.toCollection(ArrayList::new));

**10. reducing():**

List<MyObject> customList = stream.collect(Collectors.toCollection(ArrayList::new));

Optional<Integer> sum = stream.collect(Collectors.reducing(Integer::sum));

**Java Stream API**

**Que: What is Stream API in Java?**

**Ans:** The Stream API in Java is a powerful addition introduced in Java 8 that provides a way to process collections of objects in a functional style.

Basically:

Stream= Flow

* Streams are a sequence of elements that can be generated from various sources like collections, arrays, or I/O channels.
* Stream API is a way to express and process collections of objects.
* Enable us to perform operations like filtering, mapping, reducing and sorting.
* Stream does not store elements. It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.

**Que: What are the advantages of using the Java Stream API?**

**Ans:** Here are some key advantages of using the Java Stream API:

**1. Expressive and Concise Code:**

* The Stream API allows developers to write code in a more declarative and expressive style, reducing boilerplate code and making the codebase more readable and maintainable.
* Functional programming constructs such as lambda expressions and method references can be used with streams, leading to concise and elegant code.

**2. Stream Pipelines:**

* Stream pipelines provide a way to chain multiple operations (such as filtering, mapping, sorting, and reducing) together in a fluent and sequential manner.
* This stream pipeline approach simplifies complex data processing tasks and promotes a more structured and organized code structure.

**3. Lazy Evaluation:**

* Streams use lazy evaluation, meaning intermediate operations are deferred until a terminal operation is invoked.
* Lazy evaluation improves performance by avoiding unnecessary processing of elements that are not needed in the final result, especially for large data sets.

**4. Parallel Processing:**

* The Stream API supports parallel streams, allowing operations to be executed concurrently on multi-core processors.
* Parallel streams can significantly improve performance for computationally intensive tasks, leveraging the inherent parallelism of modern hardware.

**5. Immutable and Non-Interfering Operations:**

* Stream operations are designed to be non-interfering and produce immutable results, ensuring that the original data source remains unchanged during processing.
* This immutability simplifies concurrency and parallelism management, reducing the risk of data corruption or unexpected behavior.

**6. Built-in Aggregate Operations:**

* The Stream API provides built-in aggregate operations such as sum, average, max, min, count, and reduce, making it easier to perform common data aggregation tasks.
* These aggregate operations streamline code development and reduce the need for manual iteration and accumulation logic.

**7. Integration with Functional Interfaces:**

* The Stream API integrates seamlessly with functional interfaces such as Predicate, Function, Consumer, Supplier, and Comparator, enabling flexible and customizable behavior for stream processing.
* Developers can define their own functional interfaces and use them effectively with streams, enhancing code reusability and modularity.

**Que: What are the differences between Streams and Collections?**

**Ans:** Here are the key differences between Streams and Collections:

**1. Nature of Data Processing:**

* Collections are data structures that hold a collection of elements in memory, allowing random access, modification, and traversal of elements.
* Streams are not data structures; they represent a sequence of elements that can be processed sequentially or in parallel. Streams are designed for efficient data processing operations such as filtering, mapping, sorting, and reducing.

**2. Mutability:**

* Collections are mutable, meaning elements can be added, removed, or modified after the collection is created.
* Streams are immutable; once a stream is created, its elements cannot be changed. Instead, stream operations produce new streams with modified or transformed elements.

**3. Lazy Evaluation:**

* Collections perform operations eagerly, meaning all elements are processed and stored in memory when operations are executed (e.g., iterating over a collection with a for-each loop).
* Streams use lazy evaluation, executing operations only when a terminal operation is invoked. This lazy evaluation improves performance by avoiding unnecessary processing of elements that are not needed in the final result.

**4. Parallel Processing:**

* Collections do not inherently support parallel processing; developers need to write custom code using threads or ExecutorService for parallelism.
* Streams support parallel processing out-of-the-box through parallel streams. Parallel streams leverage multi-core processors to process elements concurrently, improving performance for large data sets and computationally intensive operations.

**5. Memory Footprint:**

* Collections store all elements in memory, potentially consuming significant memory space for large collections.
* Streams process elements on-the-fly without storing intermediate results, reducing memory overhead, especially for data processing tasks that involve filtering or transforming large data sets.

**Que: What are the different ways to create a Stream in Java?**

**Ans**: Here are some of the most common ways to create a Stream in Java:

**1. From a Collection:**

***List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);***

***Stream<Integer> streamFromCollection = numbers.stream();***

**2. From an Array:**

***String[] strings = {"hello", "world"};***

***Stream<String> streamFromArray = Arrays.stream(strings);***

**3. Using Stream.of() for Individual Elements**:

***Stream<Integer> streamOfElements = Stream.of(1, 2, 3, 4, 5);***

**4. From a Range of Values:**

***IntStream intStream = IntStream.range(1, 6); // Generates 1, 2, 3, 4, 5***

**5. From a Range of Values:**

***IntStream intStream = IntStream.range(1, 6); // Generates 1, 2, 3, 4, 5***

**6. Using Stream.generate() or Stream.iterate():**

***Stream<Double> randomStream = Stream.generate(Math::random); // Generates an infinite stream of random numbers***

***Stream<Integer> iterateStream = Stream.iterate(0, n -> n + 2).limit(5); // Generates 0, 2, 4, 6, 8***

* The Stream.iterate() method is used to generate an infinite Stream by repeatedly applying a function to a seed value.
* The iterate() method takes two arguments: the initial seed value and a UnaryOperator that generates the next element of the Stream.

**7. From a File:**

***Path path = Paths.get("file.txt");***

***Stream<String> lines = Files.lines(path);***

**Que: What are the different type of operations in the streams?**

**Ans**: In Java, streams provide a powerful way to perform various operations on collections of data. These operations can be broadly categorized into intermediate and terminal operations.

**Intermediate Operations:**

* Intermediate operations are typically used to transform or filter the elements of a stream.
* Intermediate Operations are the types of operations in which multiple methods are chained in a row.
* Intermediate operations transform a stream into another stream.
* It enables the concept of filtering where one method filters data and passes it to another method after processing.

**1. map():** Applies a function to each element in the stream and returns a new stream with the results.

***Stream<Integer> numbers = Stream.of(1, 2, 3, 4, 5);***

***Stream<Integer> squaredNumbers = numbers.map(x -> x \* x); // Produces 1, 4, 9, 16, 25***

**2. filter():** Selects elements from the stream based on a given predicate.

***Stream<Integer> numbers = Stream.of(1, 2, 3, 4, 5);***

***Stream<Integer> evenNumbers = numbers.filter(x -> x % 2 == 0); // Produces 2, 4***

**3. distinct():** Removes duplicate elements from the stream.

***Stream<Integer> numbers = Stream.of(1, 2, 2, 3, 3, 4, 5);***

***Stream<Integer> distinctNumbers = numbers.distinct(); // Produces 1, 2, 3, 4, 5***

**4. sorted():** Sorts the elements of the stream.

***Stream<Integer> numbers = Stream.of(5, 2, 4, 1, 3);***

***Stream<Integer> sortedNumbers = numbers.sorted(); // Produces 1, 2, 3, 4, 5***

**5. limit():** Limits the number of elements in the stream.

***Stream<Integer> numbers = Stream.of(1, 2, 3, 4, 5);***

***Stream<Integer> limitedNumbers = numbers.limit(3); // Produces 1, 2, 3***

**6. skip():** Skips a specified number of elements in the stream.

***Stream<Integer> numbers = Stream.of(1, 2, 3, 4, 5);***

***Stream<Integer> skippedNumbers = numbers.skip(2); // Skips first 2 elements, produces 3, 4, 5***

**7. flatMap():** Flattens a stream of streams into a single stream.

***List<List<Integer>> nestedList = Arrays.asList(***

***Arrays.asList(1, 2),***

***Arrays.asList(3, 4),***

***Arrays.asList(5, 6)***

***);***

***Stream<Integer> flattenedStream = nestedList.stream().flatMap(List::stream); // Produces 1, 2, 3, 4, 5, 6***

As the name suggests, is the combination of a map and a flat operation. This means you first apply the map function and then flatten the result.

Flattern= Sapat == sab mila ke barabar kr dega.

