Delhi Technological University

MC-214 Object Oriented Programming

Project Report on

Collection Framework

Submitted by

Mohit Sharma (DTU/2K14/MC/**041**)
Navjot Singh (DTU/2K14/MC/**045**)

4th Semester
(April 2016)

Acknowledgement

The successful completion of any task would be incomplete without accomplishing the people who made it possible and whose constant guidance and encouragement secured us the success.

First of all, we are grateful to the almighty for establishing us to complete this project report. We owe a debt to our faculty, Dr. S. K. Saxena for incorporating in us the idea of a creative project, helping us in undertaking this project and also for being there whenever we needed their assistance.

We also place on record, our sense of gratitude to one and all, who directly or indirectly have lent their helping hand in this venture.

Pre-Requisites

Java

> Java is a general-purpose computer programming language that is concurrent class based, object oriented and specifically designed to have as few implementation dependencies as possible.

It is intended to let application developers "write once, run anywhere" (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Java applications are typically compiled to bytecode that can run on any Java Virtual Machine (JVM) regardless of computer architecture.

Java was originally developed by James Gosling at Sun Microsystems (which has since been acquired by Oracle Corporation) and released in 1995. The language derives much of its syntax from C and C++, but it has fewer low-level facilities than both of them. The latest version is Java 8, which is the only version currently supported for free by Oracle.

Interfaces

> An interface is a reference type in Java which is similar to class but is essentially a collection of abstract methods

Interfaces are used to achieve 100% abstraction in java. Interfaces allow the implementation of multiple inheritance in Java which is otherwise, not allowed. A class "implements" one or more interfaces, thereby inheriting the abstract methods of all the interfaces implemented. Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class. However, these methods can be defined differently in each class which implements the interface.

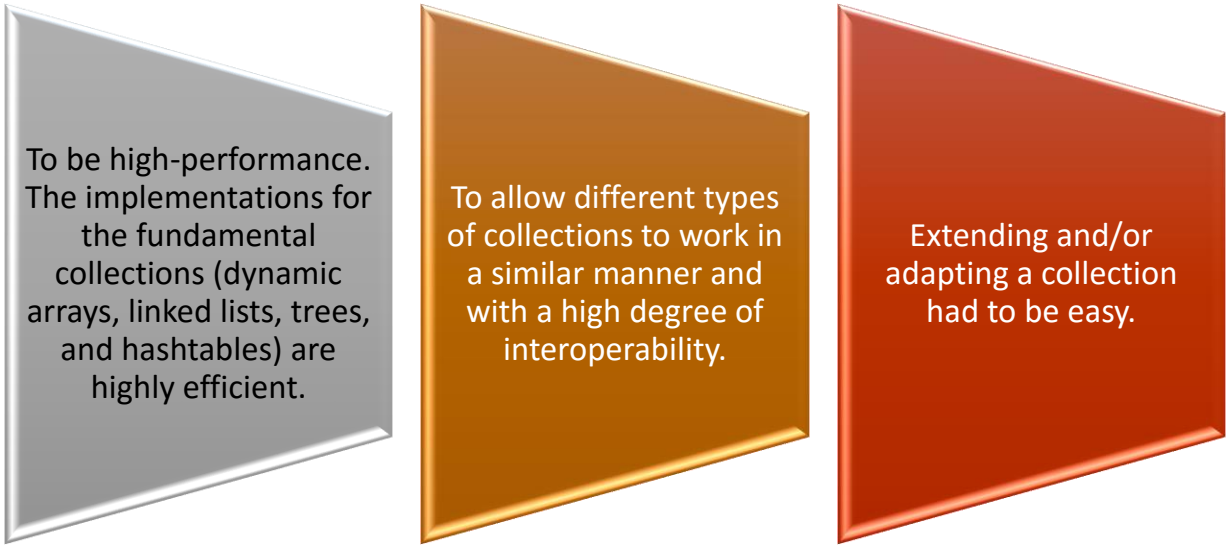
Collections

The Java collections framework (JCF) is a set of classes and interfaces that implement commonly reusable collection data structures. Although referred to as a "framework", it works in a manner of a "library". The JCF provides both interfaces that define various collections and classes that implement them. It was designed and developed primarily by Joshua Bloch, and was introduced in JDK 1.2.

