Contents

Queue in Java

Queue is an interface in Java's java.util package that represents a linear data structure following the First-In-First-Out (FIFO) principle. This means the first element added to the queue is the first one to be removed.

Key Operations:

add(E e): Adds an element to the end of the queue.
offer(E e): Adds an element to the end of the queue, but returns false if the addition fails (e.g., due to limited capacity).
remove(): Removes and returns the element at the front of the queue. If the queue is empty, it throws a NoSuchElementException.
poll(): Removes and returns the element at the front of the queue, or null if the queue is empty.
element(): Returns the element at the front of the queue without removing it. If the queue is empty, it throws a NoSuchElementException.
peek(): Returns the element at the front of the queue without removing it, or null if the queue is empty.

Common Implementations:

ArrayDeque: A resizable array implementation of a deque (double ended queue) that can also be used as a queue.
LinkedList: A linked list implementation of a queue.

Example:

```
import java.util.Queue;
import java.util.LinkedList;

public class QueueExample {
    public static void main(String[] args) {
        Queue<String> queue
```

```
 = new LinkedList<>();

        queue.add("Apple");
        queue.add("Banana");
        queue.add("Cherry");

        System.out.println("Queue: " + queue);

        String
 removed = queue.remove();
        System.out.println("Removed: " + removed);

        System.out.println("Queue: " + queue);
    }
}
```

Gfg Links for Further Reference:

Queue Interface: https://www.geeksforgeeks.org/queue-data-structure/
ArrayDeque Class:
https://www.geeksforgeeks.org/java-util-arraydeque-class-java/
LinkedList Class: https://www.geeksforgeeks.org/linked-list-in-java/

Singleton Design Pattern

Singleton pattern is a design pattern which restricts a class to instantiate its multiple objects. It is nothing but a way of defining a class. Class is defined in such a way that only one instance of the class is created in the complete execution of a program or project. It is used where only a single instance of a class is required to control the action throughout the execution.

Challenge 1 : We have a Person class. Our task is to make sure that when we create objects of Person class, the constructor gets called only once, i.e. all objects have the same hashcode.

Challenge 2 : Why do we need Singleton classes ?
(**HINT** : Driver Class for making connections to the database .)

Challenge 3 : Why does the newly created getPerson function in Person need to be static ?

References :
https://www.geeksforgeeks.org/singleton-design-pattern-introduction/
https://www.geeksforgeeks.org/singleton-design-pattern/
https://www.geeksforgeeks.org/java-singleton-design-pattern-practices-examples/

Comparable vs Comparator

Sorting is a fundamental operation in programming, and both `Comparable` and `Comparator` interfaces help us achieve this in Java. However, they serve different purposes:

Comparable

1. Defines a single natural ordering for a class.
2. Implemented by the class itself.
3. Provides the `compareTo(T obj)` method that defines how objects of the class should be compared.
4. Allows using built-in sorting methods like `Arrays.sort()` and `Collections.sort()`.
5. Example: `String`, `Integer`, and `Date` classes all implement `Comparable` for natural ordering.

Link:
https://www.geeksforgeeks.org/why-to-use-comparator-interface-rather-than-comparable-interface-in-java/

Comparator

1. Defines custom sorting logic, independent of the class being sorted.
2. Implemented as a separate class.
3. Provides the `compare(T o1, T o2)` method that defines how two objects should be compared based on specific criteria.
4. Offers more flexibility for sorting based on different attributes or in different orders.

5. Example: You might create a `Comparator` to sort `Employee` objects by salary in descending order.

Link: https://www.geeksforgeeks.org/why-to-use-comparator-interface-rather-than-comparable-interface-in-java/

Key Differences:

1. Sorting Order: `Comparable` defines a single order, while `Comparator` allows for multiple sorting criteria.
2. Class Modification: Using `Comparable` modifies the original class. `Comparator` offers independent sorting logic.
3. Use Cases: Use `Comparable` when you want a natural ordering for a class. Use `Comparator` for custom sorting needs or when you don't want to modify the original class.