Computer Science 210
Data Structures
Siena College
Fall 2017

Topic Notes: Linked Structures

Recursive Data Structures

We have seen that constructs such as arrays and `Vectors`/`ArrayLists` allow us to group together collections of elements using a single name. These work very well in many situations, but are not always going to be the most efficient option. Consider a situation where you often need to add and remove elements from near the start of a large `Vector`/`ArrayList`. Each one of those operations, for a `Vector`/`ArrayList` of n elements, is an $O(n)$ operation. If that is something we will need to do frequently, we will want to do better.

Recursion affords us another mechanism to store collections of elements. We begin this discussion with a custom class that can hold an arbitrary number of our `Ratio` objects.

See Example: RatioListApplet

This example looks much more complex than it really is, as much of the code supports the Swing GUI interface.

There are three classes here, and we will examine them one at a time.

First, the `Ratio` class is the one we saw earlier in the course, but with some of the verbose comments stripped out and a few methods added. One of these is the non-destructive `add` method you wrote for an earlier assignment.

One of a bit more interest is the `reduce` method. Hopefully you recall that to reduce a fraction to lowest terms, you find the greatest common divisor (GCD) (sometimes called the "greatest common factor") of the numerator and the denominator – the largest number that divides both evenly – then divide both the numerator and denominator by that GCD. One method to compute a GCD is called *Euclid's Method*, and that is what is implemented by the `gcd` method in the `Ratio` class. This is a recursive method!

The first line of the method is the base case: the GCD of any number with 0 is that other number. The second line is just a way to swap the order of the parameters when the first is larger than the second. The third line is the recursive case that applies Euclid's algorithm. The key thing we need to notice here is that the parameters will always become smaller on each recursive step. Eventually, the base case will come into play.

The `RatioListApplet` class implements a kind of "ratio list calculator" program. The idea is that we have a display area that can show a single ratio at any given time. We can type in numbers to change that ratio, we can reduce that ratio to lowest terms, we can store the ratio in the display in a list of ratios, we can compute and display results of a few operations on the list of ratios (the sum, the min, and the max), and finally we can reduce all ratios in the list to lowest terms.

The vast majority of the code in `RatioListApplet` constructs and manages the GUI. We are most interested in its use of the other class in this project, the `RatioList` to keep track of the list of ratios. Before we look at its implementation, let's see how it's used.

The `RatioList` instance variable starts out as `null`, indicating that there are no ratios in the list. New ratios are added to the list in the first part of the `actionPerformed` method. The construction is different from those we have seen before:

```
ratios = new RatioList(newOne, ratios);
```

where `newOne` is a `Ratio` just constructed from the values in the text fields of the display.

Think about what we see here: the constructor for the `RatioList` takes a `RatioList` as a parameter. We then replace our reference to the `RatioList` that we passed in with the new one that was just returned.

Let's shift our attention to the `RatioList` class to see how this constructor is implemented and what instance variables we find there. To this point, when we've wanted to store a collection of objects, we have used arrays and `ArrayLists`, but we find neither of those in `RatioList`. Instead, we have an instance variable to hold a single `Ratio` object, and another to hold another `RatioList`. Just like our recursive methods use themselves to as part of the solution, this class has another instance of itself as an instance variable: it is a *recursive structure*. How can this be? It's a similar idea to a recursive method, where eventually we get to a base case. Here, we eventually get to a situation where the `rest` instance variable is `null`.

Hopefully this will make more sense as we continue looking at this. We start with the constructor. Like many of our constructors, this one simply remembers its parameters in instance variables.

Consider what happens when a series of calls to the constructor is made to add `Ratio` values to our `RatioList`. (Which we will work through in class.)

Once we have seen how a `RatioList` is built up, we can consider some of the other methods. Let's start with `getMin` (and its very close cousin, `getMax`).

We want to be able to retrieve the smallest value from a `RatioList` object. Remember that an object needs to be able to answer a question like this using only the information in the provided parameters and its instance variables. Here, all we have are the two instance variables: the `Ratio` called `first` and the `RatioList` called `rest`.

Given this situation, there are two main possibilities:

1. `rest` is `null`, which means the only `Ratio` in this `RatioList` is the one in `first`, so that must be the smallest one.
2. `rest` is not `null`, in which case there is at least one other `Ratio` contained in `rest`'s instance variable, possibly more. So here, we ask `rest` what its smallest value is (after all, we're writing a method to do that), and compare that to the value we have here in `first`. The smaller of those two `Ratios` must be the smallest in the whole list!