Need of Collections

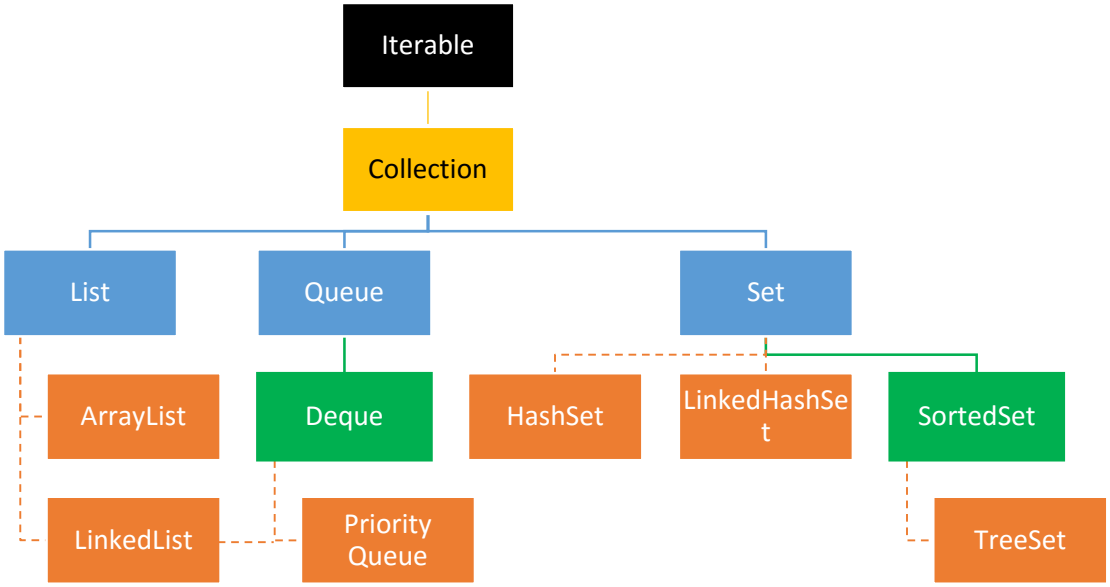
Earlier, Java provided ad hoc classes such as Dictionary, Vector, Stack and Properties to store and manipulate groups of objects. Although these classes were quite useful, but they lacked a central, unifying theme. They were not easy to extend, and did not implement a standard member interface.

Hence, the collection framework was designed to meet several goals:



The Framework

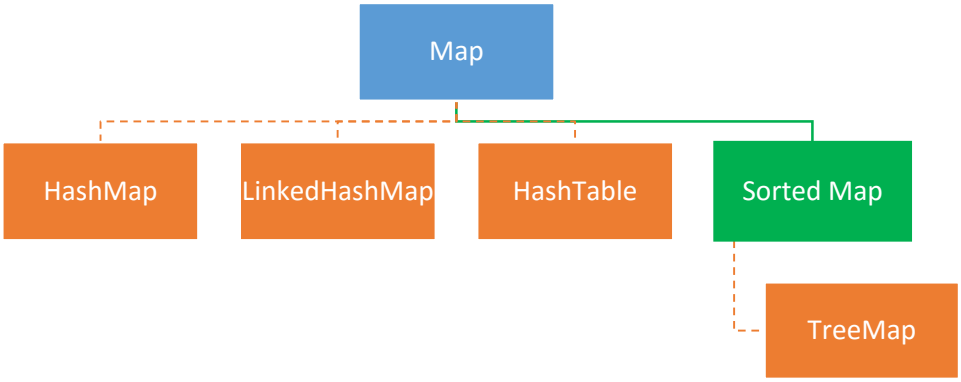
The collection framework consists of classes and interfaces, it can be represented by an organisation chart showing the hierarchical relationship between interfaces and classes. It must be noted that an interface/class extends another interface/class (represented by a solid line ____) while a class implements an interface (represented by dashed line ----). The following organisation chart shows the major classes and interfaces that form the collection framework.



