



Week 09: Iterators

CS 220: Software Design II — D. Mathias

Traversing a Linked List

```
public class LinkedList<E>{  
    private ListNode firstNode;  
    ...  
    public E get(int index) {  
        ListNode node = firstNode.getNext();  
        for(int i = 0; i < index; i++) {  
            node = node.getNext();  
        }  
        return node.value;  
    }  
}
```

Initial instinct is to use a for loop using calls to the get method

What's “wrong” with this?

we traverse the same nodes multiple times
takes extra time

```
LinkedList<String> ll = new LinkedList<>();  
// list filled with data  
for(int i = 0; i < ll.size(); i++) {  
    System.out.println(ll.get(i));  
}
```

This eliminates the extra work but...

```
// reference to the first node
LinkedList<String> ll = new LinkedList<>();

// list filled with data

ListNode<String> node = firstNode.getNext();
for(int i = 0; i < ll.size(); i++)
{
    System.out.println(node.getValue());
    node = node.getNext();
}
```

...the programmer is working with lower-level details of the list.

Iterator

An *iterator* is a class that allows a programmer to traverse the elements of a data structure in a predefined fashion

- will keep track of the current place in traversal

We will use this primarily with lists, but can be used on other (non-linear) data structures as well

Why Iterators?

1. Prevent problems we see with the previous linked list example
i.e., needing to traverse the beginning of the list multiple times to access each element
and/or forcing programmers to deal with details of a data structure
2. Generally useful for accessing all the elements in a collection in some ordered way
3. Allows for more elegant code when iterating over a collection

Implementing an Iterator

1. Ensure your list class implements the `Iterable` interface

- defines this data structure as something that provides an iterator
- requires one method: `public Iterator<T> iterator()`, which returns an iterator object
- will also allow writing foreach loops

2. Create an inner class (within list) that implements the `Iterator` interface

- basic components:
 - attribute pointing to the current node
 - constructor
 - `hasNext()` and `next()` methods

Looping Over a Linked List

explicit use of iterator

```
LinkedList<String> ll = new LinkedList<>();  
// list filled with data  
Iterator iter = ll.iterator();  
while(iter.hasNext()) {  
    System.out.println(iter.next());  
}
```

for-each loop

```
LinkedList<String> ll = new LinkedList<>();  
// list filled with data  
for(String str : ll) {  
    System.out.println(str);  
}
```

Two options for using iterators

- explicitly create and use the iterator
- use a for-each loop
 - can be used with any class that implements Iterable
 - “for each string in the list...”
 - works for arrays too!

Modifying a List While Using an Iterator

“...The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress...” (Java documentation)

- Iterators are best used to loop over and display/process a collection
- `remove()` is an optional (safe!) method for iterator
 - only guaranteed way to modify a structure during iteration
 - will not be covering it this semester
- **do not** add/move/modify elements

Exercise: Building an Iterator

By definition, an iterator should iterate over *every* item in a collection

but, we may want iterators that work in other ways

Write an iterator to return only even indices