Collections

V P JAYACHITRA

Why collections?

- Collections are used to hold a collection of objects.
- List holds objects based on order of insertion and can hold non unique collection
- Set holds objects without any order and are unique
- Maps holds objects based on a key and value pair, unique key and no specific order

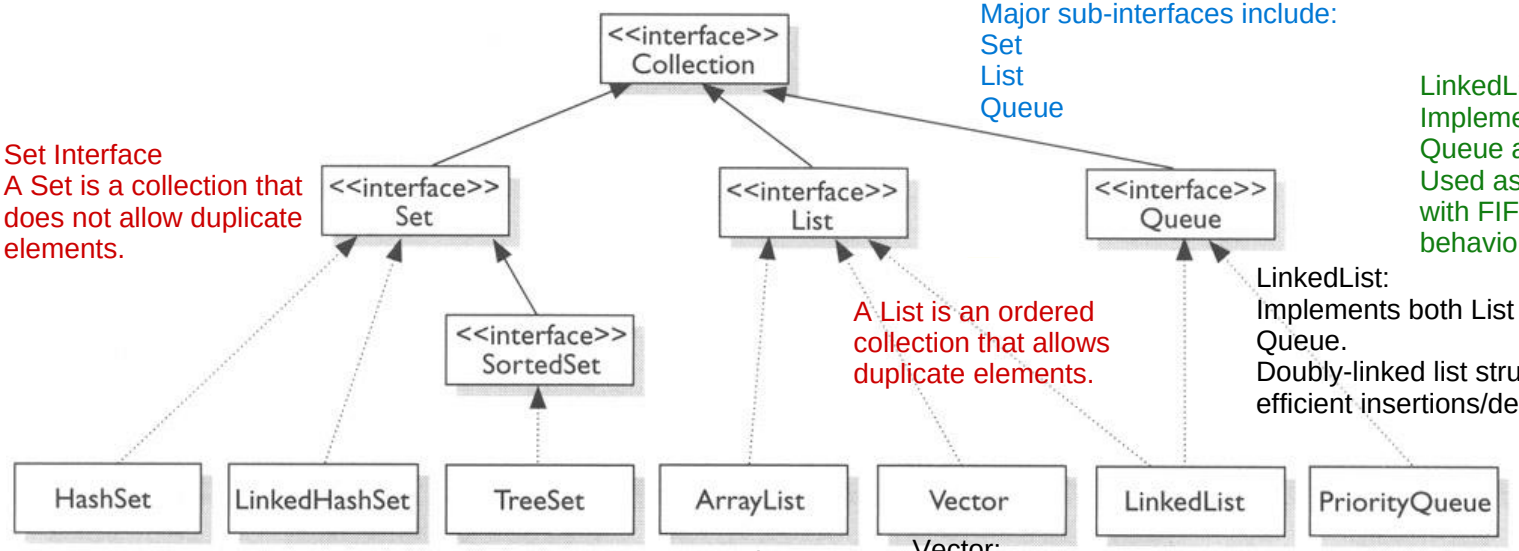
Different Collections Interfaces

- List
- Set
- Maps

Hierarchy

Collection is the root interface in the Java Collection Framework.
It provides the foundation for data structure implementations like lists, sets, and queues.

Major sub-interfaces include:
Set
List
Queue



Set Interface
A Set is a collection that does not allow duplicate elements.

A List is an ordered collection that allows duplicate elements.

LinkedList:
Implements both Queue and List.
Used as a queue with FIFO behavior.

LinkedList:
Implements both List and Queue.
Doubly-linked list structure for efficient insertions/deletions.

HashSet:
Implements a hash table to store elements.
Elements are unordered.
Allows null values.