Following in interpretations can be made from the chart-

- Collection interface implements the Iterable interface
- List, Queue & Set interfaces implement the Collection interface
- ArrayList & LinkedList implement the List interface
- Deque interface extends the Queue interface
- Priority Queue & Linked List implement the Deque interface
- HashSet and LinkedHashSet classes implement Set interface
- TreeSet implements SortedSet interface which extends Set interface

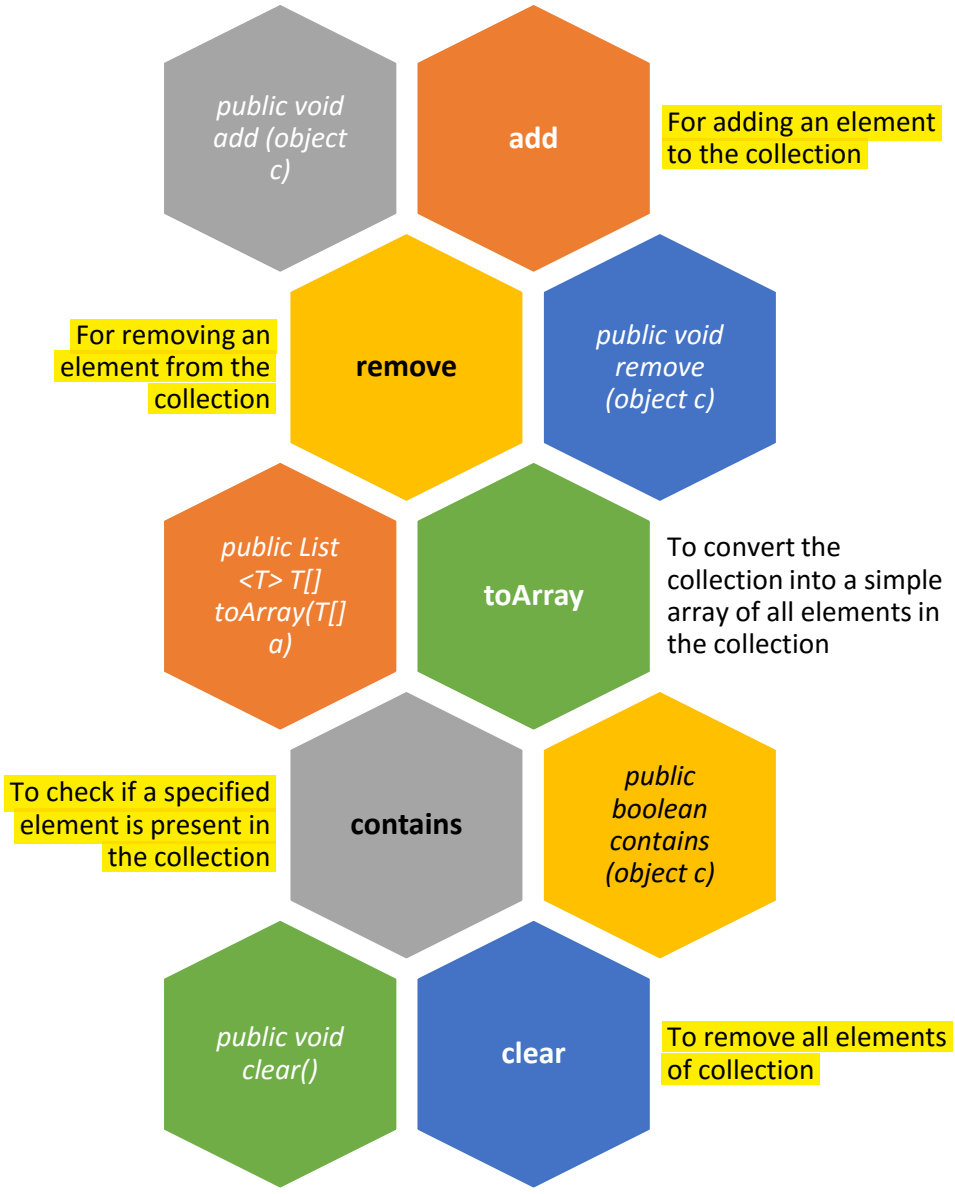
In addition to collections, the framework defines several map interfaces and classes. Although maps are not *collections* in the proper use of the term, but they are fully integrated with collections.



