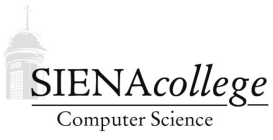


Computer Science 225
Advanced Programming
Siena College
Spring 2022

Topic Notes: List Structure Design

We have looked at the `List` interface in the Java API, and some implementations of it. We will now consider the design and implementation of a set of list structures in Bailey's structure package, with a focus on how it makes use of Java's object-oriented programming constructs.

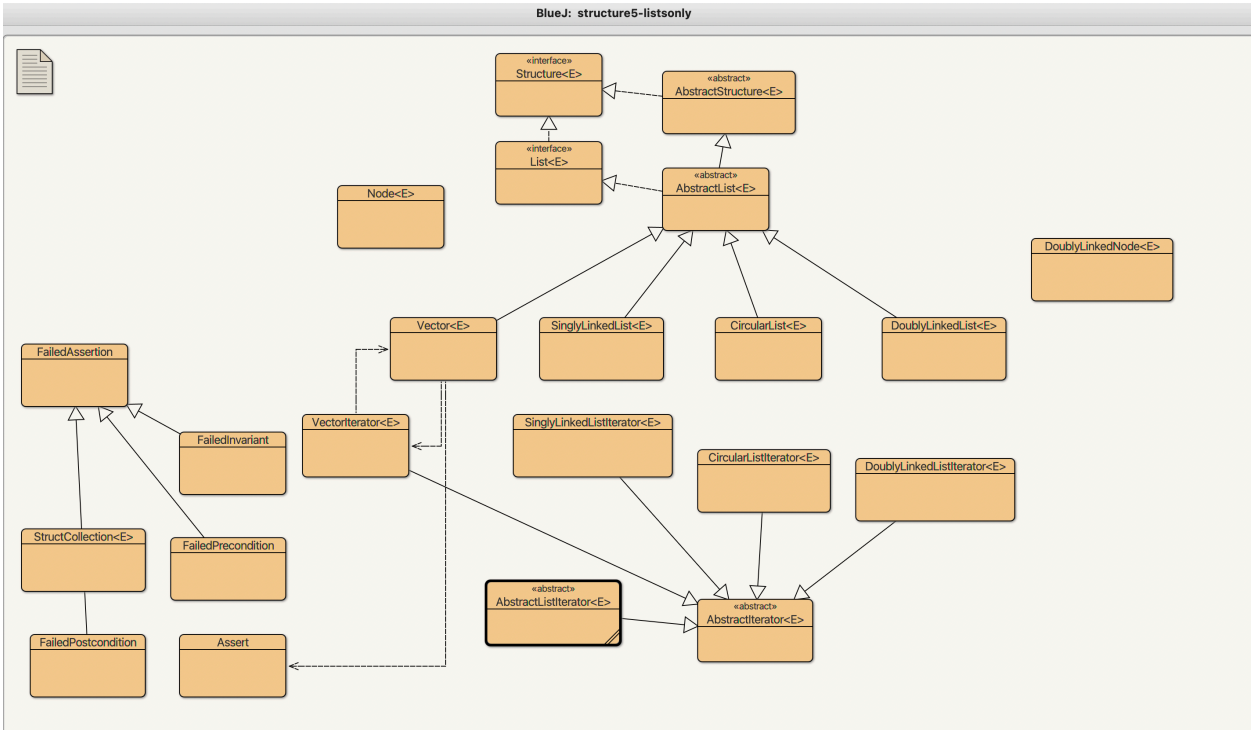
The structure package provides implementations of many standard data structures. Many are very similar in functionality to those provided by the Java API, but have full source code available.

Specifically for list structures, the package provides the following concrete classes:

- `Vector`, which is analogous to Java's `ArrayList`
- `SinglyLinkedList`
- `CircularList`
- `DoublyLinkedList`, which is analogous to Java's `LinkedList`

Our focus here will be on the way these are implemented, and the design decisions that were made to define their common functionality through interfaces and where some common functionality is factored out into an abstract class.

Bringing just the classes relevant to lists into BlueJ, we can see the interfaces, abstract classes, and concrete classes:



Interfaces

The structure package has a set of interfaces that define common features of the data structures it implements. Those relevant to list data structures are the `Structure` interface and the `List` interface.

`Structure` defines common functionality that many data structures, including lists, share. Ignoring things like iterators (which are an interesting design case study in their own right), these are:

```
public int size();
public boolean isEmpty();
public void clear();
public boolean contains(E value);
public void add(E value);
public E remove(E value);
```

The `List` interface adds functionality that lists provide that more general data structures do not. What distinguishes lists as a particular class of data structures? They are ordered by index, so a number of operations that only make sense when there's such an ordering can be defined. The following are added by structure:

```
public void addFirst(E value);
public void addLast(E value);
```

```
public E getFirst();
public E getLast();
public E removeFirst();
public E removeLast();
public E remove();
public E get();
public int indexOf(E value);
public int lastIndexOf(E value);
public E get(int i);
public E set(int i, E o);
public void add(int i, E o);
public E remove(int i);
```

Abstract Classes

One of the uses of abstract classes is to implement methods in terms of others. The abstract classes relevant to lists are `AbstractStructure` and `AbstractList`.