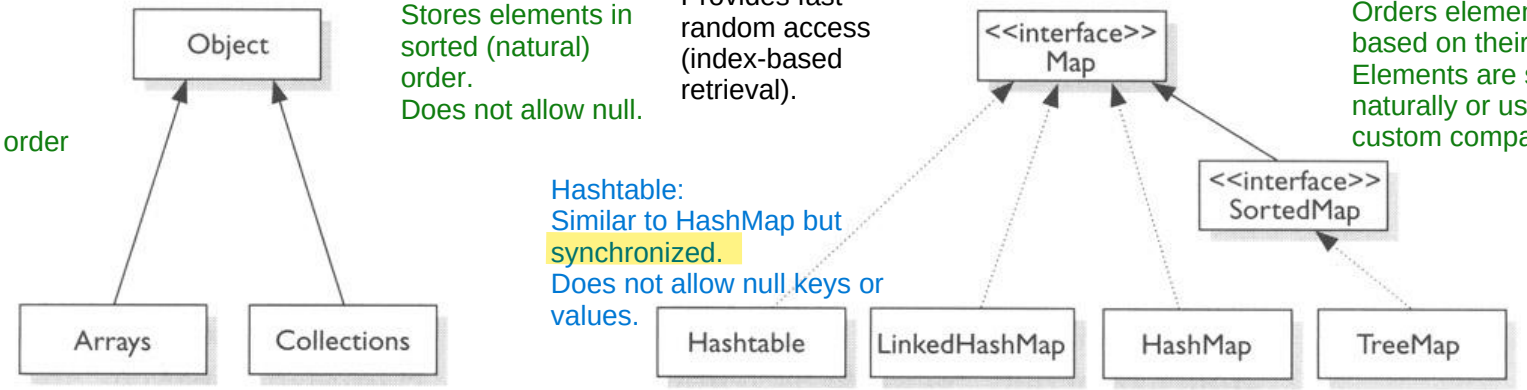
LinkedHashSet:
Extends HashSet.
Maintains insertion order using a linked list.
Allows null values.

TreeSet:
Implements SortedSet.
Stores elements in sorted (natural) order.
Does not allow null.

ArrayList:
Resizable array implementation.
Provides fast random access (index-based retrieval).

Vector:
Similar to ArrayList but is synchronized (thread-safe).
Has slightly more overhead than ArrayList.

PriorityQueue:
Orders elements based on their priority.
Elements are sorted naturally or using a custom comparator.



Hashtable:
Similar to HashMap but synchronized.
Does not allow null keys or values.

LinkedHashMap:
Extends HashMap.
Maintains insertion order of keys.

TreeMap:
Implements SortedMap.
Stores keys in sorted (natural) order.
Does not allow null keys.

HashMap:
Stores key-value pairs in a hash table.
Allows one null key and multiple null values.

List

- List is an interface with ArrayList, Vector and LinkedList as concrete implementations.
- List hold objects based on the insertion order and are non unique.
- Vector is deprecated
- Depending on the requirement ArrayList or LinkedList is used.
- Arraylist is used when more retrievals and less removals
- Linkedlist is used when more removals and less retrievals

List example

```
class mylist{
    public static void main(String args[]){
        List l = new ArrayList();
        Object o = new Object();
        l.add(o);// add o to list
        Object  x =(Object)l.get(0); // get the object in 0 position
        System.out.println(l.size()); // Print the size
    }
}
```

- This example creates an arrayList and adds an object ‘o’ to it.
- We get the object back and print the size of the list.
- Calling the get method does not remove the object from the list.
- You can use a list when the ordering is of prime importance.

```
Vector<String> vector = new Vector<>();
vector.addElement("item1");vector.addElement("item2");
```

Set

- A set interface has concrete implementation of HashSet, LinkedHashSet and TreeSet.
- The HashSet holds objects in no specific order (random order) and holds a unique set of objects.
- There is no get method in the Set interface. You will need to iterate over the set to get your object.
- Hence Set is used only as a collection of objects.
- A TreeSet stores objects in a sorted order. Every object to be inserted in the TreeSet has to implement the Comparable or Comparator interface else would result in a classcastexception.

Set example

```
class mysetExample{
	public static void main(String args[]){
		Set myset = new HashSet(); // create a hashset
		Object o = new Object(); // create an object
		boolean add = myset.add(o); // add object
		boolean addagain  myset.add(o); // add object again
		Iterator iter = myset.iterator(); // extract the iterator
		while(iter.hasNext()){
			Object o = (Object)iter.next();  // Iterate over the collection
		}
	}
}
```

- In this example we are creating a hashset and adding an object to it.
- The value of add will be true and the value of addagain will be false as the same object is already available in the set
- The iterator iterates over all the available objects in the set
- The equality of the objects is bounded by the equals(Object) and the hashcode() contract.

