Container Classes

Lists and Sets

COMP2603
Object Oriented Programming 1

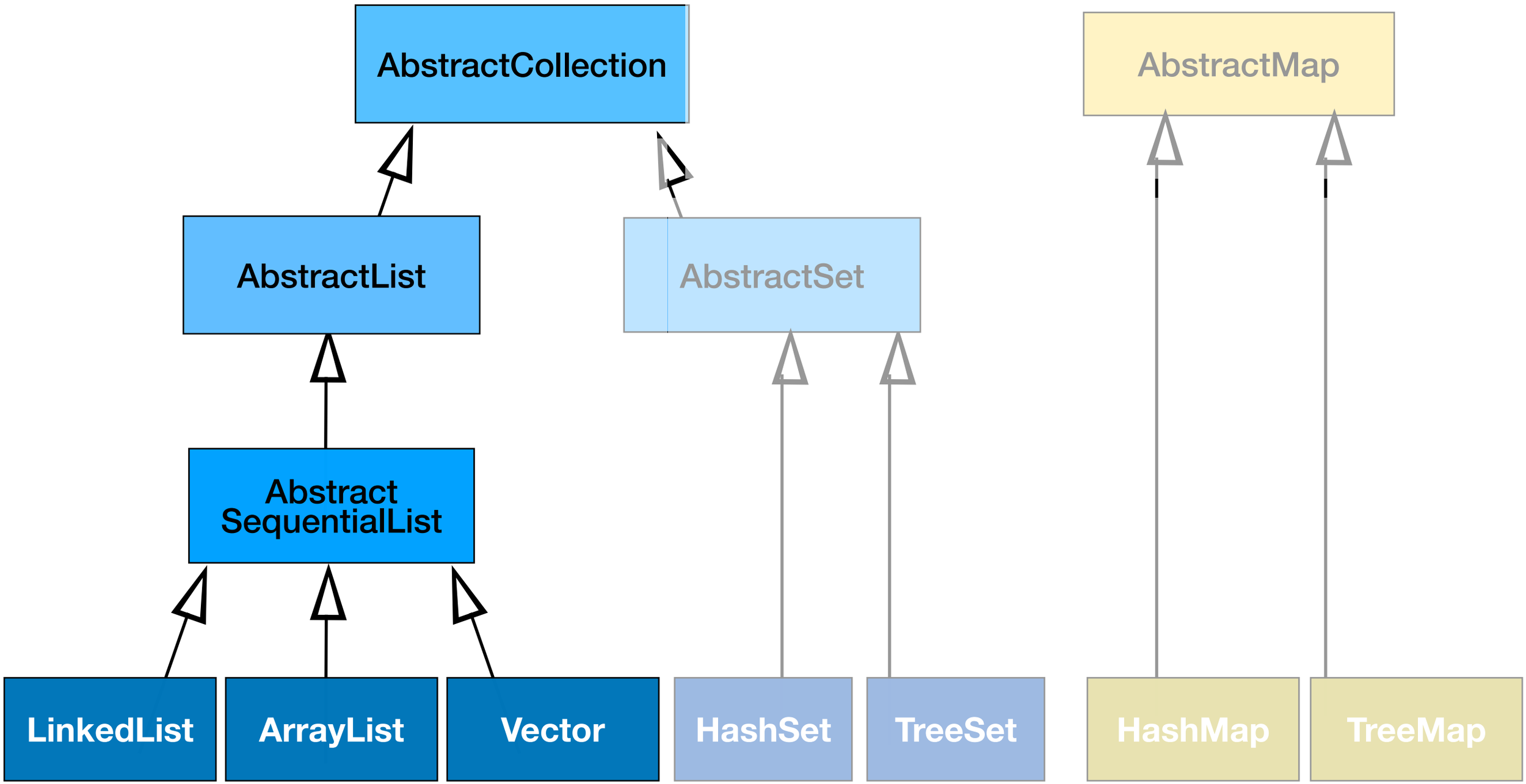
Week 9

Outline

- Java Collections Framework
- Collection Interface
- List Interface
  - Concrete Collections
    - LinkedList
    - Vector
    - ArrayList (review)
- Set Interface
  - SortedSet Interface

