# Iterator Notes

There are data types in programming that does not use indices to indicate where values are. Among those types are Sets, Queues, Stacks, and Linked Lists. In order to traverse these types, an iterator is created. An iterator is an interface that allows the programmer to go from item to item without using indices. Even some data types that do have indices, also include an iterator. An ArrayList is an example of this. Understanding iterators is essential for programming using these data types.

There are two types of iterators we will be using. There is the Iterator and the ListIterator, both of which is listed in the API. To use these iterators, you must import them into your code

**import** java.util.Iterator;

**import** java.util.ListIterator;

Create a file called Example08 and write the following:

**import** java.util.Iterator;

**import** java.util.\*;

**import** **static** java.lang.System.\*;

**public** **class** ExampleIterators {

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

{

ArrayList<Integer> list = **new** ArrayList<>();

Scanner input = **new** Scanner("42 21 17 10 22 12 9 33 4 32 28 31");

**while**(input.hasNext())

list.add(input.nextInt());

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

***out***.println(iter.next());

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

***out***.println(value);

**while**(iter.hasNext())

iter.next();

***out***.println(iter.next());

}

}

The output should look like this:

42

21

Exception in thread "main" java.util.NoSuchElementException

at java.util.ArrayList$Itr.next(Unknown Source)

at ExampleIterators.main(ExampleIterators.java:24)

Notice that there is no instantiation for the Iterator. That is because Iterator is an interface and is not instantiated. Instead, it is created by the Object that the iterator will be moving through.

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

Iterator also needs to have a generic value given to it or it will always hold an Object. Now that there is an iterator created, we need to be able to move through the iterator, and that is done with next(). next() returns the value it passes through and does not store it. It does not retain a location of where it has been. I see it as next() inhabiting the in-between space of the list it is iterating through.

list

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 42 | 21 | 17 | 10 | 22 | 12 | 9 | 33 | 4 | 32 | 28 | 31 |

iter

When next is called, iter passes through the value, returns the value 42, and does not return to its previous spot.

list

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 42 | 21 | 17 | 10 | 22 | 12 | 9 | 33 | 4 | 32 | 28 | 31 |

iter

Looking at the code, you can see that next() doesn’t store the value because when it is asked to put next() into the integer value, it places 21, the next value in the list. The iterator also has a method called hasNext(), which checks to see if there is another element to be crossed over. Notice that if the programmer tries to next() at the end of the list, an error pops up because there are no more values ahead of next.

Comment out the last line of the code and write the following

//out.println(iter.next());

iter = list.iterator();

//iter.remove();

iter.next();

iter.remove();

iter.next();

iter.next();

value = iter.next();

iter.remove();

***out***.println(value);

**while**(iter.hasNext())

iter.next();

iter.remove();

***out***.println(list);

iter.remove();

The output should look like this

42

21

10

[21, 17, 22, 12, 9, 33, 4, 32, 28]

Exception in thread "main" java.lang.IllegalStateException

at java.util.ArrayList$Itr.remove(Unknown Source)

at ExampleIterators.main(ExampleIterators.java:39)

The method remove() is a void method and does not return the value that is being removed. Please noticed the commeted section

//iter.remove();

This is commented out because remove cannot be called until after a next has occurred. In fact, if you look at the error on the bottom of the output you will notice that this occurred because there were two remove calls in a row. The second remove call did not have a next occur before it, so there was no value to remove. Let’s look at what happened in this previous code.

list

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 42 | 21 | 17 | 10 | 22 | 12 | 9 | 33 | 4 | 32 | 28 | 31 |

iter

list

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 42 | 21 | 17 | 10 | 22 | 12 | 9 | 33 | 4 | 32 | 28 | 31 |

iter

42 is removed

list

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 21 | 17 | 10 | 22 | 12 | 9 | 33 | 4 | 32 | 28 | 31 |

iter

list

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 21 | 17 | 10 | 22 | 12 | 9 | 33 | 4 | 32 | 28 | 31 |

iter

value is assigned 10

list

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 21 | 17 | 22 | 12 | 9 | 33 | 4 | 32 | 28 | 31 |

iter

10 is removed and then it is printed through value

list

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 21 | 17 | 22 | 12 | 9 | 33 | 4 | 32 | 28 | 31 |

iter

The iterator loops all the way down to the last value

list

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 21 | 17 | 22 | 12 | 9 | 33 | 4 | 32 | 28 |

iter

31 is removed, the list is printed and the program crashes on the second remove call because a next did not occur before it.

This is the only way to remove from a Collection while maintaining the iterator. If the programmer tries to resize the Collection in any other way, then a new iterator would have to be created as seen below.