Map

- The Map Interface has two concrete implementations of HashMap, LinkedHashMap, TreeMap and Hashtable.
- The Map takes in a key and value pair as input for storage.
- The value can be retrieved if the key is known.
- The Map does not maintain order and should have unique keys.
- This is an efficient use as the complexity is O(1).
- Hashtable is a synchronized version of HashMap. ConcurrentHashMap is used as a substitute to Hashtable
- TreeMap stores key, values in the sorted order based on custom ordering

Map

```
Class myMap{
    public static void main(String args[]){
    Map maps = new HashMap();
    Object o = new Object();
    maps.put(“myobj”,o);
    Object o = (Object) maps.get(“myobj”);
}
}
```

- In this example we are creating a HashMap() and adding an object with key as myobj and value as the object.
- We can retrieve the object using the key and the get method.
- If you override the hashcode and equals method, then pay special attention to the contract else the implementations can lead to weird results

1. **List**:
    1. Ordered collection that maintains insertion order
    2. Allows duplicates
    3. Provides indexed access
    4. ArrayList is most commonly used implementation
2. **Set**:
    1. Unordered collection (HashSet)
    2. No duplicates allowed
    3. TreeSet maintains natural ordering
    4. Faster lookups than List
3. **Map**:
    1. Key-value pairs
    2. Keys must be unique
    3. HashMap is unordered
    4. TreeMap maintains natural ordering of keys

```java
// Example: Student Management System
public class StudentManagementSystem {

    // Using List: (ordered, allows duplicates)
    private List<Student> studentList = new ArrayList<>();

    // Using Set: Store unique student IDs
    private Set<String> studentIds = new HashSet<>();

    // Using Map: Store student grades (studentId -> grade)
    private Map<String, Integer> studentGrades = new HashMap<>();

    public static void main(String[] args) {
        StudentManagementSystem system = new StudentManagementSystem();
        system.demonstrateCollections();
    }

    public void demonstrateCollections() {
        System.out.println("=== LIST DEMONSTRATION ===");

        // Adding students to list
        studentList.add(new Student("S001", "John"));
        studentList.add(new Student("S002", "Alice"));
        studentList.add(new Student("S003", "Bob"));

        // Accessing students by index
        System.out.println("Second student: " + studentList.get(1).getName());

        // Iterating through list
        System.out.println("\nAll Students:");
        for(Student s : studentList) {
            System.out.println(s.getName());
        }
```

```java
// Student class
class Student {
    private String id;
    private String name;

    public Student(String id, String name) {
        this.id = id;
        this.name = name;
    }

    public String getId() { return id; }
    public String getName() { return name; }
}
```

```
// 2. SET USAGE - Managing unique IDs
        System.out.println("\n=== SET DEMONSTRATION ===");

        // Adding student IDs
        studentIds.add("S001");
        studentIds.add("S002");

        // Trying to add duplicate ID
        boolean added = studentIds.add("S001");
        System.out.println("Added duplicate ID? " + added); // false

        // Checking if ID exists
        System.out.println("Contains S001? " + studentIds.contains("S001"));

        // 3. MAP USAGE - Managing grades
        System.out.println("\n=== MAP DEMONSTRATION ===");

        // Adding grades
        studentGrades.put("S001", 85);
        studentGrades.put("S002", 90);
                // Updating grade
        studentGrades.put("S001", 88); // overwrites previous grade
                // Getting grade
        System.out.println("S001's grade: " + studentGrades.get("S001"));

        // Iterating through grades
        System.out.println("\nAll Grades:");
        for(Map.Entry<String, Integer> entry : studentGrades.entrySet()) {
            System.out.println("Student ID: " + entry.getKey() + ", Grade: " + entry.getValue());
        }     } }
```

```
// Student class
class Student {
    private String id;
    private String name;

    public Student(String id, String
name) {
        this.id = id;
        this.name = name;
    }

    public String getId() { return id; }
    public String getName() { return
name; }
}
```