Reading: Chapter 13 - Container Classes: Fundamentals of Object Oriented Programming

Classes in the Java Collections Framework

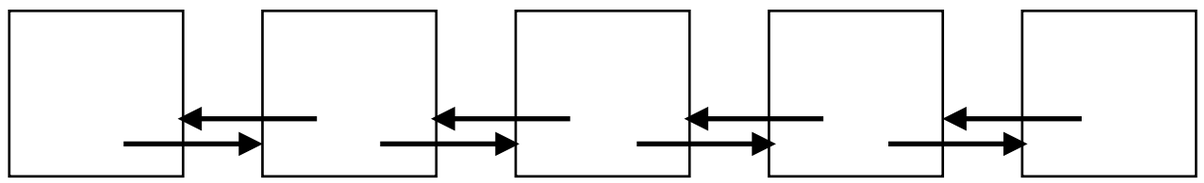


List

1. *Avoid the fog*
2. *Eat the frog*
3. *Feed the dog*
4. *Take a jog*
5. *Write on blog*
6. *Feed the dog*

Linked List

- Linear, ordered collection
- Precise control over where in the list each element is inserted.
- Supports element insertion and removal at both ends
- Doubly-linked list implementation of the List and Deque interfaces. Implements all optional list operations, and permits all elements (including null)



https://docs.oracle.com/javase/7/docs/api/java/util/List.html

https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html

List Interface

Method	Description
void add (int index, E element)	Inserts the object supplied as an argument in position index of the List. The other elements in the List are shifted down one position
E get (int index)	Returns the element at position index of the List
int indexOf(Object o)	Returns the position of the object supplied as an argument int he List, or -1 if it does not find a match. indexOf( ) uses the equals( ) method of the contained objects to check for equality with the object supplied as an argument.
E remove(int index)	Removes the object at position index of the List
E set(int index, E element)	Inserts the object supplied as element in position index of the List, overwriting the element that was there previously, if any. It returns the element that was previously stored at that position; otherwise it returns null

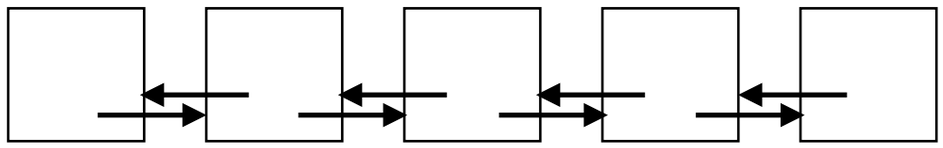
Example - Plant Class

```
public class Plant{
    private String name;

    public Plant(String n){
       name = n;
    }
    public String toString(){
       return name;
    }

}
```

LinkedList

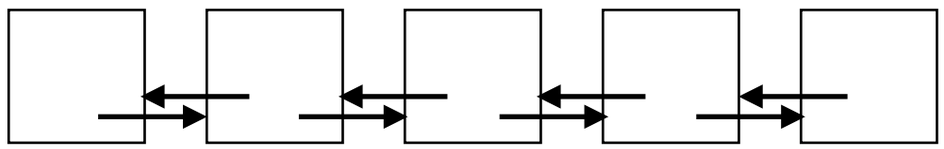
Creating and instantiating:

```
LinkedList<Plant> plants = new LinkedList<>();
```



Creating and instantiating with existing Collection:

```
LinkedList<Plant> morePlants = new LinkedList<>(plants);
```



https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html

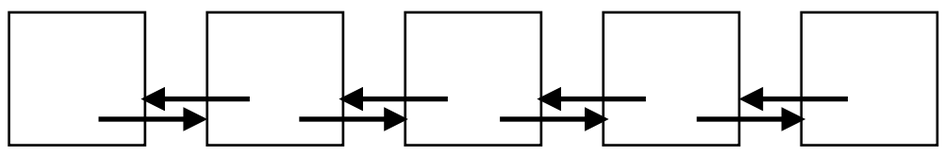
LinkedList

Getting objects

```
Plant p = plants.get(0); // Basil plant
```

Finding objects:

```
int basilIndex = plants.indexOf(p);  // index = 0
boolean hasBasil = plants.contains(p);
```



https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html

contains(Object obj)

The contains( ) method tests whether an object is in a collection. It uses the equals( ) method of the object to determine whether to return true or false.

