Iterators

Mason Vail
Boise State University Computer Science

Iterable Collections

All *Collections* in Java are expected to be *Iterable* - to implement the *Iterable* interface.

The one method of *Iterable* is iterator(), which returns a reference to an *Iterator* - another interface. The object being returned, then, is an implementation of *Iterator* that knows how to iterate over the elements in the collection in a standardized way.

One notable benefit of all *Collections* being *Iterable* is that they can be used in a for-each loop.

Iterator Purpose

The primary purpose of an *Iterator* is to navigate through all elements of a *Collection* in a standardized way. Because *Collections* may organize their elements in innumerable ways, each *Collection* needs a custom *Iterator* that knows how to navigate its specific organization.

The promise of an *Iterator* is that it will return the exact contents of the *Collection*, one element at a time. Every element will be returned exactly once, and no element will be left out.

A secondary purpose of an *Iterator* is to serve as a filtering mechanism. In addition to navigation methods, some Iterators support a method that removes the last element the *Iterator* returned during navigation.

Iterator Methods

Navigation methods:

- public boolean **hasNext()** - Returns true if the Iterator still has elements it hasn’t returned, yet. Returns false if all elements have been returned.
- public T **next()** - Returns the next element if there are unreturned elements. Throws a NoSuchElementException if all elements have been returned.

Filtering method:

- public void **remove()** - Removes the element most recently returned by next(). Throws an IllegalStateException if not preceded by a call to next(). Can only be called once for a particular call to next().

Example Code: Looping Through All Elements

```
public void showAll( List<Integer> list ) {
        Iterator<Integer> itr = list.iterator();  //get an Iterator for the list
        while( itr.hasNext() ) {                  //continue while there are remaining elements
                Integer element = itr.next();     //get next element
                System.out.println(element);      //do something with it
        }
}

public void showAll( List<Integer> list ) {
        for( Integer element : list ) {           //gets Iterator from list and builds a while loop
                System.out.println(element);      // identical to above method's loop
        }
}
```

Iterator for Universally Efficient Navigation

Either loop on the previous slide would navigate through all elements of any list implementation in O(n) time. Each element is visited exactly once and the *Iterator* does not lose its place between calls to next().

By contrast, the following loop that produces the same output would result in O(n^2) runtime for a linked list. Consistent use of *Iterator*s for navigation make for universally efficient navigation regardless of implementation.

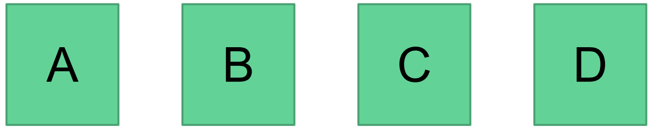
```
for ( int i = 0; i < list.size(); i++ ) {       //O(n) loop
    System.out.println( list.get(i) );  //O(n) calls to get() for linked lists
}
```

Example Code: Filtering Out All Even Integers

```
public void removeEvens( List<Integer> list ) {

    Iterator<Integer> itr = list.iterator(); //get an Iterator for the list

    while( itr.hasNext() ) {              //continue while there are remaining elements
        if( itr.next() % 2 == 0 ) {       //if the next element is even
            itr.remove();                 //remove the element last returned by next()
        }
    }

}
```

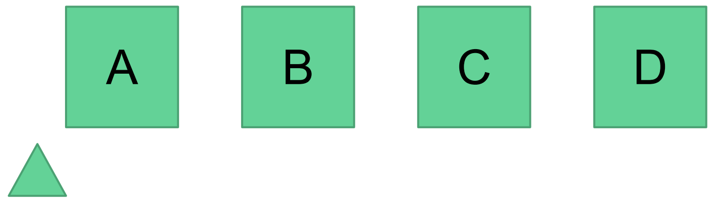
Iterator for a List

User’s abstract perspective of a list



Iterator for a List

User's abstract perspective of a list



A new Iterator is conceptually cued up in front of the first element of a list. Iterators are visualized as being between elements.

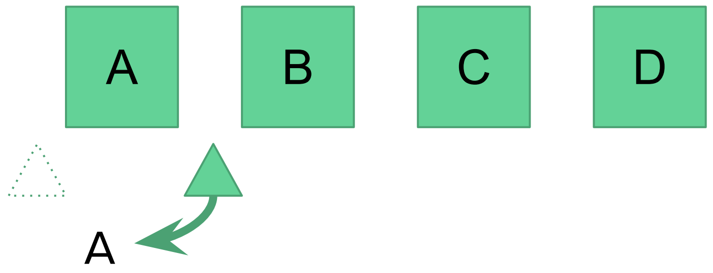
At this point, hasNext() would be expected to return true because there are still elements to the right of the Iterator that have not been returned.

next() would be expected to return element A and advance the Iterator to the position between A and B.

