BHADRINATH P S **24-NOV-2024** **INSURANCE**

JAVA OOPS & COLLECTION CONCEPTS

---

1. In java, how many child class can be created for a super class?

In Java, there is no strict limit on the number of child classes (subclasses) that can be created for a superclass. You can create as many child classes as you want, as long as they follow the rules of inheritance.

---

2. Difference between collection class and collection interface. Why do we need this to execute?

In Java, the **Collection** interface and the **Collection classes** are both part of the Java Collections Framework, but they serve different purposes. Let's explore the key differences between them and why they are important for working with collections in Java.

1. Collection Interface:

- **Definition**: The `Collection` interface is the root interface of the Java Collections Framework. It represents a group of objects, known as elements, and provides a set of basic methods that all collection classes must implement.
- **Purpose**: It provides a general contract for collections (such as lists, sets, and queues) but does not define how the data is stored or how the collection behaves (e.g., ordering, duplication).
- **Common Methods**:
  - `add()`: Adds an element to the collection.
  - `remove()`: Removes an element from the collection.
  - `size()`: Returns the number of elements in the collection.
  - `isEmpty()`: Checks if the collection is empty.
  - `contains()`: Checks if a specific element exists in the collection.
  - `clear()`: Removes all elements from the collection.

2. Collection Classes:

- **Definition**: Collection classes are concrete implementations of the `Collection` interface or its sub interfaces. They provide specific implementations of the general methods defined by the `Collection` interface.

- **Purpose**: These classes provide the actual behavior (such as how the collection is stored and how it operates) and are used to work with the collection of objects.
- **Examples of Collection Classes**:
    - `ArrayList`: A resizable array implementation of the `List` interface.
    - `HashSet`: A set implementation that uses a hash table for storage.
    - `LinkedList`: A doubly linked list implementation of the `List` and Deque interfaces.
    - `TreeSet`: A `Set` implementation that uses a red-black tree.
    - `PriorityQueue`: A queue implementation based on a priority heap.

---

3. Why Map (HashMap) is not coming under collection framework?

The reason **`Map`** (such as **`HashMap`**) is **not** part of the **Java Collection Framework** is because it operates on a **fundamentally different data structure** than the collections found in the `Collection` hierarchy (such as `List`, `Set`, or Queue).

The **`Map`** (e.g., **`HashMap`**) is **not part of the Collection Framework** because it handles **key-value pairs** (a different concept from a collection of individual elements). This design decision keeps the Map interface separate from the `Collection` interface, reflecting the conceptual difference between storing individual objects and associating one object (key) with another (value). The Map still plays a crucial role in the overall Java Collections Framework, but it is treated as a separate abstraction that complements the other data structures.

---

4. What is priority queue? Why priority queue? Difference between queue and priority? When? Why? Where to use

Aspect	Queue	Priority Queue
Order of Elements	FIFO (First-In-First-Out)	Elements are ordered based on their priority.
Processing Order	The first element added is the first to be removed.	The element with the highest (or lowest) priority is removed first, regardless of insertion order.
Implementation	Typically implemented as a **LinkedList**, **ArrayDeque**, or **CircularBuffer**.	Typically implemented using a **heap** (binary heap) but may also use other data structures.

Usage	General-purpose queue for normal task scheduling.	Task scheduling with priority, network packet handling, Dijkstra's algorithm, etc.
Example	`Queue<Integer> q = new LinkedList<>();`	`PriorityQueue<Integer> pq = new PriorityQueue<>();`
Time Complexity (Enqueue/Dequeue)	O(1) for enqueue, O(1) for dequeue (in basic implementations)	O(log n) for both enqueue and dequeue due to heap operations.

5. What and why deque in java?

A **Deque** (short for **"Double-Ended Queue"**) is a linear collection that allows elements to be added or removed from both ends—**the front (head)** and **the back (tail)**. In other words, it is a more general type of **Queue** that supports insertion and deletion of elements at both ends, rather than only one end (as in a regular queue).

- **Insertions and Deletions at Both Ends**: You can add and remove elements from both the front (head) and back (tail) of the deque.
- **Supports FIFO and LIFO**: The Deque can be used as a **FIFO (First-In-First-Out)** queue or a **LIFO (Last-In-First-Out)** stack.
- **No Capacity Restrictions** (with `ArrayDeque`): `ArrayDeque` grows dynamically and doesn't have a fixed size like an array-based queue (e.g., `Queue` or `LinkedList`).

---

6. What is Thread Safety?

Thread safety is a concept in concurrent programming that refers to the ability of a piece of code, data structure, or algorithm to function correctly when multiple threads access it simultaneously. In other words, a **thread-safe** component ensures that it behaves as expected when multiple threads are executing and interacting with it concurrently, without causing data corruption, race conditions, or unexpected behavior.