That's exactly what's done in the code. We have a recursive method operating on our recursive data structure. As with our recursive methods before, we can identify the base case: when `rest` is `null`, and the recursive step: when we call `rest.getMin()`.

With previous recursive methods, we had to make that each recursive call would get us closer to the base case. The same is true here. As long as we have constructed our `RatioList` properly, each recursive call gets us closer to the subsequent `RatioList` that has a `null` value for its `rest`.

For `getMin` and `getMax`, we are essentially doing a search. For `toString` and `getSum`, we are visiting all of the values and building up a result.

The `getSum` method computes the sum of the entire `RatioList`'s `Ratios`. It does so by determining whether there is anything in `rest`. If not, the sum is trivial: it's `first`. Otherwise, it's the sum of `first` with the result of our recursive call to `getSum`.

The `toString` method works similarly, but instead of adding `Ratios` together to accumulate the sum recursively, we concatenate the `String` representations of each `Ratio` object returned by its `toString` method.

Finally, `reduceAll` doesn't accumulate any result or search, it modifies each `Ratio` in the list. It also does so recursively. It reduces the `first` to lowest terms, using `Ratio`'s `reduce` method. Then, if the `rest` is not `null`, it makes a recursive call to reduce the rest.

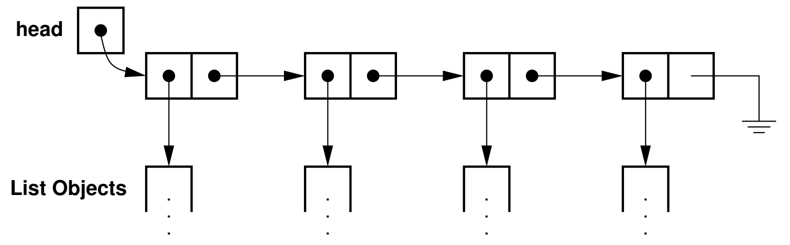
General Purpose Recursive Structures

So far, all of our structures for holding general-purpose collections of items have been very simple. We've used only arrays and `Vectors`/`ArrayLists`. These have some pretty significant limitations. `Vectors`/`ArrayLists` are resizeable, but it is an expensive operation. It's also expensive to add or remove objects from the start or the middle of a `Vector`/`ArrayList`. We can do better.

We will begin our study of more advanced data structures with *lists*. These are structures whose elements are in a *linear* order.

Singly Linked Lists

The *singly linked list* structure is built using a collection of *list nodes*. Each list node is responsible for keeping track of one *list element*. The list elements are the values we actually want to store and retrieve in our data structure.



We will study this idea by looking at a simple implementation of generic singly linked lists.

See Example: SimpleLinkedList

The list implementation is in `SimpleLinkedList.java`, and a `main` method we will use to learn how this list works is in `NotesExample.java`. As we trace through that `main` method, we will look at the details of the implementation and construct a series of memory diagrams (drawn on the board in class).

Before we start tracing through our example, we will look at the two main object types that represent a list node and the list itself.

The structure that makes up a list node has two fields:

1. `value`: the element (whose type here is specified by a type parameter) to be stored in the list.
2. `next`: a pointer to the next list node, which will be `null` for the node containing the last element.

```
class SimpleListNode<E> {

  protected E value;
  protected SimpleListNode<E> next;
}
```

And the list structure itself is no more than a reference to the list node that contains the first list element.

```
public class SimpleLinkedList<E> {

  protected SimpleListNode<E> head;
}
```

We define `SimpleListNode` in the same Java file as `SimpleLinkedList`, but `public` is not specified in the class header for `SimpleListNode`, since we aren't allowing regular users to create instances of these. Only a `SimpleLinkedList<E>` can create a `SimpleListNode`. Note that the Java file's name needs to match the `public` class it defines.

The user who wishes to use a `SimpleLinkedList` would start by constructing an empty list, like we do at the start of our `main` method in `NotesExample`:

```
    SimpleLinkedList<Double> dlist = new SimpleLinkedList<Double>();
```

As with any class, this results in a call to a constructor. All the constructor needs to do in this case is to set the `head` instance variable to `null`.

```
  public SimpleLinkedList() {
    head = null;
  }
```

Next, we'll work on adding elements. Adding an element involves two steps:

1. construct a new list node for the element
2. insert the new list node into the list

Let's think about what this will mean. We add our first element, in our example, a 17.1.

```
dlist.add(17.1);
```

We want this list to go from just an empty `head` reference, to a node pointed at by `head` which has a reference to the 17.1 as its `value` and `null` as its `next`.