***List<Integer> evens = Arrays.asList(2, 4, 6);***

***List<Integer> odds = Arrays.asList(3, 5, 7);***

***List<Integer> primes = Arrays.asList(2, 3, 5, 7, 11);***

***List<Integer> numbers = Stream.of(evens, odds, primes)***

***.flatMap(list -> list.stream())***

***.collect(Collectors.toList());***

***O*utput:** flattend list: [2, 4, 6, 3, 5, 7, 2, 3, 5, 7, 11]

**Terminal Operations**:

Terminal operations produce a result or side effect and close the stream.

**1**. **forEach**(): Performs an action for each element in the stream.

***Stream<Integer> numbers = Stream.of(1, 2, 3, 4, 5);***

***numbers.forEach(System.out::println); // Prints each number on a new line***

**2. collect():** Collects the elements of the stream into a collection or other data structure.

***Stream<Integer> numbers = Stream.of(1, 2, 3, 4, 5);***

***List<Integer> collectedNumbers = numbers.collect(Collectors.toList()); // Collects elements into a List***

**3. reduce():** Performs a reduction operation on the elements of the stream.

* The reduce method is used to reduce the elements of a stream to a single value. The reduce method takes a BinaryOperator as a parameter.

***Stream<Integer> numbers = Stream.of(1, 2, 3, 4, 5);***

***Optional<Integer> sum = numbers.reduce(Integer::sum); // Calculates the sum of elements (result: 15)***

***List number = Arrays.asList(2,3,4,5);***

***int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);***

***// Here ans variable is assigned 0 as the initial value and i is added to it.***

**4. count():** Returns the count of elements in the stream.

***Stream<Integer> numbers = Stream.of(1, 2, 3, 4, 5);***

***long count = numbers.***count(); // Returns 5

**5. anyMatch(), allMatch(), noneMatch():** Checks if any, all, or none of the elements in the stream match a given predicate.

***Stream<Integer> numbers = Stream.of(1, 2, 3, 4, 5);***

***boolean anyEven = numbers.anyMatch(x -> x % 2 == 0); // Returns true***

***boolean allEven = numbers.allMatch(x -> x % 2 == 0); // Returns false (stream consumed)***

***boolean noneEven = numbers.noneMatch(x -> x % 2 == 0); // Returns false (stream consumed)***

**6. toArray() Method:**

* The toArray() method collects the elements of a stream into an array.
* It returns an array of type Object[], and an overloaded version allows you to specify the array type explicitly.

***Stream<String> words = Stream.of("apple", "banana", "cherry");***

***String[] wordArray = words.toArray(String[]::new);***

***System.out.println("Word array: " + Arrays.toString(wordArray)); // Output: Word array: [apple, banana, cherry]***

**7. findFirst(), findAny():** Returns the first element or any element from the stream, wrapped in an Optional.

***Stream<Integer> numbers = Stream.of(1, 2, 3, 4, 5);***

***Optional<Integer> firstNumber = numbers.findFirst(); // Returns Optional[1]***

***Optional<Integer> anyNumber = Stream.of(1, 2, 3).findAny(); // Returns Optional[1] or Optional[2] or Optional[3]***

**8. min() and max() Methods:**

* The min() and max() methods are used to find the minimum and maximum elements in a stream, respectively.
* These methods return an Optional<T> object containing the minimum or maximum element, or an empty Optional if the stream is empty.

***Stream<Integer> numbers = Stream.of(3, 1, 4, 1, 5, 9, 2, 6);***

***Optional<Integer> minNumber = numbers.min(Comparator.naturalOrder());***

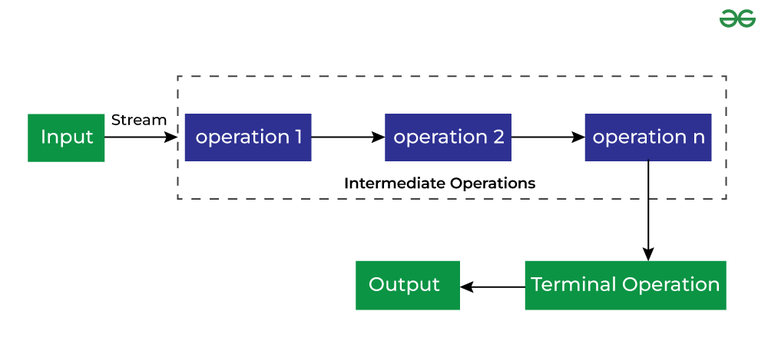
***System.out.println("Minimum number: " + minNumber.orElse(null)); // Output: Minimum number: 1***

**Optional Keyword in Java?**

The Optional<Integer> type represents an object that may or may not contain a value of type Integer. In Java, the Optional class was introduced to handle scenarios where a method may return a value or not, preventing null pointer exceptions and improving code clarity.

Here's a brief explanation of Optional:

* If a value is present, an Optional object will contain that value.
* If no value is present, the Optional object will be empty.



**Que: Difference Between Stream map() and flatMap() Functions**

**Ans:** The map() and flatMap() functions in Java's Stream API are both used for transforming elements in a stream, but they operate in slightly different ways.

**map() Function:**

* The map() function transforms each element of a stream into another element using the provided mapping function.
* It produces a one-to-one mapping between elements of the original stream and elements of the resulting stream.
* The mapping function used with map() returns a single value for each input element.

***List<String> words = Arrays.asList("hello", "world", "java");***

***Stream<Integer> lengths = words.stream().map(String::length);***

***// Resulting stream: 5, 5, 4***

**flatMap() Function**:

* The flatMap() function also transforms each element of a stream, but it deals with streams of streams.
* It flattens the elements of nested streams into a single stream.
* The mapping function used with flatMap() returns a stream for each input element, and these streams are then flattened into one stream.

***List<List<Integer>> nestedLists = Arrays.asList(***

***Arrays.asList(1, 2),***

***Arrays.asList(3, 4),***

***Arrays.asList(5, 6)***

***);***

***Stream<Integer> flattenedStream = nestedLists.stream().flatMap(List::stream);***

***// Resulting stream: 1, 2, 3, 4, 5, 6***

**Que: What are Parallel Streams?**

**Ans:** Parallel streams in Java refer to a feature introduced in Java 8 that allows streams to process elements concurrently, taking advantage of multicore processors and potentially speeding up the processing of large data sets. Unlike sequential streams, which process elements in a single thread sequentially, parallel streams distribute the workload across multiple threads, which can lead to improved performance for tasks that can be parallelized.

Here are some key points about parallel streams:

**1. Concurrency:** Parallel streams internally use the Java Fork/Join framework to split the workload into smaller tasks that can be executed concurrently by multiple threads.

**2. Multicore Processors**: Parallel streams are particularly beneficial on multicore processors, as they can utilize the available cores to process elements concurrently, potentially reducing overall processing time.

**3. Stream API Integration:** Parallel streams are seamlessly integrated into the Java Stream API, making it easy to switch between sequential and parallel processing by using the .parallel() or .sequential() methods.

**4. Thread Safety:** Parallel streams handle synchronization and thread safety internally, ensuring that stream operations are thread-safe without requiring explicit synchronization by the programmer.'

***List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);***

***// Sequential stream***

***long sequentialSum = numbers.stream().reduce(0, Integer::sum);***

***System.out.println("Sequential sum: " + sequentialSum);***

***// Parallel stream***

***long parallelSum = numbers.parallelStream().reduce(0, Integer::sum);***

***System.out.println("Parallel sum: " + parallelSum);***

In this example, parallelStream() is used to create a parallel stream, and the .reduce() operation is applied to calculate the sum of elements. The parallel stream utilizes multiple threads to process the elements concurrently, potentially improving performance for large collections.

**Que: Difference between parallel streams and sequential streams in Java?**

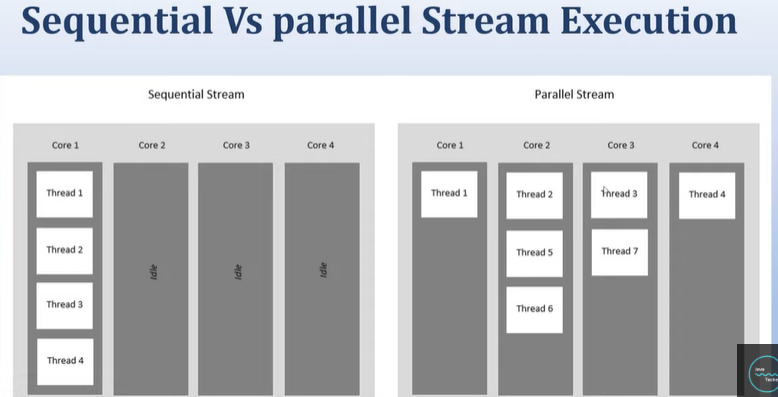
**Ans:** The main difference between parallel streams and sequential streams in Java lies in how they process elements and utilize resources such as CPU cores and threads.

