# Java Collection Framework

## Collection Framework

By the end of this session, you must be able to

- ❑ Understand the purpose and benefits of collection framework
- ❑ Understand Collection, List, Set and Queue, Deque interfaces
- ❑ Use ArrayList, LinkedList, Vector and Stack
- ❑ Use PriorityQueue, ArrayQueue, HashSet, LinkedHashSet, TreeSet
- ❑ Understand Map interface and sub interfaces
- ❑ Use HashMap, TreeMap, Legacy classes, etc
- ❑ Understand utility classes: Arrays, Collections
- ❑ Use Cursors – Iterator, ListIterator, Enumeration
- ❑ Understand and use Comparable and Comparator interface
- ❑ Understand utility classes such as Date, Calender, Scanner, etc

## The Java Collection Framework

The Java Collections Framework is a library of classes and interfaces for working with collections of objects.

A collection is an object which can store other objects, called elements. Collections provide methods for adding and removing elements, and for searching for a particular element within the collection.

## Why Collection Framework?

An array is an indexed collection of fixed number of homogeneous data elements

**Limitations of Array Objects :**

1.Arrays are fixed in size

2.Arrays can hold only homogeneous data elements

**Example:**

```
 Student[] s=new Student(1000);
 s[0]=new Student();
 s[1]=new Student();
 s[2]=new Customer();  // CE – Incompatible types
```

But this problem can be resolved by using Object type arrays.

**Example:**

```
  Object[] a= new Object[1000];
          a[0]=new Student();
          a[1]=new Customer();
```

3. Arrays concept not built based on some underlying data structures

# Collection Framework

## Advantages of Collections over Arrays:

1.Collections are grow able in nature – may increase or decrease – as per requirement

2.Collections can hold both homogeneous & heterogeneous objects

3.Every collection class is implemented based on some data structures. Readymade method support is available for every requirement

**Note:**

•Arrays can be used to hold both primitives & objects

•Collections can be used to hold only objects but not for primitives

## Collection:

A group of individual objects as a single entity is called “Collection”

# Collection Framework

## Collection framework:

A collections framework is a unified architecture for representing and manipulating collections

**The collection  framework contain the following:**

**Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently

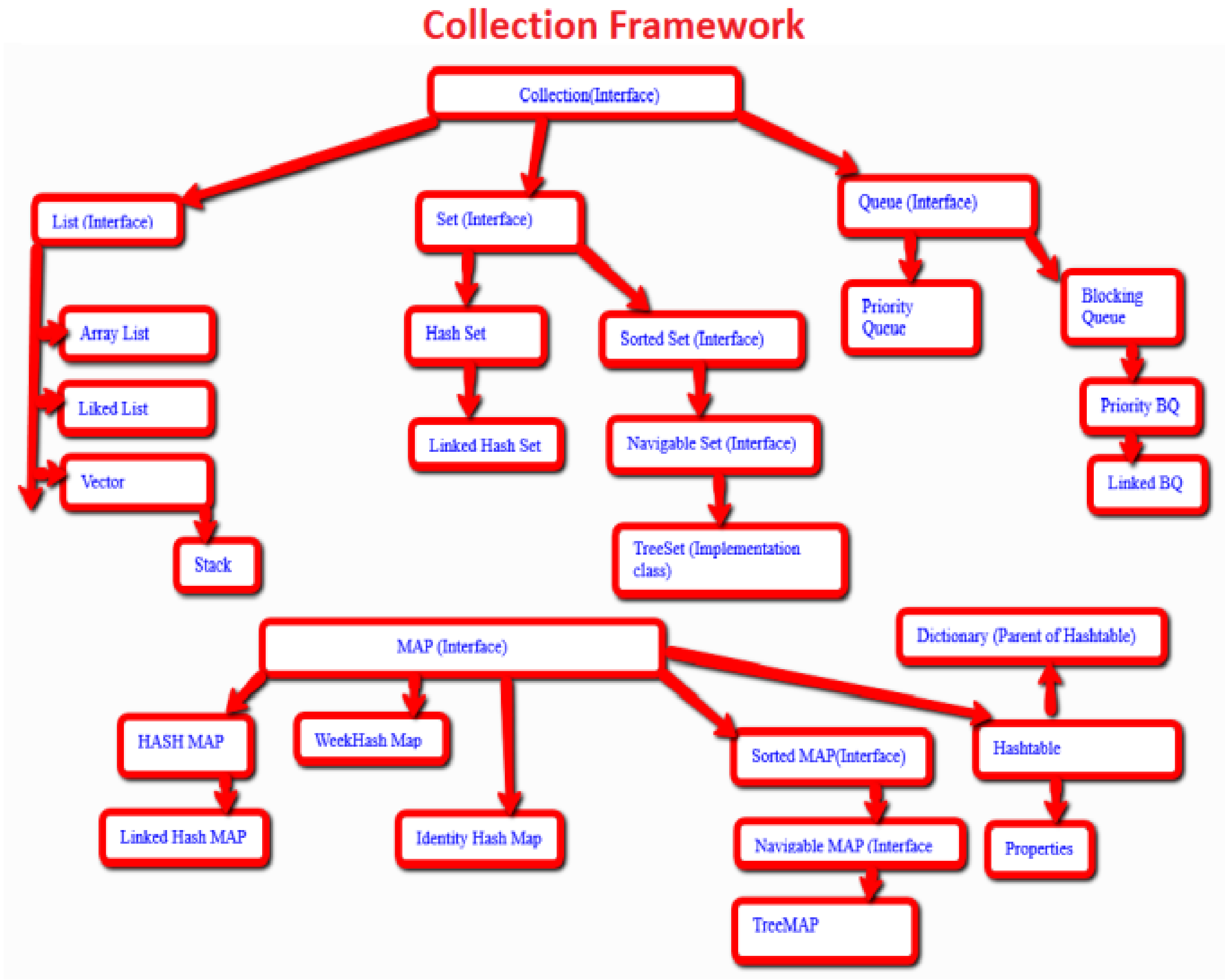
**Implementations (Classes):** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.

**Algorithms (methods):** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic

**The Java Collections Framework provides the following benefits**:

- ❖ Reduces programming effort
- ❖ Increases program speed and quality
- ❖ Allows interoperability among unrelated APIs
- ❖ Fosters software reuse, etc.,

# Collection Framework - Summary



## Key Interfaces of Collection Framework

### 1. Collection (Interface):

In general, **Collection** interface is considered as root interface of Collection Framework
**Collection** interface defines the most common methods which can be applicable for any collection object.

