WELCOME TO JAVA CLASS - WE'LL BEGIN IN A FEW MINUTES

Collections (Java Collections framework) in Java defines several classes and interfaces to represent a group of objects as a single unit - **collection object** or **container object**. A collection is a data structure which contains and processes a set (group) of data or objects. The data stored in the collection is encapsulated and the access to the data is only possible via predefined methods.

Encapsulation is a way to get private data / members of a class through public methods>

Java Collections consist of the interfaces and classes

Advantages of Collection Framework:

1. Consistent API : The API has a basic set of interfaces like Collection, Set, List, or Map. All classes (ArrayList, LinkedList, Vector, etc) that implement these interfaces have some common set of methods.
2. Reduces programming effort: A programmer doesn't have to worry about the design of Collection, and he can focus on its best use in his program.
3. Increases program speed and quality: Increases performance by providing high-performance implementations of useful data structures and algorithms.
4. No size restrictions. Unlike an array, collection objects can hold an unlimited number of items.
5. Support more flexible data structures

The following **fundamental methods of collections** are enough to define the basic behavior of a collection and provide an Iterator to step through the elements in the Collection.

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

Collection Framework - Collection Framework provides important *interfaces and classes* using which we can manage a *group of objects*.

Collections comes in the advance topics of Core Java and you shouldn't miss it for interviews. Some of the most important classes in Collection Framework are-

- **ArrayList -** It is like a *dynamic array* i.e. we don't need to declare its size, it *grows as we add elements to it* and it *shrinks as we remove elements* from it, during the *runtime* of the program**.** For more you may read **ArrayList in Java- Decodejava.com**
- **LinkedList** - It can be used to depict a *Queue*(**FIFO)** or even a *Stack*(**LIFO)**. For more on *LinkedList with simple code examples*, *you may read* **LinkedList in Java- Decodejava.com**
- **HashSet -** It stores its element by a process called **hashing.** The order of elements in HashSet is *not guaranteed***.** For more on *HashSet with simple code examples, you may read* **HashSet in Java- Decodejava.com**
- **TreeSet** - TreeSet is the best candidate when one needs to store a large number of *sorted elements* and their *fast access*. *For more on TreeSet with easy code examples, you may read* **TreeSet in Java- Decodejava.com**
- **ArrayDeque -** It can also be used to implement a *first-in, first-out*(**FIFO**) queue or a *last-in, first-out(***LIFO**) queue. *For more on ArrayDeque with code example, you may follow* **ArrayDeque Class**
- **HashMap** - HashMap stores the data in the form of **key-value** *pairs,* where **key** and **value** are **objects**. For HashMap with code examples, you may read**HashMap in Java - Decodejava.com**
- **Treemap -** TreeMap stores **key-value** *pairs* in a sorted ***ascending order*** and *retrieval speed* of an element out of a TreeMap is quite *fast*. For more on TreeMap through simple code, you may read **TreeMap in Java - Decodejava.com**

Collection Framework is one of the most important topics and it’s a little time consuming but eventually hard work pays off.

```
Collection : Root interface with basic methods like add(), remove(),
             contains(), isEmpty(), addAll(), ... etc.

Set : Doesn't allow duplicates. Example implementations of Set
      interface are HashSet (Hashing based) and TreeSet (balanced
      BST based). Note that TreeSet implements SortedSet.

List : Can contain duplicates and elements are ordered. Example
       implementations are LinkedList (linked list based) and
       ArrayList (dynamic array based)

Queue : Typically order elements in FIFO order except exceptions
        like PriorityQueue.

Deque : Elements can be inserted and removed at both ends. Allows
        both LIFO and FIFO.

Map : Contains Key value pairs. Doesn't allow duplicates.  Example
      implementation are HashMap and TreeMap.
      TreeMap implements SortedMap.

The difference between Set and Map interface is that in Set we
have only keys, whereas in Map, we have key, value pairs.
```