**1. Processing Model:**

* **Sequential Stream:** Processes elements one after the other, in a single thread, maintaining the order of elements as they appear in the stream. Operations are executed sequentially without parallelism.
* **Parallel Stream**: Processes elements concurrently using multiple threads, potentially utilizing multiple CPU cores. Elements may be processed out of order due to concurrent execution.

**2. Execution Mode:**

* **Sequential Stream**: Executes operations in a single-threaded mode, where each element flows through the stream one at a time.
* **Parallel Stream:** Executes operations in a multi-threaded mode, where elements are processed concurrently across multiple threads.



**Que: What are Predicate and Consumers?**

**Ans:**

**Predicate:**

* Predicate is a functional interface in Java that represents a boolean-valued function of one argument.
* It is used to define a condition or criteria that can be evaluated on an object.
* The Predicate interface has a single method called test(T t), which takes an argument of type T and returns a boolean value indicating whether the condition holds true or false.
* Predicates are commonly used in streams to filter elements based on certain criteria. For example, you can use a Predicate to filter a list of numbers and keep only the even numbers.
* Example of a Consumer:

***Predicate<Integer> isEven = num -> num % 2 == 0;***

***boolean result = isEven.test(10); // Returns true***

**Consumer:**

* Consumer is a functional interface in Java that represents an operation that accepts a single input argument and returns no result.
* It is used to perform an action or side-effect on an object without returning any value.
* The Consumer interface has a single method called accept(T t), which takes an argument of type T and performs the specified action.
* Consumers are commonly used in streams for forEach operations, where you apply an action to each element in the stream.
* Example of a Consumer:

***Consumer<String> printUpperCase = str -> System.out.println(str.toUpperCase());***

***printUpperCase.accept("hello"); // Prints "HELLO"***

**Serialization In Java**

**Que: What is Serialization?**

**Ans**: Serialization is the conversion of a Java object into a byte stream.

Serialization in Java refers to the process of converting an object into a format that can be easily stored, transmitted, or reconstructed later. The serialized form of an object is a sequence of bytes that includes the object's data along with information about its type and structure. Serialization is commonly used for saving object state to files, sending objects over networks, or caching objects in memory.

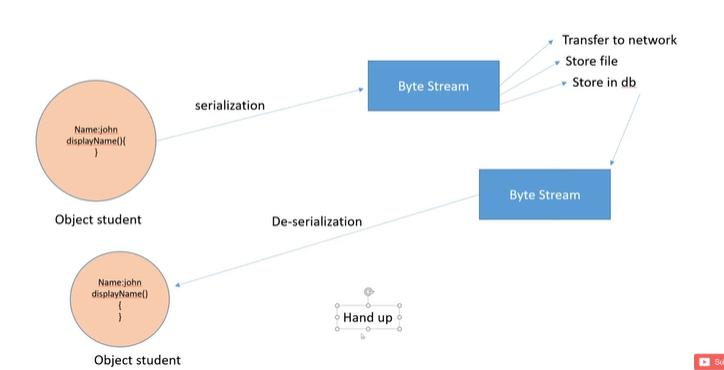
**Key Points about Serialization:**

**1. Object Serialization:**

* In Java, object serialization is achieved by implementing the java.io.Serializable interface.
* Objects of classes that implement Serializable can be converted into a stream of bytes and then restored back into objects later.

**2. Serialization Process:**

* During serialization, an object's state (instance variables) is converted into a byte stream using ObjectOutputStream.
* The byte stream contains information about the object's class, its fields, and their values.



**3. Transient Fields:**

* Fields marked as transient are excluded from serialization. This is useful for excluding sensitive data or fields that don't need to be serialized.
* Transient fields are not saved into the byte stream and are initialized with default values during deserialization.

**4. Deserialization:**

* Deserialization is the process of reconstructing an object from its serialized form.
* During deserialization, the byte stream is converted back into an object using ObjectInputStream.

**5. Serialization Formats:**

* Java's default serialization format is platform-independent, but it's not very space-efficient.
* Alternative serialization formats like JSON, XML, or Protocol Buffers (protobuf) are often used for interoperability with other platforms or for more compact serialization.

**Que: What is Serializable ?**

**Ans**: Serializable in Java refers to a marker interface that enables objects of a class to be serialized. Serialization is the process of converting an object into a stream of bytes so that it can be easily stored, transmitted over a network, or persisted to a file. The java.io.Serializable interface serves as a marker, indicating that the objects of a class can be serialized and deserialized.

**Marker Interface:**

* Serializable is a marker interface in Java, which means it doesn't have any methods or fields. It simply marks a class as serializable.
* To make a class serializable, it must implement the Serializable interface.

**Que: What is serialVersionUID?**

**Ans:** serialVersionUID in Java is a unique identifier used during the serialization and deserialization process to ensure that the version of the class being serialized and the version during deserialization are compatible. It is a special static final field that acts as a version control mechanism for serialized objects.

**Purpose:**

* serialVersionUID is used to uniquely identify versions of serializable classes.
* It ensures that serialized objects can be deserialized correctly, even if the class definition has changed between serialization and deserialization.

***class Person implements Serializable {***

***private static final long serialVersionUID = 1L; // Explicit serialVersionUID declaration***

***private String name;***

***private int age;***

**Default Calculation:**

* If you do not explicitly define serialVersionUID, Java calculates it automatically based on various aspects of the class, such as its name, fields, methods, and other structural elements.
* The default calculation can change if the class structure is modified, potentially leading to compatibility issues during deserialization.

**Explicit Declaration:**

* It's recommended to explicitly declare serialVersionUID in serializable classes to maintain versioning control and prevent compatibility issues.
* You can declare serialVersionUID as private static final long serialVersionUID = <value>; where <value> is a unique long integer.

**Exception**:

* If the serialVersionUID in the serialized data matches the one in the class, deserialization proceeds smoothly.
* If the serialVersionUID values do not match, a **InvalidClassException** or **ClassCastException** may occur, indicating a version mismatch.

**Que: What are compatible and incompatible changes in Serialization process?**

**Ans:** In the context of Java serialization, changes made to a class can be classified as either compatible or incompatible based on their impact on the serialization and deserialization process.

**Compatible Changes:**

Compatible changes are changes to a class that do not affect the serialization and deserialization process, allowing serialized objects to be deserialized without errors. Some compatible changes include:

* Adding new non-transient fields to a class is generally compatible, as existing serialized objects will simply initialize the new fields with their default values during deserialization.
* hanging a field's type to a compatible type (e.g., from int to long, or from ArrayList to List) is often compatible if the new type can represent the old type's values without loss of information.
* Adding custom serialization (writeObject()) or deserialization (readObject()) methods is compatible, as these methods can be used to handle versioning or special serialization logic.

**Incompatible Changes:**

Incompatible changes are changes to a class that can lead to serialization or deserialization errors, potentially causing data loss or runtime exceptions. Some incompatible changes include:

* Removing fields from a class is incompatible, as deserialization will fail if the serialized data contains information for the removed fields.
* Changing the names of existing fields is incompatible, as deserialization relies on field names to match with serialized data.
* Changing the serialVersionUID in a class can lead to versioning issues and is generally incompatible, as Java uses serialVersionUID to ensure version compatibility.
* Changing a field's type to an incompatible type (e.g., from int to String, or from ArrayList to HashMap) is incompatible if the new type cannot represent the old type's values correctly.
* Changing the inheritance hierarchy (e.g., superclass or implemented interfaces) can be incompatible if it affects the serialized form or behavior of objects.

**Que: What if Serialization is not available, is any other alternative way to transfer object over network?**

**Ans:** Here are some alternatives to serialization for transferring objects over the network:

**1. JSON (JavaScript Object Notation):**

* JSON is a lightweight data interchange format that is widely used for transmitting data between a server and a client over a network.

**2. XML (extensible Markup Language):**

* XML is another widely used data format for exchanging structured data between systems.

**3. Protocol Buffers (protobuf):**

* Protocol Buffers is a binary serialization format developed by Google for efficient data serialization.

**4. Message Queues:**

* Message queuing systems like Apache Kafka, RabbitMQ, or ActiveMQ provide mechanisms for exchanging messages between distributed systems.

