Lecture 13: Iterators

11:00 AM, Feb 23, 2021

Contents

Motivating Question

How can we get for-loops to work on our `LinkList` class, the way they do on built-in Java `LinkedLists`?

Objectives

By the end of this lecture, you will be able to:

- write an iterator, which allows a data structure to be traversed with a for-loop. This is important for creating data structures that are easy for other programmers to use.

1 Target Application: A Simple Vote Manager

Assume you wanted to write a small application (embodied in a class) for managing votes (for people, pizza toppings, what have you). Here's what we'd like to write, using our `LinkList` class to actually hold the votes:

```
public class Votes {
   LinkList<String> votes;

   Votes() {
       this.votes = new LinkList<String>();
   }

   // record a vote for the given name
   public void castVote(String forwho) {
       this.votes.addFirst(forwho);
```

```
    }

    // count votes received for the given name
    public int countVotesFor(String forwho) {
        int count = 0;
        for (String name : votes) {
                if (name.equals(forwho)) {
                        count = count + 1;
                }
        }
        return count;
    }
}
```

If we try to compile this code, Java will report an error that says something about `LinkList` and Iterables. That error means that the `LinkList` class is not set up for Java to use `for` loops. Our goal in this lecture is to fix that problem.

2 Iterators: Enabling `for` Loops

To see how to enable `for`-loops, it helps to look at look at the same method written with and without them so we can identify what the `for` loop must be doing under the hood. Here are two versions of a method for checking whether a list `contains` a specific method:

```
public class LinkList implements IList {
    public boolean contains(int elt) {
              Node current = this.start;
              while (current != null) {
                      if (current.item == elt) {
                              return true;
                      } else {
                              current = current.next;
                      }
              }
              return false;
     }
 }
```

```
public class LinkList implements IList {
    public boolean contains(int elt) {
        for(Integer item : this) {
           if (current.item == elt) {
              return true;
           }
        return false;
   }
}
```

Stop and Think: What lines from the `while` version must the `for`-loop be doing automatically? Actually mark the lines.

Big picture, there are really two separate tasks going on in the `contains` code: visiting all the elements in the list, and checking whether the current element is the one we were looking for. Only the **`if`** statement and the `return` are particular to `contains`. The rest (all the stuff about `current`) is just about how to visit/traverse all of the elements.

A `for`-loop handles all of the traversal internally, leaving the programmer to write the problem-specific parts. So if we want to enable someone to write a for loop over a `LinkList`, we must somewhere indicate how to traverse the data structure.

2.1 The Operations Underlying Traversal

Traversal comes down to three operations:

1. Knowing whether we are out of elements (at the end of the list)
2. Retrieving the value of the current element
3. Advancing the traversal to the next element

We can capture these three operations with a `current` variable and two methods that use it. For the moment, assume these are in the `LinkList` class (we'll change that decision shortly though).

```
Node current;

// determines whether we are at the end of the list
public boolean hasNext() {
    return (this.current != null);
}

// returns the current item and advances current to the next item
public T next() {
    // hold onto the current item
    T item = this.current.item;
    // advance current to the next item
    this.current = this.current.next;
    // return the saved current item
    return item;
}
```

If we had these methods in the `LinkList` class, we could rewrite `contains` as follows:

```
public boolean contains(T elt) {
   while (this.hasNext()) {
     if (this.next().equals(elt))
       return true;
   }
   return false;
}
```

Similarly, having this methods should help enable a `for`-loop. But we need to put them in a particular place to make that actually happen.

2.2 Defining Java Iterators

If you want to be able to use a `LinkList` (like `votes`) as the source of items for a for loop, Java needs to have access to the `hasNext` and `next` methods. Rather than leave them directly in the `LinkList` class, however, ,Java requires us to put them in a new class that implements a specific interface for these methods. Here's what that class looks like (lecture capture develops this code and labels it more carefully):

```
class LinkListIterator implements Iterator<T> {
   Node current;

   public LinkListIterator(LinkList<T> theList) {
           this.current = theList.getStart();
   }

   @Override
   public boolean hasNext() {
           return (this.current != null);
   }

   @Override
   public T next() {
           T item = this.current.item;
           this.current = this.current.next;
           return item;
   }
}
```

We put this class *inside* the `LinkList` class (since it doesn't make sense to create objects of it outside the LinkList context. The last step in enabling `for`-loops is to add a method to the `LinkList` class that returns objects from this class:

```
public Iterator<T> iterator() {
    return new LinkListIterator(this);
}
```

This method is required by an interface called *Iterable*, which is what permits for loops over a data structure. We annotate either the `LinkList` class or the *IList* interface to implement Iterable to finish everything up.

The final code is posted on the lectures page.

2.3 Reviewing the steps to create an iterator

This is a brainfull of code, so let's step back. Assume you have a class `DataStruct` that you want to let someone traverse with a for loop. What do you need to do?

1. Make `DataStruct` implement the interface `Iterable<T>`, where `T` is the type of the items inside your data structure (like Integer or String). You can either have `DataStruct` implement the `Iterable` interface directly, or you can have an interface that `DataStruct` implements extend the `iterable` Interface.

2. Put the following import in the file for the `DataStruct` class.

```
import java.util.Iterator;
```

3. Create a class nested inside `DataStruct` for the iterator. The class should look like this:

```
class DataStructIterator implements Iterator<T> {

    @Override
    public boolean hasNext() { ... }

    @Override
    public T next() { ... }
}
```

   Where again, T is the type of the item in the data structure (the same type that you write in the for-loop for each element). You may add additional variables to this class as needed (such as `current` in our `LinkList` class).

4. Add a method named `iterator` to the `DataStruct` class – this will be required since `DataStruct` implements `Iterable`. This method just creates a new iterator object, nothing more.

```
public Iterator<T> iterator() {
    return new DataStructIterator(this);
}
```

5. If you also have an interface for `DataStruct` (like `IList`), that interface needs to include

```
public Iterator<T> iterator();
```

   which means the interface file needs the same import statement as given above.

2.4 Why do we need a separate class for the iterator?

Each object in the iterator class has a copy of the `current` variable. Before we made the iterator class, we had only one `current` variable for each `LinkList` object. Imagine that you are writing an application where different parts might want to iterate over the same data structure at the same time (one part doing lookup while another displays the list contents). Two traversals can't share a `current` variable. The iterator class enables simultaneous iterators over the same data structure.

3 What Should You Take From This Lecture?

We do not expect you to memorize all of the code involved in writing an iterator (most of the staff, including the professor, have to look up the details when we need to do this). Here's what we do expect you to understand:

- Methods over sequences of items generally do two tasks: traverse the data and do the method-specific computation. `for`-loops (or built-ins like `filter` from other languages) help the programmer focus on the latter.
- If a data structure implements the `Iterable` interface, you can use a for loop to traverse it.
- If you are writing a data structure implementation and you want someone else to be able to traverse it with a `for` loop, you have to make the data structure implement the iterable interface.
- Implementing the `Iterable` interface involves creating a class that provides methods `hasNext` and `next`.
- You can always use these notes or another online source to look up the details to actually develop the iterator. You will never be asked to remember these details on a test in this course.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: `https://cs.brown.edu/courses/cs018/feedback`.