UNIT-10 List, Stacks, Queues and Priority Queues

Collection, Iterators, Lists, The Comparator interface, static methods for list and collections, Vector and Stack classes, Queues and priority Queues.(Chapter -20)

UNIT-11 Sets and Maps (Chapter 21)

Comparing the performance of Sets and Lists
singleton and unmodifiable collections and Maps

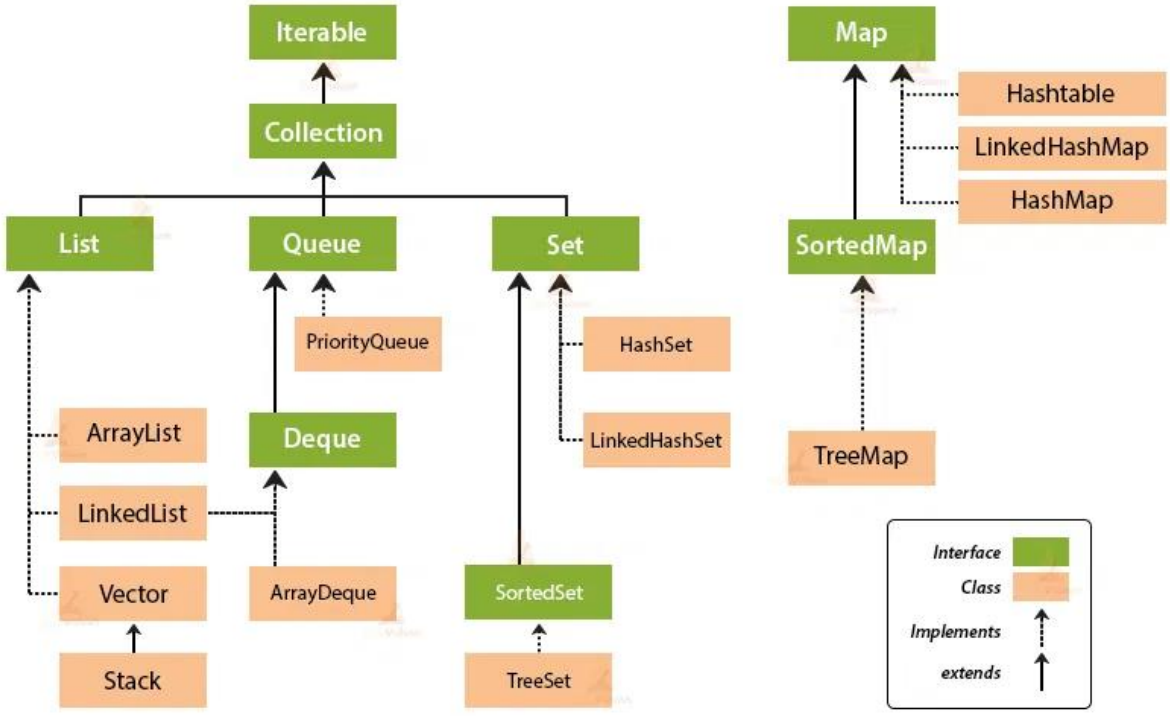
Collection

Choosing the best data structures and algorithms for a particular task is one of the keys to developing high-performance software.

The **Collection interface** defines the common operations for lists, vectors, stacks, queues, priority queues, and sets.

COLLECTION FRAMEWORK

The Java Collections Framework supports two types of containers:
■ One for storing a collection of elements is simply called a collection.
■ The other, for storing key/value pairs, is called a map.



Now let us see the following collections.

Lists

- The List interface is an ordered collection that allows us to add and remove elements like an array.
- To create a list, use one of its concrete classes: **ArrayList**, **LinkedList, Vector or Stack.**

- **Vector** is a subclass of **AbstractList**, and **Stack** is a subclass of **Vector** in the Java API.
- **Vector** is the same as **ArrayList**, except that it contains synchronized methods for accessing and modifying the vector. Synchronized methods can prevent data corruption when a vector is accessed and modified by two or more threads concurrently. For the many applications that do not require synchronization, using ArrayList is more efficient than using Vector.