- HashMap, LinkedHashMap & HashTable classes implement Map interface
- TreeMap class implements SortedMap interface which extends Map interface

Collection Interface

Almost all collections in Java are derived from the **java.util.Collection** interface. Collection interface defines the basic parts of all collections. The interface states some basic methods to perform some basic functions. These include:



All collections have an iterator that goes through all of the elements in the collection. Additionally, Collection is a generic. Any collection can be written to store any class. For example, Collection<String> can hold strings, and the elements from the collection can be used as strings without any casting required

List Interface

Basically, a list collection stores elements by insertion order (either at the end or at a specific position in the list). A list maintains indices of its elements so it allows adding, retrieving, modifying, removing elements by an integer index.

A list can store objects of any types. Primitive types are automatically converted to corresponding wrapper types, e.g. integer numbers are converted to Integer objects. It allows null and duplicate elements, and orders them by their insertion order (index).

The List is the base interface for all list types, and the ArrayList and LinkedList classes are two common List's implementations.

ArrayList

- An implementation that stores elements in a backing array. The array's size is automatically expanded to make room for new elements added into the list.
- It offers constant time for the operations like size, isEmpty, get, set, iterator, and listIterator; amortized constant time for the add operation; and linear time for other operations.
- Therefore, this implementation can be considered for fast, random access of the elements.

LinkedList

- This is an implementation that stores elements in a doubly-linked list data structure.
- It offers constant time for adding and removing elements at the end of the list; and linear time for operations at other positions in the list.
- Therefore, we can consider using a LinkedList if fast adding and removing elements at the end of the list is required.

ArrayList can be demonstrated with the following code implementation:

```
import java.util.*;
public class ArrayListDemo {
	public static void main(String[] args){
			List<Integer> myList = new ArrayList<>();
			myList.add(290);	//Adding elements of subtypes of the declared type
			myList.add(122);
			myList.add(231);
			myList.add(2,2311);	// Insertion at a particular index
			Number num = myList.get(3);			// retrieving elements
			System.out.println(num + " retrieved");
			for (Number element : myList) {			//Iterating over the list
				System.out.print(element+"\t");}
			System.out.println();
			myList.set(2, 1223);		//Updating element
			Iterator itr = myList.iterator();	//Iterating the list using Iterator
			while(itr.hasNext()){
				System.out.print(itr.next()+"\t");}
			System.out.println();
```

```
                        if(myList.contains(123)){           // Searching
                                System.out.println("Element" + 123 +" found !!");}
                        Else    {System.out.println("Element not found !!");}
                        System.out.println("The index of 1223 is :"+ myList.indexOf(1223));
                        List<String> listStrings = new ArrayList<String>();// Sorting the List
                        listStrings.add("D");
                        listStrings.add("C");
                        listStrings.add("E");
                        listStrings.add("A");
                        listStrings.add("B");
                        System.out.println("listStrings before sorting: " + listStrings);
                        Collections.sort(listStrings);
                        System.out.println("listStrings after sorting: " + listStrings);
                        List<String> sourceList = new ArrayList<String>(); //Copying a list
                        sourceList.add("A");
                        sourceList.add("B");
                        sourceList.add("C");
                        sourceList.add("D");
                        List<String> destList = new ArrayList<String>();
                        destList.add("V");
                        destList.add("W");
                        destList.add("X");
                        destList.add("Y");
                        destList.add("Z");
                        System.out.println("destList before copy: " + destList);
                        Collections.copy(destList, sourceList);
                        System.out.println("destList after copy: " + destList);
                        List<Integer> numbers = new ArrayList<Integer>();  //Reversing the List
                        for (int i = 0; i <= 10; i++)
                                numbers.add(i);
                        System.out.println("List before reversing: " + numbers);
                        Collections.reverse(numbers);
                        System.out.println("List after reversing: " + numbers);
List<String> listNames = Arrays.asList("Mohit", "Jaya", "Ayush", "Navjot", "Arpit", "Aakash");
                        System.out.println("Original list: " + listNames);
                        List<String> subList = listNames.subList(3, 5);
                        System.out.println("Sub list: " + subList);
        }
}
```