**5. RESTful APIs (Representational State Transfer):**

* RESTful APIs use HTTP methods (GET, POST, PUT, DELETE) to perform CRUD (Create, Read, Update, Delete) operations on resources.
* Objects can be converted to JSON or XML representations and sent as HTTP requests and responses.

**Que: What will happen if one the member of class does not implement Serializable interface?**

**Ans:**

* Serialization focuses on preserving and restoring the state of individual objects. Static member variables belong to the class itself (class-level state), not to individual objects (object-level state).
* Static variables are shared among all instances of a class and are associated with the class's definition, not with specific object instances.

**Que: Is constructor of class called during DeSerialization process?**

**Ans**: During the deserialization process, the constructor of a class is not called. Instead, the object is reconstructed using the data obtained from the serialized stream.

Object Creation:

* When an object is deserialized, Java allocates memory for the object and initializes its fields using default values (e.g., null for reference types, 0 for numeric types, false for boolean).
* This process bypasses the constructor, as the object is created directly from the serialized data without invoking the constructor.

**Que: Is constructor of super class called during DeSerialization process of subclass ?**

**Ans:** No, the constructor of the superclass is not called during the deserialization process of a subclass in Java. When deserializing a subclass object, Java initializes the superclass part of the object using a no-argument (default) constructor, bypassing the constructor of the superclass.

**Que**: **What values will int and Integer will be initialized to during DeSerialization process if they were not part of Serialization?**

**Ans:** Default value will be given.

* int= 0
* Integer = null

**Que: Explain the process of deserialization?**

**Ans**: The process of deserialization in Java involves reconstructing an object from its serialized form, which is a stream of bytes representing the object's state. Deserialization is commonly used to restore objects that were previously serialized, allowing data to be transmitted or stored in a persistent format. Here's a step-by-step explanation of the deserialization process:

**Object Creation:**

* When an object is deserialized, Java allocates memory for the object and initializes its fields using default values (e.g., null for reference types, 0 for numeric types, false for boolean).
* This process bypasses the constructor, as the object is created directly from the serialized data without invoking the constructor.

**Object Initialization:**

* After the object is created, Java populates its fields with the values obtained from the serialized stream.
* For non-serializable fields or transient fields, default values are used if no custom logic is provided in the readObject() method.

**Custom Deserialization Logic:**

* If the class implements Serializable and defines a readObject() method, this method is called during deserialization.
* In the readObject() method, you can perform custom initialization or post-processing logic for the deserialized object, such as initializing transient fields or handling versioning changes.

**Transient and Static Fields:**

* Transient fields are not serialized, so their values are initialized to default values during deserialization.
* Static fields are not serialized either; they belong to the class, not to individual objects, so they are not part of the serialization process and retain their values from the class definition.

**Que: What are the disadvantages of serialization in Java?**

**Ans:** The disadvantages of serialization in Java include:

* Versioning issues: Changes to serialized classes can lead to compatibility problems during deserialization.
* Performance overhead: Serialization and deserialization processes can impact application performance, especially for large objects.
* Security risks: Serialized data can be tampered with, leading to security vulnerabilities.
* Limited language compatibility: Serialized objects may not be compatible with other programming languages or platforms.
* Version control complexity: Managing version control and compatibility between serialized objects and class definitions can be complex.
* Debugging and maintenance challenges: Serialized objects can be difficult to debug and maintain directly.
* Object graph limitations: Serialization of complex object graphs can lead to issues like cyclic dependencies or excessive memory usage.
* Resource management: Serialized objects may hold references to resources, requiring careful resource management during serialization and deserialization.

**Que: What is Externalization?**

**Ans:** Externalization is used to customize the serialization method.

Externalization in Java refers to a mechanism that allows you to control the serialization and deserialization process of objects by providing custom logic for writing and reading object data. Unlike the default serialization process, where Java handles the serialization and deserialization automatically based on the object's internal state, externalization gives you more control over how object data is represented and stored.

The java.io.Externalizable interface is used for implementing externalization in Java. It extends the Serializable interface but requires you to implement two methods: writeExternal() for writing object data to a stream and readExternal() for reading object data from a stream. This allows you to define custom serialization and deserialization logic for your objects.

**Key Points about Externalization:**

**1. Custom Serialization Logic:** Externalization allows you to define custom logic for writing object data to a stream (writeExternal()) and reading object data from a stream (readExternal()).

**2. Control over Data Representation**: With externalization, you have control over how object data is represented in the serialized form. You can write only specific fields, exclude transient fields, or customize the serialization format.

**3. Performance Considerations:** Externalization can offer better performance compared to default serialization in certain scenarios, especially when dealing with large objects or complex serialization requirements.

**4. Transient Fields Handling:** Externalization gives you the ability to handle transient fields explicitly during serialization and deserialization, as they are not automatically serialized by default.

**5. Versioning and Compatibility:** Similar to default serialization, versioning and compatibility considerations are important when using externalization to ensure proper serialization and deserialization of objects across different versions of your application.

**Que: What is difference between Serialization and Externalization?**

**Ans:** Serialization and externalization are both mechanisms used in Java for object persistence, but they differ in how they handle the serialization and deserialization process and the level of control they provide to developers.

**Serialization:**

* It is a marker interface it doesn’t have any method.
* Serialization in Java refers to the process of converting an object into a stream of bytes, which can then be stored, transmitted, or persisted.
* The default serialization mechanism in Java handles the serialization and deserialization process automatically based on the object's internal state.
* Objects that need to be serialized implement the Serializable interface, and Java provides default serialization behavior unless overridden.
* Serialization includes the object's state along with metadata such as class information, which enables Java to reconstruct the object during deserialization.
* Developers can control serialization behavior through techniques like marking fields as transient, providing custom serialization logic using readObject() and writeObject() methods, or implementing Externalizable for full control over serialization/deserialization.

**Externalization:**

* It’s not a marker interface. It has method’s called writeExternal() and readExternal().
* Externalization is a specific type of serialization in Java that allows developers to have more control over the serialization and deserialization process.
* Objects that need to be externalized implement the Externalizable interface, which extends Serializable but requires implementing custom serialization (writeExternal()) and deserialization (readExternal()) logic.
* Unlike default serialization, externalization requires developers to explicitly define how object data is written to the output stream (writeExternal()) and read from the input stream (readExternal()).
* Externalization allows for finer control over data representation, exclusion of transient fields, custom serialization formats, and performance optimizations compared to default serialization in certain scenarios.
* However, externalization requires more effort from developers to implement custom serialization logic, and it is less automatic compared to default serialization.



**Interview Question on Multi-Threading in Java**

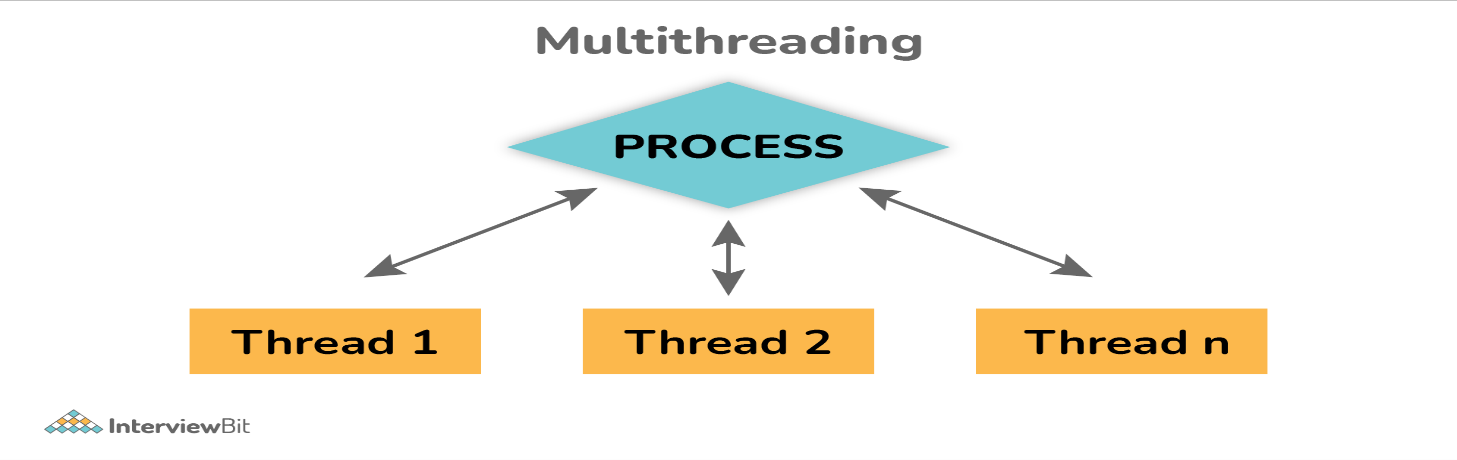
**Que: What do you meant by Multithreading in java?**

**Ans**: Multithreading refers to the ability of a program or an operating system to execute multiple threads concurrently within a single process.

