Collection Fact sheet

		Implementations			
		Hash Table	**Resizable Array**	**Balanced Tree**	**Linked List**
Interfaces	**Set**	HashSet		TreeSet	
	List		ArrayList		LinkedList
	Map	HashMap		TreeMap	

The Collection interface

Basic Operations

```
size()                  isEmpty()                   contains(Object element)
add(Object element)     remove(Object element)      iterator();
```

Bulk Operations

```
containsAll(Collection c)      addAll(Collection c)
removeAll(Collection c)        retainAll(Collection c)
clear()
```

Array Operations

```
toArray()                      toArray(Object a[])
```

The Iterator interface

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```

```
static void filter(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext();)
        if (!condition(i.next()))  //condition being
            i.remove();            //a boolean method
}
// Keep in mind that as of 1.5 you can traverse the
// collection with a for loop (but not remove)
for (Integer i : myNums)
        System.out.println(i);          // i becomes each element in turn
```

The Set Interface

A Set is a Collection that cannot contain duplicate elements. `Set` models the mathematical *set* abstraction. The `Set` interface extends `Collection` and contains *no* methods other than those inherited from `Collection`.

The List Interface

A `List` is an ordered `Collection` (sometimes called a *sequence*). Lists may contain duplicate elements. In addition to the operations inherited from `Collection`, the `List` interface includes operations for:

Positional Access

```
set(int index, Object element)              get(int index)
add(int index, Object element)              remove(int index)
addAll(int index, Collection c)
```

Search

```
indexOf(Object o)       lastIndexOf(Object o)
```

Iteration

```
listIterator()          listIterator(int index)
```

Range-view

```
subList(int from, int to)
```

Iterators

```
hasNext()       next()          hasPrevious()          previous()
nextIndex()     previousIndex()
remove()        set(Object o)   add(Object o)
```

Algorithms

- `sort` — sorts a `List` using a merge sort algorithm, which provides a fast, stable sort. (A *stable sort* is one that does not reorder equal elements.)
- `shuffle` — randomly permutes the elements in a `List`.
- `reverse` — reverses the order of the elements in a `List`.
- `rotate` — rotates all the elements in a `List` by a specified distance.
- `swap` — swaps the elements at specified positions in a `List`.
- `replaceAll` — replaces all occurrences of one specified value with another.
- `fill` — overwrites every element in a `List` with the specified value.
- `copy` — copies the source `List` into the destination `List`.
- `binarySearch` — searches for an element in an ordered `List` using the binary search algorithm.
- `indexOfSubList` — returns the index of the first sublist of one `List` that is equal to another.

The Map Interface

Basic Operations

```
put(Object key, Object value)          get(Object key)        remove(Object key)
containsKey(Object key)                containsValue(Object value)
size()                                 isEmpty()
```

Bulk Operations

```
putAll(Map t)          clear()
```

Collection Views

```
keySet()               values()       entrySet();
```

Interface for entrySet elements

```
public interface Entry {
        Object getKey();
         Object getValue();
         Object setValue(Object value);
}}
```

// Sets are an easy way of removing
// duplicates.

```
int []nums = {21,2,32,21,2,43,54,65};

HashSet asSet = new HashSet();  // faster
TreeSet asTree = new TreeSet(); // sorted

for(int i: nums){
       asSet.add(new Integer(i));
       asTree.add(new Integer(i));
}

System.out.println(asSet);
System.out.println(asTree);

/* prints
[32,2,21,43,5,54]
[2,5,21,32,43,54]   */
```

// One way to check frequency. I use some
// autoboxing which can affect speed.

```
int []nums = {21,2,32,21,2,43,54,5};
HashMap freq = new HashMap();

for(int i: nums){
       if(!freq.containsKey(i))
              freq.put(i,1);
       else{
              int n =
              ((Integer)freq.get(i)).intValue();
              freq.put(i,n+1);
       }
}
System.out.println(freq);
/* prints
{32=1, 2=2, 21=2, 43=1, 5=1, 54=1}
*/
```

ArrayList

An ArrayList is a member of the Java Collection Framework. It would be good to know the whole framework, but you don't need to just to use ArrayList. An ArrayList is a class that allows you to use a resizable array. By default ArrayLists hold objects of type Object. In Java every class is a subclass of Object so basically you can store any non-primitive value you want in an ArrayList (String, Point, JLabel, Integer, BigInteger, ArrayList ... anything.) When you take your objects out of the list you will need to cast them as their original type. If you want to store primitive types you will need to use the corresponding wrapper class. Every primitive type has a wrapper class (Integer for int, Double for double ...)