Output:

```
231 retrieved
290     122     2311    231
290     122     1223    231
Element not found !!
The index of 1223 is :2
listStrings before sorting: [D, C, E, A, B]
listStrings after sorting: [A, B, C, D, E]
destList before copy: [V, W, X, Y, Z]
destList after copy: [A, B, C, D, Z]
List before reversing: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
List after reversing: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
Original list: [Mohit, Jaya, Ayush, Navjot, Arpit, Aakash]
Sub list: [Navjot, Arpit]
```

Queue Interface

A Queue is a collection for holding elements prior to processing. Besides basic Collection operations, queues provide additional insertion, removal, and inspection operations.

Queues typically, but not necessarily, orders elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which orders elements according to priority values. Whatever ordering is used, the head of the queue is the element that would be removed by a call to remove an element.

Queue implementations generally do not allow insertion of null elements. The LinkedList implementation, which was retrofitted to implement Queue, is an exception. For historical reasons, it permits null elements.

Queue Methods

Insertion	Removal	Inspection
•add() •Inserts an element unless queue's capacity restriction is violated, in which case it throws IllegalStateException •offer() •Differs from add only in that it indicates failure to insert an element by returning false.	•remove() •Returns head of the queue. When queue is empty, it throws NoSuchElementException •poll() •Returns head of the queue. When queue is empty, it returns null	•element() •Returns head of the queue. But throws NoSuchElementException if queue is empty •peek() •Returns head of the queue. Returns null if queue is empty

Following code can be used to demonstrate Queue implementation:

```
import java.util.*;
public class QueueDemo {
    public static void main(String[] args){
        Queue<String> myQueue = new LinkedList<>();
        boolean flag = myQueue.offer("alpha");
        System.out.println("offer() exited with return value :"+ flag);
        if(flag){
                System.out.println("Element added successfully");
        }else{
                System.out.println("Element could not be added");
        }
        try{
            myQueue.add("beta");
        }catch(IllegalStateException e){
                e.printStackTrace();
        }
        String head = myQueue.poll();
        System.out.println(head + " removed from the queue");
        System.out.println("New head of the Queue is :"+ myQueue.peek());
```

```
        try{
                System.out.println("Head of the Queue is :"+ myQueue.element());
                head = myQueue.remove();
                System.out.println(head + " removed from the queue");
                System.out.println("New head of the Queue is :"+ myQueue.peek());
        }catch(NoSuchElementException e){
                e.printStackTrace();
        }
    }
}
```

Output:

```
Element added successfully
alpha removed from the queue
New head of the Queue is :beta
Head of the Queue is :beta
beta removed from the queue
New head of the Queue is :null
```

Deque Interface

Usually pronounced as deck, a deque is a double-ended-queue. A double-ended-queue is a linear collection of elements that supports the insertion and removal of elements at both end points. The Deque interface is a richer abstract data type than both Stack and Queue because it implements both stacks and queues at the same time. The Dequeue interface, defines methods to access the elements at both ends of the Deque instance.