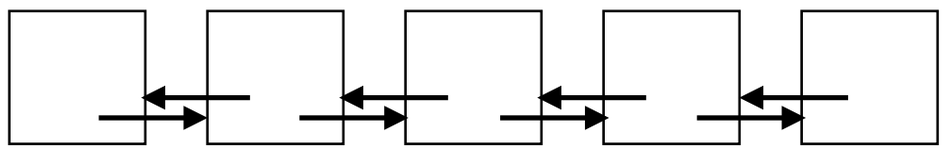
Objects that are placed in a collection should therefore override the equals( ) method. Otherwise, the equals( ) method of the Object class would be used. This method checks memory locations which does not always result in correct functionality.

https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html#contains(java.lang.Object)

Example - contains( )

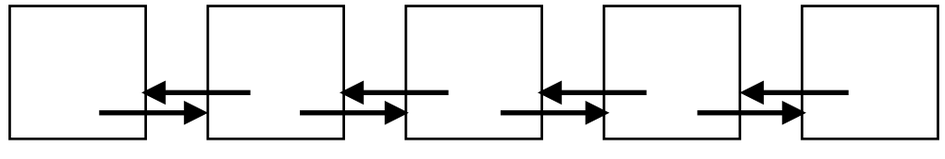
```
LinkedList<Plant> plants = new LinkedList<>();
Plant p = new Plant(“Aloe”);
Plant pA = new Plant(“Aloe”); //same plant conceptually

plants.add(p);
if(plants.contains(pA))
  System.out.println(“Aloe plant found”);
else
    System.out.println(“No Aloe plant found”);
```

Output: “No Aloe plant found”



Example 2 - Plant Class

```
public class Plant{
    private String name;
    /* rest of Plant class code */
    public boolean equals(Object obj){
       if(obj instanceof Plant){
          Plant incomingPlant = (Plant) obj;
          String plantName = incomingPlant.name;
          if(this.name.equals(plantName))
             return true;
       }
       return false;
    }
}
```



Example 2 - contains( )

```
LinkedList<Plant> plants = new LinkedList<>();
Plant p = new Plant(“Aloe”);
Plant pA = new Plant(“Aloe”); //same plant conceptually

plants.add(p);
if(plants.contains(pA))
  System.out.println(“Aloe plant found”);
else
    System.out.println(“No Aloe plant found”);
```

Output: “Aloe plant found”

LinkedList

Removing objects (top of list)

```
Plant p1 = plants.remove(); // Basil plant
```

Removing objects (index)

```
Plant p2 = plants.remove(4); // Which plant?
```

Removing objects (specific Object)

```
Plant oldPlant; // assume instantiated somewhere
Plant p3 = plants.remove(oldPlant);
```

https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html

Vector

The **`Vector`** class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index.

However, the size of a **`Vector`** can grow or shrink as needed to accommodate adding and removing items after the **`Vector`** has been created.

https://docs.oracle.com/javase/7/docs/api/java/util/Vector.html

Vector

Creating and instantiating:

```
Vector<Plant> veges = new Vector<>(); // default 10
```

Creating and instantiating with existing Collection:

```
Vector<Plant> moreVeges = new Vector<>(veges);
```

https://docs.oracle.com/javase/7/docs/api/java/util/Vector.html

Vector

Creating and instantiating:

```
Vector<Plant> veges = new Vector<>(); // default 10
```

Creating and instantiating with existing Collection:

```
Vector<Plant> moreVeges = new Vector<>(veges);
```

https://docs.oracle.com/javase/7/docs/api/java/util/Vector.html

Vector

Each vector tries to optimize storage management by maintaining a **`capacity`** and a **`capacityIncrement`**.

The **`capacity`** is always at least as large as the vector size; it is usually larger because as components are added to the vector, the vector's storage increases in chunks the size of **`capacityIncrement`**.