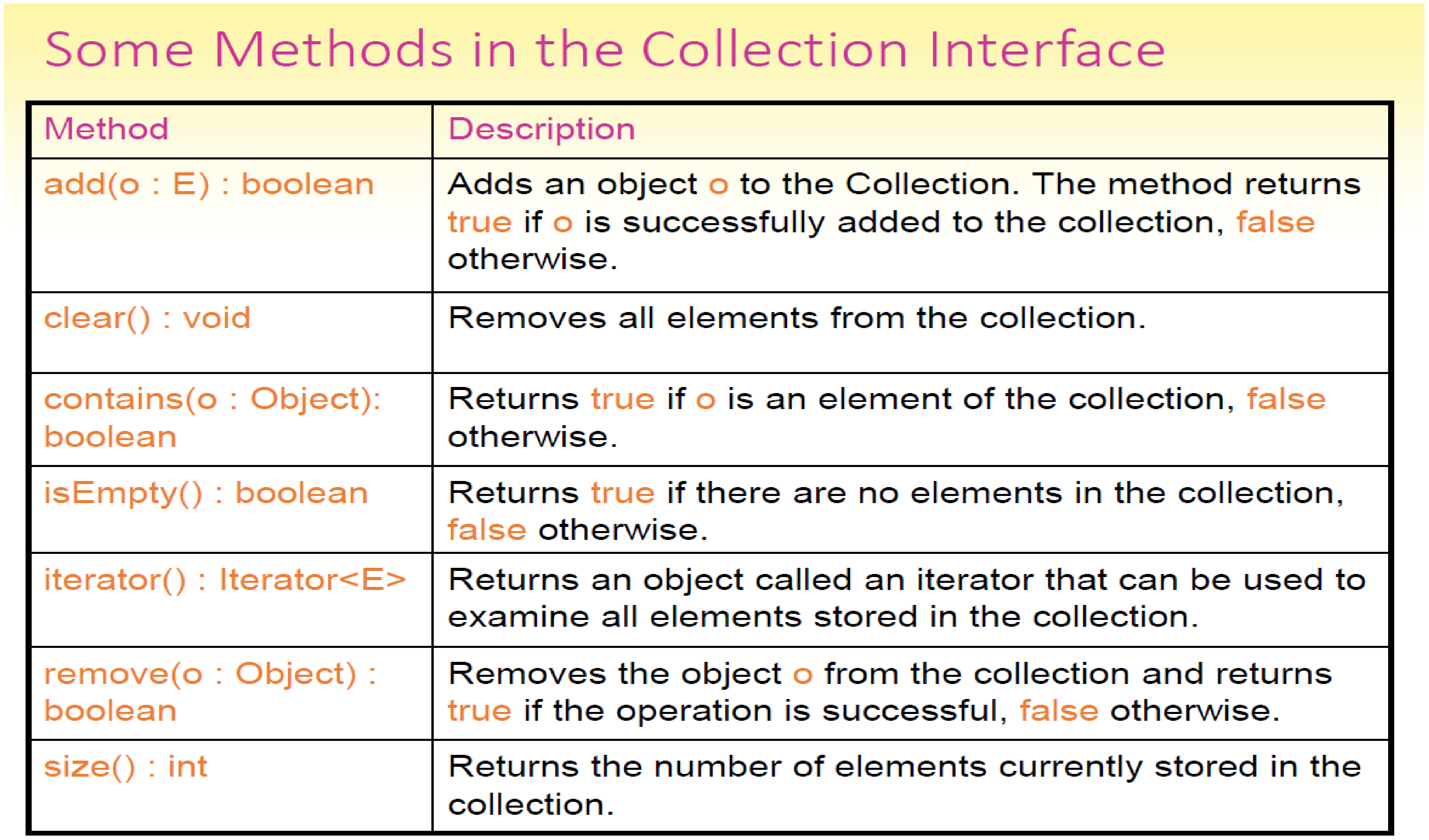
**The Collection interface contains methods that perform basic operations, such as:**
int **size()**
boolean **isEmpty()**
boolean **contains(Object element)**
boolean **add(E element)**
boolean **remove(Object element)**
iterator<E> **iterator()**

### Collection Interface Bulk Operations

Bulk operations perform an operation on an entire Collection. The following are the bulk operations:

**containsAll( )** : returns true if the target Collection contains all of the elements in the specified Collection.
**addAll( )** : adds all of the elements in the specified Collection to the target Collection.
**removeAll( )** : removes from the target Collection all of its elements that are also contained in the specified Collection.
**retainAll( ) :** it retains only those elements in the target Collection that are also contained in the specified Collection.
**clear( )** : removes all elements from the Collection.

## Commonly used methods of Collection interface



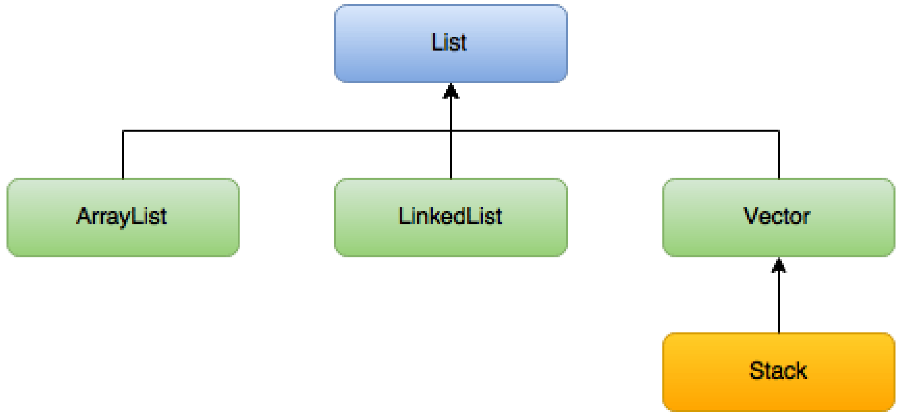
### Some Methods in the Collection Interface

| Method | Description |
|---|---|
| add(o : E) : boolean | Adds an object o to the Collection. The method returns true if o is successfully added to the collection, false otherwise. |
| clear() : void | Removes all elements from the collection. |
| contains(o : Object): boolean | Returns true if o is an element of the collection, false otherwise. |
| isEmpty() : boolean | Returns true if there are no elements in the collection, false otherwise. |
| iterator() : Iterator<E> | Returns an object called an iterator that can be used to examine all elements stored in the collection. |
| remove(o : Object) : boolean | Removes the object o from the collection and returns true if the operation is successful, false otherwise. |
| size() : int | Returns the number of elements currently stored in the collection. |

# Key Interfaces of Collection Framework

## 2. List (Interface):

- ✓ It is a child interface of Collection
- ✓ **A List is an ordered Collection (**sometimes called a sequence**).** Lists may contain duplicate elements.
- ✓ Used to represent a group of individual objects where **insertion order is preserved & duplicates** are allowed



*Vector & Stack are legacy classes

**In addition to the operations inherited from Collection, the List interface includes operations for the following:**

- ✓ **Positional access :** manipulates elements based on their numerical position in the list.
  This includes methods such as **get(), set(), add(), addAll(), and remove**()

- ✓ **Search** : searches for a specified object in the list and returns its numerical position.
  Search methods include **indexOf()** and **lastIndexOf()**

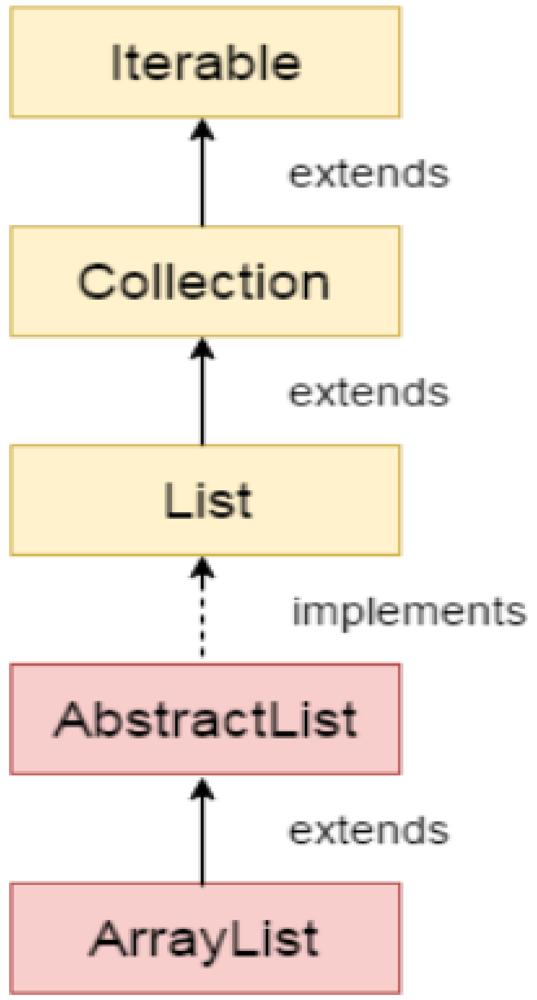
- ✓ **Iteration** : extends Iterator semantics to take advantage of the list's sequential nature. The **listIterator** methods provide this behavior- hasPrevious(), next() and previous(), hasNext(), etc

