

Purpose of Iterators

Commonly want to perform some process on all the data(objects) in a data structure

- visualizations or other output
- saving or transmitting the data
- computing conversions or summaries
- performing some kind of update/change on each object

Iterators

- The client code can move the Iterator by explicitly calling its next method, or by using an enhanced for statement
- The Iterator stays on its current list item until it is needed
- An Iterator traverses in $O(n)$ while a list traversal using `get()` calls in a linked list is $O(n^2)$

Iterator Interface: <https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html>

hasNext

```
boolean hasNext()
```

Returns `true` if the iteration has more elements. (In other words, returns `true` if `next()` would return an element rather than throwing an exception.)

Returns:

`true` if the iteration has more elements

next

```
E next()
```

Returns the next element in the iteration.

Returns:

the next element in the iteration

Throws:

`NoSuchElementException` - if the iteration has no more elements

remove

```
default void remove()
```

Removes from the underlying collection the last element returned by this iterator (optional operation). This method can be called only once per call to `next()`. The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

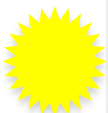
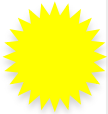
Implementation Requirements:

The default implementation throws an instance of `UnsupportedOperationException` and performs no other action.

Throws:

`UnsupportedOperationException` - if the remove operation is not supported by this iterator

`IllegalStateException` - if the `next` method has not yet been called, or the `remove` method has already been called after the last call to the `next` method



Using Iterator explicitly

In the following loop, we process all items in `List<Integer>` through an Iterator

```
Iterator<Integer> iter = aList.iterator();
```

List method that returns and iterator

```
while (iter.hasNext()) {
```

Make sure not at the end of the List

```
    int value = iter.next();
```

```
    // Do something with value
```

Get the next value

```
    ...
```

```
}
```

Using Iterator in enhanced for statement

In the following loop, we process all items in List<Integer> through an Iterator

```
for(Integer value: aList) {  
    ...  
}
```

remove() method

- You can use the Iterator `remove()` method to remove items from a list as you access them
- `remove()` deletes the most recent element returned by a call to `next()`
- The difference between removing directly from a linked list vs. using an iterator to remove
 - Linked List: must walk down the list each time, then remove, so in general it is $O(n)$
 - Iterator: removes items without starting over at the beginning, so in general it is $O(1)$

Example using remove()

```
28 public static void removeDivisibleBy(Iterable<Integer> values,  
29                                     int div) {  
30     Iterator<Integer> iter = values.iterator();  
31     while (iter.hasNext()) {  
32         int nextInt = iter.next();  
33         if (nextInt % div == 0) {  
34             iter.remove();  
35         }  
36     }  
37 }  
38
```

Example using remove() with Iterable and Comparable

```
39 public static <T extends Comparable<T>> void removeBelowBound
40 (Iterable<T> values, T lowerbound)
41 {
42     Iterator<T> iter = values.iterator();
43     while (iter.hasNext()) {
44         if (iter.next().compareTo(lowerbound) < 0) {
45             iter.remove();
46         }
47     }
48 }
```


ListIterator

- Iterator limitations
 - Traverses List only in the forward direction
 - Provides a remove method, but no add method
 - You must advance the Iterator using your own loop if you do not start from the beginning of the list
- ListIterator extends Iterator, overcoming these limitations
<https://docs.oracle.com/javase/8/docs/api/java/util/ListIterator.html>

Why bother with Iterators?

- Lists have indices so we *can* use an index to access all the objects(`myList.get(i)`)
- If the underlying structure is a linked chain, this becomes $O(n^2)$. An Iterator can do it in $O(n)$ because it directly accesses the underlying structure!
- An iterator can provide a way for client code to traverse all the data in a collection like a bag or set, even without access to indexed elements!