**Iterators – Fail fast Vs Fail safe**

September 22nd, 2009[CertPal](http://www.certpal.com/blogs/author/certpal/)[Leave a comment](http://www.certpal.com/blogs/2009/09/iterators-fail-fast-vs-fail-safe/#respond)[Go to comments](http://www.certpal.com/blogs/2009/09/iterators-fail-fast-vs-fail-safe/#comments)

Iterators can be designed so that they are [fail fast](http://en.wikipedia.org/wiki/Fail-fast) or [fail safe](http://en.wikipedia.org/wiki/Fail-safe). Depending on the underlying implementation of the Iterator, a ConcurrentModificationException is thrown if the Collection is modified while Iterating over the data structure. It pays to understand how an Iterator will behave under both conditions. Lets try to implement fail fast Vs fail safe iterators of our own.

Our data structure for this example is pretty simple. It defines an interface that abstracts set and get operations on a structure. How the underlying classes handle invalid set operations or the size of the structure is implementation dependent

**A data structure interface:**

**public** **interface** Data<T> **extends** Iterable<T>

{

**int** size();

T getElement(**int** position);

**void** setElement(**int** position, T t);

}

An underlying implementation of such an Interface, could be say an array of Integers whose size is fixed. Invalid indexes on bounds are not allowed and the internal structure will not grow or shrink. An implementation is given below

**An array of integers implementing the interface:**

**public** **class** ArrayOfIntegers **implements** Data<Integer>

{

**private** **static** **int** DEFAULT\_SIZE = 1024;

**private** **int** mods = 0;

**private** Integer[] integers = **new** Integer[DEFAULT\_SIZE];

@Override

**public** Integer getElement(**int** position)

{

checkRange(position);

**return** integers[position];

}

@Override

**public** **void** setElement(**int** position, Integer integer)

{

checkRange(position);

mods++;

integers[position] = integer;

}

@Override

**public** **int** size()

{

**return** integers.length;

}

@Override

**public** Iterator<Integer> iterator()

{

*// return new FailFastIter();*

*// return new NoFailIter();*

**return** **null**;

}

**private** **void** checkRange(**int** position)

{

**if** (position > DEFAULT\_SIZE || position < 0)

{

**throw** **new** ArrayIndexOutOfBoundsException(position);

}

}

*// Iterator implementations go here as a private class*

}

If we wanted to Iterate over the ArrayOfIntegers structure, there are 2 ways to do it. Either ensure that the underlying data structure Integer[] integers is not modified while we iterate over ArrayOfIntegers, or make a copy of Integer[] integers so that any changes made to the internal structure will not affect the caller in any way. Let us look at 2 private Iterator classes that we can place into the ArrayOfIntegers class that will help us achieve both flavors of Iteration

**Fail fast:**

**private** **class** FailFastIter **implements** Iterator<Integer>

{

**int** currentIndex = 0;

**int** check = -1;

**int** size = integers.length;

**public** FailFastIter()

{

check = mods;

}

@Override

**public** **boolean** hasNext()

{

checkForModification();

**if** (currentIndex < size)

{

**return** **true**;

}

**return** **false**;

}

@Override

**public** Integer next()

{

checkForModification();

Integer result = integers[currentIndex];

currentIndex++;

**return** result;

}

@Override

**public** **void** remove()

{

**throw** **new** UnsupportedOperationException();

}

**private** **void** checkForModification()

{

**if** (check != mods)

{

**throw** **new** ConcurrentModificationException();

}

}

}

The fail fast Iterator in this example, refers to the internal data structure directly. Every modification to the internal structure is tracked using the ‘**mods**‘ variable. Our iterator stores this value when it was initially created using the ‘**check**‘ variable. Both variables are compared every time the hasNext() or next() methods are called. If they are unequal then it means that the underlying structure was changed. This is when the code throws a ConcurrentModificationException

**Fail safe:**

**private** **class** NoFailIter **implements** Iterator<Integer>

{

**int** currentIndex = 0;

Integer[] internal = Arrays.copyOf(integers, size());

**int** internalSize = -1;

**public** NoFailIter()

{

internalSize = internal.length;

}

@Override

**public** **boolean** hasNext()

{

**if** (currentIndex < internalSize)

{

**return** **true**;

}

**return** **false**;

}

@Override

**public** Integer next()

{

**if** (hasNext())

{

Integer result = internal[currentIndex];

currentIndex++;

**return** result;

}

**else**

{

**throw** **new** NoSuchElementException();

}

}

@Override

**public** **void** remove()

{

**throw** **new** UnsupportedOperationException();

}

}

The fail safe iterator makes a copy of the internal array data structure and uses it to iterate over the elements. This prevents any concurrent modification exceptions from being thrown if the underlying data structure changes. Of course, the overhead of copying the entire array is introduced. Both implementation can be tested using a program

**Iterator test for both iterators:**

**public** **class** IterTest

{

**public** **static** **void** main(String... args)

{

**new** IterTest().go();

}

**public** **void** go()

{

Data<Integer> dataArray = **new** ArrayOfIntegers();

**for** (**int** iCounter = 0; iCounter < 100; iCounter++)

{

dataArray.setElement(iCounter, iCounter);

}

Iterator<Integer> iterator = dataArray.iterator();

**while** (iterator.hasNext())

{

System.out.println(iterator.next());

dataArray.setElement(1, 12);

}

}

}

By changing the Iterator implementation that the iterator() method returns, this test program will either throw a ConcurrentModificationException or iterate over all the elements without knowing what changed under the hood.