- In the Java Collections Framework, **Stack** is implemented as an extension of Vector. The Stack class extends Vector to provide a last-in, first-out data structure.

Sets

- The Set interface allows us to store elements in different sets similar to the set in mathematics. It cannot have duplicate elements.
- You can create a set using one of its three concrete classes: **HashSet, LinkedHashSet, or TreeSet.**

Queues

- The Queue interface is used when we want to store and access elements in **First In, First Out(FIFO)** manner.
- In FIFO, first element is removed first and last element is removed at last.
- To create a list, use one of its concrete classes: **PriorityQueue or LinkedList**

- A queue is a first-in, first-out data structure. Elements are appended to the end of the queue and are removed from the beginning of the queue.
- In a priority queue, elements are assigned priorities. When accessing elements, the element with the highest priority is removed first.
- The **PriorityQueue** class provides the facility of using queue. But it does not orders the elements in FIFO manner. It inherits **AbstractQueue** class.

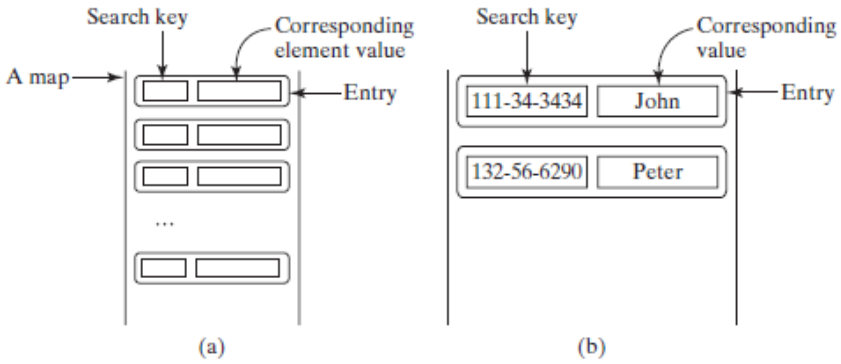
The declaration for java.util.PriorityQueue class is as follows:

```
public class PriorityQueue<E> extends AbstractQueue<E> implements Serializable
```

- By default, the priority queue orders its elements according to their natural ordering using **Comparable**.
- The element with the least value is assigned the highest priority and thus is removed from the queue first.

Map

- Map is like a dictionary that provides a quick look up to retrieve a value using a key.
- *You can create a map using one of its three concrete classes:* **HashMap, LinkedHashMap, or TreeMap.**
- A *map* is a container object that stores a collection of key/value pairs.
- It enables fast retrieval, deletion, and updating of the pair through the key.
- A map stores the values along with the keys. The keys are like indexes.
- In **List**, the indexes are integers.
- In **Map**, the keys can be any objects.
- A map cannot contain duplicate keys. Each key maps to one value.
- A key and its corresponding value form an entry stored in a map.



Methods of Collection Interface

The Collection interface includes various methods that can be used to perform different operations on objects. These methods are available in all its subinterfaces.

- add() - inserts the specified element to the collection
- size() - returns the size of the collection
- remove() - removes the specified element from the collection
- iterator() - returns an iterator to access elements of the collection
- addAll() - adds all the elements of a specified collection to the collection
- removeAll() - removes all the elements of the specified collection from the collection
- clear() - removes all the elements of the collection

❖ List interface

In Java, the List interface is an ordered collection that allows us to store and access elements sequentially. It extends the Collection interface.

Concrete Classes that Implement List

Since List is an interface, we cannot create objects from it. In order to use functionalities of the List interface, we can **ArrayList,,LinkedList,Vector and Stack** class.

How to use List?

In Java, we must import `java.util.List` package in order to use `List`.

```
// ArrayList implementation of List
List<String> list1 = new ArrayList<>();

// LinkedList implementation of List
List<String> list2 = new LinkedList<>();
```

Here, we have created objects *list1* and *list2* of classes `ArrayList` and `LinkedList`. These objects can use the functionalities of the `List` interface.