- ✓ **Range-view** : The **subList()** method performs arbitrary range operations on the list.
  Ex: list.subList(fromIndex, toIndex).clear(); // removes those ranged values

## The List Interface Methods

| | |
|---|---|
| add(index:int, el:E) : void | Adds the element el to the collection at the given index. Throws IndexOutOfBoundsException if index is negative, or greater than the size of the list. |
| get(index:int):E | Returns the element at the given index, or throws IndexOutBoundsException if index is negative or greater than or equal to the size of the list. |
| indexOf(o:Object):int | Returns the least (first) index at which the object o is found; returns -1 if o is not in the list. |
| lastIndexOf(o:Object):int | Returns the greatest (last) index at which the object o is found; returns -1 if o is not in the list. |
| listIterator():ListIterator<E> | Returns an iterator specialized to work with List collections. |
| remove(index:int):E | Removes and returns the element at the given index; throws IndexOutOfBoundsException if index is negative, or greater than or equal to the size of the list. |
| set(index:int, el:E):E | Replaces the element at index with the new element el. |

## Array List Demo



```
import java.util.*;

public class ALDemo1 {
        public static void main(String[]
args) {

        ArrayList al=new ArrayList();
        al.add("Sadhu");
        al.add("Sreenivas");
        al.add("35");
        al.add("8.5");
        al.add("Hyderabad");
        al.add("500089");
        al.add("true");
        System.out.println(al);

      // Collections.sort(al);
        System.out.println(al);

        Iterator itr=al.iterator();
        while(itr.hasNext())
            System.out.println(itr.next());

           }
}
```

## The Iterator Interface

Iterators implement the Iterator interface. This interface specifies the following methods:

- hasNext() : boolean
- next() : E
- remove() : void

The remove() method is optional, so not all iterators have it.

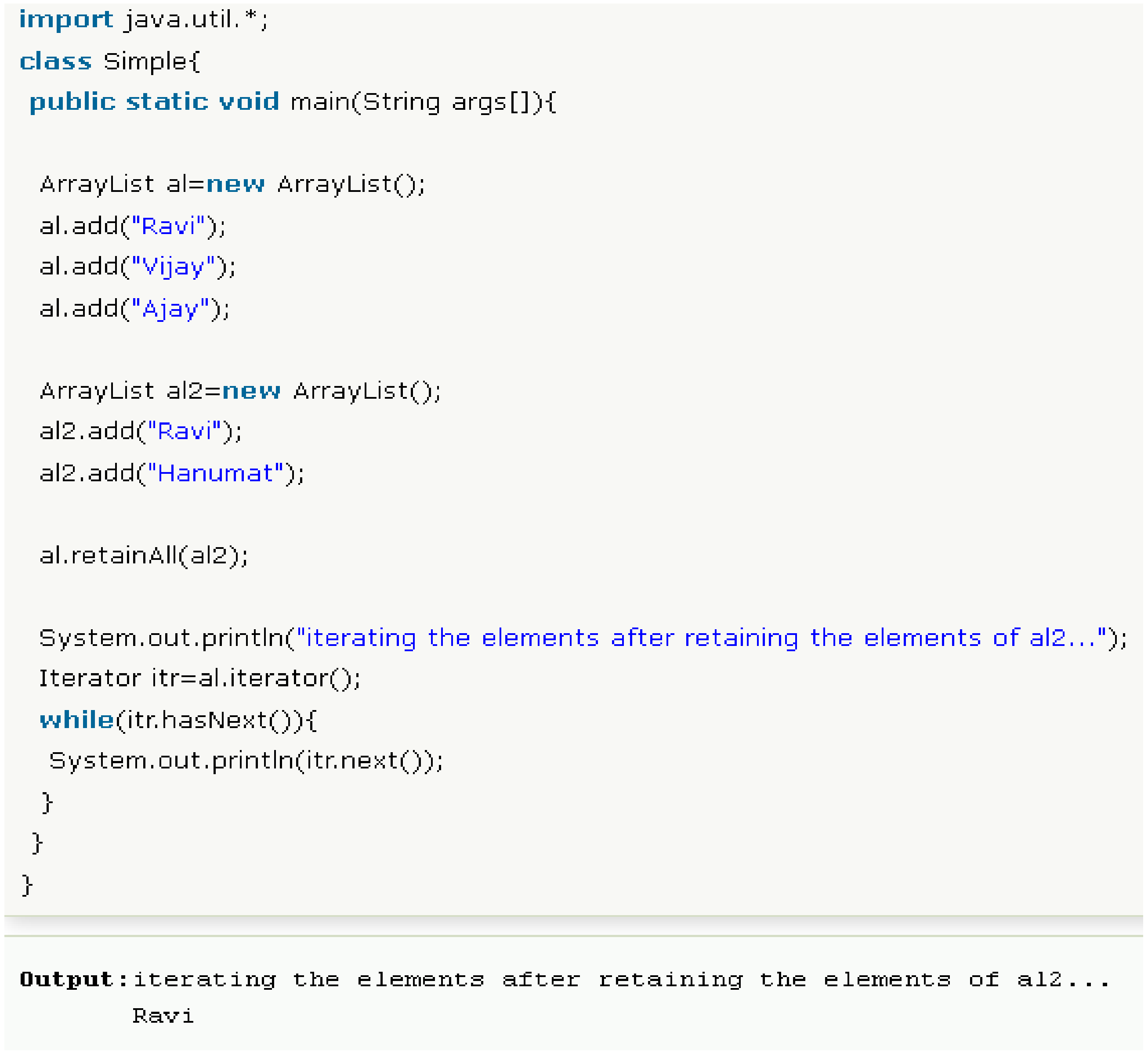
```
List<String> nameList = new ArrayList<String>();
  String [ ] names = {"Ann", "Bob", "Carol"};
  // Add to arrayList
  for (int k = 0; k < names.length; k++)
    nameList.add(names[k]);

  // Display name list using an iterator
  Iterator<String> it = nameList.iterator();    // Get the iterator
  while (it.hasNext())                         // Use the iterator
    System.out.println(it.next());
```

## Array List Demo

```
import java.util.ArrayList;

import java.util.*;

public class ALDemo {
    public static void main(String[] args) {
        ArrayList al=new ArrayList();
        int[] a={5,7,9,2,3,1};
        Arrays.sort(a);
        for(int i:a)
        System.out.println(i);

        al.add("sreenivas");
        al.add(10);
        al.add(101);
        al.add(3.14);
        al.add('s');
        al.add(true);
```

```
ArrayList al1=new ArrayList();
        al1.add("vasu");
       al1.add("sreenivas");
        al.addAll(al);
        al.add(1,"Sadhu");
        al.add(0,“cdac");
        Collections.sort(al1);
        System.out.println(al);

        Iterator itr=al.iterator();
        while(itr.hasNext()){
        System.out.println(itr.next());
        }

       }
}
```