It makes sense to throw ConcurrentModificationException from an Iterator in certain cases. Under other scenarios you just don’t care if the Iterator will fail or survive an internal structure change. Taking a leaf out of the java Collection API, take a look at the Iterator behavior of an ArrayList Vs that of a CopyOnWriteArrayList.

**Iterator test for 2 iterators within the JDK API:**

**public** **class** IterTest2

{

**public** **static** **void** main(String... args)

{

**new** IterTest2().go();

}

**public** **void** go()

{

List<String> arList = **new** ArrayList<String>();

List<String> copyList = **new** CopyOnWriteArrayList<String>();

populate(arList);

populate(copyList);

iterate(arList);

iterate(copyList);

}

**private** **void** populate(List<String> list)

{

**for** (**int** iCounter = 0; iCounter < 100; iCounter++)

{

list.add(**new** Integer(iCounter).toString());

}

}

**private** **void** iterate(List<String> list)

{

**try**

{

**for** (String x : list)

{

System.out.println(x);

list.add(x);

}

}

**catch** (RuntimeException e)

{

e.printStackTrace();

}

}

}

The CopyOnWriteArrayList does not fail when the internal structure changes, but ArrayList does. This example better highlights the problem of fail fast Vs fail safe iterators. The next time your write a custom iterator for your classes, consider this decision.

**PS:** The data structure defined here was just to illustrate the problem. You could certainly design this better by first providing an Abstract implementation that houses a private Iterator class, which will provide the default Iterator implementation for all subclasses. If the previous sentence sounds confusing, have a look at the **Itr**private class within the **AbstractList**class in the java source. The interface and implementation are much richer than the example provided here.

**Difference between fail-fast Iterator vs fail-safe Iterator in Java**

fail-fast Iterators in Java

[Difference between fail-safe vs fail-fast iterator in java](http://2.bp.blogspot.com/-wrzDeQGAe1I/TWu8pLuLr4I/AAAAAAAAADE/V017G-6Q61w/s1600/java_logo_50_50.jpg)As name suggest **fail-fast Iterators** fail as soon as they realized that *structure of Collection has been changed since iteration has begun*. Structural changes means adding, removing or updating any element from collection while one thread is Iterating over that collection. fail-fast behavior is implemented by keeping a modification count and if iteration thread realizes the change in modification count it throwsConcurrentModificationException.

Java doc says this is not a guaranteed behavior instead its done of "best effort basis", So application programming can not  rely on this behavior. Also since multiple threads are involved while updating and checking modification count and this check  is done without synchronization, there is a chance that Iteration thread still sees a stale value and might not be able to detect any change done by parallel threads. Iterators returned by most of JDK1.4 collection are fail-fast including Vector, [ArrayList](http://javarevisited.blogspot.com/2011/05/example-of-arraylist-in-java-tutorial.html), HashSet etc. to read more about Iterator see my post [What is Iterator in Java](http://javarevisited.blogspot.com/2011/10/java-iterator-tutorial-example-list.html).

fail-safe Iterator in java

Contrary to fail-fast Iterator, **fail-safe iterator** doesn't throw any Exception if Collection is modified structurally

while one thread is Iterating over it because they work on clone of Collection instead of original collection and that’s why they are called as fail-safe iterator. Iterator of CopyOnWriteArrayList is an example of fail-safe Iterator also iterator written by ConcurrentHashMap keySet is also fail-safe iterator and never throw ConcurrentModificationException in Java.

That’s all on **difference between fail-safe vs fail-fast Iterator in Java**, As I said due to there confusing or not to easy differentiation they are quickly becoming [popular java collection questions](http://javarevisited.blogspot.com/2011/11/collection-interview-questions-answers.html) asked on various level of java interview. Let me know if you are aware of any other difference between fail-fast and fail-safe iterator.

I don't know much regarding this.But I can explain the concept.   
Fail-safe and Fail-fast are the concepts relevant from the context of iterators.   
Fail-safe : If an iterator has been created on a collection object and some other thread tries to modify the collection object "structurally”, a concurrent modification exception will be thrown. It is possible for other threads though to invoke "set" method since it doesn't modify the collection "structurally”.However, if prior to calling "set", the collection has been modified structurally, "IllegalArgumentException" will   
be thrown.   
Fail-fast : For example, it is not generally permissible for one thread to modify a   
Collection while another thread is iterating over it. In general, the results of the iteration are undefined under these circumstances. Some Iterator   
implementations (including those of all the general purpose collection implementations provided by the JRE) may choose to throw this exception if this behavior is detected. Iterators that do this are known as fail-fast iterators,   
as they fail quickly and cleanly, rather that risking arbitrary, non-deterministic behavior at an undetermined time in the future.