An application can increase the capacity of a vector before inserting a large number of components; this reduces the amount of incremental reallocation.

https://docs.oracle.com/javase/7/docs/api/java/util/Vector.html

Vector

Adding objects:

```
veges.add(new Plant(“Tomato”));  // index = 0;
veges.add(new Plant(“Potato”)); // index = 1
veges.add(1, new Plant(“Carrot”)); //insert at 1
```

https://docs.oracle.com/javase/7/docs/api/java/util/Vector.html

Vector

The iterators returned by this class's `iterator` and `listIterator` methods are *fail-fast*: if the vector is structurally modified at any time after the iterator is created, in any way except through the iterator's own `remove` or `add` methods, the iterator will throw a `ConcurrentModificationException`.

Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behaviour at an undetermined time in the future. The `Enumerations` returned by the `elements` method are *not* fail-fast.

https://docs.oracle.com/javase/7/docs/api/java/util/Vector.html

Vector

Traversal (with Iterator):

```
Iterator<Plant> iter = veges.iterator();
while( iter.hasNext()){
  Plant p = iter.next();
  System.out.println(p);
}
```

Output:
Tomato
Carrot
Potato

Vector

Traversal (with Iterator):

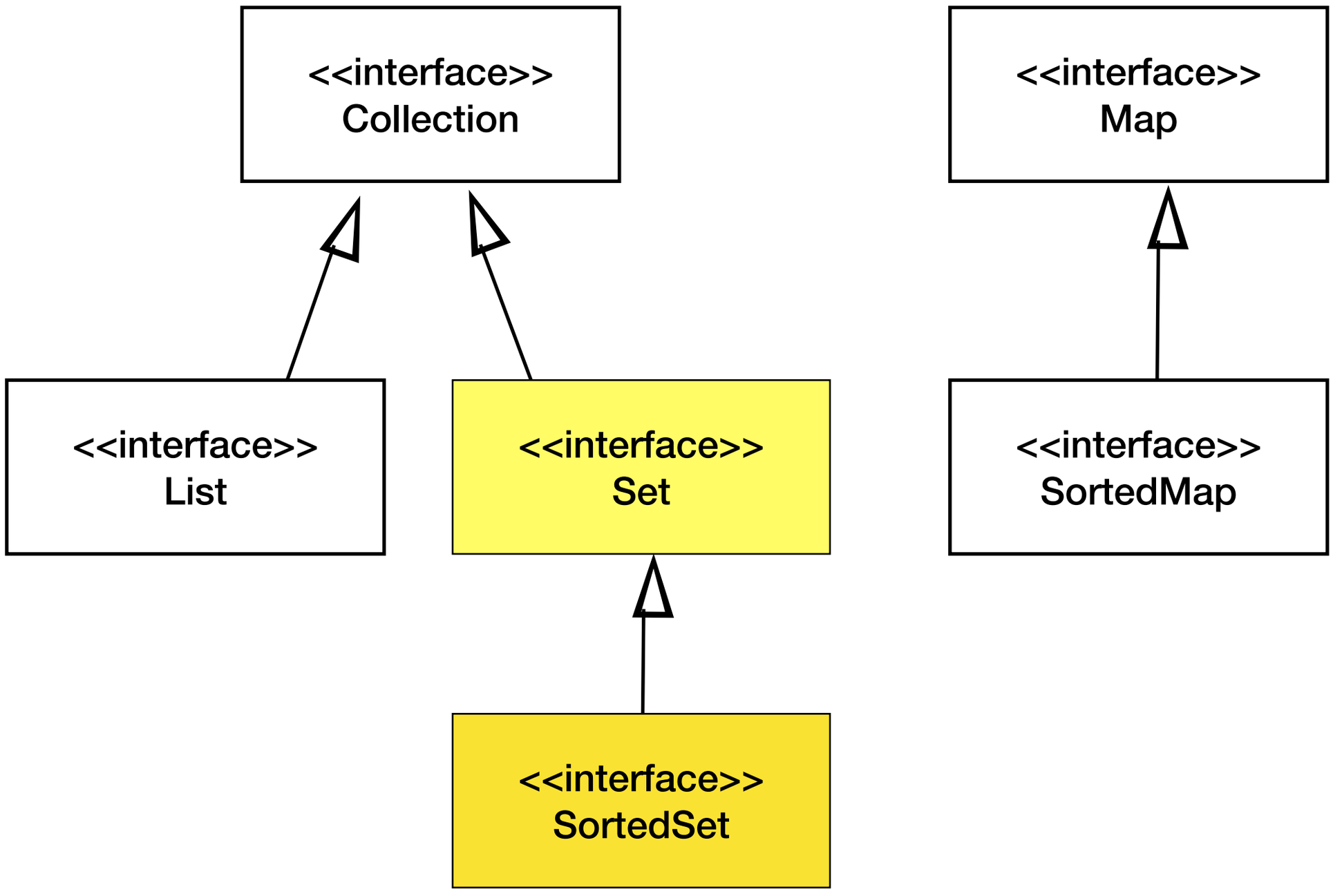
```
Iterator<Plant> iter = veges.iterator();
while( item.hasNext()){
   Plant p = item.next();
   System.out.println(p);
   iter.remove(p);
}

for(Plant p: veges){
    System.out.println(p);
   iter.remove(p); // ConcurrentModificationException
}
```