Deque Methods

Insertion	Removal	Retrieve
•The addfirst and offerFirst methods insert elements at the beginning of the Deque. •The methods addLast and offerLast insert elements at the end of theDeque instance.	•The removeFirst and pollFirst methods remove elements from the beginning of the Deque. •The removeLast and pollLast methods remove elements from the end.	•The methods getFirst and peekFirst retrieve the first element of the Deque. These methods dont remove the value from the Deque. •Similarly, the methods getLast and peekLast retrieve the last element.

Additional Predefined Methods
• removeFirstOccurence, removes the first occurence of the specified element • removeLastOccurence; removes the last occurence of the specified element in the Deque.

Following code can be used to demonstrate Queue implementation:

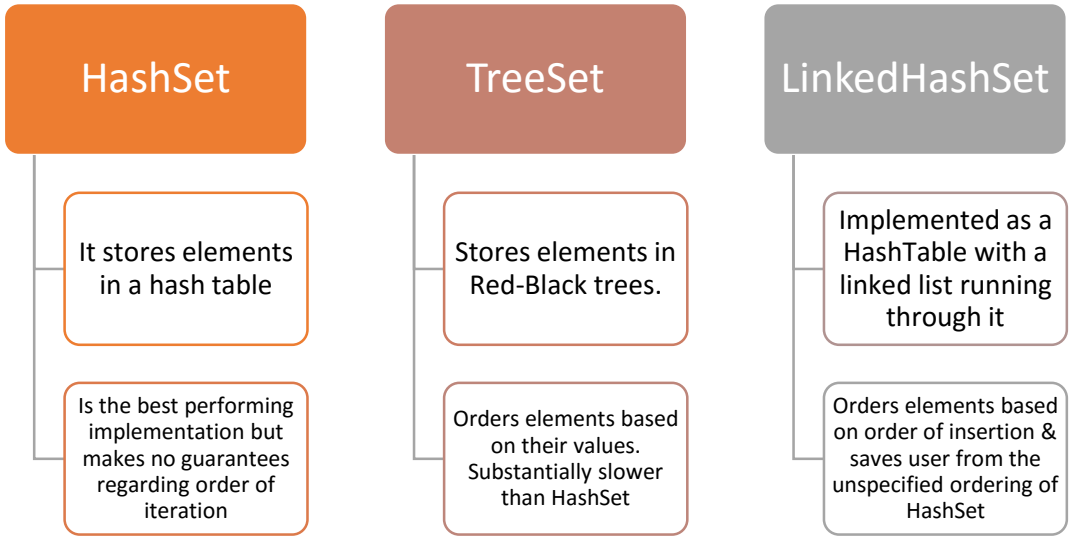
```
import java.util.*;
public class DequeueDemo {
    public static void main(String[] args){
        Deque<String> myDeque = new LinkedList<>();
        try{
                myDeque.add("Ayush");
                myDeque.addFirst("Mohit");
                myDeque.addLast("Navjot");
        }catch(IllegalStateException e){
                e.printStackTrace();}
        myDeque.offer("Arpit");
        myDeque.offerFirst("Jaya");
        myDeque.offerLast("Aakash");
        System.out.println("Iterator through the queue elements.");
        Iterator itr = myDeque.iterator();
        while(itr.hasNext()){
                System.out.print(itr.next()+"\t");}
        System.out.println("\nReverse iterator through the queue elements.");
        Iterator rev_itr = myDeque.descendingIterator();
        while(rev_itr.hasNext()){
                System.out.print(rev_itr.next()+"\t");}
        System.out.println("\nThe head of the Deque is "+ myDeque.peek());
        String head = myDeque.pop();
        System.out.println(head+" is popped out");
        System.out.println(myDeque);
        System.out.println("The head of the Deque is "+myDeque.getFirst());
        System.out.println("The tail of the Deque is "+myDeque.getLast());
        try{
                head = myDeque.removeFirst();
                System.out.println(head+" is removed out");
                System.out.println(myDeque);
                System.out.println("The head of the Deque is "+myDeque.getFirst());
                head = myDeque.removeLast();
                System.out.println(head+" is removed out");
                System.out.println(myDeque);
                System.out.println("The tail of the Deque is "+myDeque.getLast());
        }catch(NoSuchElementException e){
                e.printStackTrace();}
        head = myDeque.pollFirst();
        System.out.println("The head of the Deque is "+myDeque.getFirst());
        head = myDeque.pollLast();
        System.out.println(head+" is removed out");
        System.out.println(myDeque);
        System.out.println("The tail of the Deque is "+myDeque.getLast());
    }
}
```