The `List` interface includes all the methods of the `Collection` interface. Its because `Collection` is a super interface of `List`.

Methods of the `List` interface are:

- `get()` - helps to randomly access elements from lists
- `toArray()` - converts a list into an array
- `contains()` - returns `true` if a list contains specified element

1. Implementing the ArrayList Class

```
import java.util.List;
import java.util.ArrayList;

class Main {
    public static void main(String[] args) {
        // Creating list using the ArrayList class
        List<Integer> numbers = new ArrayList<>();

        // Add elements to the list
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        System.out.println("List: " + numbers);

        // Access element from the list
        int number = numbers.get(2);
        System.out.println("Element at index 2: " + number);

        numbers.set(2, 1000); // Replaces 3rd element
        System.out.println("Replaced 3rd element:" + numbers);

        // Remove element from the list
        int removedNumber = numbers.remove(1);
        System.out.println("Removed Element at index 1: " + removedNumber);

          // Returns true if 1000 present
        System.out.println("Does list have 1000?" + numbers.contains(1000));

        numbers.remove(Integer.valueOf(1000)); // This will remove 1000 from the list
```

```
        System.out.println("Removed element 1000, New list: " + numbers);

        numbers.clear(); //This will remove all the elements from the list.
        System.out.println("Cleared list" + numbers);
    }
}
```

Output

```
List: [1, 2, 3]
Element at index 2: 3
Replaced 3rd element:[1, 2, 1000]
Removed Element at index 1: 2
Does list have 1000?true
Removed element 1000, New list: [1]
Cleared list[]
```

2. Implementing the LinkedList Class

```
import java.util.List;
import java.util.LinkedList;

class Main {
    public static void main(String[] args) {
        // Creating list using the ArrayList class
        List<Integer> numbers = new LinkedList<>();

        // Add elements to the list
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        System.out.println("List: " + numbers);

        // Access element from the list
        int number = numbers.get(2);
        System.out.println("Element at index 2: " + number);

        numbers.set(2, 1000); // Replaces 3rd element
        System.out.println("Replaced 3rd element:" + numbers);

        // Remove element from the list
        int removedNumber = numbers.remove(1);
        System.out.println("Removed Element at index 1: " + removedNumber);

          // Returns true if 1000 present
        System.out.println("Does list have 1000?" + numbers.contains(1000));

        numbers.remove(Integer.valueOf(1000)); // This will remove 1000 from
the list
        System.out.println("Removed element 1000, New list: " + numbers);

       numbers.clear(); //This will remove all the elements from the list.
       System.out.println("Cleared list" + numbers);
    }
}
```

Output

```
List: [1, 2, 3]
Element at index 2: 3
Replaced 3rd element:[1, 2, 1000]
Removed Element at index 1: 2
Does list have 1000?true
Removed element 1000, New list: [1]
Cleared list[]
```

3. Implementing the Vector Class

```
import java.util.*;
class Main {
	public static void main(String[] args)
	{
		int n = 5;
		Vector<Integer> v = new Vector<>(n);
		// Appending new elements at the end of the vector
		for (int i = 1; i <= n; i++)
			v.add(i);
		// Printing elements
		System.out.println(v);
		// Remove element at index 3
		v.remove(3);
		// Displaying the vector after deletion
		System.out.println(v);
		// Using set() method to replace 12 with 21
		System.out.println("The Object that is replaced is:"+ v.set(0, 21));
		// iterating over vector elements using for loop
		for (int i = 0; i < v.size(); i++)
			System.out.print(v.get(i) + " ");
	}
}
```

Output

```
[1, 2, 3, 4, 5]
[1, 2, 3, 5]
The Object that is replaced is: 1
21 2 3 5
```

