Collections

1. If we want to represent group of individual objects as a single entity then we go for Collections.
2. Collections are **growable in size.**
3. We can add the elemets over the size then **JVM will not rise any Exception.**
4. Collections can have **Heterogeneous elements**, If we are adding **different type of elements JVM will not throw Error.**
5. In Collections we have predefined Methods.

Arrays

1. Arrays are **fixed in size.**
2. We cannot add the elements over the size if we add the elements over the size, then **JVM will throw an error.**
3. Arrays can have only **Homogeneous elements.** If we are adding the **elements which are not same** then **JVM will throw an Error.**

Diff bw list and set and map

List:

1.It is the child interface of collection.

2.If we want to represent a group of individual objects as a **single entity** where **"duplicates are allowed and insertion order must be preserved"** then we should go for List interface.

Collection(I) → List(I)→ ArrayList(C), LinkedList(C) and Vector→ Stack

Set:

1.It is the child interface of collection.

2.If we want to represent a group of individual objects as **single entity** where "**duplicates are not allowed and insertion order is not preserved"** then we should go for Set interface.

3.Set Interface does **not contain any new methods.**

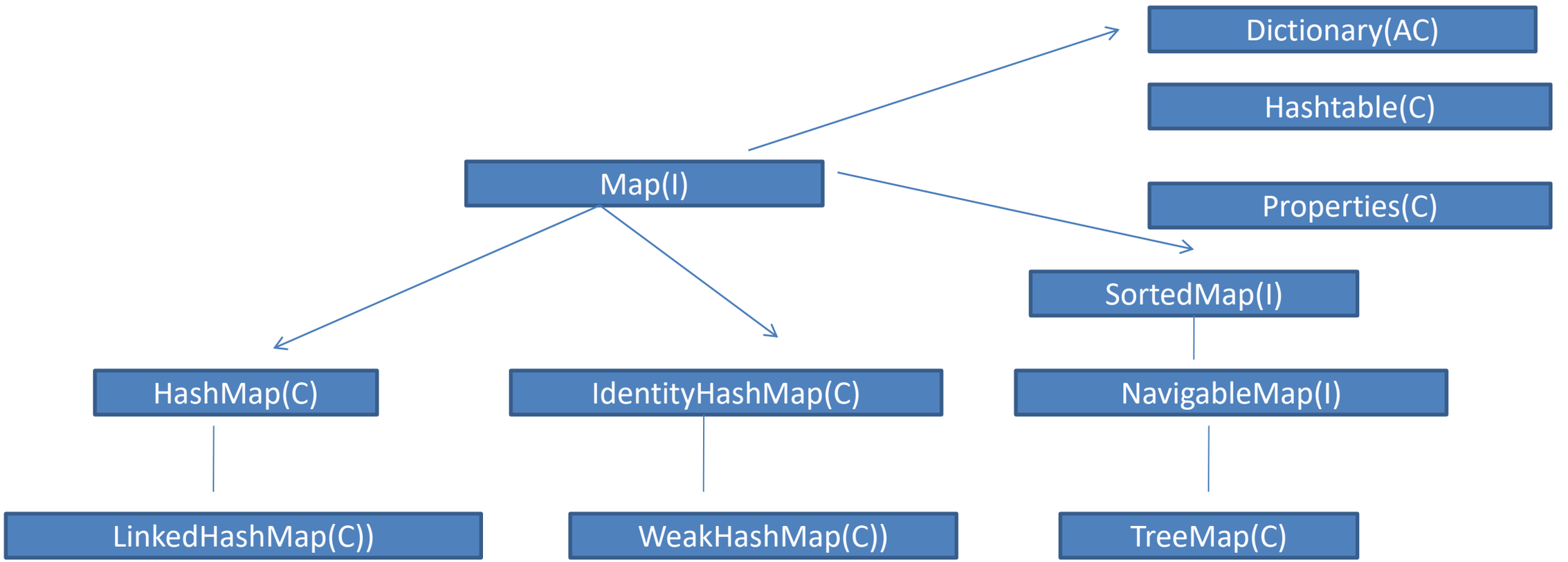
So we have to use only Collection Interface methods.

Collection(I)→ Set(I)→ HashSet(C) →LinkedHashSet(C)

Collection(I)→ Set(I)→ SortedSet(I) →NavigableSet(I)→ TreeSet(C)

Map:

1. Map is **Not Child Interface** of **Collection**. Hence we cant apply **collections interface methods here.**
2. If we want to represent a group of objects as **key-value pairs** then we should go for **Map Interface.**
3. Duplicate **keys** are not allowed but **values** can be **duplicated**.
4. Each **key-value** pair is known as **one entity.**



Type	Data Structure
ArrayList	Re-sizable Array
LinkedList	Double Linked List
Stack	Linear Data Structure
Vector	
HashSet	HashTable
LinkedHashSet	HashTable and LinkedList
Hashset	HashTable
LinkedHashSet	HashTable and LinkedList
TreeSet	TreeMap
HashMap	HashTable
LinnkedHashMap	HashTable and LinkedList
TreeMap	Red Black Tree