List
ArrayList
Set
Map

LinkedList

can store the duplicate elements.
maintains the insertion order
with LinkedList data manipulation is faster

```
public class LinkedListDemo {
public static void main(String args[]) {
        LinkedList ll = new LinkedList();// create a linked list
       // add elements to the linked list
       ll.add("F");
        ll.add("B");
       ll.add("D");
       ll.add("E");
       ll.add("C");
       ll.addLast("Z");
       ll.addFirst("A");
       ll.add(1, "A2");
        System.out.println("Original contents of ll: " + ll);
        // remove elements from the linked list
       ll.remove("F");
       ll.remove(2);
        System.out.println("Contents of ll after deletion: " + ll);

   ll.removeFirst();  // remove first and last elements
       ll.removeLast();
        System.out.println("ll after deleting first and last: " + ll);
        // get and set a value
        Object val = ll.get(2);
       ll.set(2, (String) val + " Changed");
        System.out.println("ll after change: " + ll);
  }
}
```

Vector

Vector is a grow-able *array* of **objects**, whose size is designed to be changed at run time.

dynamic array to store the data elements

very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.

Vector is a grow-able *array* of **objects**, whose size is designed to be changed at run time. Dynamic array to store the data elements (objects). Very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.

```
public static void main(String args[]) {
        Vector v = new Vector(3, 2);   // initial size is 3, increment is 2
        System.out.println("Initial size: " + v.size());
        System.out.println("Initial capacity: " + v.capacity());
        v.addElement(new Integer(1));
        v.addElement(new Integer(2));
        v.addElement(new Integer(3));
        v.addElement(new Integer(4));
        System.out.println("Capacity after four additions: " + v.capacity());
        v.addElement(new Double(5.45));
        System.out.println("Current capacity: " + v.capacity());
        v.addElement(new Double(6.08));
        v.addElement(new Integer(7));
   System.out.println("Current capacity: " + v.capacity());
        v.addElement(new Float(9.4));
        v.addElement(new Integer(10));
   System.out.println("Current capacity: " + v.capacity());
        v.addElement(new Integer(11));
        v.addElement(new Integer(12));
        System.out.println("First element: " + (Integer)v.firstElement());
        System.out.println("Last element: " + (Integer)v.lastElement());
        if(v.contains(new Integer(3)))
        System.out.println("Vector contains 3.");
        // enumerate the elements in the vector.
        Enumeration vEnum = v.elements();
        System.out.println("\nElements in vector:");
        while(vEnum.hasMoreElements())
        System.out.print(vEnum.nextElement() + " ");
        System.out.println();
```

```
 }
}
```

Hashtable

Hashtable internally contains buckets in which it stores the key/value pairs.

The Hashtable uses the key's hashcode to determine to which bucket the key/value pair should map.

does not accept null key or value.

does not accept duplicate keys.