As the name suggests**, it is the ability of a CPU to execute multiple threads independently at the same time but share the process resources simultaneously.** Its main purpose is to provide simultaneous execution of multiple threads to utilize the CPU time as much as possible.

**It is a Java feature where one can subdivide the specific program into two or more threads to make the execution of the program fast and easy.**

Multithreading allows a program to perform multiple tasks simultaneously, improving performance, responsiveness, and resource utilization.



**Que: What are the features of MultiThreading in java?**

**Ans:** Here are some key Features of multi-threading:

**1. Concurrency:**

* Multithreading enables concurrent execution of tasks within a program. Multiple threads can execute different parts of the code concurrently, allowing for efficient use of CPU resources and better overall system performance.

**2. Responsiveness:**

* Multithreading improves responsiveness in applications by allowing tasks to run in the background while the main thread continues to handle user interactions or other critical operations.

**3. Parallelism:**

* Multithreading facilitates parallelism, where tasks can be divided into smaller subtasks that can run concurrently on multiple CPU cores. This can significantly reduce the time required to complete complex computations or process large amounts of data.

**4. Resource Utilization:**

* Multithreading helps in better utilization of system resources such as CPU time, memory, and I/O operations. By utilizing idle CPU cycles and overlapping I/O operations with computation, multithreaded applications can achieve higher efficiency.

**5. Scalability:**

* Multithreading contributes to the scalability of applications, allowing them to handle increasing workloads and accommodate more users without sacrificing performance.

**6. Asynchronous Programming:**

* Multithreading is essential for implementing asynchronous programming models, such as event-driven or non-blocking I/O, where tasks can execute independently and asynchronously without blocking the main thread.

**Que: What are the benefits of using Multithreading?**

Ans: There are various benefits of multithreading as given below:

* Allows program to run continuously even if a part of it is blocked.
* Improve performance as compared to traditional parallel programs that use multiple processes.
* Allows to write effective programs that utilize maximum CPU time.
* Improves the responsiveness of complex applications or programs.
* Increase use of CPU resources and reduce costs of maintenance.
* Saves time and parallelism tasks.
* If an exception occurs in a single thread, it will not affect other threads as threads are independent.
* Less resource-intensive than executing multiple processes at the same time.

**Que: What is Thread in Java?**

**Ans:** Threads are basically the lightweight and smallest unit of processing that can be managed independently by a scheduler. Threads are referred to as parts of a process that simply let a program execute efficiently with other parts or threads of the process at the same time.

Using threads, one can perform complicated tasks in the easiest way. It is considered the simplest way to take advantage of multiple CPUs available in a machine. They share the common address space and are independent of each other.

**Que: What are different ways of implementing thread in Java?**

**Ans**: In Java, there are several ways to implement threads, each offering different levels of control and flexibility over thread creation and management. Here are the main ways of implementing threads in Java:

**1. Extending the Thread Class:**

One way to create a thread is by extending the Thread class and overriding its run() method to define the thread's behavior.

**class MyThread extends Thread {**

**public void run() {**

**// Thread behavior/logic here**

**}**

**}**

**// Creating and starting a thread**

**MyThread thread = new MyThread();**

**thread.start();**

**2. Implementing the Runnable Interface:**

Another approach is to implement the Runnable interface and pass an instance of the implementing class to a Thread object's constructor.

This approach is preferred because it allows you to extend other classes as well since Java supports single inheritance.

**class MyRunnable implements Runnable {**

**public void run() {**

**// Thread behavior/logic here**

**}**

**}**

**// Creating a thread using Runnable interface**

**Thread thread = new Thread(new MyRunnable());**

**thread.start();Using Lambda Expressions (Java 8 and later):**

With Java 8 and later versions, you can use lambda expressions to create threads more concisely, especially when implementing simple tasks.

**3. Using Lambda Expressions (Java 8 and later):**

With Java 8 and later versions, you can use lambda expressions to create threads more concisely, especially when implementing simple tasks.

***// Creating a thread using lambda expression***

***Thread thread = new Thread(() -> {***

***// Thread behavior/logic here***

***});***

***thread.start();***

**4. Using Executor Framework:**

Java provides the Executor framework in the java.util.concurrent package for managing thread execution.

You can create threads using ExecutorService and submit tasks for execution.

***ExecutorService executor = Executors.newFixedThreadPool(5);***

***executor.submit(() -> {***

***// Thread behavior/logic here***

***});***

***executor.shutdown();***

**5. Using Callable and Future:**

Similar to Runnable, you can implement the Callable interface and submit tasks to an ExecutorService using Future objects to get the result or handle exceptions.

***Callable<Integer> task = () -> {***

***// Thread behavior/logic here***

***return result;***

***};***

***ExecutorService executor = Executors.newSingleThreadExecutor();***

***Future<Integer> future = executor.submit(task);***

***try {***

***Integer result = future.get(); // Get the result***

***} catch (InterruptedException | ExecutionException e) {***

***// Handle exceptions***

***}***

***executor.shutdown();***

**Que: What are Runnable and Callable?**

**Ans:** Both Runnable and Callable are interfaces in Java that are used for defining tasks or operations that can be executed asynchronously in a multi-threaded environment. However, they have some key differences in terms of their purpose, return values, and exception handling:

**Runnable Interface:**

* **Purpose**: The Runnable interface is designed for tasks that do not return a result or throw checked exceptions. It represents a task that can be executed by a thread.
* **Method**: The Runnable interface defines a single method called run(), which does not take any arguments and does not return a value.
* **Execution**: A Runnable task is executed by passing it to a Thread object or an executor service's execute() method.

***public class MyRunnable implements Runnable {***

***public void run() {***

***// Task implementation***

***}***

***}***

**Callable Interface:**

* **Purpose**: The Callable interface is designed for tasks that return a result and can throw checked exceptions. It represents a task that produces a result when executed.
* **Method**: The Callable interface defines a single method called call(), which returns a result of type V and can throw Exception.
* **Execution**: A Callable task is typically executed by submitting it to an executor service's submit() method, which returns a Future object representing the task's result.

***public class MyCallable implements Callable<Integer> {***

***public Integer call() throws Exception {***

***// Task implementation***

***return 42;***

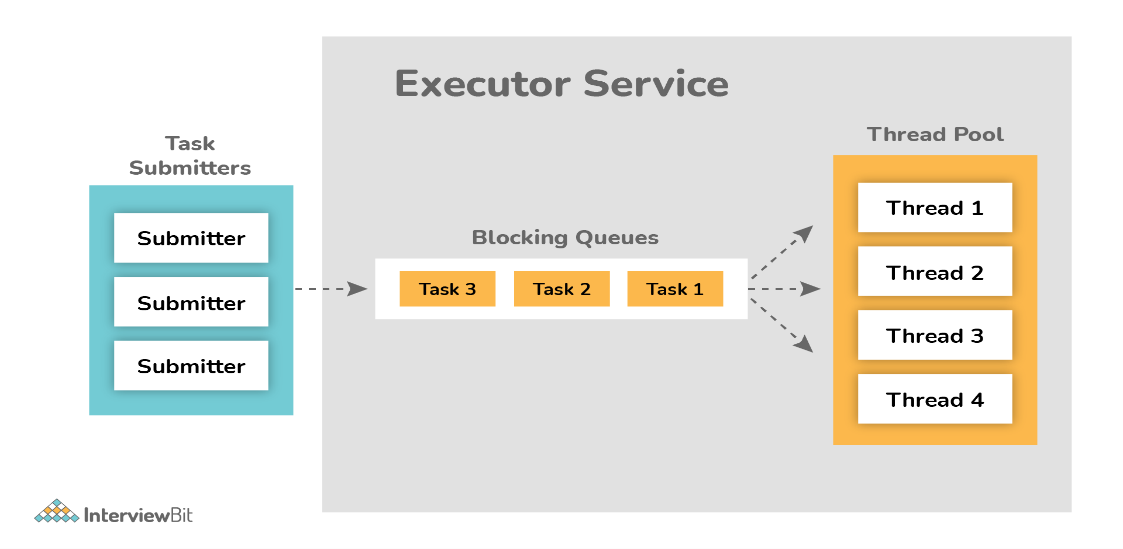
***}***

***}***

**Que: What is the ExecutorService interface?**

**Ans**: The ExecutorService interface in Java is part of the java.util.concurrent package and provides a higher-level API for managing and executing concurrent tasks using a pool of threads. It extends the Executor interface and adds functionalities for task submission, execution, and termination.