Our `SimpleListNode` has two `add` methods. The one-parameter version just calls that two-parameter version with a position parameter of 0. So that's the method we'll look at.

```
public void add(int pos, E obj) {...}
```

The method first does a little error checking. Turns out any add at position 0 is the same in our implementation, regardless of whether the list is empty. The code below takes care of this caee.

```
if (pos == 0) {
   head = new SimpleListNode<E>(obj, head);
   return;
}
```

We replace the list's `head` reference (which is `null` in this case) with a reference to a new list node. The constructor for `SimpleListNode` is straightforward enough, just storing the two parameters (the value and the `next` reference) in the node's instance variables:

```
public SimpleListNode(E value, SimpleListNode<E> next) {
    this.value = value;
    this.next = next;
}
```

In this case, the node's `next` instance variable is set to `null`, as the previous value of the list's `head` reference.

Now, we add another element at the start of the list.

```
dlist.add(23.0);
```

The same code is used for this case, but now the list's `head` reference starts as a reference to the list node responsible for the 17.1. That reference becomes the `next` of the new node we're creating for 23.0, which itself becomes the new `head` of the entire list.

The situation becomes a bit more interesting when we want to add a position other than 0.

```
dlist.add(1, -3.5);
```

To achieve this, we need to find the list node after which a new node will need to be inserted to store the new element, and have that new node's `next` reference point to the rest of the existing list beyond that point.

```
// we are adding somewhere else, find entry after which we will
// insert our item
int i = 0;
SimpleListNode<E> finger = head;
while (i < pos-1) {
    i++;
    finger = finger.next();
    if (finger == null) {
        throw new IndexOutOfBoundsException("Attempt to add at position " + pos
    }
}
// finger points at the node after which we want to add
// so create the new object with finger's next as its next
// and set finger's next to the new node.
// note that this also works for the case when we are adding
// to the end
finger.setNext(new SimpleListNode<E>(obj, finger.next()));
```

This makes use of trivial accessor and mutator methods that allow us to retrieve and modify a list node's `next` field.

The loop here follows `next` references, counting how many nodes we have "skipped over". Once we have skipped `pos-1` nodes, our `finger` reference points at the node after which we need to insert. Note that if the `finger` ever becomes `null`, it means the `pos` at which we were asked to insert a value is too large given the current contents of the list.

Once we know where we are inserting, the last line of code in the method accomplishes exactly what we need:

1. A new list node is created to store the element we are adding,
2. that new list node's `next` reference is set to the `next` reference of the "finger" node, and

3. the “finger” node’s `next` reference is set to the new list node we just created.

Turns out this same code works when we add at the end of the list:

```
dlist.add(3, 1.1);
```

The only difference is that `finger`’s `next` reference will be `null`, resulting in the new node’s `next` reference being `null`, which is exactly what we want for the new last node in our list.

Now that we can construct and populate `SimpleLinkedList`s, we consider the `get` method. Like with corresponding method of `Vector`/`ArrayList`, it can retrieve the element at any position in the list.

Our method call

```
double val = dlist.get(2);
```

results in a call to the method:

```
public E get(int pos) {

    // negative positions not allowed
    if (pos < 0) {
        throw new IndexOutOfBoundsException("Attempt to get from a negative pos
    }

    if (head == null) {
        throw new IndexOutOfBoundsException("Attempt to get from an empty list'
    }

    SimpleListNode<E> finger = head;
    int i = 0;
    while (i < pos) {
        i++;
        finger = finger.next();
        if (finger == null) {
            throw new IndexOutOfBoundsException("Attempt to get element " + pos
        }
    }
    return finger.value();
}
```

Other than some error checking, the code is strikingly similar to add. The only difference here is that once we’ve followed the correct number of `next` references to find the list node that contains the element we are trying to retrieve, we simply access that value (with the `value` accessor method of `SimpleListNode`) and return it.

The next operation used by our `NotesExample` `main` method is `set`:

```
dlist.set(1, -7.0);
```

The code for the `set` method (not reproduced here) is almost identical to `get`. Instead of returning the value at the desired position, we just set it to the new value and return the old value.