Because no calls have been made to next(), remove() would be expected to throw an IllegalStateException.

Iterator for a List

User’s abstract perspective of a list



When next() is called, the Iterator moves to the position between A and B and returns A, the element it just passed over.

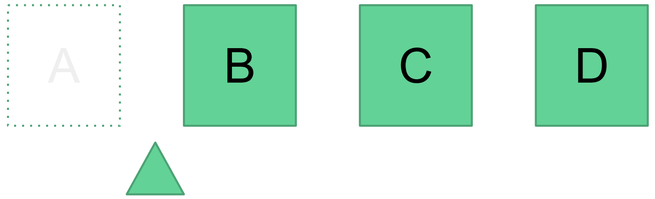
At this point, hasNext() would be expected to return true because there are still elements to the right of the Iterator that have not been returned.

next() would be expected to return element B and advance the Iterator to the position between B and C.

remove() would be expected to remove element A, the last returned element, from the list.

Iterator for a List

User's abstract perspective of a list



If remove() is called, element A is removed from the list. The Iterator is still in front of element B.

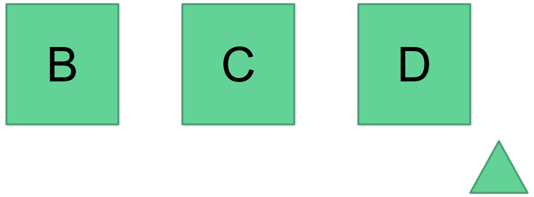
At this point, hasNext() would be expected to return true because there are still elements to the right of the Iterator that have not been returned.

next() would be expected to return element B and advance the Iterator to the position between B and C.

Another call to remove() before first calling next() would be expected to throw an IllegalStateException. remove() cannot be called twice in succession.

Iterator for a List

User's abstract perspective of a list



After 3 more calls to next(), the Iterator is positioned after the last element, D.

At this point, hasNext() would be expected to return false because all elements have been returned.

next() would be expected to throw a NoSuchElementException.

remove() would remove element D from the list and the Iterator would still be positioned after the last element, which would now be C.

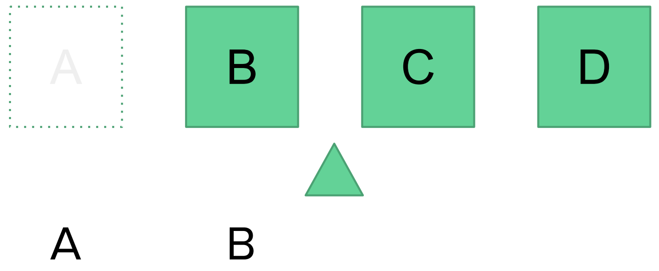
“Fail-fast” and ConcurrentModificationException

*Iterator*s promise to reliably return the contents of their *Collection*. When hasNext() finally returns false, you should be assured that the values returned from next() were the complete current contents of the *Collection*.

If any unknown source (i.e. not the *Iterator*’s own remove()) changes the *Collection* while an *Iterator* is still active, however, it cannot guarantee that the change has not invalidated what it has shown you. As soon as an *Iterator* detects a change to the list, calling any of its methods should result in a ConcurrentModificationException. The *Iterator* self-destructs as soon as it is able rather than give the impression that it is still trustworthy. This behavior is called “fail-fast.”

Iterator Interactions

User’s abstract perspective of a list

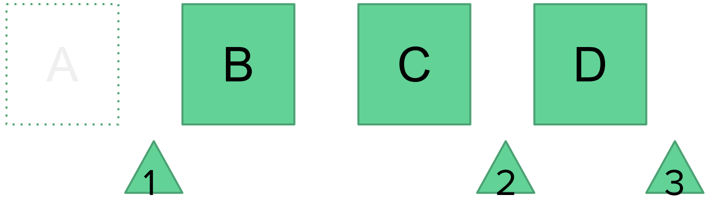


If a change is made to the list in other code, say through a list method like removeFirst(), the Iterator was not involved in that change and must fail-fast. In this example, the Iterator is entirely justified in failing-fast. It has reported that the list contains A, but it no longer does.

A ConcurrentModificationException is now expected when attempting to use any of the Iterator’s methods.

Iterator Interactions

User’s abstract perspective of a list



A single Iterator is able to continue operating without failing-fast when its own remove() method is used. The user of the Iterator is aware of the change they have made.

However, when multiple Iterators are active simultaneously, if one of them changes the list with its remove() method, the others were not involved and they are expected to fail-fast.

In this example, Iterator 1 has just removed element A. Iterators 2 and 3 are expected to throw ConcurrentModificationExceptions the next time they are used. They do not know what changed and their reported values are no longer trustworthy.

Iterators

Mason Vail
Boise State University Computer Science