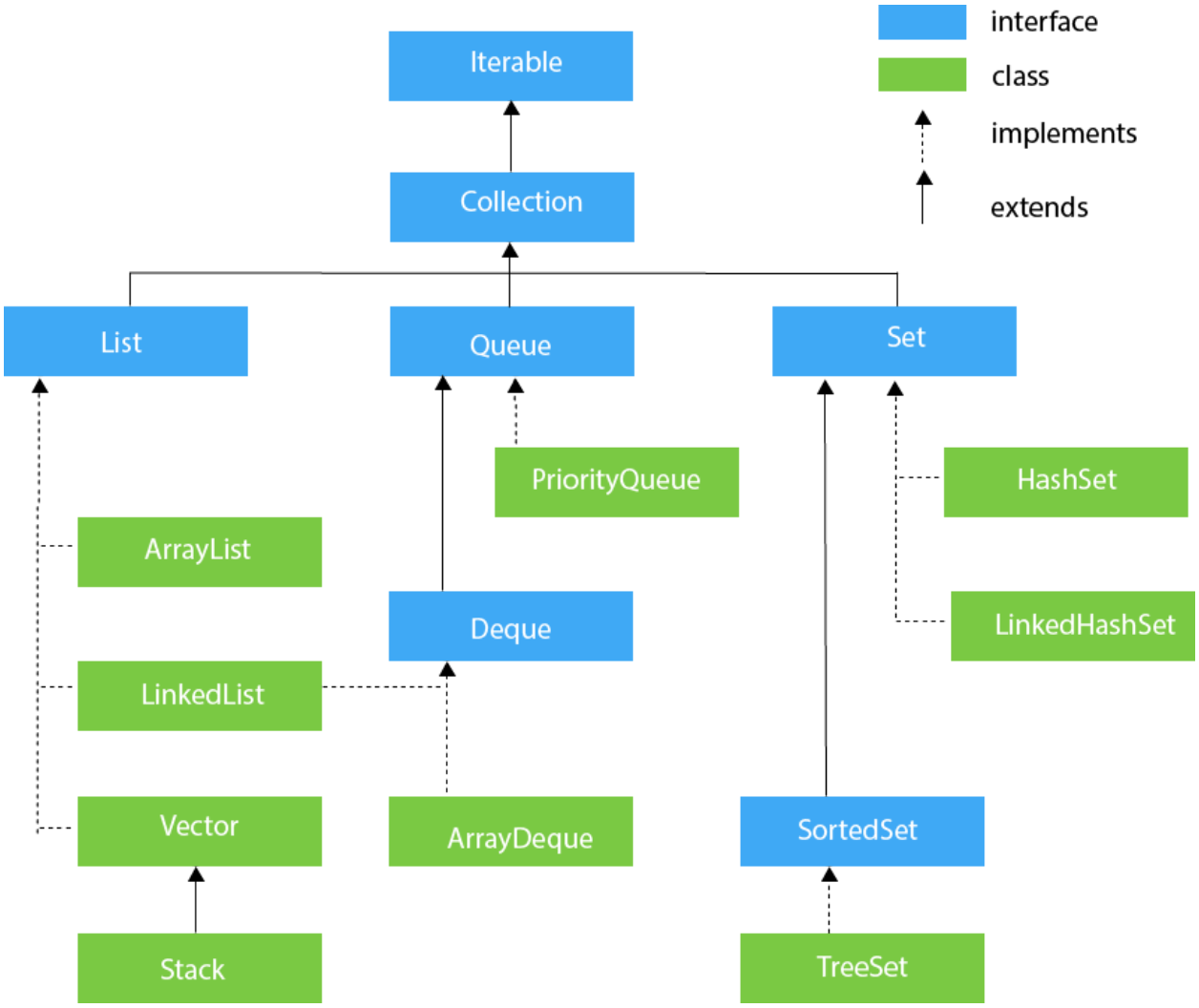
stores key-value pairs in hash table data structure which internally maintains an array of list. Each list may be referred as a bucket. In case of collisions, pairs are stored in this list.

The function to get bucket location from Key's hashcode is called hash function

```
import java.util.Hashtable;
import java.util.Iterator;

public class HashTableDemo {
	public static void main(String[] args) {
		Hashtable<Integer, String> hashtable = new Hashtable<Integer, String>();
		hashtable.put(1, "A");
		hashtable.put(2, "B");
		hashtable.put(3, "C");

		System.out.println(hashtable);

		String value = hashtable.get(1);
		System.out.println(value);

		hashtable.remove(3);

		Iterator<Integer> itr = hashtable.keySet().iterator();

		while (itr.hasNext()) {
			Integer key = itr.next();
			String mappedValue = hashtable.get(key);
			System.out.println("Key: " + key + ", Value: " + mappedValue);
		}
	}
}
```

Iterator and ListIterator

https://howtodoinjava.com/java-collections/



A Set is a Collection of sorted elements with no duplicates.