Vector

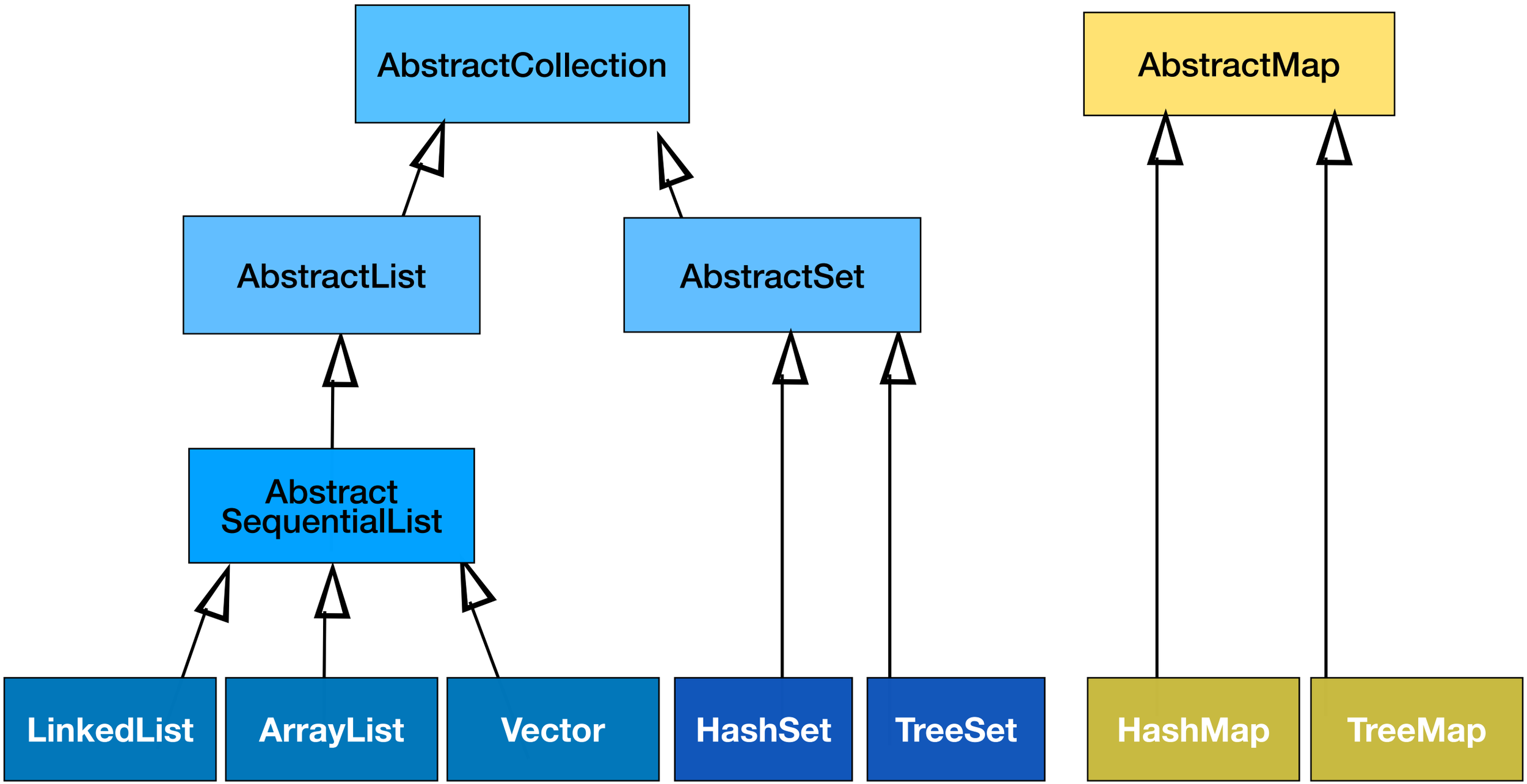
`Vector` is synchronized. If a thread-safe implementation is not needed, it is recommended to use `ArrayList` in place of `Vector`.

