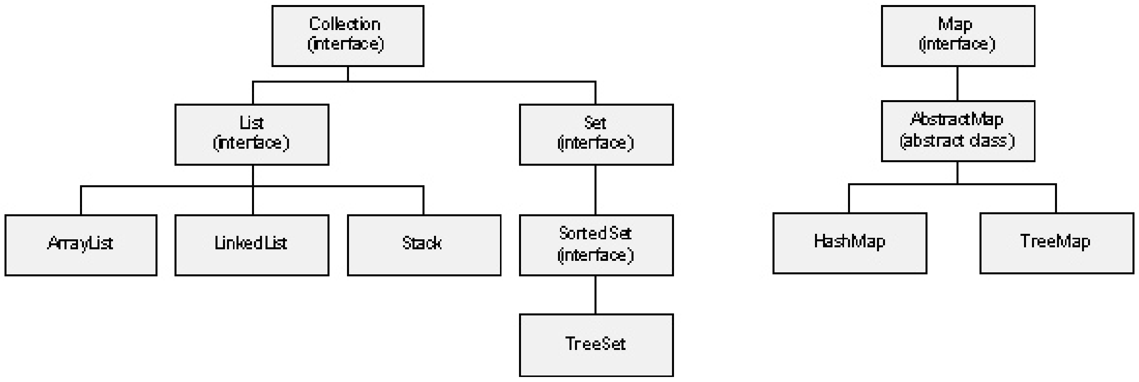
Return to Module Overview Page (https://onlinelearning.berkeley.edu/courses/665040/pages/module-six)

6.5: Collections

The Collections Framework is actually rooted in two interfaces: the Collection Interface and the Map interface. The figure below shows the hierarchy of some commonly used framework classes with Collection and Map as the roots.



Collection is an interface with methods to manage a group of elements. These methods include adding, removing and retrieving elements. Classes that implement the Collection interface have all of these methods in common. The table below shows a list of commonly used methods.

Table: List of Commonly Used Methods

Method	Description
boolean add(E e)	Add element to the collection
boolean remove(E e)	Remove element from the collection
int size( )	Returns the number of elements in the collection
void clear( )	Deletes all of the elements in the collection
boolean isEmpty( )	Returns a boolean on whether there are elements in the collection
Iterator iterator( )	Returns the iterator object for the collection
boolean contains(E e)	Returns a boolean on whether e is in the collection

Note #1: The type E is just a placeholder for a type. Since the entire Collections Framework allows generic types, you can substitute E with any type you choose. We will see how to do this later..

Note #2: We will use iterators throughout this section but discuss it in detail in the next section..

Classes that implement the List interface are suitable for elements that have to be in order (such as a top 10 list or a queue listing). Concrete classes of the List interface include ArrayList, LinkedList and Stack.

In addition to the Collection interface methods, List classes inherit the methods in table below. Since elements in a list are ordered, you can refer to them with an index.

Table: Methods

Method	Description
E get(int index)	Return the element at index
E set(int index, E e)	Set element at index
void add(int index, E e)	Inserts element e at index
E remove(int index)	Remove element at index. Returns the element

ArrayList works like an array, but with more features. Like an array, specific element retrieval takes constant time (O(1)). Unlike an array, space is automatically allocated when adding elements. Insert, add and delete operations occur in linear time (O(n)). Because a range of data needs to be shifted, these operations are not immediate. The more the data, the more time consuming it is.

The code below demonstrates adding elements and iterating through an ArrayList.

```
String [] languages = { "Python", "Java", "Perl",  "Basic", "Python" };

ArrayList al = new ArrayList();
for (String s : languages) {
   al.add(s);
}

for (Iterator iter = al.iterator(); iter.hasNext(); ) {
   String lang =  (String)iter.next();
   System.out.println(lang);
}
```

Note that the output will be in the same order the elements were added:

```
Python

Java

Perl

Basic

Python
```

A LinkedList is a data structure optimized for tasks that require fast addition and deletion of elements. Instead of a series of elements that are accessible through an index like an ArrayList, a LinkedList is composed of elements with a reference (link) to another element. Think of steel chains in a link. Insert and delete operations are fast (O(1)), because it just updates the links of the element. There would be no need to shift any of the elements like in an ArrayList. Element retrieval, however, is linear (O(n)). It has to iterate through the entire list to access a specific element.

A Stack is a LIFO (last in first out) collection. It includes push(E e) method to insert an element at the end of the Stack (last in) and pop( ) method to remove from its beginning (first out). The Stack is generally not used with an index.

Set classes contain only unique elements (such as employees or snowflake designs). The Set interface does not add any new methods, but provides a division from List classes. HashSet and TreeSet implement the Set interface.

The HashSet class operates just like a List, but will only add elements that do not already exist in a HashSet. The order of the elements is not guaranteed and may move over time.

TreeSet class works like the HashSet but elements that are inserted are sorted automatically. The order of the elements is guaranteed over time. You can also provide a Comparator object during instantiation for custom sorting.

The code below demonstrates adding elements to a TreeSet and iterating through it.

```
String [] languages = { "Python", "Java", "Perl",  "Basic", "Python" };
TreeSet ts = new TreeSet();
for (String s : languages) {
   ts.add(s);
}

for (Object o : ts) {
   System.out.println(o);
}
```

Notice that the print out of the iteration is unique and sorted alphabetically, unlike in our ArrayList example:

```
Basic

Java

Perl

Python
```