**java Iterator**

The Java Iterator interface represents an object capable of iterating through a collection of Java objects, one object at a time. The Iterator interface is one of the oldest mechanisms in Java for iterating collections of objects (although not the oldest - Enumerator predated Iterator).

* It is used to traverse a collection object elements one by one.
* It is available since Java 1.2 Collection Framework.
* It is applicable for all Collection classes. So it is also known as Universal Java Cursor.
* It supports both READ and REMOVE Operations.
* Compare to Enumeration interface, Iterator method names are simple and easy to use.
* Java Iterator Basic Example

A screenshot of a cell phone

Description automatically generated

A close up of a logo

Description automatically generated

**Obtaining an Iterator**

Most often that is how you will interact with an Iterator by obtaining it from some Java object that contains multiple nested objects. The standard Java collection interface Collection contains a method called iterator(). By calling iterator() you can obtain an iterator from the given Collection.

You can also obtain an Iterator from many of the Java Collection data structures, e.g. a List, Set, Map, Queue, Deque or Map.

### Java Iterator Methods

In this section, we will discuss about Java Iterator methods in-brief. We will explore these methods in-depth with some useful examples in the coming section.

* boolean hasNext():Returns true if the iteration has more elements.
* E next(): Returns the next element in the iteration.
* default void remove(): Removes from the underlying collection the last element returned by this iterator.
* default void forEachRemaining(Consumer action): Performs the given action for each remaining element until all elements have been processed or the action throws an exception.

**Iterating an Iterator**

You iterate the objects in an Iterator using a while loop. Here is an example of iterating the elements of a Java Iterator using a while loop:

Iterator iterator = list.iterator();

while(iterator.hasNext()) {

Object nextObject = iterator.next();

}

There are two methods to pay attention to in the above Java example. The first method is the Iterator hasNext() method which returns true if the Iterator contains more elements. In other words, if the Iterator has a next element.

The second method to pay attention to is the next() method. The next() method returns the next element of the collection the Iterator traverses.

**Iteration Order**

The order in which the elements contained in a Java Iterator are traversed depends on the object that supplies the Iterator. For instance, an iterator obtained from a List will iterate through the elements of that List in the same order the elements are stored internally in the List. An Iterator obtained from a Set, on the other hand, does not make any guarantees about the exact sequence the elements in the Set are iterated in.

**Java List Iterator**

Here is an example of obtaining a Java Set Iterator from a List instance:

List list = new ArrayList();

list.add("123");

list.add("456");

list.add("789");

Iterator iterator = list.iterator();

**Java Set Iterator**

Here is an example of obtaining a Java Set Iterator from a Set instance:

Set set = new HashSet();

set.add("123");

set.add("456");

set.add("789");

Iterator iterator = set.iterator();

**Modification During Iteration**

Some collections do not allow you to modify the collection while you are iterating it via an Iterator. In that case you will get a ConcurrentModificationException the next time you call the Iterator next() method. The following example results in a ConcurrentModificationException when executed:

List<String≫ list = new ArrayList<≫();

list.add("123");

list.add("456");

list.add("789");

Iterator<String≫ iterator = list.iterator();

while(iterator.hasNext()) {

String value = iterator.next();

if(value.equals("456")){

list.add("999");

}

}

The ConcurrentModificationException is thrown because the Iterator gets out of sync with the collection, if you modify the collection while iterating it via the Iterator.

**Remove Elements During Iteration**

The Java Iterator interface has a remove() method which lets you remove the element just returned by next() from the underlying collection. Calling remove() does not cause a ConcurrentModificationException to be thrown. Here is an example of removing an element from a collection during iteration of its Iterator:

List list = new ArrayList<>();

list.add("123");

list.add("456");

list.add("789");

Iterator iterator = list.iterator();

while(iterator.hasNext()) {

String value = iterator.next();

if(value.equals("456")){

iterator.remove();

}

}

# Java Collections - Overview

|  |
| --- |
|  |

In order to understand and use the Java Collections API effectively it is useful to have an overview of the interfaces it contains. So, that is what I will provide here.

There are two "groups" of interfaces: Collection's and Map's.

Here is a graphical overview of the Collection interface hierarchy:

A picture containing clock

Description automatically generated

And here is a graphical overview of the Map interface hierarchy:

A screen shot of a computer

Description automatically generated

You can find links to explanations of most (if not all) of these interfaces and implementations in the sub-menu at the top right of this page. That top-menu exists on all pages in this trail.

Java - How to Use Iterator?

Often, you will want to cycle through the elements in a collection. For example, you might want to display each element. The easiest way to do this is to employ an iterator, which is an object that implements either the Iterator or the ListIterator interface.

Iterator enables you to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements.