4. Implementing the Stack Class

Stack Methods

Since Stack extends the Vector class, it inherits all the methods Vector. To learn about different Vector methods, visit Java Vector Class. Besides these methods, the Stack class includes 5 more methods that distinguish it from Vector.

```
import java.util.Stack;
```

```
public class Main {
    public static void main(String[] args) {
        Stack<String> animals= new Stack<>();
        animals.push("Lion");
        animals.push("Dog");
        animals.push("Horse");
        animals.push("Cat");
        System.out.println("Stack: " + animals);
        System.out.println(animals.peek());
        animals.pop();
        System.out.println("Stack: " + animals);
        System.out.println(animals.peek());
        // Search an element
        int index = animals.search("Horse");
        System.out.println("Index of Horse: " + index);
        // Check if stack is empty
        boolean result = animals.empty();
        System.out.println("Is the stack empty? " + result);

    }
}
```

Output

```
Stack: [Lion, Dog, Horse, Cat]
Cat
Stack: [Lion, Dog, Horse]
Horse
Index of Horse: 1
Is the stack empty? false
```

❖ Set interface

The **Set** interface extends the **Collection** interface, as shown in Figure. It does not introduce new methods or constants, but it stipulates that an instance of Set contains no duplicate elements.

1. HashSet Class

The **HashSet** class is a concrete class that implements Set. You can create an empty hash set using its no-arg constructor or create a hash set from an existing collection.

```
import java.util.*;
 public class Main {
 public static void main(String[] args) {
 // Create a hash set
 Set<String> set = new HashSet<>();
 // Add strings to the set
 set.add("London");
 set.add("Paris");
 set.add("New York");
 set.add("San Francisco");
 set.add("Beijing");
 set.add("New York");
```

```
System.out.println(set);
// Display the elements in the hash set
for (String s: set) {
System.out.print(s.toUpperCase() + " ");
}
}
}
```

Output

```
[San Francisco, Beijing, New York, London, Paris]
SAN FRANCISCO BEIJING NEW YORK LONDON PARIS
```

There is no particular order for the elements in a hash set. To impose an order on them, you need to use the **LinkedHashSet** class, which is introduced in the next section.

2. LinkedHashSet Class

LinkedHashSet extends **HashSet** with a linked-list implementation that supports an ordering of the elements in the set. The elements in a HashSet are not ordered, but the elements in a LinkedHashSet can be retrieved in the order in which they were inserted into the set.

```
import java.util.*;
 public class Main {
 public static void main(String[] args) {
 // Create a hash set
 Set<String> set = new LinkedHashSet<>();

 // Add strings to the set
 set.add("London");
 set.add("Paris");
 set.add("New York");
 set.add("San Francisco");
 set.add("Beijing");
 set.add("New York");
 System.out.println(set);

 // Display the elements in the hash set
 for (Object element: set)
 System.out.print(element);
 }
 }
```

Output

```
[London, Paris, New York, San Francisco, Beijing]
LondonParisNew YorkSan FranciscoBeijing
```

To impose a different order (e.g., increasing or decreasing order), you can use the **TreeSet** class.

3. TreeSet Class

SortedSet is a subinterface of **Set**, which guarantees that the elements in the set are sorted.

```
import java.util.*;
 public class Main {
 public static void main(String[] args) {
 // Create a hash set
 Set<String> set = new HashSet<>();
 // Add strings to the set
 set.add("London");
 set.add("Paris");
 set.add("New York");
 set.add("San Francisco");
 set.add("Beijing");
 set.add("New York");
 TreeSet<String> treeSet = new TreeSet<>(set);
 System.out.println("Sorted tree set: " + treeSet);

 // Use the methods in SortedSet interface
 System.out.println("first(): " + treeSet.first());
 System.out.println("last(): " + treeSet.last());
 System.out.println("headSet(\"New York\"): " +
 treeSet.headSet("New York"));
 System.out.println("tailSet(\"New York\"): " +
 treeSet.tailSet("New York"));

 // Use the methods in NavigableSet interface
 System.out.println("lower(\"P\"): " + treeSet.lower("P"));
 System.out.println("higher(\"P\"): " + treeSet.higher("P"));
 System.out.println("floor(\"P\"): " + treeSet.floor("P"));
 System.out.println("ceiling(\"P\"): " + treeSet.ceiling("P"));
 System.out.println("pollFirst(): " + treeSet.pollFirst());
 System.out.println("pollLast(): " + treeSet.pollLast());
 System.out.println("New tree set: " + treeSet);
 }
 }
```