The `AbstractStructure` class uses only methods defined in the `Structure` interface. Some examples:

```
public boolean isEmpty() {
    return size() == 0;
}

public boolean contains(E value) {
    Iterator<E> i = iterator();
    while (i.hasNext()) {
        if (i.next().equals(value)) return true;
    }
    return false;
}
```

There is also a default version of the `hashCode` function.

In `AbstractList`, these are some of the implementations provided:

```
public void addFirst(E value) {
    add(0,value);
}

public void addLast(E value) {
    add(size(),value);
}
```

```
    public E getFirst() {
        return get(0);
    }

   public E getLast() {
        return get(size()-1);
    }

    public E removeFirst() {
        return remove(0);
    }

    public E removeLast() {
        return remove(size()-1);
    }

    public void add(E value) {
        addLast(value);
    }

    public E remove() {
        return removeLast();
    }

    public E get() {
        return getLast();
    }

    public boolean contains(E value) {
        return -1 != indexOf(value);
    }
```

Having many of these defined allows the concrete list implementations to avoid the need to define their own versions of these. Though they can do so if an implementation can be more efficient than the generic version defined in the abstract class.

Concrete Classes

The four concrete list classes must define remaining `List` and/or `Structure` methods that were not provided by the abstract classes, and will override some that are provided.

The methods from the interfaces that are not provided at all in the abstract classes:

```
    public int size();
```

```
public void clear();
public E remove(E value);
public int indexOf(E value);
public int lastIndexOf(E value);
public E get(int i);
public E set(int i, E o);
public void add(int i, E o);
public E remove(int i);
```

Vector

Remember, the `Vector` class defines an array-based list, much like the Java API's `ArrayList` and `Vector` classes.

We will not look at every method, but some of note:

```
public void add(E obj) {
    ensureCapacity(elementCount+1);
    elementData[elementCount] = obj;
    elementCount++;
}

public boolean contains(E elem) {
    int i;
    for (i = 0; i < elementCount; i++) {
        if (elem.equals(elementData[i])) return true;
    }
    return false;
}
```

These versions of the `add` and `contains` methods override the ones in `AbstractList`, with code that is a bit more efficient because they access the `Vector`'s internal array directly.

The `remove` "by element" method is written in terms of other methods:

```
public E remove(E element) {
    E result = null;
    int i = indexOf(element);
    if (i >= 0) {
        result = get(i);
        remove(i);
    }
    return result;
}
```

The `size` method is implemented in a very straightforward manner, using the element count:

```
public int size() {
    return elementCount;
}
```

`SinglyLinkedList`

The singly-linked list implementation is very similar to the `SimpleLinkedList` we have worked with.

While the `addFirst` method has a version provided in `AbstractList`, the `SinglyLinkedList` defines its own that is a bit more efficient:

```
public void addFirst(E value) {
    head = new Node<E>(value, head);
    count++;
}
```

Similarly for methods like `addLast`, `getFirst`, `getLast`, `removeFirst`, and `removeLast`. Versions are provided that override the ones in `AbstractList`. This also means that the general-purpose `add` method can use these to handle special cases:

```
public void add(int i, E o) {
    Assert.pre((0 <= i)  && (i <= size()),
               "Index in range.");
    if (i == size()) {
        addLast(o);
    } else if (i == 0) {
        addFirst(o);
    } else {
        Node<E> previous = null;
        Node<E> finger = head;
        // search for ith position, or end of list
        while (i > 0) {
            previous = finger;
            finger = finger.next();
            i--;
        }
        // create new value to insert in correct position
        Node<E> current = new Node<E>(o,finger);
        count++;
        // make previous value point to new value
```

```
            previous.setNext(current);
        }
    }
```

Note also that this is a counted list, so it can have an efficient `size` method:

```
public int size() {
    return count;
}
```

The `clear` method can also be very simple (though in the end, Java's garbage collector will have some work to do to clean up all of the list nodes):

```
public void clear() {
    head = null;
    count = 0;
}
```

We will not look in detail at `CircularList` or `DoublyLinkedList`, as the design decisions are similar. But of course, some operations can be implemented more efficiently.

Some key points from this case study:

- The interfaces define the common functionality that allow the lists to be used by programs interchangeably.
- The abstract classes define some methods in terms of others, so the concrete classes do not need to do so.
- The concrete classes fill in remaining details to satisfy the interfaces, and override methods defined by the abstract classes.