ExecutorService interface is basically a sub-interface of Executor interface with some additional methods or features that help in managing and controlling the execution of threads. It enables us to execute tasks asynchronously on threads.



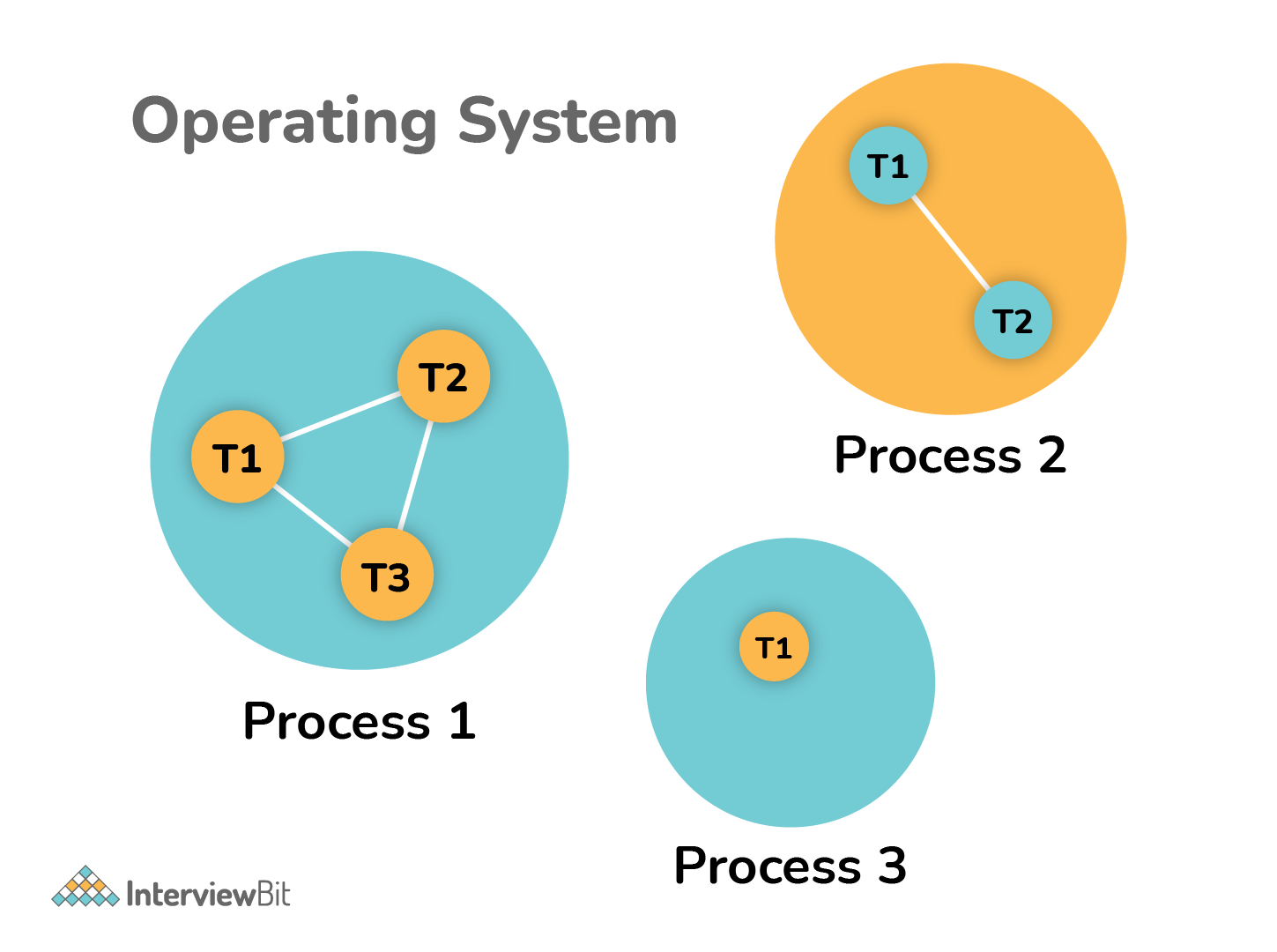
**Que: What is Future Interface?**

**Ans:** In Java, the Future interface represents the result of an asynchronous computation or task that may not have completed yet. It allows you to retrieve the result of a task asynchronously, handle exceptions, and check the status of the task's completion. The Future interface is commonly used in conjunction with asynchronous programming, concurrency, and the ExecutorService framework.

**Que: What's the difference between thread and process?**

**Ans:**

* Thread: It simply refers to the smallest units of the particular process. It has the ability to execute different parts (referred to as thread) of the program at the same time.
* Process: It simply refers to a program that is in execution i.e., an active program. A process can be handled using PCB (Process Control Block).



**Que: What's the difference between User thread and Daemon thread?**

**Ans:**

* **User Thread (Non-Daemon Thread)**: In Java, user threads have a specific life cycle and its life is independent of any other thread. JVM (Java Virtual Machine) waits for any of the user threads to complete its tasks before terminating it. When user threads are finished, JVM terminates the whole program along with associated daemon threads.
* **Daemon Thread**: In Java, daemon threads are basically referred to as a service provider that provides services and support to user threads. There are basically two methods available in thread class for daemon thread: setDaemon() and isDaemon().

**Que: What is Demon Thread?**

**Ans:** A daemon thread in Java is a special type of thread that runs in the background, providing services to other threads or performing tasks that are not critical to the application's functionality.

Deomon Thread is a background thread with least priority.

* JVM does not wait for daemon threads to finish their tasks before termination.
* These threads are normally created by JVM. They are also used internally by the JVM for system-level operations like garbage collection, finalization, and other maintenance activities.

e.g.: Garbage collector is a daemon thread.

MS word autosave option.

**Que: How can we create daemon threads?**

**Ans**: For creating a demon thread we follow the same process as createing the normal(user thread).

* We can create daemon threads in java using the thread class setDaemon(true). setDaemon(true): It is used to mark the current thread as daemon thread or user thread.
* isDaemon(): This method is generally used to check whether the current thread is daemon or not. If the thread is a daemon, it will return true otherwise it returns false.

***class MyDaemonThread extends Thread {***

***public void run() {***

***// Daemon thread behavior/logic here***

***}***

***}***

***// Creating and starting a daemon thread***

***MyDaemonThread daemonThread = new MyDaemonThread();***

***daemonThread.setDaemon(true); // Mark as daemon thread***

***daemonThread.start();***

**Que: What is the synchronization process? Why use it?**

**Ans:** Synchronization in Java refers to the process of controlling access to shared resources or critical sections of code by multiple threads to ensure thread safety and prevent data corruption or race conditions. The main purpose of synchronization is to establish mutual exclusion, which allows only one thread at a time to access or modify shared resources.

* **Critical Sections:** Critical sections refer to parts of code where shared resources (variables, data structures, etc.) are accessed or modified by multiple threads.
* **Locks**: Synchronization is typically achieved using locks. A lock is like a key that allows only one thread to enter a critical section at a time. If a thread holds the lock, other threads must wait until the lock is released before they can enter the critical section.
* **Mutual Exclusion**: Mutual exclusion ensures that only one thread can execute a critical section at any given time. This prevents concurrent access issues, race conditions, and data inconsistencies.

**Que: What is the lock interface? Why is it better to use a lock interface rather than a synchronized block.?**

**Ans:** The Lock interface defines methods for acquiring and releasing locks, which are used to achieve mutual exclusion and prevent multiple threads from accessing shared resources simultaneously.

Key methods of the Lock interface include lock() to acquire the lock, unlock() to release the lock, and tryLock() to attempt to acquire the lock without blocking.

**Que: What is Lifecycle of a Thread?**

**Ans**: The lifecycle of a thread in Java involves several states and transitions, each representing a different stage of the thread's execution.