Output:

```
Iterator through the queue elements.
Jaya    Mohit   Ayush   Navjot  Arpit    Aakash
Reverse iterator through the queue elements.
Aakash  Arpit    Navjot  Ayush   Mohit   Jaya
The head of the Deque is Jaya
Jaya is popped out
[Mohit, Ayush, Navjot, Arpit, Aakash]
The head of the Deque is Mohit
The tail of the Deque is Aakash
Mohit is removed out
[Ayush, Navjot, Arpit, Aakash]
The head of the Deque is Ayush
Aakash is removed out
[Ayush, Navjot, Arpit]
The tail of the Deque is Arpit
Ayush is removed out
[Navjot, Arpit]
The head of the Deque is Navjot
Arpit is removed out
[Navjot]
The tail of the Deque is Navjot
```

Set Interface

A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction. The Set interface contains *only* methods inherited from Collection and adds the restriction that duplicate elements are prohibited. Set instances to be compared meaningfully even if their implementation types differ.



Basic Operations

- The size operation returns the number of elements in the Set (*Cardinality*).
- isEmpty checks if set is empty
- The add method adds the specified element to the Set if it is not already present and returns a boolean indicating whether the element was added.
- The remove method removes the specified element from the Set if it is present and returns a boolean indicating whether the element was present.
- The iterator method returns an Iterator over the Set.

Bulk Operations

- Perform standard Set-algebraic operations. Suppose s1 and s2 are sets. Here's what bulk operations do:
  - s1.containsAll(s2) — returns true if s2 is a **subset** of s1
  - s1.addAll(s2) — transforms s1 into the **union** ofs 1 and s2
  - s1.retainAll(s2) — transforms s1 into the intersection of s1 and s2
  - s1.removeAll(s2) — transforms s1 into the (asymmetric) set difference of s1 and s2

Following code can be used to demonstrate Set implementation:

```
import java.util.*;
public class mohit_sharma {
    public static void main(String[] args){
        Set<String> s = new HashSet<>();
        Scanner mohit = new Scanner(System.in);
        for(int i =0;i< 5;i++){
                String variable = mohit.next();

                s.add(variable);
        }
        Set<String> s1 = new TreeSet<String>();
        for(int i =0;i< 5;i++){
                String variable = mohit.next();

                s1.add(variable);
        }
        Set<String> Union = new HashSet<>(s);
        Union.addAll(s1);
        Set<String> Intersection = new HashSet<>(s);
        Intersection.retainAll(s1);
        Set<String> Difference= new HashSet<>(s);
        Difference.removeAll(s1);

                System.out.println(s.size()+ " distinct words"+ s);
                System.out.println(s1.size()+ " distinct words"+ s1);
                System.out.println(Union.size()+ " distinct words"+ Union);
                System.out.println(Intersection.size()+ " distinct words" + Intersection);
                System.out.println(Difference.size()+ " distinct words"+ Difference);
    }
}
```