## Example of retainAll() method:



```
import java.util.*;
class Simple{
 public static void main(String args[]){

  ArrayList al=new ArrayList();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ajay");

  ArrayList al2=new ArrayList();
  al2.add("Ravi");
  al2.add("Hanumat");

  al.retainAll(al2);

  System.out.println("iterating the elements after retaining the elements of al2...");
  Iterator itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

```
Output:iterating the elements after retaining the elements of al2...
       Ravi
```

## Linked List Demo

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

public class LLDemo1 {

    public static void main(String[] args) {

        LinkedList al=new LinkedList();
        al.add("Sadhu");
        al.add("Sreenivas");
        al.add("35");
        al.add("8.5");
        al.add("Hyderabad");
        al.add("500089");
        al.add("true");
        al.addFirst("Mr");
        al.addLast("false");
        System.out.println(al);
 // Collections.sort(al);
        System.out.println(al);
        ListIterator itr=al.listIterator();
        while(itr.hasNext())
        {
            System.out.println(itr.next());

        }
         System.out.println(itr.previous());
    }
}
```

## Key Interfaces of Collection Framework

**Most polymorphic algorithms in the Collections class apply specifically to List.**

**sort(list l)** :sorts a List using a merge sort algorithm, which provides a fast, stable sort.

**shuffle ( )** : randomly permutes the elements in a List.

**reverse ( ):** reverses the order of the elements in a List.

**rotate ( ):** rotates all the elements in a List by a specified distance.

**swap ( ):** swaps the elements at specified positions in a List.

**replaceAll ( ):** replaces all occurrences of one specified value with another.

**fill( ):** overwrites every element in a List with the specified value.

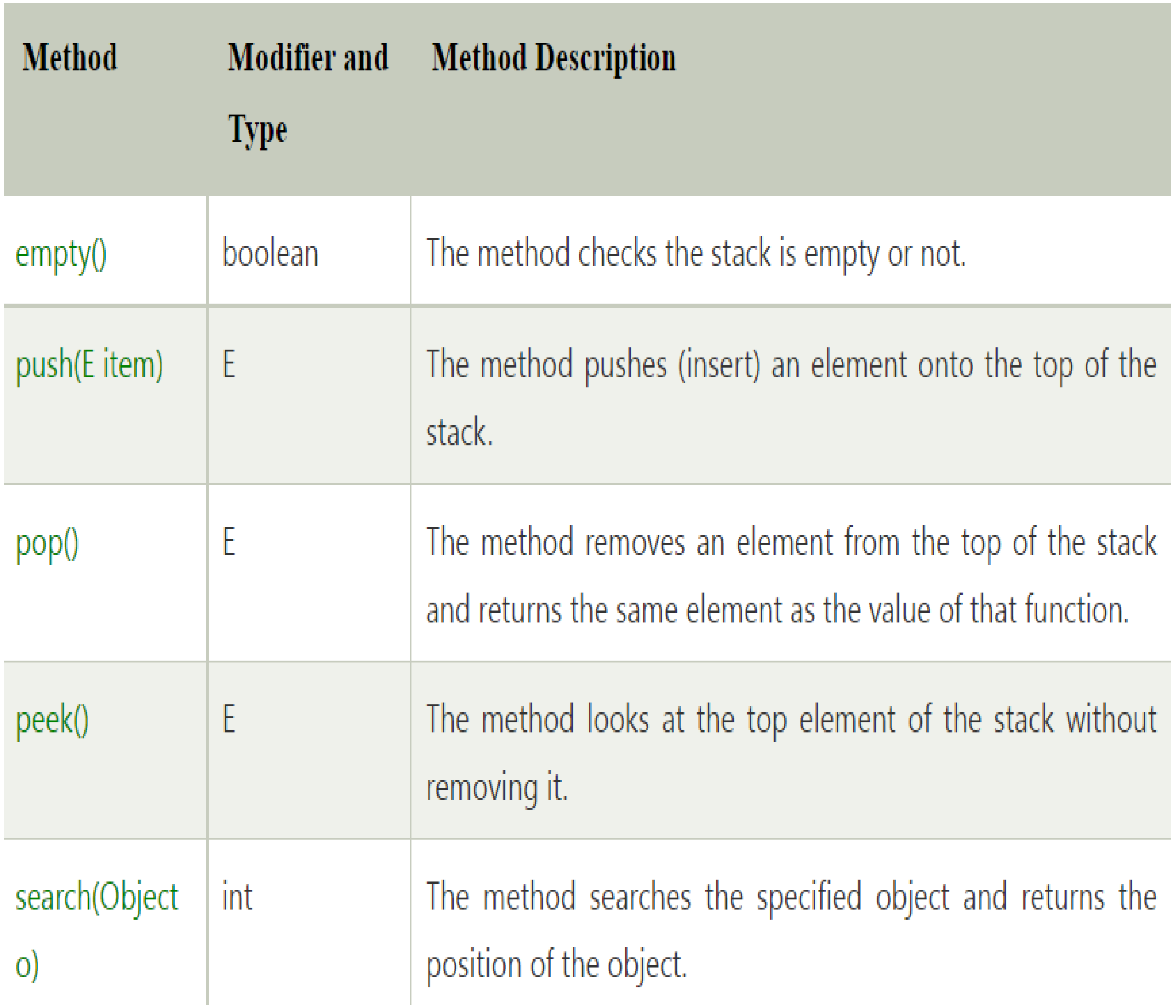
**copy ( ):** copies the source List into the destination List.