**1. New State:**

When a thread object is created using the new keyword or by instantiating a subclass of Thread, it is in the "New" state. At this stage, the thread has been created but has not yet started its execution.

**2. Runnable State:**

After calling the start() method on a thread object, it transitions to the "Runnable" state. In this state, the thread is ready to run, but it may not be currently executing if the CPU is busy with other threads.

Threads in the "Runnable" state can be further categorized based on their execution status:

* **Running**: The thread is actively executing its code and using CPU resources.
* **Ready**: The thread is ready to run but waiting for CPU time to be allocated.
* **Waiting**: The thread is waiting for a notification, I/O operation, or time delay.

**3. Sleep State:**

* Threads can enter the "Sleep" state using the Thread.sleep() method or other sleep-related mechanisms. When a thread is in the sleep state, it temporarily suspends its execution for a specified period of time.
* While in the sleep state, the thread is not actively using CPU resources, allowing other threads to run concurrently.



**4. Blocked State:**

* Threads can enter the "Blocked" state when they are waiting for a resource that is currently held by another thread. This can occur in synchronized blocks or methods when a thread attempts to acquire a lock held by another thread.
* For example, if a thread attempts to enter a synchronized block but the lock is held by another thread, it will be blocked until the lock is released by the other thread.

**5. Terminated State:**

* Threads transition to the "Terminated" state when their run() method completes execution or when they are explicitly stopped using methods like Thread.stop() (deprecated and not recommended).
* Once terminated, a thread cannot be restarted or resumed. It is no longer eligible for execution and its resources are released by the JVM during garbage collection.

**Que: What are the wait() and sleep() methods?**

**Ans:**

**wait() Method:**

* It causes the current thread to wait and go to sleep until some other threads call the notify () or notifyAll() method for the object’s monitor (lock).
* wait() is a method defined in the Object class and is used for inter-thread communication and synchronization.
* It is typically used in conjunction with the notify() and notifyAll() methods to implement thread coordination and signaling mechanisms.
* When a thread calls wait(), it releases the lock on the object it is synchronized on, allowing other threads to acquire the lock and execute synchronized code.

**sleep() Method:**

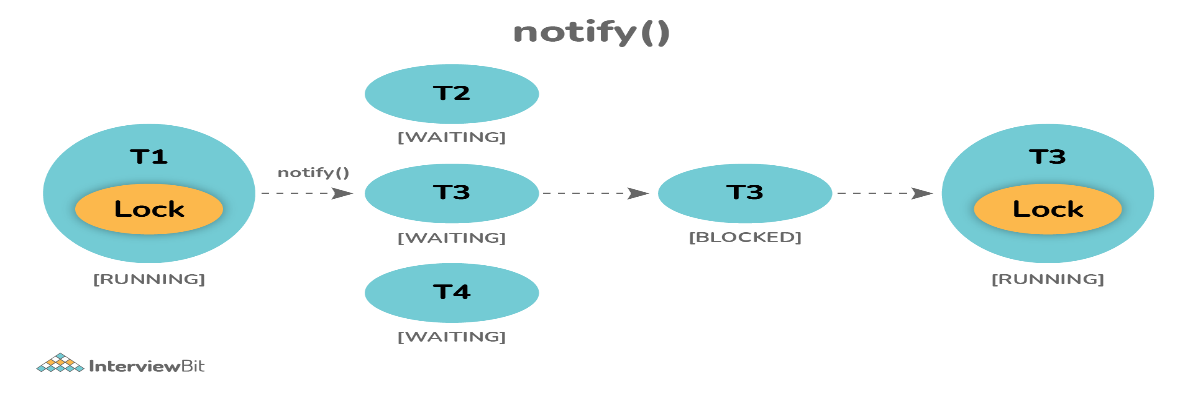
* As the name suggests, it is a static method that pauses or stops the execution of the current thread for some specified period.
* It doesn’t release the lock while waiting and is mostly used to introduce pause on execution. It is defined in thread class, and no need to call from a synchronized context.
* Unlike wait(), sleep() does not require synchronization and can be called from any context, including non-synchronized blocks or methods.

***Thread.sleep(1000); // Sleeps the thread for 1 second***

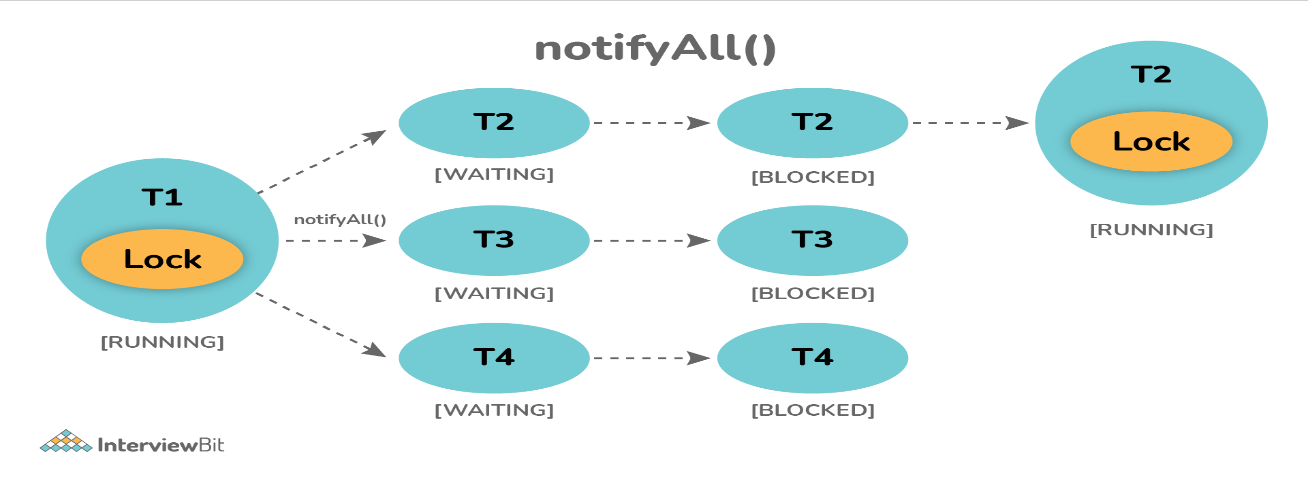
**Que: What’s the difference between notify() and notifyAll()?**

**Ans**:

**notify**(): It sends a notification and wakes up only a single thread instead of multiple threads that are waiting on the object’s monitor.



**notifyAll**(): It sends notifications and wakes up all threads and allows them to compete for the object's monitor instead of a single thread.



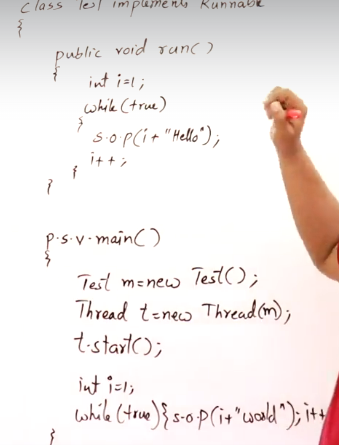
**Que: What is the start() and run() method of Thread class?**

**Ans:**

**start():** In simple words, the start() method is used to start or begin the execution of a newly created thread. When the start() method is called, a new thread is created and this newly created thread executes the task that is kept in the run() method. One can call the start() method only once.

**run()**: In simple words, the run() method is used to start or begin the execution of the same thread. When the run() method is called, no new thread is created as in the case of the start() method. This method is executed by the current thread. One can call the run() method multiple times.

* Whenever you extends a class from thread class and you want to achieve multi-threading then you must override Run() Method.
* whatever logic you want just write is public void run().
* Run() is the starting point of the thread.
* In main method, create an object of myThread class🡪 Now the thread is created.
* Then we have to class start() method then only we can start the thread.
* Start() is the built-in method in thread class.
* These two things will be running simultaneously.



