Lecture 9: Recap and Iterators

Contents

1 Recap: Where We’ve Been and Where We Are Going .......................... 1
  1.1 What have we done in CS18 so far? ........................................... 1
  1.2 Looking Forward ..................................................................... 2

2 Target Application: A Simple Vote Manager ........................................ 2

3 Iterators: Enabling for Loops ............................................................... 3
  3.1 The Operations Underlying Traversal .......................................... 4
  3.2 Defining Java Iterators ................................................................ 4
  3.3 Reviewing the steps to create an iterator ...................................... 5
  3.4 Why do we need a separate class for the iterator? ......................... 6

4 What Should You Take From This Lecture? ......................................... 6

Motivating Question

How can we get for-loops to work on our LinkedList 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 Recap: Where We’ve Been and Where We Are Going

1.1 What have we done in CS18 so far?

OO Code Organization

- Learned that Java (as with other OO languages) organizes code into classes, where a class corresponds to a type of user-defined data that may have fields/components. All functions that operate on that type live in that class; these are called methods.
• Learned that objects are concrete values created from a class. Objects are created with the `new` construct.

• Learned that we can share common code across classes putting it in a new class that each of the original classes `extends`. The idea of one class extending (getting the fields and methods of) another is called inheritance. The extended class is called the parent class.

• Learned that when we need a field or parameter that whose value can be from one of several classes, we create an `interface`. An interface lists the names and input/output types of methods that every class that implements the interface must provide.

Programming for Sharing and Mutating Data

• Learned how to update fields in objects, and how that affects the environment and the contents of memory.

• Learned how to arrange and name objects so that changes through one name are visible through the other. This is essential for sharing data across different parts of a program.

• Learned about `for` and `while` loops, the constructs for iterating over lists (and other data structures, as we will see). While these aren’t about sharing or mutating, languages that allow you to change the values of fields and variables (known as imperative languages) typically use these loops to traverse data structures whose items have a linear sequence.

Implementing Data Structures

• Practiced implementing data structures from CS 17/19/111 as Java classes. We’ve done binary search trees and lists so far.

• Looked at two implementations of lists: a `NodeList` class that resembles lists from the fall semester but doesn’t enable sharing updates across names, and `LinkList` which does enable sharing.

1.2 Looking Forward

Now that we have different implementations of lists, we want to work towards three goals:

1. Writing code in such a way that we could easily change which list implementation we are using down the road. This can be valuable if your data grows to a point that you need to make certain operations faster or more space efficient without changing your application code.

2. Implementing data structures in such a way that they enable being swapped out (the complement to the first goal, if you are the one building the data structure implementation that someone else will use).

3. General OO coding practices for providing data structures and types that will be used by others.

These three goals will be the theme of this week.
2 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:

```java
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;
    }
}
```

Take a look – what aspects of this does our LinkList and IList code from the last lecture still need to support?

- Our LinkList has integer elements, but here we want Strings.
- Java gives as error if we try to write a for loop over our class (something about Iterables, whatever those are ...)

The notes from the last lecture (though not lecture itself) showed how to let a class take types as parameters. So we’ll assume we’ve fixed the first issue and focus instead on the second.

3 Iterators: Enabling for Loops

To see how to enable for-loops, it helps to go back to the contains method we wrote last class:

```java
class LinkList implements IList {
    public boolean contains(int elt) {
        Node current = this.start;
        while (current != null) {
            if (current.elt == elt) {
                return true;
            }
            current = current.next;
        }
        return false;
    }
```
There were 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 is particular to contains. The rest is just about how to visit/traverse all of the elements.

When we look at a for-loop, it similarly handles all of the traversal internally, and focuses on the computation for the task at hand. 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.

### 3.1 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).

```java
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:
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.

3.2 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):

```java
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:

```java
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.
3.3 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 that extends the `Iterable` Interface.

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

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

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

   ```java
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.

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

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

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

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

3.4 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.
4 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:

- 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.

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](https://cs.brown.edu/courses/cs018/feedback)