We next have a couple of calls to `contains`

```
boolean hasVal = dlist.contains(23.0);
System.out.println("Has a 23.0? " + hasVal);
hasVal = dlist.contains(0.5);
System.out.println("Has a 0.5? " + hasVal);
```

We again search through list nodes, but instead of counting how many we have skipped over to find particular one, we check the value in each node until we either find the one we're searching for (in which case the method returns `true`), or reach the end of the list, indicated by the "finger" becoming `null` (in which case the method returns `false`).

The basic structure is the same as `get` and `set`.

```
public boolean contains(E obj) {

    // easy when the list is empty
    if (head == null) return false;

    // otherwise look for it
    SimpleListNode<E> finger = head;
    while (finger != null) {
        if (finger.value().equals(obj)) return true;
        finger = finger.next();
    }
    return false;
}
```

Next up, we call `size`, which is also pretty straightforward. Here, we traverse the list nodes, counting how many we see. When the `finger` becomes `null`, we are out of list nodes to count, and return that count.

```
public int size() {
    SimpleListNode<E> finger = head;
    int count = 0;
    // count up the number of list nodes until we get a null next
    while (finger != null) {
        count++;
        finger = finger.next();
    }
    return count;
}
```

That code is easy enough, but it is quite inefficient ($O(n)$ for an n-element list). More on that later.

Now, let's consider a harder one: `remove()`. In general, we will want to be able to remove items by value or by index. We'll just implement by index.

There are a number of cases that we need to make sure are considered, though some of the more specific might end up being taken care of by general cases:

1. remove the only item from a list (in which case it also was the first and the last)
2. remove the first item in a list from a list with at least two elements
3. remove the last item in a list from a list with at least two elements
4. remove an item from the middle of a list from a list with at least two elements

Our running example in `NotesExample` tests all of these cases.

Let's see how the `remove` implementation handles these.

```
public E remove(int pos) {
```

After some error checks, we can take care of the "first item" case, where it turns out not to matter if it is also the only item.

```
if (pos == 0) {
    E retval = head.value();
    head = head.next();
    return retval;
}
```

The list's `head` reference gets replaced with the `next` reference of the first node (which contains the element we wish to remove). Simple enough.

In other cases, we need to find the item we want to remove and adjust some references to "bypass" the node that contains the element we're removing.

The key idea here is that we need to have our "finger" on the element *before* the one we want to remove, since that's the one whose `next` pointer will need to be adjusted.

```
    // remove an item at a non-first position
    SimpleListNode<E> finger = head;
    int count = 0;
    // find the item before the one we want to remove
    while (count < pos-1) {
        count++;
        finger = finger.next();
```

```
            if (finger == null) {
                throw new IndexOutOfBoundsException("Attempt to remove element at i
            }
        }
        // finger is pointing to item pos-1
        // make sure there is something at pos
        if (finger.next() == null) {
            throw new IndexOutOfBoundsException("Attempt to remove element at index
        }
        E retval = finger.next().value();
        finger.setNext(finger.next().next());
        return retval;
```

Removing everything is very simple.

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

What about all those list nodes? We still have references to them! Not to worry, Java's garbage collector will clean them up.

However, not all languages are garbage collected like Java. In C or C++, you need to be careful to `free` (in C) or `delete` (in C++) all of the objects you no longer need.

Let's consider the complexity of our operations.

- `add(0)` : $O(1)$
- `add(i)` : $O(i)$
- `add(n)` : $O(n)$
- `get/set(0)` : $O(1)$
- `get/set(i)` : $O(i)$
- `get/set(n-1)` : $O(n)$
- `remove(0)` : $O(1)$
- `remove(i)` : $O(i)$
- `remove(n-1)` : $O(n)$
- `get` all values in sequence : $O(n^2)$ (hey, we need an `Iterator`!)
- `size()` : $O(n)$ (hey, we can do better if we remember this)

How do these compare to similar operations on `Vectors`/`ArrayLists`?

- adding at the front is easier.
- adding at the end is harder.
- adding in the middle, well it depends where.
- the cost is consistent, though, since there is no reallocation and copying to grow the structure.
- removing at the front is easier.
- removing at the end is harder.
- removing in the middle is probably similar.
- getting/setting an arbitrary value is harder.

What about space usage?

- there are no empty slots like we have in `Vectors`
- but there's an extra reference for each object stored! That's $O(n)$ space overhead.

We still have a couple of problems with this implementation that we'd like to address. First, the $O(n^2)$ traversal is no good – we need an `Iterator`.