Before you can access a collection through an iterator, you must obtain one. Each of the collection classes provides an iterator( ) method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time.

In general, to use an iterator to cycle through the contents of a collection, follow these steps −

* Obtain an iterator to the start of the collection by calling the collection's iterator( ) method.
* Set up a loop that makes a call to hasNext( ). Have the loop iterate as long as hasNext( ) returns true.
* Within the loop, obtain each element by calling next( ).

For collections that implement List, you can also obtain an iterator by calling ListIterator.

The Methods Declared by Iterator

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **boolean hasNext( )**  Returns true if there are more elements. Otherwise, returns false. |
| 2 | **Object next( )**  Returns the next element. Throws NoSuchElementException if there is not a next element. |
| 3 | **void remove( )**  Removes the current element. Throws IllegalStateException if an attempt is made to call remove( ) that is not preceded by a call to next( ). |

The Methods Declared by ListIterator

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **void add(Object obj)**  Inserts obj into the list in front of the element that will be returned by the next call to next( ). |
| 2 | **boolean hasNext( )**  Returns true if there is a next element. Otherwise, returns false. |
| 3 | **boolean hasPrevious( )**  Returns true if there is a previous element. Otherwise, returns false. |
| 4 | **Object next( )**  Returns the next element. A NoSuchElementException is thrown if there is not a next element. |
| 5 | **int nextIndex( )**  Returns the index of the next element. If there is not a next element, returns the size of the list. |
| 6 | **Object previous( )**  Returns the previous element. A NoSuchElementException is thrown if there is not a previous element. |
| 7 | **int previousIndex( )**  Returns the index of the previous element. If there is not a previous element, returns -1. |
| 8 | **void remove( )**  Removes the current element from the list. An IllegalStateException is thrown if remove( ) is called before next( ) or previous( ) is invoked. |
| 9 | **void set(Object obj)**  Assigns obj to the current element. This is the element last returned by a call to either next( ) or previous( ). |

Example

Here is an example demonstrating both Iterator and ListIterator. It uses an ArrayList object, but the general principles apply to any type of collection.

Of course, ListIterator is available only to those collections that implement the List interface.

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

Iterator vs Foreach In Java

**Background :**

[**Iterator**](http://geeksquiz.com/how-to-use-iterator-in-java/) is an interface provided by collection framework to traverse a collection and for a sequential access of items in the collection.

// Iterating over collection 'c' using [terator](http://geeksquiz.com/how-to-use-iterator-in-java/)

for (Iterator i = c.iterator(); i.hasNext(); )

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

[**For each**](https://www.geeksforgeeks.org/for-each-loop-in-java/)loop is meant for traversing items in a collection.

// Iterating over collection 'c' using [**for-each**](https://www.geeksforgeeks.org/g-fact-40-foreach-in-c-and-java/)

for (Element e: c)

System.out.println(e);

We read the ‘:’ used in for-each loop as “in”. So loop reads as “for each element e in elements”, here elements is the collection which stores Element type items.

**Note :** In Java 8 using lambda expressions we can simply replace for-each loop with

elements.forEach (e -> System.out.println(e) );

**Difference between the two traversals**

In for-each loop, we can’t modify collection, it will throw a [ConcurrentModificationException](https://docs.oracle.com/javase/7/docs/api/java/util/ConcurrentModificationException.html)on the other hand with iterator we can modify collection.

Modifying a collection simply means removing an element or changing content of an item stored in the collection. This occurs because for-each loop implicitly creates an iterator but it is not exposed to the user thus we can’t modify the items in the collections.

**When to use which traversal?**

* If we have to modify collection, we can use Iterator.
* While using nested for loops it is better to use for-each loop, consider the below code for better understanding.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java program to demonstrate working of nested iterators  // may not work as expected and throw exception.  import java.util.\*;    public class Main  {      public static void main(String args[])      {          // Create a link list which stores integer elements          List<Integer> l = new LinkedList<Integer>();            // Now add elements to the Link List          l.add(2);          l.add(3);          l.add(4);            // Make another Link List which stores integer elements          List<Integer> s=new LinkedList<Integer>();          s.add(7);          s.add(8);          s.add(9);            // Iterator to iterate over a Link List          for (Iterator<Integer> itr1=l.iterator(); itr1.hasNext(); )          {              for (Iterator<Integer> itr2=s.iterator(); itr2.hasNext(); )              {                  if (itr1.next() < itr2.next())                  {                      System.out.println(itr1.next());                  }              }          }      }  } |

Output:

Exception in thread "main" java.util.NoSuchElementException

at java.util.LinkedList$ListItr.next(LinkedList.java:888)

at Main.main(Main.java:29)

The above code throws java.util.NoSuchElementException.