**binarySearch( ):** searches for an element in an ordered List using the binary search algorithm.

**indexOfSubList( ):** returns the index of the first sublist of one List that is equal to another.

**lastIndexOfSubList( ):** returns the index of the last sublist of one List that is equal to another.

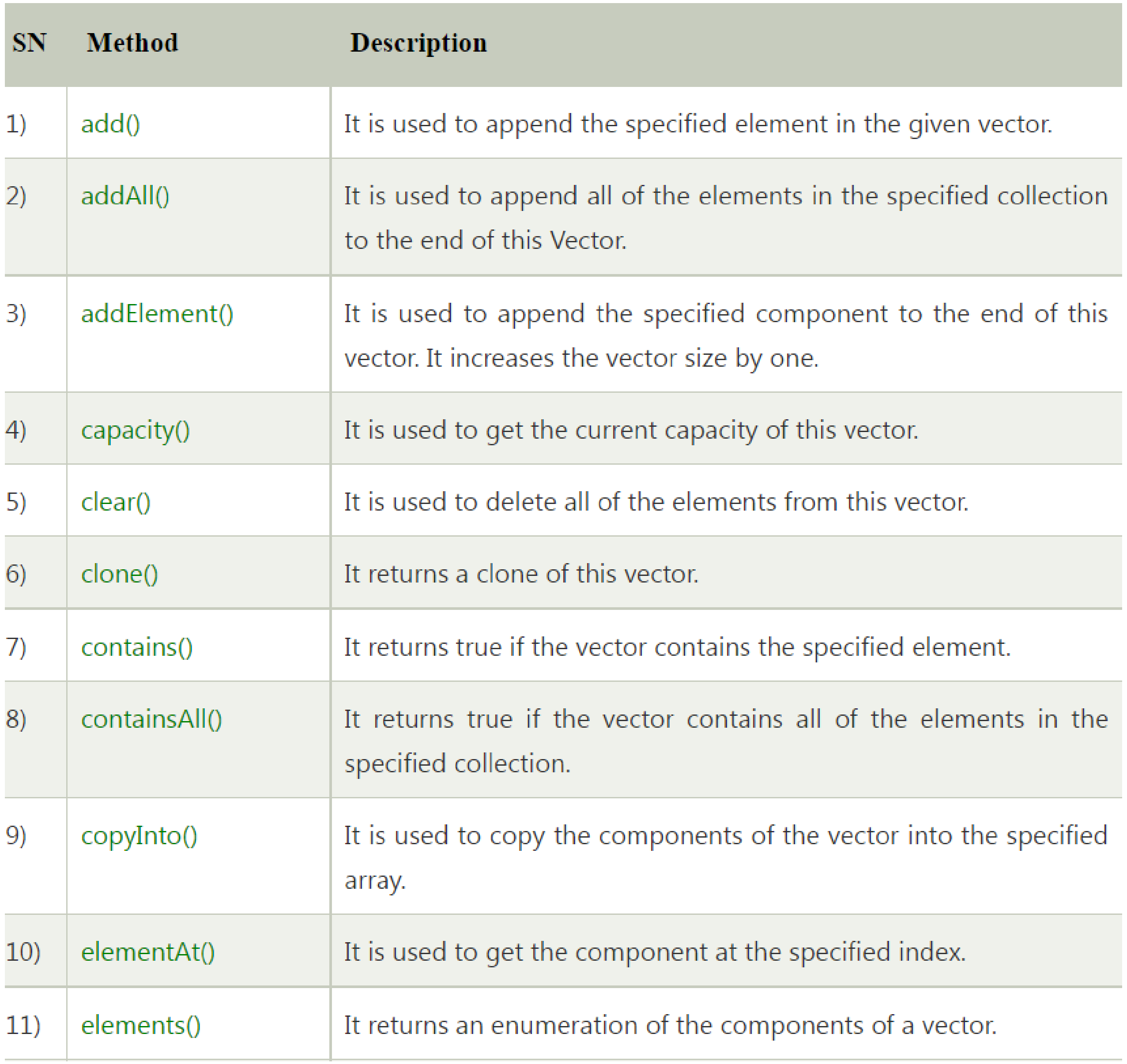
## Stack Demo



| Method | Modifier and Type | Method Description |
|---|---|---|
| empty() | boolean | The method checks the stack is empty or not. |
| push(E item) | E | The method pushes (insert) an element onto the top of the stack. |
| pop() | E | The method removes an element from the top of the stack and returns the same element as the value of that function. |
| peek() | E | The method looks at the top element of the stack without removing it. |
| search(Object o) | int | The method searches the specified object and returns the position of the object. |

```
import java.util.*;
public class StackDemo {
    public static void main(String[] args) {

        Stack s=new Stack();
        s.push(10);
        s.push(20);
        s.push(30);
        s.push(40);
        s.push(50);
        System.out.println(s);
        System.out.println(s.peek());
        System.out.println(s.pop());
        System.out.println(s.pop());
        System.out.println(s.peek());
    }

}
```

## Vector Demo



| SN | Method | Description |
|---|---|---|
| 1) | add() | It is used to append the specified element in the given vector. |
| 2) | addAll() | It is used to append all of the elements in the specified collection to the end of this Vector. |
| 3) | addElement() | It is used to append the specified component to the end of this vector. It increases the vector size by one. |
| 4) | capacity() | It is used to get the current capacity of this vector. |
| 5) | clear() | It is used to delete all of the elements from this vector. |
| 6) | clone() | It returns a clone of this vector. |
| 7) | contains() | It returns true if the vector contains the specified element. |
| 8) | containsAll() | It returns true if the vector contains all of the elements in the specified collection. |
| 9) | copyInto() | It is used to copy the components of the vector into the specified array. |
| 10) | elementAt() | It is used to get the component at the specified index. |
| 11) | elements() | It returns an enumeration of the components of a vector. |

```
import java.util.*;

public class VectorDemo {

    public static void main(String[] args) {

        Vector v=new Vector();
        System.out.println(v.capacity());
        v.add("Sadhu");
        v.addElement("Sreeni");
        v.add("Hyderabad");
        System.out.println(v);

        v.remove(0);
        Collections.sort(v);
        Enumeration e=v.elements();
       while(e.hasMoreElements())
            System.out.println(e.nextElement());


        Iterator itr=v.iterator();
        while(itr.hasNext())
            System.out.println(itr.next());
    }
}
```

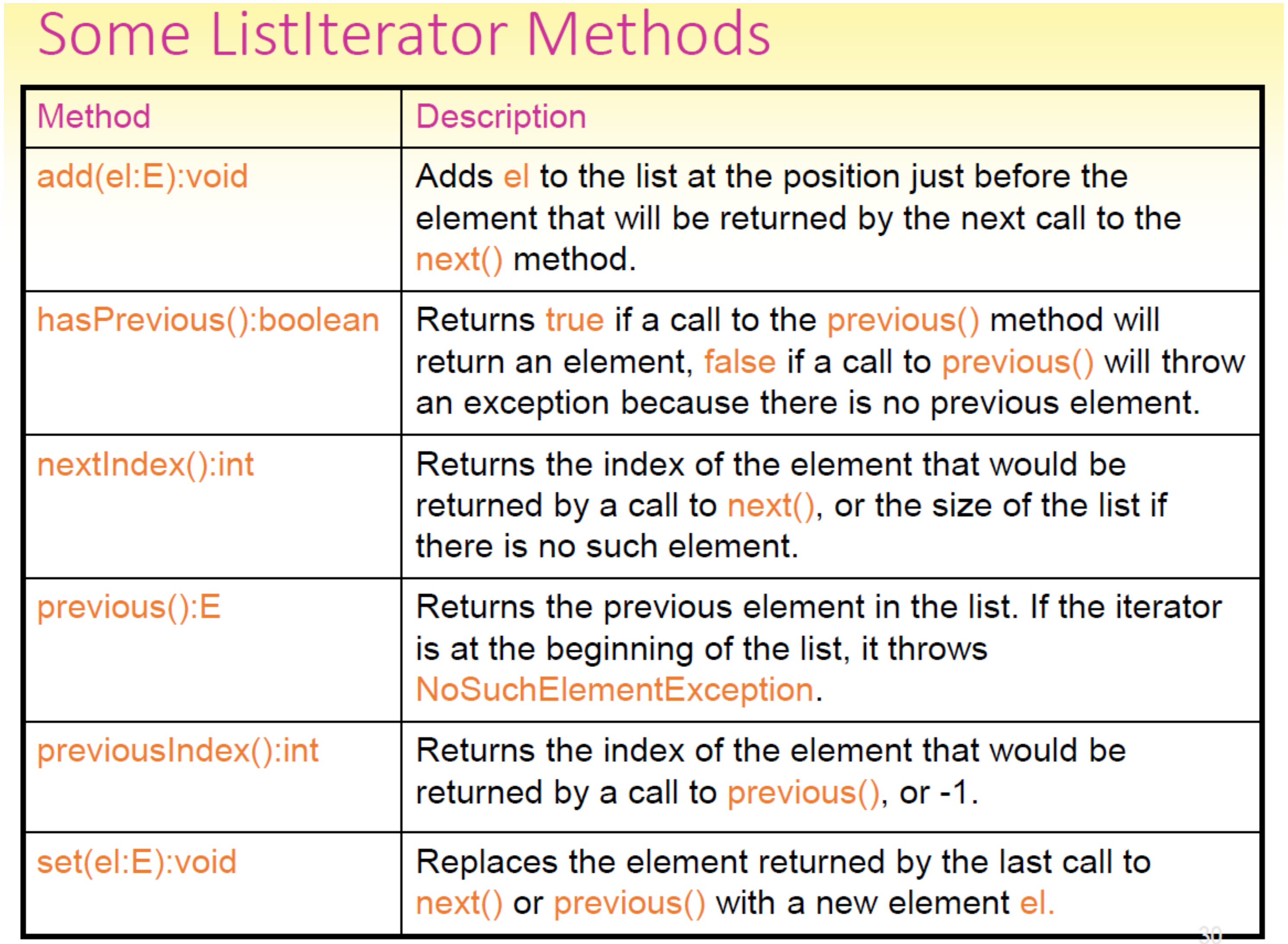
## ListIterator

The ListIterator extends Iterator by adding methods for moving backward through the list (in addition to the methods for moving forward that are provided by Iterator)