Output

```
Sorted tree set: [Beijing, London, New York, Paris, San Francisco]
first(): Beijing
last(): San Francisco
headSet("New York"): [Beijing, London]
tailSet("New York"): [New York, Paris, San Francisco]
lower("P"): New York
higher("P"): Paris
floor("P"): New York
ceiling("P"): Paris
pollFirst(): Beijing
pollLast(): San Francisco
New tree set: [London, New York, Paris]
```

treeSet.first() returns the first element in **treeSet** (line 20)
treeSet.last() returns the last element in **treeSet** (line 21).
treeSet.headSet("New York") returns the elements in **treeSet** before New York (lines 22–23).
treeSet.tailSet("New York") returns the elements in **treeSet** after New York, including New York (lines 24–25).
treeSet.lower("P") returns the largest element less than **P** in **treeSet** (line 28).
treeSet.higher("P") returns the smallest element greater than **P** in **treeSet** (line 29).
treeSet.floor("P") returns the largest element less than or equal to **P** in **treeSet** (line 30).
treeSet.ceiling("P") returns the smallest element greater than or equal to **P** in **treeSet** (line 31).
treeSet.pollFirst() removes the first element in **treeSet** and returns the removed element (line 32).
treeSet.pollLast() removes the last element in **treeSet** and returns the removed element (line 33).

❖ Queue interface

PriorityQueue class

Methods

- add() - Inserts the specified element to the queue. If the queue is full, it throws an exception.
- offer() - Inserts the specified element to the queue. If the queue is full, it returns false.
- remove() - removes the specified element from the queue
- poll() - returns and removes the head of the queue

```
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Queue;
public class Main {
    public static void main(String[] args) {
        Queue<Integer> pq = new PriorityQueue<>(Comparator.reverseOrder());
        pq.offer(40);
        pq.offer(12);
        pq.offer(24);
        pq.offer(36);
        System.out.println(pq);
        pq.poll();
        System.out.println(pq);
        System.out.println(pq.peek());
        pq.remove(12);
        System.out.println(pq);
    }
}
```

Output

```
[40, 36, 24, 12]
[36, 12, 24]
36
[36, 24]
```

❖ Map interface

```
import java.util.*;
 public class Main {
 public static void main(String[] args) {
 // Create a HashMap
 Map<String, Integer> hashMap = new HashMap<>();
 hashMap.put("Smith", 30);
 hashMap.put("Anderson", 31);
 hashMap.put("Lewis", 29);
 hashMap.put("Cook", 29);

 System.out.println("Display entries in HashMap");
 System.out.println(hashMap + "\n");

 // Create a TreeMap from the preceding HashMap
 Map<String, Integer> treeMap =  new TreeMap<>(hashMap);
 System.out.println("Display entries in ascending order of key");
 System.out.println(treeMap);

 // Create a LinkedHashMap
 Map<String, Integer> linkedHashMap =
 new LinkedHashMap<>(16, 0.75f, true);
 linkedHashMap.put("Smith", 30);
 linkedHashMap.put("Anderson", 31);
 linkedHashMap.put("Lewis", 29);
 linkedHashMap.put("Cook", 29);

 // Display the age for Lewis
 System.out.println("\nThe age for " + "Lewis is " +
 linkedHashMap.get("Lewis"));

 System.out.println("Display entries in LinkedHashMap");
 System.out.println(linkedHashMap);
 }
 }
```