https://docs.oracle.com/javase/7/docs/api/java/util/Vector.html

Interfaces in the Java Collections Framework



Classes in the Java Collections Framework



The Set Interface

The Set interface represents an unordered collection of objects that contains no duplicate elements.

It therefore cannot contain two elements e1 and e2 where e1.equals(e2).

A Set can contain at most one null element.

The Set interface declares the same methods as its super-interface, Collection.

The Set Interface
(same as Collection)

Method	Description
boolean add (E o)	Inserts the object of the specified type into the collection; returns true if the object was added, false otherwise
boolean addAll (Collection c)	Inserts all the objects from the specified collection into the current collection
void clear( )	Removes all the elements from the collection
boolean contains (Object o)	Returns true id the specified object is present in the colleciton, and false otherwise
boolean isEmpty( )	Returns true if there are no elements in the collection, and false otherwise
boolean remove(Object o)	Deletes the specified object from the collection
int size( )	Returns the number of elements currently in the collection

https://docs.oracle.com/javase/7/docs/api/java/util/Set.html

The Set Interface - Adding Elements

The Set interface does not allow the add( ) and addAll( ) methods to duplicate elements to the Set.

If a Set already contains the element being added, its add( ) and addAll( ) methods return false.

In order for a Set to determine if it already contains an element, it uses the equals( ) method of the element to check if the element is equal to an existing element in the Set.

It is therefore necessary to override the equals( ) method of the objects that will be inserted into a Set.

The Sorted Set Interface

The SortedSet interface is a Set that sorts its elements and guarantees that elements are enumerated in sorted order.

It declares a few methods of its own such as first( ) and last( ) which return the lowest and highest elements in the set, respectively (as determined by the sort order).

Comparable Interface

This interface imposes a total ordering on the objects of each class that implements it.

This ordering is referred to as the class's natural ordering, and the class's **compareTo** method is referred to as its natural comparison method.

https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html

int compareTo(Object obj)

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html

Comparable Interface

The natural ordering for a class C is said to be consistent with equals if and only if e1.compareTo(e2) == 0 has the same boolean value as e1.equals(e2) for every e1 and e2 of class C.

Note that null is not an instance of any class, and e.compareTo(null) should throw a NullPointerException even though e.equals(null) returns false.

https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html

Example 1 - Plant Class

```
public class Plant implements Comparable{
    private String name;
    …
    …
    // Compare by Name – ascending A–Z
    public int compareTo(Object obj){
      if(obj instanceof Plant){
        Plant p = (Plant)obj;
         return name.compareTo(p.name);
      }
      throw new ClassCastException(“Not a Plant”);
    }
}
```

Example 2 - Plant Class

```
public class Plant implements Comparable{
    private int ID;
    …
    …// 1 , 2
    // Compare by ID – ascending order
    public int compareTo(Object obj){
       if(obj instanceof Plant){
         Plant p = (Plant)obj;
          if(this.ID == p.ID) return 0;
          if(this.ID > p.ID) return 1;
          if(this.ID < p.ID) return -1;
       }
       throw new ClassCastException(“Not a Plant”);
    }
}
```