hasPrevious() : boolean

previous() : E

## ListIterator Demo



### Some ListIterator Methods

| Method | Description |
|---|---|
| add(el:E):void | Adds el to the list at the position just before the element that will be returned by the next call to the next() method. |
| hasPrevious():boolean | Returns true if a call to the previous() method will return an element, false if a call to previous() will throw an exception because there is no previous element. |
| nextIndex():int | Returns the index of the element that would be returned by a call to next(), or the size of the list if there is no such element. |
| previous():E | Returns the previous element in the list. If the iterator is at the beginning of the list, it throws NoSuchElementException. |
| previousIndex():int | Returns the index of the element that would be returned by a call to previous(), or -1. |
| set(el:E):void | Replaces the element returned by the last call to next() or previous() with a new element el. |

```
import java.io.*;
import java.util.*;

public class ListIteratorDemo {

    public static void main(String[] args) {
        LinkedList l=new LinkedList();
        l.add("sachin");
        l.add("saurabh");
        l.add("yuvi");
       l.add("dhoni");
        l.add("zaheer");
        System.out.println(l);

        ListIterator litr=l.listIterator();
        while(litr.hasNext()){
            String s=(String)litr.next();
            if(s.equals("zaheer"))
                litr.remove();
            if(s.equals("sachin"))
                litr.set("Virat");

             }
        System.out.println(l);
    }
```

## Storing user-defined class objects:

```
class Student{
 int rollno;
 String name;
 int age;
 Student(int rollno,String name,int age){
  this.rollno=rollno;
  this.name=name;
  this.age=age;
 }
}
```

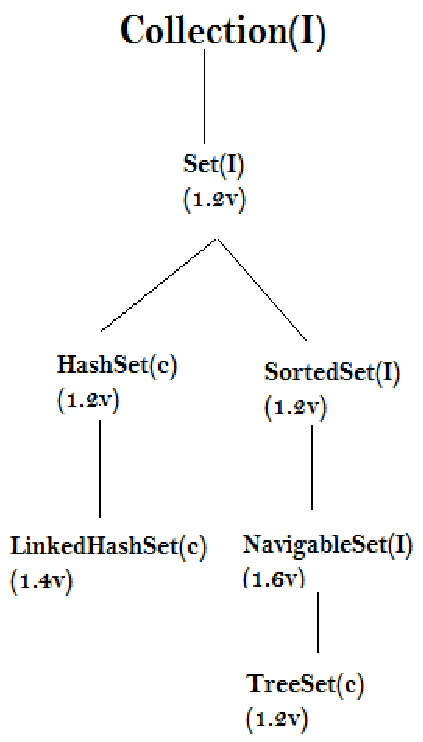
```
import java.util.*;
class Simple{
 public static void main(String args[]){

  Student s1=new Student(101,"Sonoo",23);
  Student s2=new Student(102,"Ravi",21);
  Student s2=new Student(103,"Hanumat",25);

  ArrayList al=new ArrayList();
  al.add(s1);
  al.add(s2);
  al.add(s3);

  Iterator itr=al.iterator();
  while(itr.hasNext()){
    Student st=(Student)itr.next();
    System.out.println(st.rollno+" "+st.name+" "+st.age);
  }
 }
}
```

```
Output:101 Sonoo 23
       102 Ravi 21
       103 Hanumat 25
```

# Key Interfaces of Collection Framework

## 3. Set (Interface):

- ✓ It is a child interface of Collection
- ✓ A Set is a Collection that cannot contain duplicate elements
- ✓ Used to represent a group of individual objects where **insertion order is not preserved & duplicates are not** allowed
- ✓ The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited



**There are three general-purpose Set implementations**:
HashSet, TreeSet, and LinkedHashSet.

**HashSet :**
- ✓ stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration.

**TreeSet :**
- ✓ stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashSet.

**LinkedHashSet:**
- ✓ implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order).
- ✓ LinkedHashSet spares its clients from the unspecified, generally chaotic ordering provided by HashSet at a cost that is only slightly higher.

## Key Interfaces of Collection Framework

### Basic Operations on Set:

**size()** method returns the number of elements in the Set (its cardinality)

**isEmpty()** method returns whether the set is empty or not

**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

**remove()** method removes the specified element from the Set if it is present and returns a boolean indicating whether the element was present

**iterator()** method returns an Iterator over the Set

### 4. SortedSet (Interface):

- ✓ It is a child interface of Set
- ✓ Used to represent a group of individual objects according to **some sorting order**

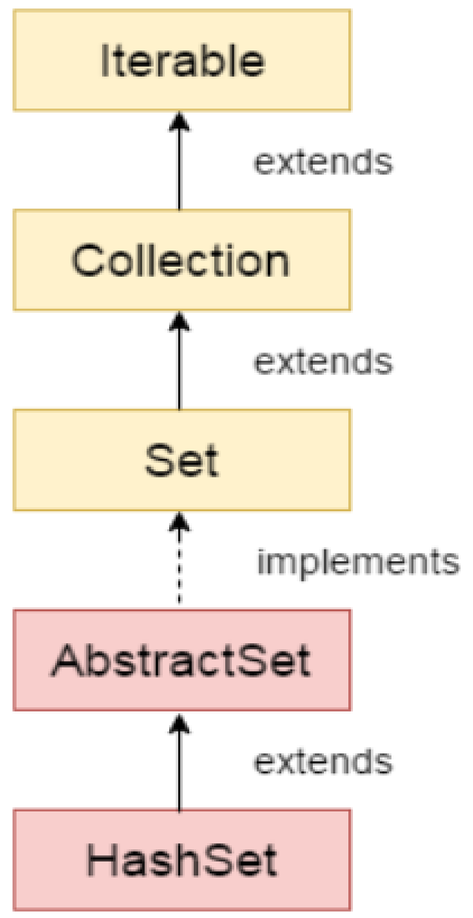
### 5. NavigableSet (Interface):

- ✓ It is a child interface of SortedSet
- ✓ Defines several methods for **navigation** purpose

## Some HashSet Constructors

| | |
|---|---|
| HashSet() | Creates an empty HashSet object with a default initial capacity of 16 and load factor of 0.75. |
| HashSet(int initCapacity, float loadFactor) | Creates an empty HashSet object with the specified initial capacity and load factor. |
| HashSet(int initCapacity) | Creates an empty HashSet object with the specified initial capacity and a load factor of 0.75. |

# HashSet Demo



```
import java.util.HashSet;

import java.util.*;
public class HashSetDemo {

    public static void main(String[] args) {

        LinkedHashSet h=new LinkedHashSet();
        h.add("TS");
        h.add("AP");
        h.add("UP");
        h.add("TS"); // false
        h.add(123);
        h.add(321);
       h.add(null);
        System.out.println(h);

        Iterator i=h.iterator();
        while(i.hasNext()){
            System.out.println(i.next());
        }

    }
}
```

