**Fail-Fast Vs Fail-Safe Iterator in Java**

The collections which are there from Java 1.2 (or even legacy) like ArrayList, Vector, HashSet all have fail-fast iterator whereas Concurrent collections added in Java 1.5 like ConcurrrentHashMap, CopyOnWriteArrayList, CopyOnWriteArraySet have fail-safe iterator.

>fail-fast iterator throws ConcurrentModificationException if the underlying collection is structurally modified in any way except through the iterator's own remove or add methods.

fail-safe iterator doesn't throw ConcurrentModificationException.

Note that structural modification is any operation that adds or deletes one or more elements; merely setting the value of an element (in case of list) or changing the value associated with an existing key (in case of map) is not a structural modification.

In case of fail-safe iterator, ConcurrentModificationException is not thrown as the fail-safe iterator makes a copy of the underlying structure and iteration is done over that snapshot.

Since iteration is done over a copy of the collection so interference is impossible and the iterator is guaranteed not to throw ConcurrentModificationException.

**Difference Between Comparable and Comparator in Java**

Comparable interface in Java is defined as follows-

public interface Comparable<T> {

public int compareTo(T o);

}

Classes implementing this interface need to provide sorting logic in compareTo() method.

In the case of Comparator interface, class that is to be sorted doesn't implement it. Comparator can be implemented by some other class, as an anonymous class or as lambda expression (from Java 8). Comparators can be passed to a sort method (such as Collections.sort or Arrays.sort) to allow precise control over the sort order.

@FunctionalInterface

public interface Comparator<T> {

int compare(T o1, T o2);

}

>Comparable interface is in java.lang package.

Comparator interface is in java.util package.

>Comparable interface provides public int compareTo(T o); method which needs to be implemented for sorting the elements.

Comparator interface provides int compare(T o1, T o2); method which needs to be implemented for sorting the elements.

>The class which has to be sorted should implement the comparable interface (sorting logic is in the class which has to be sorted), and that implementation becomes the natural ordering of the class.Some other class can implement the Comparator interface not the actual class whose objects are to be sorted. That way there may be many comparators and depending on the ordering needed specific comparator can be used.

>When Comparable is used to sort the list we can use Collections.sort(List).

When Comparator is used to sort the list,Collections.sort(List, Comparator).

**Sorting ArrayList in descending order**

Use method **reverseOrder()** provided by Collections class

// sorting the list in descending order

Collections.sort(cityList, Collections.reverseOrder());

//Displaying the list

for(String city : cityList){

System.out.println("Name " + city);

}

**Using a custom comparator.**

// sorting the list in descending order

Collections.sort(cityList, new MyComparator());

//Displaying the list

for(String city : cityList){

System.out.println("Name " + city);

}

//Custom comparator class

class MyComparator implements Comparator<String>{

@Override

public int compare(String o1, String o2) {

return o2.compareTo(o1);

}

}

**Inter-thread Communication Using wait(), notify() And notifyAll() in Java**

Another important point about wait(), notify() and notifyAll() methods in Java is that they can only be called from a synchronized context, as these methods are about releasing the monitor and acquiring it again. Threads acquire monitor(lock) when entering a synchronized method (or block) so it makes sense to call them from synchronized context.

**wait()** method in Java

Wait method tells the current thread (thread which is executing code inside a synchronized method or block) to give up monitor and go to sleep, until another thread invokes the notify() or notifyAll() method for this object.

General form of wait method in Java

public final void wait() throws InterruptedException

There are two more overloaded wait methods

public final void wait(long timeout) throws InterruptedException

Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

synchronized (obj) {

while (condition not true){

obj.wait();

}

}

notify() method in Java

Wakes up a single thread that is waiting on this object's monitor. If more than one threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Note that the thread which comes out of waiting because of the notify() method will not be able to proceed until the current thread relinquishes the lock on this object. The awakened thread just changes to the runnable state and it is ready to be scheduled again.

synchronized (obj) {

while (condition not true){

obj.wait();

}

// When condition holds true

..

..

obj.notify();

}

**notifyAll()** method in Java

Wakes up all the threads that called wait() on the same object. As explained in notify() any one of the threads will be granted access to the object.

Generally you'll use code similar to as given below for calling notifyAll method.

synchronized (obj) {

while (condition not true){

obj.wait();

}

// When condition holds true

..

..

obj.notifyAll();

}