Problems with normal iterations on Collection.

In this post we will see how to iterate the collection using the default method provided in Iterable<E> interface.

But before we proceed to new things let us see what problems we face in other iterating methods.

* First, let us traverse a list using size() and get(int index) methods. Refer [ArrayList<E> class from here](http://data-structure-learning.blogspot.com/2015/05/explaination-arraylist-class.html).

/\*\*

\* As we have discussed List<E> is backed by an array and

\* hence it provides us RandomAccess to elements.

\* We leverage the concept of E get(int index); method

\* from List<E> interface and traverse till the end of List<E>.

\* \*/

System.***out***.print("Iteration using Random Access to Array index ");

**for** (**int** i = 0; i < list.size(); i++) {

System.***out***.print(list.get(i) + " ");

}

This method will produce the result as expected. But there is problem. The problem is for loop condition “i < list.size(); ”. And why is that? Because whenever you will look this loop you need to pay extra attention to this condition and think it is “<”, “>”, “>=” or “<=”. Also you need to see how Iist loop progressing “++” or “--” or something else.

* Next, let us use enhanced for loop or for-each loop.

/\*\*

\* This is foreach loop.

\* It is used to iterate over array or Iterable<E>

\* Pronounce it as for each number of type Integer in list print it.

\* \*/

System.***out***.print("Using for-each loop ");

**for** (Integer number : list) {

System.***out***.print(number + " ");

}

This method will also produce the result as expected. So what the problem is? Well, for-each loop uses Iterator beneath it, but it does not provide remove() method. So using remove method will result in [ConcurrentModificationException](http://data-structure-learning.blogspot.com/2015/05/concurrentmodificationexception.html).

* Now, let us traverse the List<E> using Iterator<E>.

/\*\*

\* Use iterator to iterate through the list

\* hasNext()

\* return true if list has next element.

\* return false if end of list is reached.

\* next()

\* returns the next element in list.

\* \*/

System.***out***.print("Using Itertor ");

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

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

System.***out***.print(iterator.next() + " ");

}

Iterator works fine. There are no problems in the looping mechanism but there is one subtle issue. See the “iterator” object is defined outside the scope of loop. This can create problem when we try to take another iterator and confuse in the names and accidently use same names. See the code.

List<String> languages = *populateLanguages*();

Iterator<String> iterator=languages.iterator();

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

System.***out***.println(iterator.next()+" ");

}

List<String> databases=*populateDatabases*();

Iterator<String> iterator2=databases.iterator();

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

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

}

Scope of the iterator is more hence we can confuse in names and end up wasting time. Look, we used previous Iterator<E> object instead of using new one. This does not result in error. But it definitely wastes time.

There is a solution to this. We can use default method forEach(..).

In next post we will see how to use this method for different ways.