23. Collections in Java - Part 2 | Comparator vs Comparable | Priority-Queue.

Queue

* Queue is an interface, child of Collection Interface

Methods	Usage
add()	* Insert element into the queue. * True if Insertion is Sucessful and Exception if insertion Fails * Null element not allowed for insertion, will throw NPE
offer()	* Insert element into queue * True if Insertion is sucessful, * and Exception if Insertion Fails
⋮	

{Rest Few more methods are there}

Priority Queue (Internally based on Heap)

- Min Priority Queue → MinHeap
- Max Priority Queue → Max Heap
- Elements are ordered according to either Natural ordering or Comparator provided during queue construction time.

Comparator V/s Comparable

Comparator and Comparable both provides a way to sort the collection of Objects.

Ex:

Arrays.sort ( ) —(internally using)→ DualPivotQuicksort.sort( )

↳ Step inside sort method to see

↳ if we want to sort it using our Custom way we can provide our own Comparator.

Ex:

↳ For sorting collection Objects.

How to sort the
Object Array

Comparator [Functional Interface]

abstract int compare (T obj1, T obj2);

* Sorting algorithm uses this compare method of Comparator of 2 variables and decide whether to swap the variables or Not.

* Method returns

1: if obj1 > obj2

0: if obj1 == obj2

-1: if obj1 < obj2

* Mostly in algorithm, if method return 1, swap the values.

Comparable

* int compareTo(T obj2)

* If you are ~~[illegible]~~ using Arrays.sort(arr); then Comparable is used.

* If you don't provide anything, it uses the ~~[illegible]~~ compareTo() implementation present in default classes

When we are using

Arrays.sort(arr, Lambda expression)

$\Downarrow$

Internally its calling.

sort(T[] a,
Comparator<? super T> c)

wildcard,

If we pass Integer Array, So the Comparator of Integer or its parent class will be accepted

```
@FunctionalInterface
interface Comparator<T> {

    int compare(T obj1, T obj2);

}

Integer[] arr = {1,2,5,4}

Arrays.sort(arr, (Integer a, Integer b) -> a-b);

                    (Implementation, that is being
                                          called)
sort( T[] a, Comparator<? super T> c)

        ~~Comparable~~

public ~~sta~~ class PriorityQueue<E> extends AbstractQueue<E> {

  private static final int DEFAULT_INITIAL-CAPACITY =11;

  public PriorityQueue( Comparator<? super E> comparator){

        => Constructor with Comparator

  }

}
                                  max PriorityQueue, keeping biggest
                                                    at first
PriorityQueue <Integer> maxPQ = new PriorityQueue<>( (Integer a, Integer b)
                                                        -> b-a);
```

For Comparator, if you are not using Lambda expression,

```
public class ~~[illegible]~~ CustomComparator implements Comparator<Integer>{
  @Override
  public int compare(Integer a, Integer b){
      return a-b;
  }
}
```

Use your own custom comparator

```
Collections.sort( ~~arr~~, new CustomComparator());
```

Comparable

```
@FunctionalInterface
public interface Comparable<T>{

  public int compareTo(T o);
}
```

We only have one

(****) Now since this ~~class~~ function has only 1 argument, hence we need to use it with the class itself.

Example

```
public class Car implements Comparable<Car>{
    @override the compareTo method here
}
```

You can also use the comparator as Follows as we lets say you have class Car, and you want to sort it using Comparator

```
public class Car implements Comparator <Car> {

   String carName;

   @Override
   public int compare ( (Car obj1, Car obj2) ) {
     return     obj1.carName.compareTo( obj2.carName );
   }
}.
```

this class just became a comparator. Nothing ~~to~~ else. You can still use your custom Comparator

Comparable only having 1 way of doing it.

```
public class Car implements Comparable <Car> {

   String name;

   Car ( String name ) { this.name = name; }

   @Override
   public int compareTo ( (Car obj2) ) {
      return  { this.name.compareTo( obj2.name ); }
   }
```

this is what basically obj1, you can decide your ordering by that