Comment the last line and add the following to the code

//iter.remove();

list.add(value);

***out***.println(list);

iter.next();

The output will look like this

[21, 17, 22, 12, 9, 33, 4, 32, 28, 10]

Exception in thread "main" java.util.ConcurrentModificationException

at java.util.ArrayList$Itr.checkForComodification(Unknown Source)

at java.util.ArrayList$Itr.next(Unknown Source)

at ExampleIterators.main(ExampleIterators.java:43)

The programmer has added the value 10 to the list. But now the iterator is no longer valid because the modification to ArrayList was done without using an iterator. In fact, iterator does not allow the programmer to add to Collection when used. When the programmer must modify the Collection, then they must reset the iterator. In fact, when the iterator gets to the end of the Collection, there is no other choice but to reset the iterator.

## List Iterator

This is one of the reasons that Listiterator is useful, it does the same thing that iterator does, but allows more flexibility to the iterator. It extends the Iterator interface. We have never addressed that, but a programmer can extend an interface.

Create a new class called ExampleListIterator and put the following in

**import** java.util.ListIterator;

**import** java.util.\*;

**import** **static** java.lang.System.\*;

**public** **class** ExampleListIterator {

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

{

ArrayList<Integer> list = **new** ArrayList<>();

Scanner input = **new** Scanner("42 21 17 10 22 12 9 33 4 32 28 31");

**while**(input.hasNext())

list.add(input.nextInt());

ListIterator<Integer> iter = list.listIterator();

***out***.println(list);

iter.add(39);

***out***.println(list);

iter.next();

iter.next();

iter.next();

iter.add(38);

***out***.println(list);

}

}

The output looks like this

[42, 21, 17, 10, 22, 12, 9, 33, 4, 32, 28, 31]

[39, 42, 21, 17, 10, 22, 12, 9, 33, 4, 32, 28, 31]

[39, 42, 21, 17, 38, 10, 22, 12, 9, 33, 4, 32, 28, 31]

List Iterator allows the user to add to the list using add(int value). When it does add to the list, it always adds behind the next value. Let’s walk through the code:

list

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 42 | 21 | 17 | 10 | 22 | 12 | 9 | 33 | 4 | 32 | 28 | 31 |

iter

list

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 42 | 21 | 17 | 10 | 22 | 12 | 9 | 33 | 4 | 32 | 28 | 31 |

iter

add(39) places the value 39 behind the iter.

list

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 42 | 21 | 17 | 10 | 22 | 12 | 9 | 33 | 4 | 32 | 28 | 31 |

iter

next is called 3 times

list

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 42 | 21 | 17 | 38 | 10 | 22 | 12 | 9 | 33 | 4 | 32 | 28 | 31 |

iter

add(38) places 38 behind the iter.

There is also the new methods previous and hasPrevious. These methods work just like next except they go the opposite direction. The programmer should be aware that previous will create an error if iter is sitting at the front of the list, just like next creates an error when it is at the end of the list. The method set will behave like remove, in that it is a void method and will only act after a previous or a next occurs.

Add the following to the code

iter.next();

iter.set(3);

***out***.println(list);

iter.add(5);

iter.set(36);

The output is this

[39, 42, 21, 17, 38, 3, 22, 12, 9, 33, 4, 32, 28, 31]

Exception in thread "main" java.lang.IllegalStateException

at java.util.ArrayList$ListItr.set(Unknown Source)

at ExampleListIterator.main(ExampleListIterator.java:30)

In the first part of the new code we see we are adding 3 into the list, after obeying the rule of having next or add occur

list

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 42 | 21 | 17 | 38 | 3 | 22 | 12 | 9 | 33 | 4 | 32 | 28 | 31 |

iter

After printing we add 5, which is fine

list

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 42 | 21 | 17 | 38 | 3 | 5 | 22 | 12 | 9 | 33 | 4 | 32 | 28 | 31 |

iter

But then we try to use set, before a next or a previous is called. This leads to a crash because iter is no longer at the position it used to be and doesn’t have a value to replace.

Comment out the previous line and add the following line of code

//iter.set(36);

***out***.println(list);

***out***.println(iter.nextIndex());

***out***.println(iter.previousIndex());

The output should be

[39, 42, 21, 17, 38, 3, 5, 22, 12, 9, 33, 4, 32, 28, 31]

7

6

The last 2 methods to look at is the nextIndex and the previousIndex which will give you the index value of the List that you are working with. This is not a method I focus on but it is a useful method. It tells you the indexes that the iter is in-between.

## forEachRemaining

The Iterator class as well as the ListIterator class has another method in it called the forEachRemaining method. This method is a lambda method that incorporates Consumers.

add the following to the previous code

iter = list.listIterator();

iter.next();

iter.next();

iter.next();

iter.forEachRemaining(k -> System.***out***.println(k));

the output should be

7

6

17

38

3

5

22

12

9

33

4

32

28

31

The consumer k is every object that remains in the iter that the iter hasn’t crossed. The -> says what we are going to do with that value. Finally we show what we are going to do with the consumer. There are a lot of limitations to what I show you here, because lambda has lots of depth. But it is a good starting point and you can experiment with how to work with this method.