Some Basic ArrayList Examples: (for each of these examples I have only given the contents of main for space reasons.

```
String [] nms = {"Joe","Sue","Sam","An"};
ArrayList names = new ArrayList();

for(int i=0; i<nms.length; i++)
       names.add(nms[i]);

System.out.println(names);

//outputs:  [Joe, Sue, Sam, An]
```

Example 1:

This simple example shows how to construct an ArrayList, how to use the add Method to add objects to the ArrayList and the fact that we can println an entire list. When we println each element will be displayed based on its toString method.

```
String [] nms ={"Joe","Sue","Sam","An"};
ArrayList names = new ArrayList(Arrays.asList(nms));

System.out.println(names);

//outputs:  [Joe, Sue, Sam, An]
```

Example 2:

The Arrays class has a static method that allows us to convert an array to a List. I can use addAll to add all elements after the ArrayList is made or I can simply add the List to the constructor

```
String [] nms = {"Joe", "Sue", "Sam", "An"};
ArrayList names = new ArrayList(Arrays.asList(nms));

for(int i=0; i<names.size(); i++)
       System.out.print(((String)names.get(i)).toUpperCase()+",");

//outputs:  JOE,SUE,SAM,AN,
```

Example 3:

This is a simple example that shows if I need to do something with my original objects I need to type cast them when I take them out. (names.get(i) returns an object of class Object, to treat that as a String object I use (String) in Front.)

```
ArrayList nums = new ArrayList();
int tot=0;

for(int i=0; i<10; i++){
       int n = (int)(Math.random()*50);
       nums.add(new Integer(n));
}
for(int i=0; i<10; i++){
       tot += ((Integer)nums.get(i)).intValue();
}
System.out.println(nums +"\n"+tot);

/*e.g. output:
[35, 41, 28, 0, 8, 40, 1, 26, 7, 5]
191    */
```

Example 4:

This is an example of wrapping ints in Integer objects. (note: I can't type cast between int and Integer I can only type cast from class to class (if compatible) or Primitive to Primitive)

```
int []n ={12,54,23,65,5,3,12};
int []sn ={5,3,12};

ArrayList nums = new ArrayList();
ArrayList subnums = new ArrayList();

for(int i=0; i<n.length; i++)
        nums.add(new Integer(n[i]));
for(int i=0; i<sn.length; i++)
        subnums.add(new Integer(sn[i]));

System.out.println(nums);                       // [12, 54, 23, 65, 5, 3, 12]
Collections.sort(nums);
System.out.println(nums);                       // [3, 5, 12, 12, 23, 54, 65]
Collections.rotate(nums,2);
System.out.println(nums);                       // [54, 65, 3, 5, 12, 12, 23]
System.out.println(Collections.indexOfSubList(nums,subnums));          // -1
Collections.sort(subnums);
System.out.println(subnums);                    // [3, 5, 12]
System.out.println(Collections.indexOfSubList(nums,subnums));          // 2
nums.removeAll(subnums);
System.out.println(nums);                       // [54, 65, 23]
```

Example 5:

This is an example of using some Collections algorithms, and one Collections method. See how they are different. If you are going to use collections you need to become fairly familiar with all of the methods at your disposal.

New In JDK 5.0

Three new features in Java 5.0 make using Collections easier, they are:

- **Generics**
- **Autoboxing/unboxing**
- **New For Loop Syntax**

Generics:

Like a lot of the names in Java, the name makes more sense if you know C++. As far as we are concerned Generics allow us to be more specific. When we set up a collection we can specify what type of Objects we will be storing. When we do this we no longer need to cast the objects when we remove them from the list they come out as the type that is stored in the collection. This also forces us to only store one type of object in our collection (this is usually a good thing.) For Example:

ArrayList<Integer> nums = new ArrayList<Integer>();

// This ArrayList only holds Integers, the get method will return Integer

Autoboxing/unboxing

Autoboxing/unboxing allows your program to automatically convert between primitive types and their corresponding wrapper classes (int ←→ Integer, double ←→ Double ...) Autoboxing/unboxing is a bit of a controversial topic. The problems are it causes a performance hit, and allows for messier programs where programmers don't know what type their data is.

New For Loop Syntax

The new for loop syntax exists in many languages. In most languages it is referred to as a "For Each" loop. Basically our loop control variable "becomes" each element in the array/collection rather than simply becoming an index into the array.

```
ArrayList<Integer> numList = new ArrayList<Integer>();
Integer x=new Integer(54);

System.out.println(x*10);     // Basic unboxing Example

int []n = {4,5,6,7,8,9};

for(int i:n)                  // new for syntax
      numList.add(i);         // because our list only holds Integers Java
                              // knows to convert (autoboxing)

for(int i=0; i<numList.size(); i++)       // normal for loop, no unboxing
      System.out.println(numList.get(i).intValue());

for(int i:numList)                  // For Each loop with unboxing
      System.out.println(i*2);
```