Input:

```
q
w
e
r
ts
de
rf
e
w
q
```

Output:

```
5 distinct words[w, e, ts, r, q]
5 distinct words[de, e, q, rf, w]
7 distinct words[de, w, e, ts, r, q, rf]
3 distinct words[w, e, q]
2 distinct words[ts, r]
```

Map Interface

A Map is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value. It models the mathematical *function* abstraction. Java contains three general-purpose Map implementations:

1. HashMap
2. TreeMap
3. LinkedHashMap

Their behaviour and performance are precisely analogous to HashSet, TreeSet, and LinkedHashSet. The Map interface includes the following components-

Basic Operations

- The basic operations of Map (put, get, containsKey, containsValue, size, and isEmpty) behave exactly like their counterparts in Hashtable.

Bulk Operations

- The clear operation removes all the mappings from the Map.
- The putAll operation is the Map analogue of the Collection interface's addAll operation

Collection View

- Collection view methods allow a Map to be viewed as a Collection in these three ways:
  - keySet — the Set of keys contained in theMap.
  - Values — The Collection of values contained in the Map. This Collection is not a Set, because multiple keys can map to the same value.
  - EntrySet — the Set of key-value pairs contained in the Map. The Map interface provides a small nested interface called Map.Entry, the type of the elements in this Set

The most common class that implements the Map interface is the Java HashMap. A HashMap is a hash table based implementation of the Map interface. It does not maintain any order among its elements and does not guarantee a constant order over time. Following code gives a simple implementation of Java HashMap:

```
import java.util.*;
public class MapDemo {
    public static void main(String[] args){
        Map<String, String> dept_student = new TreeMap<>();
        dept_student.put("WEBD", "Mohit Sharma");
        dept_student.put("APPD", "Arpit Gogia");
        dept_student.put("CODER", "Aakash Dabas");
        dept_student.put("DR", "Ayush Sinha");
        dept_student.put("COOLDUDE", "Navjot Singh");
        dept_student.put("NONTECH", "Jaya Ranjan");
        Set<String> keys =dept_student.keySet();
        for(String token: keys)
        System.out.println("Character :"+token +" Name :"+ dept_student.get(token));
        System.out.println("Total Entries :"+dept_student.size());
        String searchKey = "NONTECH";
        if(dept_student.containsKey(searchKey)){
               System.out.println("Element found !!");
               System.out.println("Character: "+searchKey+" Name :"+
dept_student.get(searchKey));
String temp = dept_student.remove(searchKey);
               System.out.println("Character: "+searchKey+" Name :"+temp+" Removed from
Map");
        }
        String searchValue = "Mohit Sharma";
        if(dept_student.containsValue(searchValue)){
               System.out.println("Element found !!");}
        dept_student.clear();
        System.out.println("After clear operation, size: " + dept_student.size());
    }
}
```

Output:

```
Character :APPD Name :Arpit Gogia
Character :CODER Name :Aakash Dabas
Character :COOLDUDE Name :Navjot Singh
Character :DR Name :Ayush Sinha
Character :NONTECH Name :Jaya Ranjan
Character :WEBD Name :Mohit Sharma
Total Entries :6
Element found !!
Character: NONTECH Name :Jaya Ranjan
Character: NONTECH Name :Jaya Ranjan Removed
from Map
Element found !!
After clear operation, size: 0
```

Benefits of the Java Collections Framework

The Java Collections Framework provides the following benefits:

Reduces Programming Effort

- By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work. By facilitating interoperability among unrelated APIs, the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs.

Increases Program Speed & Quality

- It provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. Because you're freed from the drudgery of writing your own data structures, you'll have more time to devote to improving programs' quality and performance.

Allows inter-operability among unrelated APIs

- The collection interfaces are the vernacular by which APIs pass collections back and forth. If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings, our APIs will interoperate seamlessly, even though they were written independently.

Reduces effort to learn and to use new APIs

- Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections. There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them. With the advent of standard collection interfaces, the problem went away.

Reduces effort to design new APIs

- This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.

Fosters software reuse

- New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.