Map

stores values in key-value pairs
keys are unique - a single key only appears once in the Map
a key can map to only one value
a key is an object that is used to retrieve a value at a later date
values do not have to be unique

Map methods:

Object put(Object key, Object value)

Object get(Object key)

Object remove(Object key)

boolean containsKey(Object key)

boolean containsValue(Object value)

int size()

boolean isEmpty()

```
package mapdemo;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class MapDemo {
	public static void main(String[] args) {
		Map<Integer, String> map = new HashMap<Integer, String>();
		map.put(1, "Sapin");
		map.put(2, "Biju");
		map.put(3, "");
		map.put(4, "Nepal");
		map.put(9, "Naveen");
		map.put(10, "Nabil");
		map.put(5, "Kalam");
		map.put(6, "Nikhil");
		map.put(7, "Samir");
		map.put(8, "Agni");
		System.out.println(map);
		System.out.println("Is this map empty? " + map.isEmpty());

		System.out.println(map.values());
```

```
		System.out.println(map.toString());

		Map<Integer, String> anotherMap = new HashMap<Integer, String>();

		map.putAll(anotherMap);
		System.out.println("Another Map values: " + anotherMap);

		System.out.println(map.values()); // prints all the values - not the keys
		System.out.println(map.keySet()); // prints all the keys - not the values

		System.out.println("Print entry set:" + map.entrySet());

		System.out.println("The size of the Map is: " + map.size());

		System.out.println("Is there a value for key 3? " + map.containsKey(3)); // If the 3rd Key
contains a value
		System.out.println("Is there a value - Agni? " + map.containsValue("Agni")); // If the
value Agni is present

		System.out.println("Comparing two maps that contains exactly same set of keys and
values : " + map.equals(map));

		System.out.println("Looping through map elements:");
		for (int i = 0; i < map.size(); i++) {
			System.out.println(map.get(i)); // why null?
		}

		System.out.println("The HashCode of this map is : " + map.hashCode()); // HashCode is
the memory reference of

// objects in the integer form
		System.out.println("Clearing a Map:");
		map.clear();
		System.out.println("Map cleared :" + map);
		// putAll(Map m)
	}
}
```

Declaring a Set and printing its individual values:

```
package traversal;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class PrintingElementsOfASet {
public static void main(String[] args) {                    // main method
        Set<String> mySet = new HashSet<String>();    // Set declaration
                mySet.add("dime");
                mySet.add("penny");
                mySet.add("cent");
                mySet.add("quarters");
                mySet.add("pence");
                mySet.add("bill");

// using Iterator class:
                Iterator<String> itr = mySet.iterator();
                while (itr.hasNext()) {
                        String element = itr.next();
                        System.out.println(element);
                }

// using advanced (enhanced) for loop:
                for (String item : mySet) {
                        System.out.println(item);
                }

// iterating over Set using forEach method:
                mySet.forEach(System.out::println);
        }
}
```

LinkedList

can store the duplicate elements.
maintains the insertion order
with LinkedList data manipulation is faster

```
public class LinkedListDemo {

public static void main(String args[]) {

        LinkedList ll = new LinkedList();// create a linked list

       // add elements to the linked list

      ll.add("F");

       ll.add("B");

      ll.add("D");

      ll.add("E");

      ll.add("C");

      ll.addLast("Z");

      ll.addFirst("A");

      ll.add(1, "A2");

       System.out.println("Original contents of ll: " + ll);

       // remove elements from the linked list

      ll.remove("F");

      ll.remove(2);

       System.out.println("Contents of ll after deletion: " + ll);


   ll.removeFirst();  // remove first and last elements

      ll.removeLast();

       System.out.println("ll after deleting first and last: " + ll);

       // get and set a value
```

```
        Object val = ll.get(2);

        ll.set(2, (String) val + " Changed");

        System.out.println("ll after change: " + ll);

  }

}
```

Vector

Vector is a grow-able *array* of objects, whose size is designed to be changed at run time. Dynamic array to store the data elements. Very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.

```
public static void main(String args[]) {
        Vector v = new Vector(3, 2);   // initial size is 3, increment is 2
        System.out.println("Initial size: " + v.size());
        System.out.println("Initial capacity: " + v.capacity());
        v.addElement(new Integer(1));
        v.addElement(new Integer(2));
        v.addElement(new Integer(3));
        v.addElement(new Integer(4));
        System.out.println("Capacity after four additions: " + v.capacity());
        v.addElement(new Double(5.45));
        System.out.println("Current capacity: " + v.capacity());
        v.addElement(new Double(6.08));
        v.addElement(new Integer(7));
    System.out.println("Current capacity: " + v.capacity());
        v.addElement(new Float(9.4));
        v.addElement(new Integer(10));
    System.out.println("Current capacity: " + v.capacity());
        v.addElement(new Integer(11));
        v.addElement(new Integer(12));
        System.out.println("First element: " + (Integer)v.firstElement());
        System.out.println("Last element: " + (Integer)v.lastElement());
        if(v.contains(new Integer(3)))
        System.out.println("Vector contains 3.");
        // enumerate the elements in the vector.
```

```
        Enumeration vEnum = v.elements();
        System.out.println("\nElements in vector:");
        while(vEnum.hasMoreElements())
        System.out.print(vEnum.nextElement() + " ");
        System.out.println();
  }
}
```