**Que: What will happen if we don’t override the thread class run() method?**

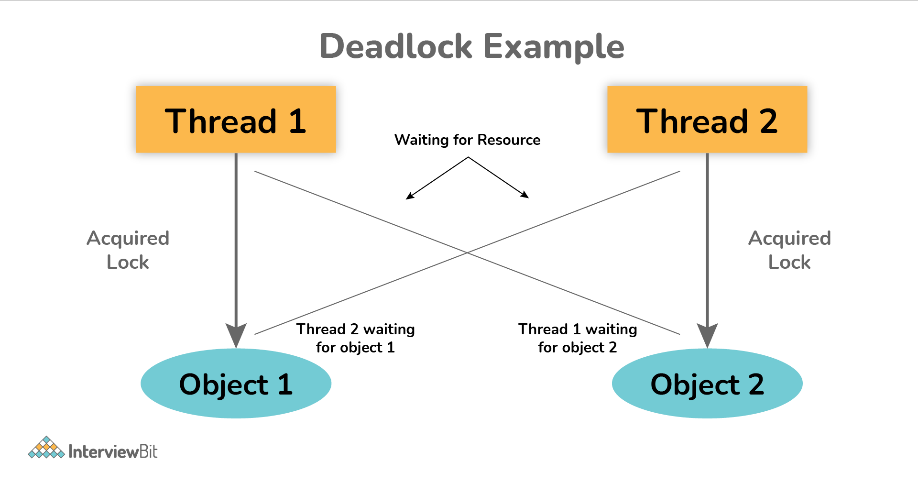
**Ans:** Nothing will happen as such if we don’t override the run() method. The compiler will not show any error. It will execute the run() method of thread class and we will just don’t get any output because the run() method is with an empty implementation.

**Que: Is it possible to call the run() method directly to start a new thread?**

**Ans**: No, it's not possible at all. You need to call the start method to create a new thread otherwise run method won't create a new thread. Instead, it will execute in the current thread.

**Que: Explain the meaning of the deadlock and when it can occur?**

**Ans**: Deadlock, as the name suggests, is a situation where multiple threads are blocked forever. It generally occurs when multiple threads hold locks on different resources and are waiting for other resources to complete their task.



**Que: Explain thread pool?**

**Ans:** A thread pool in Java is a collection of pre-initialized threads that are ready to execute tasks.

Main function is to :

1. Thread pools manage thread creation, destruction, and recycling, reducing overhead and resource consumption.

2. By reusing threads, thread pools can handle multiple tasks efficiently and achieve better throughput.

3. Some thread pools support task prioritization, allowing critical tasks to be executed with higher priority.

**Que: Explain volatile variables in Java?**

**Ans:** In Java, the volatile keyword is used to indicate that a variable's value may be modified by different threads.

* The primary purpose of declaring a variable as volatile is to ensure its visibility across threads. When a thread writes to a volatile variable, the updated value is immediately visible to other threads reading the same variable.
* This keyword cannot be used with classes and methods, instead can be used with variables.

**Que: How do threads communicate with each other?**

**Que: What do you mean by inter-thread communication?**

**Ans:** Inter-thread communication is a process or mechanism using which multiple threads can communicate with each other. Threads can communicate using three methods i.e., wait(), notify(), and notifyAll().

**Que: What is atomic variable?**

**Ans:** An atomic variable in Java refers to a type of variable that supports atomic operations, meaning that operations on the variable are guaranteed to be executed atomically.

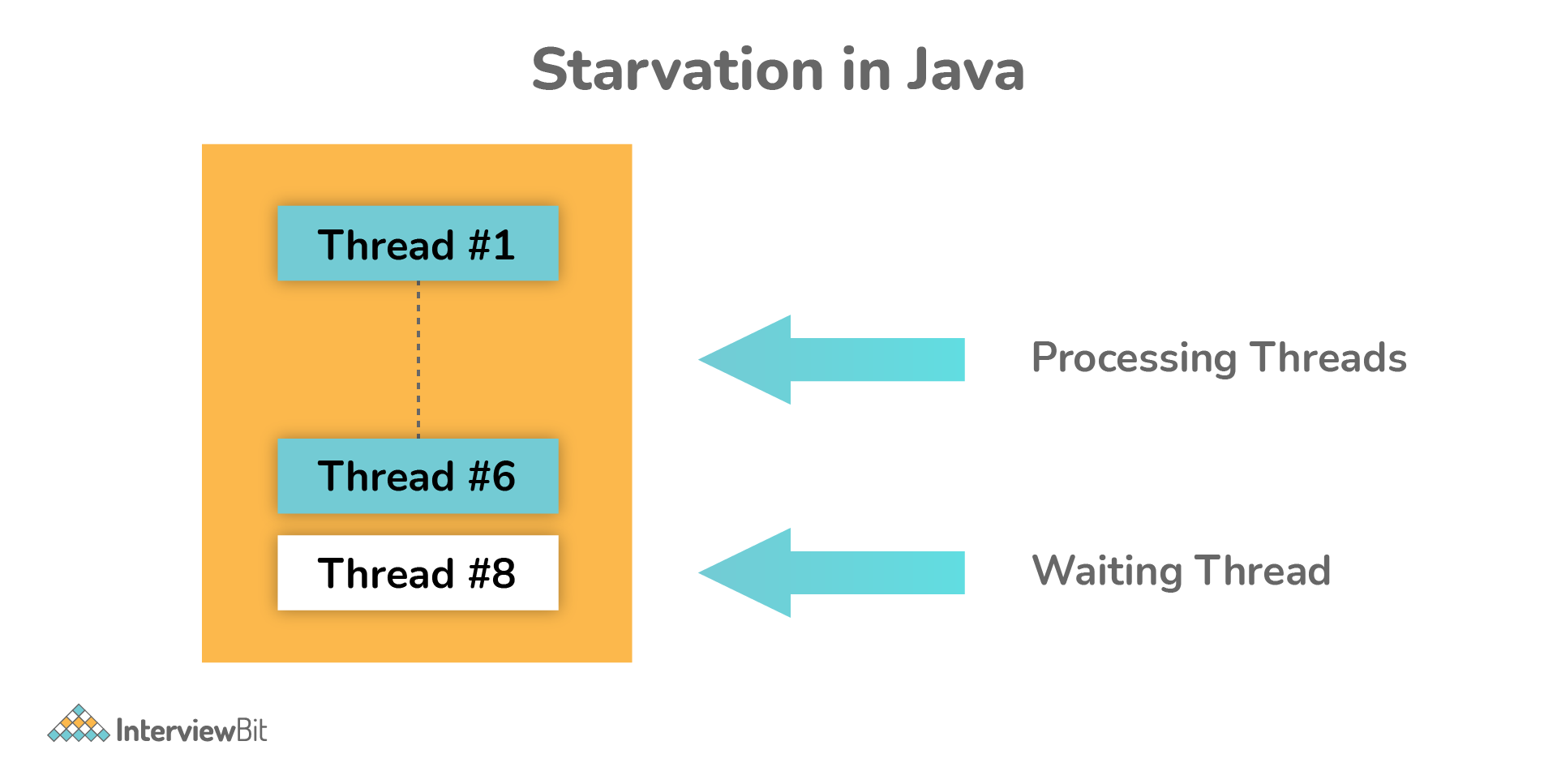
* Atomicity ensures that the operation is indivisible and not subject to interference from other threads, making it safe for concurrent access by multiple threads without explicit synchronization.
* "Atomically" refers to an operation that is performed as a single, indivisible unit. It means that the operation either completes entirely or does not execute at all, and no other thread can interrupt or observe the operation in an incomplete or partially executed state.
* Atomically mean ki eek thread jab ispr kam krega toh pura khatam krega ya toh karega hi nahi ekdm.
* Atomic variables support atomic operations such as reading, writing, and updating the variable's value in a way that appears to other threads as if it occurred instantaneously and without interruption.
* Atomic variables are suitable for scenarios where multiple threads access shared variables concurrently and atomicity is required.

**Que: What is thread starvation?**

**Ans:**

Thread starvation is basically a situation or condition where a thread won’t be able to have regular access to shared resources and therefore is unable to proceed or make progress. This is because other threads have high priority and occupy the resources for too long. This usually happens with low-priority threads that do not get CPU for its execution to carry on.

Starvation: Suppose we have multiple thread out of which one thread has highest priority, so it may happen that the thread will hold the CPU for longer time and the other thread will not get the CPU, This is starvation.



**Que: Explain context switching.**

**Ans:** Context switching is basically an important feature of multithreading. It is referred to as switching of CPU from one thread or process to another one. It allows multiple processes to share the same CPU. In context switching, the state of thread or process is stored so that the execution of the thread can be resumed later if required.

**Que: Explain thread priority.**

**Ans**: Thread priority simply means that threads with the highest priority will get a chance for execution prior to low-priority threads. One can specify the priority but it's not necessary that the highest priority thread will get executed before the lower-priority thread. Thread scheduler assigns processor to thread on the basis of thread priority. The range of priority changes between 1-10 from lowest priority to highest priority.