## Use of the Car Class with a HashSet

```
public  static void main(String [ ] args)
 {
   Set<Car> carSet = new HashSet<Car>();
   Car [ ] myRides = {
                       new Car("TJ1", "Toyota"),
                       new Car("GM1", "Corvette"),
                       new Car("TJ1", "Toyota Corolla")
                      };
   // Add the cars to the HashSet
   for (Car c : myRides)
      carSet.add(c);

   // Print the list using an Iterator
   Iterator it = carSet.iterator();
   while (it.hasNext())
      System.out.println(it.next());
 }
```

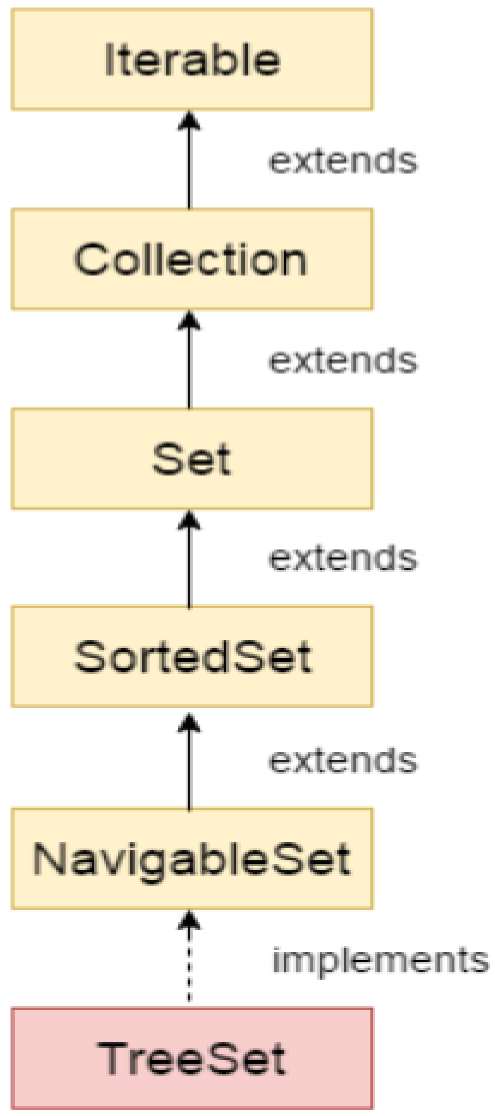
## TreeSet

A TreeSet stores elements based on a natural order defined on those elements.

The natural order is based on the values of the objects being stored .

By internally organizing the storage of its elements according to this order, a TreeSet allows fast search for any element in the collection.

## TreeSet Demo



```
import java.util.*;
public class TreeSetDemo {

    public static void main(String[] args) {

        TreeSet t=new TreeSet();
        t.add(10);
        t.add(9);
        t.add(11);
        t.add(5);
        t.add(15);
        //t.add(null); not possible
       // t.add("ABC");
        System.out.println(t);
        System.out.println(t.first());
        System.out.println(t.last());
        System.out.println(t.headSet(11));
        System.out.println(t.tailSet(10));

Iterator itr=t.iterator();
        while(itr.hasNext())
            System.out.println(itr.next());
    }
}
```

## The Comparable Interface

In Java, a class defines its natural order by implementing the Comparable interface:

```
public interface Comparable<T>
{
    int compareTo(T other);
}
```

The compareTo method returns a negative value, or zero, or a positive value, to indicate that the calling object is less than, equal to, or greater than the other object.

## Sorting Strings Using a TreeSet

```
import java.util.*;
public class Test
{
  public  static void main(String [ ] args)
   {
     // Create TreeSet
     Set<String> mySet = new TreeSet<String>();
     // Add Strings
     mySet.add("Alan");
     mySet.add("Carol");
     mySet.add("Bob");
     // Get Iterator
     Iterator it = mySet.iterator();
     while (it.hasNext())
     {
        System.out.println(it.next());
     }
   }
}
```

## Comparators

A comparator is an object that can impose an order on objects of another class.

This is different from the Comparable interface, which allows a class to impose an order on its own objects.

```
Interface Comparator <T>
{
  int compare(T obj1, T obj2);
  boolean equals(Object o);
}
```

The compare(x, y) method returns a negative value, or zero, or a positive value, according to whether x is less than, equal to, or greater than y.

```java
public class Test
{
  public  static void main(String [ ] args)
   { // Create Comparator
     RevStrComparator comp = new RevStrComparator();
     Set<String> mySet = new TreeSet<String>(comp);
     // Add strings
     mySet.add("Alan");
     mySet.add("Carol");
     mySet.add("Bob");
     // Get Iterator
     Iterator it = mySet.iterator();
     while (it.hasNext())
     {
        System.out.println(it.next());
     }
   }
}
```

```java
import java.util.*;

class RevStrComparator implements Comparator<String>

{

  public int compare(String s1, String s2)

  {

     return  - s1.compareTo(s2);    // Note the negation op

  }

}
```

```
import java.util.Comparator;
import java.util.TreeSet;

public class MySetWithCompr {

    public static void main(String a[]){

        TreeSet<String> ts = new TreeSet<String>(new MyComp());
        ts.add("RED");
        ts.add("ORANGE");
        ts.add("BLUE");
        ts.add("GREEN");
        System.out.println(ts);
    }
}

class MyComp implements Comparator<String>{

    @Override
    public int compare(String str1, String str2) {
        return str1.compareTo(str2);
    }

}
```

Output:

```
[BLUE, GREEN, ORANGE, RED]
```

```
public class MyArrayDuplicates {

        public static void main(String a[]){
                String[] strArr = {"one","two","three","four","four","five"};
                //convert string array to list
                List<String> tmpList = Arrays.asList(strArr);
                //create a treeset with the list, which eliminates duplicates
                TreeSet<String> unique = new TreeSet<String>(tmpList);
                System.out.println(unique);
        }
}
```

Output:

```
[five, four, one, three, two]
```

## Example shows how to sort **TreeSet** using **Comparator** with **user defined objects**

```
import java.util.Comparator;
import java.util.TreeSet;

public class MyCompUserDefine {

  public static void main(String a[]){
    //By using name comparator (String comparison)
    TreeSet<Empl> nameComp = new TreeSet<Empl>(new MyNameComp());
    nameComp.add(new Empl("Ram",3000));
    nameComp.add(new Empl("John",6000));
    nameComp.add(new Empl("Crish",2000));
    nameComp.add(new Empl("Tom",2400));
    for(Empl e:nameComp){
      System.out.println(e);
    }
    System.out.println("===========================");
    //By using salary comparator (int comparison)
    TreeSet<Empl> salComp = new TreeSet<Empl>(new MySalaryComp());
    salComp.add(new Empl("Ram",3000));
    salComp.add(new Empl("John",6000));
    salComp.add(new Empl("Crish",2000));
    salComp.add(new Empl("Tom",2400));
    for(Empl e:salComp){
      System.out.println(e);
    }
  }
}
```

```
class MyNameComp implements Comparator<Empl>{

  @Override
  public int compare(Empl e1, Empl e2) {
    return e1.getName().compareTo(e2.getName());
  }
}
```

```
class Empl{

  private String name;
  private int salary;
```

```
class MySalaryComp implements Comparator<Empl>{

  @Override
  public int compare(Empl e1, Empl e2) {
    if(e1.getSalary() > e2.getSalary()){
      return 1;
    } else {
      return -1;
    }
  }
}
```