---

7. Thread & Multi-Threading in Java

A **thread** is the smallest unit of execution within a process. A process can have multiple threads, and each thread has its own execution path. Threads within a process share the same memory space (heap), but each thread has its own stack memory, program counter, and local variables.

In Java, a **thread** can be created and run using two primary methods:

1. **Extending the `Thread` class**
2. **Implementing the `Runnable` interface**

A. Extending the `Thread` class

You can create a thread by subclassing the `Thread` class and overriding its `run()` method, which defines the code that will be executed by the thread.

B. Implementing the `Runnable` Interface

Alternatively, you can create a thread by implementing the `Runnable` interface. This approach allows you to define the `run()` method in a class that is not a subclass of `Thread`. This is a more flexible approach, especially if the class already extends another class.

Multi-threading in Java allows multiple threads to run concurrently within a program, enabling tasks to be performed in parallel. This can improve the performance of programs that have tasks that can be executed independently, such as handling multiple user requests in a server or performing parallel computations.

Java provides built-in support for multi-threading through the **`Thread`** class and the **`Runnable`** interface. This allows developers to execute multiple threads concurrently, improving efficiency in tasks like I/O operations, user interface management, and parallel computing.

---

8. Data Structures in Java Similar to Dictionary

`Map` Interface:

The `Map` interface in Java represents a collection of key-value pairs (like a dictionary or associative array in other languages).
The keys are unique, and each key maps to exactly one value.
Some common implementations of the `Map` interface in Java include:

i. `HashMap`
ii. `TreeMap`
iii. `LinkedHashMap`
iv. `Hashtable` (an older class, now largely replaced by `HashMap`).

---

9. Difference Between HashMap, HastSet, TreeSet, TreeMap.

Feature	HashMap	HashSet	TreeSet	TreeMap
Type	Key-Value Pair Collection (Map)	Collection (Set)	Collection (Set)	Key-Value Pair Collection (Map)
Implements	`Map<K, V>`	`Set<E>`	`NavigableSet<E>`	`Map<K, V>`
Order	No specific order (unordered)	No specific order (unordered)	Sorted (based on natural ordering or comparator)	Sorted by key (based on natural ordering or comparator)
Duplicates	Allows only unique keys, values can be duplicates	Does not allow duplicates (unique elements only)	Does not allow duplicates (unique elements only)	Allows only unique keys, values can be duplicates
Null Values	Allows one `null` key and multiple `null` values	Allows `null` elements	Does not allow `null` elements	Does not allow `null` key or value
Thread-Safety	Not synchronized	Not synchronized	Not synchronized	Not synchronized
Use Case	Stores key-value pairs, fast access to values based on keys	Stores unique elements, fast membership testing	Stores unique elements in sorted order	Stores key-value pairs in sorted order, fast access based on keys

10. Difference Between Throw and throws.

1. `throw`:

- **Purpose**: `throw` is used to **explicitly throw an exception** from a method or a block of code.
- **How It Works**: When you use `throw`, you're creating an instance of an exception and throwing it to be handled by an appropriate `catch` block or, if not caught, to propagate it up the call stack.
- **Usage**: It is followed by an instance of an exception (either a built-in exception or a custom exception).
- **Can Throw**: Can throw **checked** and **unchecked** exceptions.

2. `throws`:

- **Purpose**: `throws` is used in a **method signature** to **declare that a method can throw an exception**. It does not throw an exception itself, but it informs the calling code that the method might throw one or more exceptions, which must be handled.
- **How It Works**: By declaring `throws` in a method's signature, you are telling the compiler that this method might throw specific checked exceptions, and it must be handled either by catching the exception or by declaring it to be thrown by the calling method.
- **Can Only Declare**: `throws` can only be used with **checked exceptions.**
- **Multiple Exceptions**: You can declare multiple exceptions separated by a comma.

---

11. 15(Fifteen) Interface Names

- **`Runnable`**: For defining tasks to be executed by a thread.
- **`Comparable`**: For comparing objects of the same type (e.g., sorting).
- **`Cloneable`**: For creating a copy of an object.
- **`Serializable`**: For making objects serializable.
- **`Iterable`**: For enabling iteration over a collection.
- **`Iterator`**: For traversing elements of a collection.
- **`Map`**: For key-value pairs.
- **`List`**: For ordered collections.
- **`Set`**: For collections without duplicates.
- **`Queue`**: For a FIFO (First-In-First-Out) collection.
- **`Deque`**: For a collection supporting both ends insertion/removal.
- **`Comparator`**: For custom sorting/comparison logic.
- **`Function`**: For functions that take one input and return a result.
- **`Predicate`**: For Boolean operations on a single argument.
- **`Supplier`**: For supplying results without taking any input.