Different List Operations

Add data using indices:

1. by Using add(index) method adding element
2. by Using add(index, element) method adding specified index with element
3. by Using add(index, element) method If we add new element at the index position 2, but here old element is moved to next index accordingly
4. by Using addAll(index, element/object-ref) inserts all of the elements in the specified collection into this list at the specified position

Update data using indices:

1. by Using set(index, element) method we update the index and old element is replaced with new element
2. by Using set(index, element) method we cannot add new index

Get data using indices:

1. by Using get(index) method we can get the element if there is no element we get error

Remove data using indices:

1. by Using remove(index) method we can remove the element and reaming elements with index are moved to right
2. by using HashSet constructor remove duplicate elements from ArrayList

Find data using indices:

1. by Using indexOf(element) method we can get the index position if no index we get -1
2. by Using lastIndexOf(element) we can get the last occurrence

Constructor:

1. by using Arraylist Constructor we can store other class object reference to it `new ArrayList<String>(arrayList);`
2. by Using size() method returns the elements present in list

by Using add(index, element) method adding specified index with element

```
List<String> list = new ArrayList<String>();
list.add(0, "NameZero");
list.add(1, "NameOne");
list.add(2, "NameTwo");
list.add(3, "NameThree");
list.add(4, "NameFour");
System.out.println(list); // [NameZero, NameOne, NameTwo, NameThree, NameFour]

//we cannot add the higher index directly, it is index out of range
list.add(8, "NameEight"); // java.lang.IndexOutOfBoundsException:
```

by Using add(index, element) method adding specified index with element

```
List<String> list = new ArrayList<String>();
list.add(0, "NameZero");
list.add(1, "NameOne");
list.add(2, "NameTwo");
list.add(3, "NameThree");
list.add(4, "NameFour");
System.out.println(list);
System.out.println(list.size());

// If we add new  element at the index position 2, but here old element is moved to next index accordingly
list.add(2, "NameTwoDuplicate"); // 5
System.out.println(list); // [NameZero, NameOne, NameTwoDuplicate, NameTwo, NameThree, NameFour]
System.out.println(list.size()); // 6
```

by Using addAll(index, element/object-ref) inserts all of the elements in the specified collection into this list at the specified position

```
List<String> list1 = new ArrayList<String>();
list1.add(0, "Zero");
list1.add(1, "One");
list1.add(2, "Two");
list1.add(3, "Three");
list1.add(4, "Four");
System.out.println(list1); // [Zero, One, Two, Three, Four]

List<String> list2 = new ArrayList<String>();
list2.add("TwoD");
list2.add("ThreeD");
list2.add("ThreeD");
System.out.println(list2); // [TwoD, ThreeD, ThreeD]

//Inserts all of the elements in the specified collection into this list at the specified position
list1.addAll(2, list2);
System.out.println(list1); // [Zero, One, TwoD, ThreeD, ThreeD, Two, Three, Four]
```

by Using set(index, element) method we update the index and old element is replaced with new element

```
List<String> list1 = new ArrayList<String>();
list1.add(0, "Zero");
list1.add(1, "One");
list1.add(2, "Two");
list1.add(3, "Three");
list1.add(4, "Four");
System.out.println(list1); // [Zero, One, Two, Three, Four]

//set can modify the existed index element
list1.set(2, "TwoDuplicate");
System.out.println(list1); // [Zero, One, TwoDuplicate, Three, Four]

//set cannot add new index element
list1.set(5, "Five");
System.out.println(list1); // java.lang.IndexOutOfBoundsException:
```

by Using get(index) method we can get the element if there is no element we get error

```
List<String> list = new ArrayList<String>();
list.add(0, "Zero");
list.add(1, "One");
list.add(2, "Two");
list.add(3, "Three");
System.out.println(list); // [Zero, One, Two, Three]

System.out.println(list.get(3)); // Three
System.out.println(list.get(8)); // java.lang.IndexOutOfBoundsException:
```

by Using remove(index) method we can remove the element and reaming elements with index are moved to right

```
List<String> list = new ArrayList<String>();
list.add(0, "Zero");
list.add(1, "One");
list.add(2, "Two");
list.add(3, "Three");
System.out.println(list); // [Zero, One, Two, Three]
list.remove(2);
System.out.println(list); // [Zero, One, Three]
list.remove(1);
System.out.println(list); // [Zero, Three]
list.remove(0);
System.out.println(list); // [Three]
list.remove(0);
System.out.println(list); // []
```

