Differentiate between Iterator and Enumeration?

Iterator

- It can traverse both legacy as well as non legacy elements.
- It is slower than Enumeration.
- It can perform `remove` operations while traversing the collection.
- It is fail-fast.

Enumeration

- It can traverse only legacy elements.
- It is faster than an Iterator.
- It can perform only `traverse` operations on the collection.
- It is not a fail-fast.

What is Fail-fast and Fail-safe iterations in Java Collections?

Using iterations we can traverse over the collections objects. The iterators can be either fail-safe or fail-fast. *Fail-safe* iterators means they will not throw any exception even if the collection is modified while iterating over it. Whereas *Fail-fast* iterators throw an exception(*ConcurrentModificationException*) if the collection is modified while iterating over it.

Consider an example:

```
ArrayList<Integer> integers = new ArrayList<>();
integers.add(1);
integers.add(2);
integers.add(3);
Iterator<Integer> itr = integers.iterator();
while (itr.hasNext()) {
Integer a = itr.next();
integers.remove(a);
}
```

As arrayLists are fail-fast above code will throw an exception.
First a will have value = 1, and then 1 will be removed in same iteration.
Next when a will try to get next(), as the modification is made to the list, it will throw an exception here.

However if we use an fail-safe collection e.g. CopyOnWriteArrayList then no exception will occur:

```
List integers = new CopyOnWriteArrayList();
integers.add(1);
integers.add(2);
```

```
integers.add(3);
Iterator  itr = integers.iterator();
while (itr.hasNext()) {
Integer a = itr.next();
integers.remove(a);
}
```

Here if we print the element a, then all the elements will be printed.

Fail-Fast Iterators internal working:

Every fail fast collection has a **modCount** field, to represent how many times the collection has changed/modified.
So at every modification of this collection we increment the modCount value. For example the modCount is incremented in below cases:
1. When one or more elements are removed.
2. When one or more elements are added.
3. When the collection is replaced with other collection.
4. When the collection is sorted.
So everytime there is some change in the collection structure, the mod count is incremented.

Now the iterator stores the modCount value in the initialization as below:

```
int expectedModCount = modCount;
```

Now while the iteration is going on, expectedModCount will have old value of modCount.
If there is any change made in the collection, the modCOunt will change and then an exception is thrown using:

```
if (modCount != expectedModCount)
throw new ConcurrentModificationException();
```

This code is used in most of the iterator methods e.g.
1. next()
2. remove()
3. add()

So if we make any changes to the collection, the modCount will change, and expectedModCount will not be hence equal to the modCount. Then if we use any of the above methods of iterator, the ConcurrentModificationException will be thrown.

***Note**: If we remove/add the element using the remove() or add() of iterator instead of collection, then in that case no exception will opccur. It is because the remove/add methods of iterators call the*

remove/add method of collection internally, and also it reasigns the expectedModCount to new modCount value

```
ArrayList.this.remove(lastRet);
cursor = lastRet;
lastRet = -1;
expectedModCount = modCount;andArrayList.this.add(i, e);
expectedModCount = modCount;
```

So the below code is safe as we are removing the element from the iterator here:

```
Iterator<Integer> itr = integers.iterator();
while (itr.hasNext()) {
if (itr.next() == 2) {
// will not throw Exception
itr.remove();
}
}
```

Whereas the below code will throw an exception as we are removing the element from the collection here:

```
Iterator<Integer> itr = integers.iterator();
while (itr.hasNext()) {
if (itr.next() == 3) {
// will throw Exception on
// next call of next() method
integers.remove(3);
}
}
```

Fail-Safe Iterators internal working:

Unlike the fail-fast iterators, these iterators traverse over the clone of the collection. So even if the original collection gets structurally modified, no exception will be thrown. E.g. in case of CopyOnWriteArrayList the original collections is passed and is stored in the iterator:

```
public Iterator<E> iterator() {
return new COWIterator<E>(getArray(), 0);
}
```

here iterator() method returns the iterator of the CopyOnWriteArrayList. As we can see, it passes the getArray() in the constructor of the iterator. This getArray() has all the collection elements.
Now the iterator(COWIterator here) will save this to traverse upon as:

```
COWIterator(Object[] elements, int initialCursor) {
cursor = initialCursor;
snapshot = elements;
}
```

So the original collection elements are saved in the snapshot field variable.So all the iterator methods will work on this snapshot method. So even if there is any change in the original collection, no exception will be thrown. But note the the iterator will not reflect the latest state of the collection

```
Iterator<Integer> itr = integers.iterator();
while (itr.hasNext()) {
int a = itr.next();
if (a == 1) {
integers.remove(Integer.valueOf(a));
}
System.out.print(a);
}
```

So as we are removing the element from the collection. the collection now has only elements **2 and 3** in it. But the iterator will print all the elements **1,2,3** because it traverses over the snapshot of the collection elements.We can print the collection elements after the above code. It will print only **2,3** as:

```
Iterator<Integer> itr = integers.iterator();
while (itr.hasNext()) {
int a = itr.next();
System.out.print(a);
}
```

Q.What is the diff between Iterator and ListIterator in java

Iterator	ListIterator
Can traverse elements present in Collection only in the forward direction.	Can traverse elements present in Collection both in forward and backward directions.
Helps to traverse Map, List and Set.	Can only traverse List and not the other two.
Cannot modify or replace elements present in Collection	We can modify or replace elements with the help of set(E e)
Cannot add elements and it throws ConcurrentModificationException.	Can easily add elements to a collection at any time.
Certain methods of Iterator are next(), remove() and hasNext().	Certain methods of ListIterator are next(), previous(), hasNext(), hasPrevious(), add(E e).