Output

```
Display entries in HashMap
{Lewis=29, Smith=30, Cook=29, Anderson=31}

Display entries in ascending order of key
{Anderson=31, Cook=29, Lewis=29, Smith=30}

The age for Lewis is 29
Display entries in LinkedHashMap
{Smith=30, Anderson=31, Cook=29, Lewis=29}
```

Advantages of Collection:

- If you want to store the group of individual objects into a single entity then we should go for collections.
- Collections accept both Homogeneous and Heterogeneous objects.
- We do not need to provide fixed size to store data into the collections means collections are resizable in nature.
- Every collection class having particular underlying data structure which is useful to delete, insert, and retrieve elements present in collections without providing any external code.
- By using the concept of **generics (generic classes with generic type)** we can avoid typesafe problem.

Collection Interface VS Collections Class

Collection	Collections
It is an interface.	It is a utility class.
It is used to represent a group of individual objects as a single unit.	It defines several utility methods that are used to operate on collection.
The Collection is an interface that contains a static method since java8. The Interface can also contain abstract and default methods.	It contains only static methods.

Java List vs. Set

- Both the List interface and the Set interface inherits the Collection interface. However, there exists some difference between them.
- Lists can include duplicate elements. However, sets cannot have duplicate elements.
- Elements in lists are stored in some order. However, elements in sets are stored in groups like sets in mathematics.

Java Iterator Interface

- **Iterators** in Java are used in the Collection framework to **retrieve elements one by one**.
- It is a universal iterator as we can apply it to any Collection object.
- It has a subinterface *ListIterator*.
- By using Iterator, **we can perform both read and remove operations**.

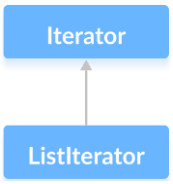
- Iterator must be used whenever we want to enumerate elements in all Collection framework implemented interfaces like Set, List, Queue, Deque and also in all implemented classes of Map interface

- The iterator for most simple java collections just **keeps a pointer of where in the collection the iterator is currently at**.
- Calling ***.next()*** will advance the iterator.
- It doesn't copy the elements, and just returns the next element from the collection.
- All the Java collections include an ***iterator()*** method. This method returns an instance of iterator used to iterate over elements of collections.

Methods of Iterator

The `Iterator` interface provides 4 methods that can be used to perform various operations on elements of collections.

- ***hasNext()*** - returns `true` if there exists an element in the collection
- ***next()*** - returns the next element of the collection
- ***remove()*** - removes the last element returned by the `next()`
- ***forEachRemaining()*** - performs the specified action for each remaining element of the collection




```
import java.util.ArrayList;
import java.util.Iterator;

class Main {
  public static void main(String[] args) {
    // Creating an ArrayList
    ArrayList<Integer> numbers = new ArrayList<>();
    numbers.add(10);
    numbers.add(30);
    numbers.add(20);
    System.out.println("ArrayList: " + numbers);

    // Creating an instance of ListIterator
    Iterator<Integer> iterate = numbers.iterator();
    System.out.println("Iterating over ArrayList using Iterator:");

      while(iterate.hasNext()) {
       System.out.print(iterate.next() + ", ");
    }
  }
}
```

Output

```
ArrayList: [10, 30, 20]
Iterating over ArrayList:
10, 30, 20,
```

Enumeration Interface In Java

- **java.util.Enumeration** interface is one of the predefined interfaces, whose object is used for retrieving the data from collections framework variable(like Stack, Vector, HashTable etc.) in a forward direction only and not in the backward direction.
- This interface has been superceded by an iterator.

The Enumeration Interface defines the functions by which we can enumerate the elements in a collection of elements.

- For new code, Enumeration is considered obsolete.
- However, several methods of the legacy classes such as vectors and properties, several API classes, application codes use this Enumeration interface.

Important Features

- Enumeration is Synchronized.
- It does not support adding, removing, or replacing elements.
- Elements of legacy Collections can be accessed in a forward direction using Enumeration.
- Legacy classes have methods to work with enumeration and returns Enumeration objects.