by Using indexOf(element) method we can get the index position if no index we get -1
by Using lastIndexOf(element) we can get the last occurrence

```
List<String> list = new ArrayList<String>();
list.add(0, "A");
list.add(1, "B");
list.add(2, "C");
list.add(3, "B");
list.add(4, "E");
list.add(5, "B");
list.add(6, "D");
System.out.println(list); //[A, B, C, B, E, B, D]

int indexOf(Object o);
System.out.println(list.indexOf("B")); // 1
System.out.println(list.indexOf("Z")); // -1

 int lastIndexOf(Object o);
System.out.println(list.lastIndexOf("B")); // 5
```

by using Arraylist Constructor we can pass object reference to it

```
ArrayList<String> arrayList = new ArrayList<String>();
System.out.println(arrayList); // []
arrayList.add("One");
arrayList.add("Two");
arrayList.add("Three");
arrayList.add("Four");
arrayList.add("Five");
System.out.println(arrayList); // [One, Two, Three, Four, Five]

ArrayList<String> list = new ArrayList<String>(arrayList);
list.add("Jan");
list.add("Feb");
list.add("Mar");
System.out.println(list); // [One, Two, Three, Four, Five, Jan, Feb, Mar]
```

by using HashSet constructor remove duplicate elements from ArrayList

```
List<String> list = new ArrayList<String>();
list.add("A");
list.add("B");
list.add("C");
list.add("B");
list.add("D");
list.add("A");
System.out.println(list); // [A, B, C, B, D, A]

Set<String> set = new HashSet<String>(list);
System.out.println(set); [A, B, C, D]
```

Difference between ArrayList and LinkedList

ArrayList and **LinkedList** both implements **List Interface** and maintains insertion order. **Both are non synchronized classes.**
Collection(I) → List(I)→ ArrayList(C), LinkedList(C) and Vector(C)→ Stack(C)

ArrayList	LinkedList
Introduced in 1.2 version	Introduced in 1.2 version
ArrayList internally uses a **dynamic array** to store its elements.	LinkedList uses **Doubly Linked List** to store its elements.
Duplicate Objects are allowed **(Same Object)**	Duplicate Objects are allowed **(Same Object)**
ArrayList is slow as array manipulation is slower.	LinkedList is faster being **node** based as not much bit shifting required.
ArrayList implements only **List**.	LinkedList implements **List** as well as **Queue**. It can acts as a queue as well.
ArrayList is faster in **storing and accessing data.**	LinkedList is faster in **manipulation of data..**
Allows Sorting	Allows Sorting

add() and set() and remove() in LinkedList()

```
List<String> list = new LinkedList<String>();
list.add(0,"NameZero");
list.add(1,"NameOne");
list.add(2,"NameTwo");
list.add(3, "NameThree");
list.add(4, "NameFour");
list.add(5, "NameFive");
list.set(5, "Sai Kiran");   // [NameZero, NameOne, NameTwo, NameThree, NameFour, Sai Kiran]
System.out.println(list);
list.remove(5); // remove element
System.out.println(list);//  [NameZero, NameOne, NameTwo, NameThree, NameFour]
```

```
//null value in ArrayList, LinkedList, List
ArrayList<String> l = new ArrayList<String>();
l.add(0, null); // public void add(int index, E element)
l.add(null); // public boolean add(E e)
l.add(null);
System.out.println(l); // [null, null, null]

LinkedList<String> list = new LinkedList<String>();
list.add(0, null); // public void add(int index, E element)
list.add(1, null);
list.add(2, null);
list.add(null); // public boolean add(E e)
list.add(null);
System.out.println(list); // [null, null, null, null, null]

List<String> l1 = new ArrayList<String>();
l1.add(0, null); // boolean add(E e);
l1.add(null); // void add(int index, E element);
l1.add(null);
System.out.println(l1); // [null, null, null]
```

sort()

default void sort(Comparator<? super E> c) {}

```
LinkedList<String> ll = new LinkedList<String>();
ll.add("A");
ll.add("C");
ll.add("B");
System.out.println(ll); // [A, C, B]
ll.sort(null); // need to pass null
System.out.println(ll); // [A, B, C]
```

sort(null)
sort() method expects null as argument for sorting List, ArrayList and LinkedList
If the specified comparator is null then all elements in this list must implement the Comparable interface and the elements' natural ordering should be used.

public void sort(Comparator<? super E> c) {}

```
ArrayList<String> l = new ArrayList<String>();
l.add("A");
l.add("C");
l.add("B");
System.out.println(l); // [A, C, B]
l.sort(null); // need to pass null
System.out.println(l); // [A, B, C]
```

LinkedList Methods:

```
LinkedList<Integer> list = new LinkedList<>();
list.add(10);
list.add(20);
list.add(30);
list.add(40);
list.add(50);
list.addFirst(100);
System.out.println(list); //[100, 10, 20, 30, 40, 50]
list.removeFirst();
list.removeLast();
System.out.println(list); //[10, 20, 30, 40]
list.addLast(200);
System.out.println(list);  // [10, 20, 30, 40, 200]
```