In the above code we are calling the next() method again and again for itr1 (i.e., for List l). Now we are advancing the iterator without even checking if it has any more elements left in the collection(in the inner loop), thus we are advancing the iterator more than the number of elements in the collection which leads to NoSuchElementException.

for-each loops are tailor made for nested loops. Replace the iterator code with the below code.

## ****The**** Iterator ****Interface****

To start, we need to obtain an Iterator from a Collection; this is done by calling the iterator() method.

For simplicity, we'll obtain Iterator instance from a list:

|  |  |
| --- | --- |
| 1  2 | List<String> items = ...  Iterator<String> iter = items.iterator(); |

The Iterator interface has three core methods:

### **2.1.** hasNext()

The hasNext() method can be used for checking if there's at least one element left to iterate over.

It's designed to be used as a condition in while loops:

|  |  |
| --- | --- |
| 1  2  3 | while (iter.hasNext()) {      // ...  } |

### **2.2.** next()

The next() method can be used for stepping over the next element and obtaining it:

|  |  |
| --- | --- |
| 1 | String next = iter.next(); |

**It's good practice to use hasNext() before attempting to call next().**

Iterators for Collections don't guarantee iteration in any particular order [unless particular implementation provides it.](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html#iterator())

### **2.3.** remove()

Finally, if we want to **remove the current element from the collection,** we can use the remove:

|  |  |
| --- | --- |
| 1 | iter.remove(); |

This is a safe way to remove elements while iterating over a collection without a risk of a [ConcurrentModificationException.](https://www.baeldung.com/java-concurrentmodificationexception)

### **2.4. Full**Iterator **Example**

Now we can combine them all and have a look at how we use the three methods together for collection filtering:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | while (iter.hasNext()) {      String next = iter.next();      System.out.println(next);        if( "TWO".equals(next)) {          iter.remove();      }  } |

This is how we commonly use an Iterator,we check ahead of time if there is another element, we retrieve it and then we perform some action on it.

### **2.5. Iterating With Lambda Expressions**

As we saw in the previous examples, it's very verbose to use an Iterator when we just want to go over all the elements and do something with them.

Since Java 8, we have the forEachRemaining method that allows the use of lambdas to processing remaining elements:

|  |  |
| --- | --- |
| 1 | iter.forEachRemaining(System.out::println); |

## ****3. The**** ListIterator ****Interface****

ListIteratoris an extension that adds new functionality for iterating over lists:

|  |  |
| --- | --- |
| 1 | ListIterator<String> listIterator = items.listIterator(items.size()); |

Notice how we can provide a starting position which in this case is the end of the List.

### **3.1.** hasPrevious() **and**previous()

ListIterator can be used for backward traversal so it provides equivalents of hasNext() and next():

|  |  |
| --- | --- |
| 1  2  3 | while(listIterator.hasPrevious()) {      String previous = listIterator.previous();  } |

### 3.2. nextIndex() and previousIndex()

Additionally, we can traverse over indices and not actual elements:

|  |  |
| --- | --- |
| 1  2 | String nextWithIndex = items.get(listIterator.nextIndex());  String previousWithIndex = items.get(listIterator.previousIndex()); |

This could prove very useful in case we need to know the indexes of the objects we're currently modifying, or if we want to keep a record of removed elements.

### **3.3.** add()

The add method, which, as the name suggests, allows us to add an element **before the item that would be returned by next() and after the one returned by previous():**

|  |  |
| --- | --- |
| 1 | listIterator.add("FOUR"); |

### **3.4.** set()

The last method worth mentioning is set(),which lets us replace the element that was returned in the call to next() or previous():

|  |  |
| --- | --- |
| 1  2  3  4 | String next = listIterator.next();  if( "ONE".equals(next)) {      listIterator.set("SWAPPED");  } |

It's important to note that **this can only be executed if no prior calls to add() or remove() were made.**

### **3.5. Full** ListIterator **Example**

We can now combine them all to make a complete example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | ListIterator<String> listIterator = items.listIterator();  while(listIterator.hasNext()) {      String nextWithIndex = items.get(listIterator.nextIndex());      String next = listIterator.next();      if("REPLACE ME".equals(next)) {          listIterator.set("REPLACED");      }  }  listIterator.add("NEW");  while(listIterator.hasPrevious()) {      String previousWithIndex       = items.get(listIterator.previousIndex());      String previous = listIterator.previous();      System.out.println(previous);  } |

In this example, we start by getting the ListIterator from the List, then we can obtain the next element either by index –**which doesn't increase the iterator's internal current element**– or by calling next.

**iterator or for loop which is faster**

**Iterator** and for-each **loop** are **faster** than simple for **loop** for collections with no random access, while in collections which allows random access there is no performance change with for-each **loop**/for **loop**/**iterator**.