Declaration

```
public interface Enumeration<E>
```

Creating Enumeration Object

```
Vector v = new Vector();
Enumeration e = v.elements();
```

```
// Java program to test Enumeration
import java.util.Vector;
import java.util.Enumeration;

public class EnumerationClass {

    public static void main(String args[])
    {
        Enumeration months;
        Vector<String> monthNames = new Vector<>();

        monthNames.add("January");
        monthNames.add("February");
        monthNames.add("March");
        monthNames.add("April");
        months = monthNames.elements();

        while (months.hasMoreElements()) {
            System.out.println(months.nextElement());
        }
    }
}
```

Output

```
January
February
March
April
```

Enumeration Vs Iterator In Java : Interfaces

Enumeration	**Iterator**
Using *Enumeration*, you can only traverse the collection. You can't do any modifications to collection while traversing it.	Using *Iterator*, you can remove an element of the collection while traversing it.
Enumeration is introduced in JDK 1.0	*Iterator* is introduced from JDK 1.2
Enumeration is used to traverse the legacy classes like *Vector*, *Stack* and *HashTable*.	*Iterator* is used to iterate most of the classes in the collection framework like *ArrayList*, *HashSet*, *HashMap*, *LinkedList* etc.
Methods : *hasMoreElements()* and *nextElement()*	Methods : *hasNext()*, *next()* and *remove()*
Enumeration is fail-safe in nature. It doesn't throw any exceptions if a collection is modified while iterating	*Iterator* is fail-fast in nature.
Enumeration is not safe and secured due to it's fail-safe nature.	*Iterator* is safer and secured than *Enumeration*.

According to Java API Docs, *Iterator* is always preferred over the *Enumeration*. Here is the note from the Enumeration Docs.
NOTE: The functionality of this interface is duplicated by the Iterator interface. In addition, Iterator adds an optional remove operation, and has shorter method names. New implementations should consider using Iterator in preference to Enumeration.

The Comparator interface

Java provides two interfaces to sort objects using data members of the class:

1. Comparable
2. Comparator

Using Comparable Interface [Covered in UNIT-6]
A comparable object is capable of comparing itself with another object.

The class itself must implements the **java.lang.Comparable** interface to compare its instances.

Consider a Movie class that has members like, rating, name, year. Suppose we wish to sort a list of Movies based on year of release. We can implement the Comparable interface with the Movie class, and we override the method compareTo() of Comparable interface.

Using Comparator

- A comparator interface is used to order the objects of user-defined classes.
- We create multiple separate classes (that implement Comparator) to compare by different members.
- **Collections** class has a second sort() method and it takes Comparator.
- The sort() method invokes the **compare**() to sort objects.
- A comparator object is capable of comparing two objects of the same class**.**
- Following function compare obj1 with obj2.

Syntax:
public int compare(Object obj1, Object obj2):

To compare, we need to do 3 things :

1. Create a class that implements Comparator (and thus the compare() method that does the work previously done by compareTo()).
2. Make an instance of the Comparator class.
3. Call the overloaded sort() method, giving it both the list and the instance of the class that implements Comparator.

public void sort(List list, ComparatorClass c)

