Fail fast and fail safe

Iterators in java are used to iterate over the Collection objects.Fail-Fast iterators immediately throw *ConcurrentModificationException* if there is **structural modification** of the collection. Structural modification means adding, removing or updating any element from collection while a thread is iterating over that collection. Iterator on ArrayList, HashMap classes are some examples of fail-fast Iterator.

- Concurrent Modificaiton Exception can be thrown from a single threaded or multi threaded env.
- Single Threaded : If the collection is modified by any object after the iterator has been created, the iterator throws this exception. The iterator provides its own method to modify the collection.

Fail-Safe iterators don't throw any exceptions if a collection is structurally modified while iterating over it. This is because, they operate on the clone of the collection, not on the original collection and that's why they are called fail-safe iterators. Iterator on CopyOnWriteArrayList, ConcurrentHashMap classes are examples of fail-safe Iterator.

How Fail Fast Iterator works ?

To know whether the collection is structurally modified or not, fail-fast iterators use an internal flag called *modCount* which is updated each time a collection is modified.Fail-fast iterators checks the *modCount* flag whenever it gets the next value (i.e. using *next()* method), and if it finds that the *modCount* has been modified after this iterator has been created, it throws *ConcurrentModificationException*.

```
// Java code to illustrate
// Fail Fast Iterator in Java
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
```

```
public class FailFastExample {
	public static void main(String[] args)
	{
		Map<String, String> cityCode = new HashMap<String, String>();
		cityCode.put("Delhi", "India");
		cityCode.put("Moscow", "Russia");
		cityCode.put("New York", "USA");

		Iterator iterator = cityCode.keySet().iterator();

		while (iterator.hasNext()) {
			System.out.println(cityCode.get(iterator.next()));

			// adding an element to Map
			// exception will be thrown on next call
			// of next() method.
			cityCode.put("Istanbul", "Turkey");
		}
	}
}
```

Output :

```
India
Exception in thread "main" java.util.ConcurrentModificationException
    at java.util.HashMap$HashIterator.nextNode(HashMap.java:1442)
    at java.util.HashMap$KeyIterator.next(HashMap.java:1466)
    at FailFastExample.main(FailFastExample.java:18)
```

Important points of fail-fast iterators :

- These iterators throw ConcurrentModificationException if a collection is modified while iterating over it, by any other object other than the iterator itself.
- They use original collection to traverse over the elements of the collection.
- These iterators don’t require extra memory.
- Ex : Iterators returned by ArrayList, Vector, HashMap.

Note 1(from java-docs): The fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of

unsynchronized concurrent modification. Fail-fast iterators throw *ConcurrentModificationException* on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.

Note 2 : If you remove an element via Iterator *remove()* method, exception will not be thrown. However, in case of removing via a particular collection *remove()* method, *ConcurrentModificationException* will be thrown. Below code snippet will demonstrate this:

```
// Java code to demonstrate remove

// case in Fail-fast iterators




import java.util.ArrayList;

import java.util.Iterator;




public class FailFastExample {
```

```
public static void main(String[] args)

{

        ArrayList<Integer> al = new ArrayList<>();

        al.add(1);

        al.add(2);

        al.add(3);

        al.add(4);

        al.add(5);



        Iterator<Integer> itr = al.iterator();

        while (itr.hasNext()) {
```

```
            if (itr.next() == 2) {

                    // will not throw Exception

                    itr.remove();

            }

}



System.out.println(al);



itr = al.iterator();

while (itr.hasNext()) {

            if (itr.next() == 3) {
```

```
                    // will throw Exception on

                    // next call of next() method

                    al.remove(3);

                }

            }

        }

}
```

Fail Safe Iterator

First of all, there is no term as fail-safe given in many places as Java SE specifications does not use this term. I am using this term to demonstrate the difference between Fail Fast and Non-Fail Fast Iterator. These iterators make a copy of the internal collection (object array) and iterates over the copied collection. Any structural modification done to the iterator affects the copied collection, not original collection. So, original collection remains structurally unchanged.

- **Fail-safe iterators allow modifications of a collection while iterating over it.**

- **These iterators don’t throw any Exception if a collection is modified while iterating over it.**
- **They use copy of original collection to traverse over the elements of the collection.**
- **These iterators require extra memory for cloning of collection. Ex : ConcurrentHashMap, CopyOnWriteArrayList**

// Java code to illustrate

// Fail Safe Iterator in Java

```
import java.util.concurrent.CopyOnWriteArrayList;

import java.util.Iterator;



class FailSafe {

    public static void main(String args[])

    {

        CopyOnWriteArrayList<Integer> list

            = new CopyOnWriteArrayList<Integer>(new Integer[] { 1, 3, 5, 8 });
```

```
            Iterator itr = list.iterator();

            while (itr.hasNext()) {

                Integer no = (Integer)itr.next();

                System.out.println(no);

                if (no == 8)



                    // This will not print,

                    // hence it has created separate copy

                    list.add(14);

            }

        }

}
```

Also, those collections which don’t use fail-fast concept may not necessarily create clone/snapshot of it in memory to avoid ConcurrentModificationException. For example, in case of ConcurrentHashMap, it does not operate on a separate copy

although it is not fail-fast. Instead, it has semantics that is described by the official specification as weakly consistent(memory consistency properties in Java). Below code snippet will demonstrate this:

Example of Fail-Safe Iterator which does not create separate copy

```
// Java program to illustrate
// Fail-Safe Iterator which
// does not create separate copy
import java.util.concurrent.ConcurrentHashMap;
import java.util.Iterator;

public class FailSafeItr {
   public static void main(String[] args)
   {

      // Creating a ConcurrentHashMap
      ConcurrentHashMap<String, Integer> map
         = new ConcurrentHashMap<String, Integer>();

      map.put("ONE", 1);
      map.put("TWO", 2);
      map.put("THREE", 3);
```

```
        map.put("FOUR", 4);

        // Getting an Iterator from map
        Iterator it = map.keySet().iterator();

        while (it.hasNext()) {
            String key = (String)it.next();
            System.out.println(key + " : " + map.get(key));

            // This will reflect in iterator.
            // Hence, it has not created separate copy
            map.put("SEVEN", 7);
        }
    }
}
```

Note(from java-docs) : The iterators returned by ConcurrentHashMap is weakly consistent. This means that this iterator can tolerate concurrent modification, traverses elements as they existed when iterator was constructed and may (but not guaranteed to) reflect modifications to the collection after the construction of the iterator.

Difference between Fail Fast Iterator and Fail Safe Iterator

The major difference is fail-safe iterator doesn’t throw any Exception, contrary to fail-fast Iterator.This is because they work on a clone of Collection instead of the original collection and that’s why they are called as the fail-safe iterator.