Remember, an iterator must remember some state about the collection it's visiting. With our `Vector` iterator example, we just needed to remember the index of the next item to be returned. Remembering the index doesn't help us here. We need to remember something about the internals of the list to make this work. The most useful thing to remember here is the list node – that "finger" we used in most of the methods we've looked at.

We'll implement our iterator as an extension of the structure package's `AbstractIterator` abstract class:

```
class SimpleListIterator<E> extends AbstractIterator<E> { ...
```

This is not a public class, since no one except our `SimpleLinkedList` is allowed to construct one.

We need to have data to support the regular iterator operations, plus be able to reset the iterator, so we need to have our iterator remember the head of the list and the "finger":

```
    protected SimpleListNode<E> current;
    protected SimpleListNode<E> head;
```

and to construct one, we need to have the head of the list passed in:

```
    public SimpleListIterator(SimpleListNode<E> t) {
        head = t;
        reset();
    }
```

To reset, we just set the current to `head`.

```
    public void reset() {
        current = head;
    }
```

So the `current` pointer always points to the next node whose value has *not yet been returned*. From this, we can construct the remaining methods:

```
    public boolean hasNext() {
        return current != null;
    }

    public E next() {
        E temp = current.value();
        current = current.next();
        return temp;
    }

    public E get() {
        return current.value();
    }
```

And in the `SimpleLinkedList` class, we have a method to create one:

```
    public Iterator<E> iterator() {
        return new SimpleListIterator<E>(head);
    }
```

We also make our `SimpleLinkedList` implement `Iterable` so we can use it in enhanced for loops.

Maintaining an element count

We can improve the efficiency of the `size()` method by maintaining an extra instance variable that tracks how many elements are in the list.

Which methods would need to change to do this?

- `count` needs to be initialized in the constructor
- increment `count` in `add`
- decrement `count` in `remove`
- reset `count` to 0 in `clear`
- simplify `size` to return `count`

This seems worthwhile. We've added some work to our methods, but it's all $O(1)$, and we only added a single `int` to the size of the structure. The biggest disadvantage of adding a count is that we could forget to update it in some circumstance, leading to an inconsistent structure. For example, there are three different cases in the `add` method, and we need to make sure we increment the count in each.

This is exactly what we find in the structure package's `SinglyLinkedList` implementation.

See Structure Source:
`structure5/SinglyLinkedList.java`

Maintaining a `tail` pointer

There's another enhancement we can consider. Adding elements to the end of the list is an operation we might like to make as efficient as possible. It's an $O(n)$ operation in this implementation because we need to follow all the links from the head to find the last element, so we can add it.

We can fix that by maintaining another reference to the tail of the list. Some things to note about such an implementation:

- An empty list has both `head` and `tail` as a `null` reference , a list with one element has both `head` and `tail` as a reference to that one element's list node, while in all other cases, `head` and `tail` refer to different list nodes.
- Our `add` operation in the final position is straightforward (the new node is stored at the `tail` node's `nextElement` and the `tail` is updated to refer to the new node. This is $O(1)$.
- The `tail` reference makes a `get` or `set` of the last element an $O(1)$ time operation.
- But a `remove` from the end is still $O(n)$. We need `tail`'s predecessor to be able to remove the last item, and we have to search all the way from the beginning to find it.

Still, this seems like a worthwhile enhancement. We add just one extra reference and have some efficiency benefits.

That said, it adds coding complexity to the `add` and `remove` methods since we must worry about resetting the `tail` field. Even adding at the beginning may have to reset `tail` field (think about why).

Circular Lists

Can we simplify things further without changing the "Big-O" behavior by noticing that tail node of list has a `nextElement` field that is always "wasted" – it is always equal to `null`!

If we use that field to point to the beginning of the list, then we don't need a separate `head` field.

This is a *circularly linked list*. The `head` is always found as `tail.nextElement()`!

An implementation of this is in the `CircularList` in the structure package.

See Structure Source:
`structure5/CircularList.java`

- With this implementation, `add` at the end is $O(1)$.
- But now it takes one extra dereference (*i.e.*, following a reference) to get to `head`.
- Only `contains`, `remove`, and `removeLast` are still $O(n)$.
- `contains` and `remove` involve searches and seem likely always to be $O(n)$ (unless we attempt to keep list in order and do binary search - which has its own problems - we'll consider this later..).
- However, why can't we make `removeLast` $O(1)$? The problem is that we need to know the predecessor in order to delete an element from the list.