How do the sort() method of Collections class work?
Internally the ***sort()*** method does call ***compare()*** method of the classes it is sorting. To compare two elements, it asks "Which is greater?" Compare method returns -1, 0, or 1 to say if it is less than, equal, or greater to the other. It uses this result to then determine if they should be swapped for their sort.

```
import java.util.*;

class Student {
      int rollno;
      String name;
      // Constructor

      public Student(int rollno, String name)
      {
            // This keyword refers to current instance itself
            this.rollno = rollno;
            this.name = name;
      }
```

```
	// To print student details in main()
	  @Override
	public String toString()
	{
		return this.rollno + " " + this.name + " ";
	}
}


class Sortbyroll implements Comparator<Student> {
	  @Override
	public int compare(Student a, Student b)
	{

		return a.rollno - b.rollno;
	}
}

class Sortbyname implements Comparator<Student> {
	  @Override
	public int compare(Student a, Student b)
	{

		return a.name.compareTo(b.name);
	}
}

class Main {

	// Main driver method
	public static void main(String[] args)
	{
		ArrayList<Student> ar = new ArrayList<Student>();

		ar.add(new Student(111, "Mahi"));
		ar.add(new Student(131, "Astha"));
		ar.add(new Student(121, "Sakshi"));
		ar.add(new Student(101, "Nitya"));

		System.out.println("Unsorted");
		for(Student e: ar)
			System.out.println(e.name + " " + e.rollno);

		// Sorting student entries by roll number
		Collections.sort(ar, new Sortbyroll());
		System.out.println("\nSorted by rollno");
		for(Student e: ar)
			System.out.println(e.name + " " + e.rollno);

		// Sorting student entries by name
		Collections.sort(ar, new Sortbyname());
		System.out.println("\nSorted by name");
		for(Student e: ar)
			System.out.println(e.name + " " + e.rollno);
	}
}
```

Output

```
Unsorted
Mahi 111
Astha 131
Sakshi 121
Nitya 101

Sorted by rollno
Nitya 101
Mahi 111
Sakshi 121
Astha 131

Sorted by name
Astha 131
Mahi 111
Nitya 101
Sakshi 121
```

Comparing the Performance of Sets and Lists

Sets are more efficient than **Lists** for storing non-duplicate elements. Lists are useful for accessing elements through the index.

Following program creates an ArrayList and HastSet to check if which takes less time and performs better when searched for particular element in the list.

```java
import java.util.*;

public class Main {
  static final int N = 5000;

  public static void main(String[] args) {
    // Add numbers 0, 1, 2, ..., N - 1 to the array list
    List<Integer> list = new ArrayList<>();
    for (int i = 0; i < N; i++)
      list.add(i);
    Collections.shuffle(list); // Shuffle the array list

    // Create a hash set, and test its performance
    Collection<Integer> set1 = new HashSet<>(list);
    System.out.println("Member test time for hash set is " +
      getTestTime(set1) + " milliseconds");

    // Create an array list, and test its performance
    Collection<Integer> list1 = new ArrayList<>(list);
    System.out.println("Member test time for array list is " +
      getTestTime(list1) + " milliseconds");
  }
```

```
23
24    public static long getTestTime(Collection<Integer> c) {
25      long startTime = System.currentTimeMillis();
26
27      // Test if a number is in the collection
28      for (int i = 0; i < N; i++)
29        c.contains((int)(Math.random() * 2 * N));
30
31      return System.currentTimeMillis() - startTime;
32    }
33  }
```

```
Member test time for hash set is 4 milliseconds
Member test time for array list is 91 milliseconds
```

Singleton and Unmodifiable Collections and Maps

You can create singleton sets, lists, and maps and unmodifiable sets, lists, and maps using the **static** methods in the **Collections** class.

The **Collections** class contains the static methods for lists and collections. It also contains the methods for creating immutable singleton sets, lists, and maps, and for creating read-only sets, lists, and maps, as shown in Figure.

java.util.Collections	
+singleton(o: Object): Set	Returns an immutable set containing the specified object.
+singletonList(o: Object): List	Returns an immutable list containing the specified object.
+singletonMap(key: Object, value: Object): Map	Returns an immutable map with the key and value pair.
+unmodifiableCollection(c: Collection): Collection	Returns a read-only view of the collection.
+unmodifiableList(list: List): List	Returns a read-only view of the list.
+unmodifiableMap(m: Map): Map	Returns a read-only view of the map.
+unmodifiableSet(s: Set): Set	Returns a read-only view of the set.
+unmodifiableSortedMap(s: SortedMap): SortedMap	Returns a read-only view of the sorted map.
+unmodifiableSortedSet(s: SortedSet): SortedSet	Returns a read-only view of the sorted set.

This type of view is like a reference to the actual collection. But you cannot modify the collection through a read-only view. Attempting to modify a collection through a readonly view will cause an **UnsupportedOperationException**.