**Collections**

So far we have used two relatively low-level data structures, arrays and linked lists using ListNodes. We will now introduce two more-developed data structures, ArrayLists and LinkedLists. Look at your cheat sheet to see what these data structures can do to themselves.

Recall that you wrote DLL so that it acts just like Java’s LinkedList. In today’s lab, you will write TJArrayList<E> so that it acts just like Java’s ArrayList<E>.

In the course of the year we will also use Stacks, Queues, PriorityQueues, Maps, and Sets. Their methods are also specified on your cheat sheet. All these data structures are a part of Java’s Collections Framework, which is a hierarchy of interfaces, abstract classes, and concrete classes. Sun's tutorial at <http://java.sun.com/docs/books/tutorial/collections/index.html> contains a helpful discussion.

In the simplified Java hierarchies below, *italics* signifies an interface and **bold** signifies a concrete class that implements the interface.

*Collection*  
 *List<E>*

*Map<K, V>*

**HashMap<K, V>**

**TreeMap<K, V>**

**ArrayList<E>**

**LinkedList<E>**

**Stack<E>**

*Queue<E>*

Not in the Collections hierarchy:

arrays using [ ]

**ListNode**

**TreeNode**

graphs

**PriorityQueue<E>**

*Set<E>*

**HashSet<E>**

**TreeSet<E>**

Notice that List<E> is an interface. An interface's methods are abstract and without code. The interface specifies what Lists need to do: add objects, remove objects, and so on. All the code to actually accomplish the tasks is in the concrete subclass, either ArrayList<E> or LinkedList<E>.

Here are some useful features of the Collections Framework:

1. The Collection Framework requires that every Collection have a toString method that all by itself returns a nicely-formatted list that includes square brackets and commas, e.g. [Andy, Bianca, Connor]. You don’t need a loop to print the elements of a Collection, unlike printing the elements of an array or of a linked list of ListNodes.

1. The Collection Framework requires that some Collections, including Lists and Sets, contain an inner, private object, called an "**iterator**," that allows the data structure to traverse through its data items one-by-one. Programmers do not need to know how the data is actually structured. All the programmer needs to know is how to call the iterator’s methods, which is what the API is for. That’s the point. A later lab will introduce you to iterators and what they can do.
2. Java 1.5 also provided the **for-each loop** to be used to visit each item in Lists or Sets. For example, here is the code using for-each to println the List or Set containing Student objects referenced by theList:

for( Student s : theList )

System.out.println( s + ", " );

Notice that, if the List is an ArrayList, the for-each loop has direct access to each item, without using index numbers. That is so convenient that they even made the for-each loop backward compatible with an old-style array. Please use it when appropriate.

for( int n : array)

System.out.println ( n + " " );

1. The overloaded class method Collections.sort(<T>) sorts its argument T in ascending order, whether the argument is an ArrayList or a LinkedList. In this course sometimes you are allowed to use Collections.sort
2. The generic <E> data type cannot be used by primitive types such as int, double, or char . You must declare your Collection object using a wrapper class such as Integer, Double, or Character. On the other hand, when adding or removing ints or doubles, Java automatically converts (or wraps) these either to Integer or Double. All the mathematical operations on Integer and Double objects work as you would expect.

**Exercises**

1. Which of the following statements compile? Assume the necessary libraries are imported.

1. List<String> list = **new** List<String>();//does not work because List is an interface
2. List<Double> list = **new** ArrayList<Double>();
3. List<Comparable> list = **new** LinkedList<Comparable>();
4. ArrayList<Widget> list = **new** List<Widget>();
5. ArrayList<String> list = **new** ArrayList<>();
6. LinkedList<Polkadot> list = **new** List<Polkadot>();
7. LinkedList<Integer> list = **new** ArrayList<Integer>();

2. Mark *True* or *False* and explain:

\_True\_a. A list can contain multiple references to the same object.

There can be multiple pointers to the same object

\_True\_b. The same object may belong to two different lists.

Yes, aslong as the lists point

False\_c. java.util.List's remove method destroys the object after it has been removed.

It removes the Object from the loop

\_False\_d. java.util.List's add method makes a copy of the object and adds it to the list.

\_True\_e. Two references can refer to the same list.

Two pointers at the same list

3. Are these two statements interchangeable?

Object obj = list.getLast();

Object obj = list.get(list.size() – 1);

Why or why not?

Yes, as they both refer to the last List element, but not all lists have a Get Last.

4. In Java Collections is an abstract base class called AbstractList. In that class are at least two abstract methods called iterator and size . Why are these methods declared to be abstract?