Doubly linked lists

In our implementations so far, references only go from the front to the back of the list. Why not put them in the other direction instead? Well we could, and remove from the end would be fast, but then it would be harder to delete from the front.

So... we can put references in both directions.

This is a more significant change: our list nodes now need more information. We need an extra field to hold a reference to the previous node, but the space overhead remains $O(n)$.

```
class DoublyLinkedNode<E> {

  protected E data;
  protected DoublyLinkedNode<E> nextElement;
  protected DoublyLinkedNode<E> previousElement;
}
```

With this defined, it is now easy to define a *doubly linked list*. This time we'll keep track of both the first (`head`) and last (`tail`) elements of the list so we can get to `tail` quickly.

This is the `DoublyLinkedList` in the structure package.

See Structure Source:
`structure5/DoublyLinkedList.java`

Note that the constructor for `DoublyLinkedNode`s automatically sets back pointers to maintain consistency.

`removeLast` is now $O(1)$, but the tradeoff is that now all addition and removal operations must set one extra pointer in the list node. We must also worry about maintaining both the `head` and `tail` of the list, with somewhat more complicated cases arising when adding and removing from a 0-, 1-, or 2-element list.

Design of List Classes

We now have a number of ways to implement linked list structures. To some extent, these are all interchangeable functionally. We can add, retrieve, remove, search, though there are time and space tradeoffs involved.

We would like to be able to use them interchangeably. That is, if we write code that uses a `SinglyLinkedList` and determine that we want instead to use a `DoublyLinkedList` or even a `Vector`, we should be able to switch without rewriting our code. After all, each of these classes has substantially the same methods available.

This ability is provided by Java's interfaces and abstract classes.

We start by defining an interface that we'll use for a variety of implementations of lists:

- Accessors: `isEmpty()`, `size()`, `get()`, `contains()`, `iterator()`
- Mutators: `add()`, `set()`, `remove()`, `clear()`

See Structure Source:
`structure5/List.java`

First notice that it extends `Structure`. This means a `List` requires the basic operations we expect on any of our structures.

See Structure Source:
`structure5/Structure.java`

The text has a simple example of reading in successive lines from a text and adding each line to the end of a list if it doesn't duplicate an element already in the list. This is easily handled with the operations provided.

`AbstractList`**s**

The `List` interface requires a lot of methods, many of which will be the same in all of the implementations. So the structure package defines an abstract class for this:

See Structure Source:
`structure5/AbstractList.java`

Then each of our actual list implementations `extends AbstractList`, and needs only to fill in the methods not already provided.

For example, the `isEmpty` implementation in `AbstractList` is written in terms of the `size` method, so individual implementations need not provide their own `isEmpty`.

`Vector` as a `List`

Given this, we can imagine another implementation of the `List` interface.

`Vector` already provides all of the methods required by the `List` interface. In the structure package, `Vector extends AbstractList`.

So we can use a `Vector` as a "list" as well. Some of the operations are more expensive, but anywhere we want a `List`, we can use a `Vector`.

This leaves 4 classes that implement structure's `List` interface (all by extending `AbstractList`) in the structure package:

- `SinglyLinkedList`
- `CircularList`
- `DoublyLinkedList`
- `Vector`

You should understand how each of these structures work, know how to use them correctly, should be able to develop the internals of any of the methods in these classes, and should understand the time and space complexities of the implementations.

Lists in the Java API

Java's builtin API provides a variety of generic classes that perform list-like functionality. Many of these can be used interchangeably because they implement the `java.util.List<E>` interface.

The `List` interface defines a set of common operations that a number of API classes provide. Programmers can also write their own classes that satisfy the `List` interface. As long as a user of a class that implements the interface restricts his or her usage to the methods specified in the interface, different implementations can be swapped in with the only change being in the construction of the object that implements `List`. Consider this example:

See Example: ListInterfaceDemo

We create lists of three of the types that implement the `List` interface and then use them in various ways. The main thing to notice here is that the name of the actual type of the list being used (in this case, `ArrayList`, `Vector`, and `LinkedList`) appears only when constructing each list. We could change any of those to any other type that implements the `List` interface, and the remaining code would continue to work unchanged.

Note that the Java API's `java.util.LinkedList` class provides a doubly linked list implementation.

Recall that different underlying structures do different things more or less efficiently than others, so depending on which operations we expect to use, it might make more sense to use one structure over another. But by writing as much of our code using only the methods provided by an interface common to the options, we can switch among actual implementations later with minimal effort!