Stack Demo:

```
package collectionsdemo;
import java.util.Collections;
import java.util.Iterator;
import java.util.Stack;

public class StackDemo {
	public static void main(String args[]) {
		Stack<String> stk = new Stack<String>();
		stk.add("Ostrich");		stk.add("Iguana");		stk.add("Boa");
		stk.add("Crocodile");		stk.add("Zebra");		stk.add("Jackle");
		stk.add("Anaconda");		stk.add("Tapir");
		System.out.println(stk);
		stk.pop(); // removes the first one
		System.out.println(stk); // the first one removed

		// printing individual values using for loop:
		for (int i = 0; i < stk.size(); i++) {
			System.out.println(stk.get(i));

		}


		Sorting elements (ascending or alphabetical order):
		Collections.sort(stk);

		Sorting elements (descending or reverse alphabetical order):
		Collections.sort(stk, Collections.reverseOrder());


		Iterator<String> itr = stk.iterator();
		while (itr.hasNext()) {
			System.out.println(itr.next());
		}
	}
}
```

Queue Demo:

```
package collectionsdemo;
import java.util.LinkedList;
import java.util.Queue;
public class QueueDemo {
	public static void main(String[] args) {
		Queue<Integer> q = new LinkedList<>();
		q.add(111);

		for (int i = 0; i < 5; i++) {			// Adds elements {0, 1, 2, 3, 4} to queue
			q.add(i);
			}
		System.out.println("Elements of queue: " + q);	// Display contents of the queue.

		int removedele = q.remove();				// To remove the head of queue.
		System.out.println("removed element-" + removedele);

		System.out.println(q);

		int head = q.peek();					// To view the head of queue
		System.out.println("head of queue-" + head);

		// Rest all methods of collection interface,
		// Like size and contains can be used with this
		// implementation.
		int size = q.size();
		System.out.println("Size of queue-" + size);
	}
}
```

QueueDemo:

```
package collectionsdemo;
import java.util.Deque;
import java.util.Iterator;
import java.util.LinkedList;

public class DequeDemo {
	public static void main(String[] args) {
		Deque<String> myDeque = new LinkedList<String>();
		myDeque.add("new york");		myDeque.addFirst("boston");
		myDeque.addLast("miami");		myDeque.push("richmond");
		myDeque.offer("wyoming");		myDeque.offerFirst("lexington");
		myDeque.offerLast("hartford");
		System.out.println(myDeque	+ "\n");

		// Iterate through the queue elements.
		Iterator iterator = myDeque.iterator();
		while (iterator.hasNext())
			System.out.println("\t" + iterator.next());

		// Reverse order iterator
		Iterator reverse = myDeque.descendingIterator();
		System.out.println("Reverse Iterator");
		while (reverse.hasNext())
			System.out.println("\t" + reverse.next());

		// Peek returns the head, without deleting it from the dequeue
		System.out.println("Peek " + myDeque.peek());
		System.out.println("After peek: " + myDeque);

		// Pop returns the head, and removes it from the deque
		System.out.println("Pop " + myDeque.pop());
		System.out.println("After pop: " + myDeque);

		// We can check if a specific element exists in the deque
		System.out.println("Contains Boston? " + myDeque.contains("boston"));

		myDeque.removeFirst();			// We can remove the first / last element.
		myDeque.removeLast();
		System.out.println("Deque after removing " + "first and last: " + myDeque);
	}
}
```