# CMPSCI 187:Programming With Data Structures

Lecture #21: Lists in the Collections Framework
28 October 2011

# Lists in the Collections Framework

- L&C's List Classes versus the Real Java Classes
- Abstract Classes and Methods
- `AbstractCollection` and `AbstractList`
- The `Vector` and `ArrayList` Classes
- The `LinkedList` Class

# List Classes: L&C Versus Real Java

- L&C's pedagogical mission is to show you what lists are and how they work.
- As with stacks and queues, they define ADT's that say what methods make a list a list, and what those methods do. We've now seen their `ListADT`, `OrderedListADT`, and `UnorderedListADT` interfaces, reflection the distinction between lists that are kept sorted and lists that are not.
- They then define classes implementing these interfaces -- we have seen `ArrayList`, `LinkedList`, and DoubleList. (Note that the first two are different from the classes in Collections with the same names.) In the last two lectures we've seen several details of these implementations.
- But lists in the Collections framework of the actual Java library are quite different. As a working programmer, you are more likely to use Collections.

# Lists in the Collections Framework

- The designers of Java wanted industrial-strength, highly flexible data structures that could be used by a variety of programmers for a variety of purposes. They were concerned that their work be understandable and clear, but not specifically that it could be taught easily to undergraduates.

- Like L&C, they came up with a single linked data structure to implement lists, called `LinkedList`. While L&C have a single array-based data structure, they have two: `ArrayList` and `Vector`. (These two are quite similar to each other.)

- All three library classes implement **indexed** lists, where the user can refer directly to the i'th element in the list for any number i.

- Our knowledge of L&C's implementations will usually be enough to let us determine the big-O running time of operations in the library classes.

# Abstract Classes and Methods

- The Java library list classes sit at the bottom of a hierarchy of several interfaces, classes, and abstract classes. This is because inheritance is used at every point -- methods are written as high in the hierarchy as possible so that one piece of code may serve many different classes.

- We have interfaces at the top of the hierarchy and classes at the bottom. In between are some **abstract classes**. An abstract class (see L&C Appendix B.14, pages 506-508) cannot be instantiated in objects, but unlike interfaces it may contain code for its methods. This code is inherited by all classes below it, and may be run for real objects of those classes.

- Some of the methods of an abstract class may be **abstract methods**. These, like the methods of an interface, are defined but not coded. They must be overridden by methods with real code in any instantiable class below. We use an abstract method when the code will depend on implementation choices made at a lower level of the class hierarchy.

# The Class `AbstractCollection`

- Above all the list types are three interfaces: `Iterable`, extended by `Collection`, extended by `List`. (All are generic, of course.) An `Iterable` must have an `iterator` method, creating an `Iterator`. A `Collection` has methods to add, test for, or remove either a specific element or *all* the elements that are in another `Collection`, along with `isEmpty`, `size`, and `toArray`. Finally, `List` brings in the indexed list operations such as `indexOf`, `get`, and `set`.

- The class `AbstractCollection` has implementations for some of the `Collection` operations, but `iterator` and `size` are left abstract because these two operation's code will be quite different for array-based and linked lists.

- The operations that modify the structure (by adding or removing elements) are called "optional" and are left unsupported in `AbstractCollection`. To make a modifiable collection, we need a real `add` and an `iterator` that creates Iterators that support `remove`.

# The Class `AbstractList`

- The `AbstractList` abstract class leaves the methods `get` and `size` as abstract -- implement them and you have an unmodifiable list.

- To have a modifiable list of a constant size you implement the optional method `set`, and to be able to change the size you implement `add` and `remove`. The `iterator` and `listIterator` methods use these other methods, so you don't need separate implementations for them. (A `ListIterator` is an `Iterator` that can traverse the list in either direction, and can replace elements as well as remove them.)

- The two array-based list classes extend `AbstractList`, while `LinkedList` also extends another interface called `AbstractSequentialList`. That class defines `get`, `set`, `add`, and `remove` in terms of `listIterator`, instead of the other way round as in `AbstractList`.

# The `Vector` and `ArrayList` Classes

- Java 1.0 contained `Vector` as its primary array-based list class. In Java 1.2 and thereafter, the `ArrayList` class and the `List` interface were introduced, and Vector was retrofitted to implement `List`.

- Both work much like L&C's `ArrayList` class -- they have resizable arrays of T objects and support all the `List` operations. The user can access the `capacity` and alter it with the methods `ensureCapacity` and `trimToSize` in either class.

- There are very few differences -- `Vector` can spawn off an `Enumeration` object (an obsolete version of an `Iterator`), and `Vector` is **synchronized** while `ArrayList` is not. This refers to the possibility that multiple threads will attempt to modify the same object at the same time. If a synchronized method is running on an object, no other thread can do anything to that object until the first method is done.

# The `LinkedList` Class

- This class operates pretty much like L&C's `DoubleList` and supports both the regular and optional operations of the `List` interface -- it is an indexed list. Like any such `List`, it is also a `Deque` and a `Queue`.

- All three list classes also implement the `Serializable` and `Cloneable` interfaces. The first allows you to write an object to a file, read it back, and reconstruct it just as it was. The second allows the `Object.clone` method to work, creating a **deep copy** of the object, as long as a `clone` method is written for the class.

- The two array-based list classes also implement the interface `RandomAccess`. This interface has no methods, but is a marker for other algorithms saying that `get` and similar methods run in O(1) time. Naturally, then, `LinkedList` does not implement this interface, as `get` is O(n) there.