	List	Set	Map
dec	List<String> list; ArrayList<String> arrayList; LinkedList<String> linkedList;	Set<String> set; HashSet<String> hashSet; TreeSet<String> treeSet;	Map<String, Integer> map; HashMap<String, Integer> hashMap; TreeMap<String, Integer> treeMap;
init	list = new ArrayList<>();	set = new HashSet<>();	map = new HashMap<>();

add	list.add("d");
	list.add(0, "d");
	list.addAll(collection);
		set.add("d");
		set.addAll(collection);
			map.put("key", value);
			map.putAll(otherMap);
			map.putIfAbsent("key", value);

access	String elem = list.get(0);
	list.indexOf("a");
	list.contains("a");
		set.contains("a");
		for(String s : set) {
		// iterate
		}
			Integer val = map.get("key");
			map.containsKey("key");

remove	list.remove(0);
	list.remove("a");
	list.clear();
		set.remove("a");
		set.removeAll(collection);
		set.clear();
			map.remove("key");
			map.remove("key", value);
			map.clear();

iterate	for(String s : list) {}
	list.forEach(item -> ...);
	Iterator<String> it =
	list.iterator();
		for(String s : set) {}
		set.forEach(item -> ...);
		Iterator<String> it =
		set.iterator();
			for(Map.Entry e : map.entrySet()){}
			map.forEach((k,v) -> ...);
			for(String key : map.keySet()){}

common	list.size();
	list.isEmpty();
	list.toArray();
		list.size();
		list.isEmpty();
		list.toArray();
			map.size();
			map.isEmpty();
			map.keySet();

	List	Set	Map
dec	**Ordered collection that maintains insertion**	Unordered collection (HashSet)	Key-value pairs
init	**Allows duplicates**	No duplicates allowed	Keys must be unique
add	Provides indexed access	TreeSet maintains natural ordering	HashMap is unordered
access	ArrayList is most commonly used implementation	Faster lookups than List	TreeMap maintains natural ordering of keys
	// List Operations **List<String> list = new ArrayList<>();** **list.add(element);** **// O(1) amortized** **list.get(index);** **// O(1)** **list.remove(index);** **// O(n)** **list.contains(element);** **// O(n)**	**// Set Operations** **Set<String> set = new HashSet<>();** **set.add(element);** **// O(1)** **set.remove(element);** **// O(1)** **set.contains(element);** **// O(1)**	**// Map Operations** **Map<String, Integer> map = new HashMap<>();** **map.put(key, value);** **// O(1)** **map.get(key);** **// O(1)** **map.remove(key);** **// O(1)** **map.containsKey(key);** **// O(1)**

```
public class CollectionDifferences {
    public static void main(String[] args) {
        // 1. ArrayList vs LinkedList
        List<String> arrayList = new ArrayList<>();  // Better for random access
        List<String> linkedList = new LinkedList<>(); // Better for insertions/deletions

        // 2. HashSet vs TreeSet
        Set<String> hashSet = new HashSet<>();      // Unordered, O(1) operations
        Set<String> treeSet = new TreeSet<>();      // Sorted, O(log n) operations

        // 3. HashMap vs TreeMap
        Map<String, Integer> hashMap = new HashMap<>();    // Unordered, O(1)
operations
        Map<String, Integer> treeMap = new TreeMap<>();    // Sorted by key, O(log n)
```

```
// Demonstrating Differences
        // 1. Order Maintenance
    List<String> list = new ArrayList<>();
    list.add("A"); list.add("C"); list.add("B");
    System.out.println("List (maintains order): " + list);  // [A, C, B]

    Set<String> hashset = new HashSet<>();
    hashset.add("A"); hashset.add("C"); hashset.add("B");
    System.out.println("HashSet (no order): " + hashset);   // Random order

    TreeSet<String> treeset = new TreeSet<>();
    treeset.add("A"); treeset.add("C"); treeset.add("B");
    System.out.println("TreeSet (sorted): " + treeset);     // [A, B, C]

    // 2. Duplicates
    list.add("A");
    System.out.println("List with duplicates: " + list);    // [A, C, B, A]

    hashset.add("A");
    System.out.println("Set without duplicates: " + hashset); // No duplicate A
     // 3. Null Values
    list.add(null);               // Allows null
    hashset.add(null);            // Allows one null
    // treeset.add(null);         // Throws NullPointerException
}
```

Further Reading

- http://docs.oracle.com/javase/tutorial/collections/interfaces/index.html
- http://java.sun.com/developer/onlineTraining/collections/Collection.html