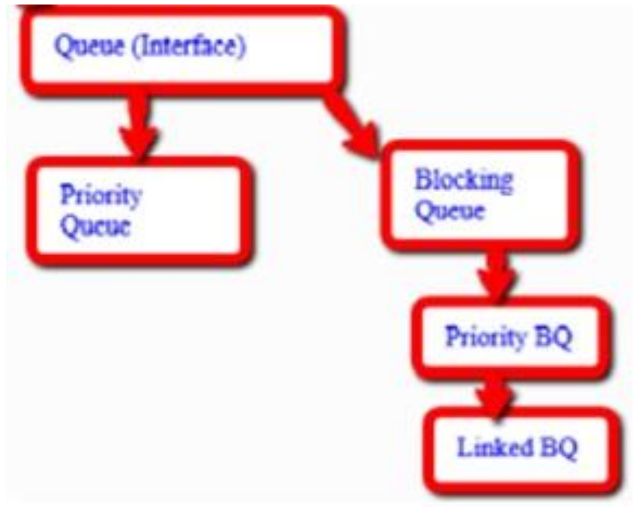
## Key Interfaces of Collection Framework

### 6. Queue (Interface):

- ✓ It is a child interface of Collection
- ✓ Used to represent a group of individual objects **prior to processing**
- ✓ A Queue is a collection for holding elements prior to processing.
- ✓ Besides basic Collection operations, queues provide additional insertion, removal, and inspection operations.

**The Queue interface follows:**

✓
```
public interface Queue<E> extends Collection<E> {
 E element();  // return head
 boolean offer(E e);  // add
 E peek(); // return head
 E poll();  //remove
 E remove();  // remove
  }
```



Queue Interface Structure

| Type of Operation | Throws exception | Returns special value |
|---|---|---|
| **Insert** | add(e) | offer(e) |
| **Remove** | remove() | poll() |
| **Examine** | element() | peek() |

# Key Interfaces of Collection Framework

## 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, defines methods to access the elements at both ends of the Deque instance.
- ✓ Methods are provided to **insert, remove, and examine** the elements**.**

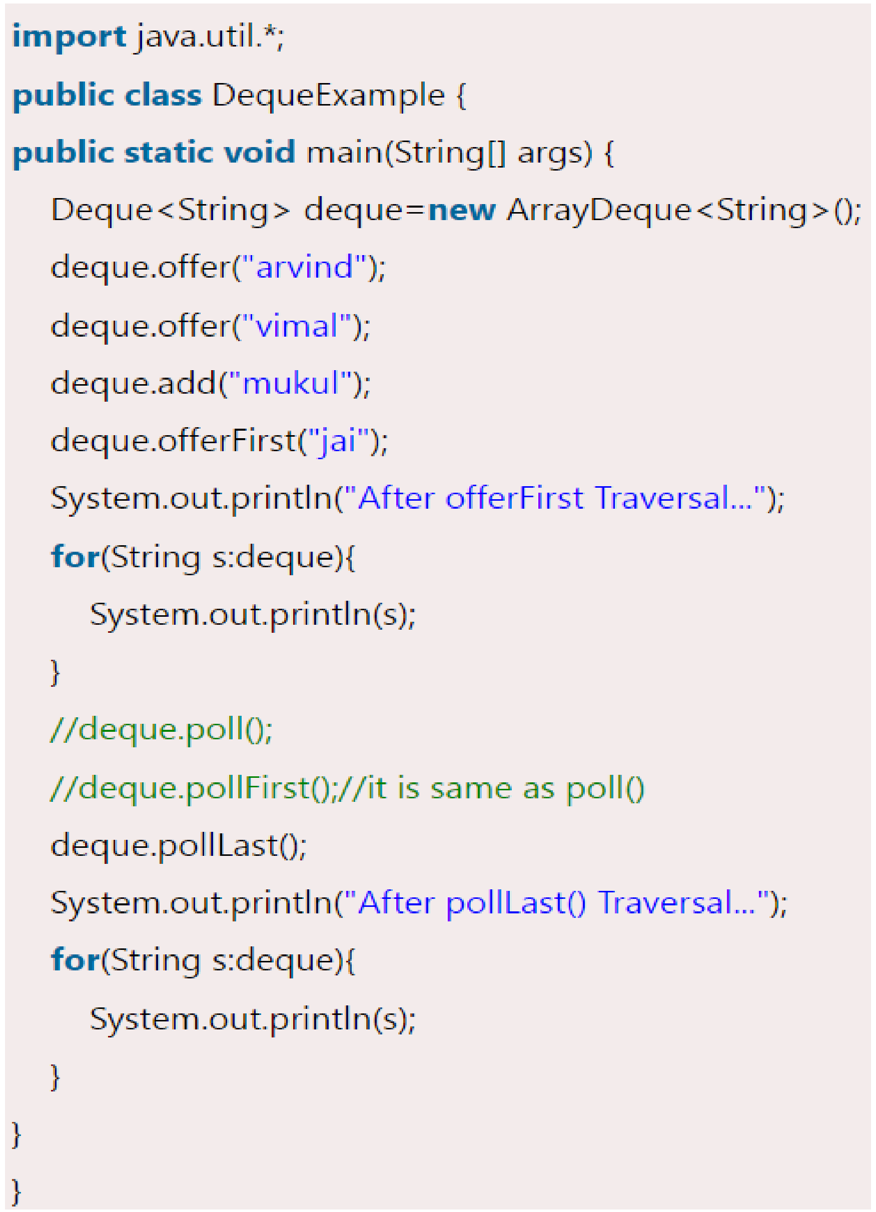
| | **Deque Methods** | |
|---|---|---|
| ***Type of Operation*** | ***First Element (Beginning of the Deque instance)*** | ***Last Element (End of the Deque instance)*** |
| **Insert** | addFirst(e)<br>offerFirst(e) | addLast(e)<br>offerLast(e) |
| **Remove** | removeFirst()<br>pollFirst() | removeLast()<br>pollLast() |
| **Examine** | getFirst()<br>peekFirst() | getLast()<br>peekLast() |

- ❖ All the above interfaces (**Collection, List, Set, SortedSet, NavigableSet, Queue**) used to represent a group of individual objects only.
- ❖ To represent group of objects as key-value pairs , then Map interface has to be used

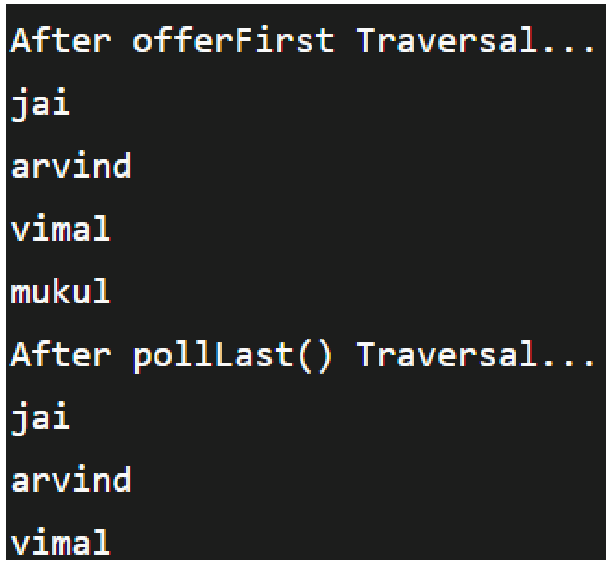
## PriorityQueue Demo

```
import java.util.*;
public class PriorityQueueDemo {
    public static void main(String[] args) {

        PriorityQueue pq=new
PriorityQueue();
        pq.add(10);
        pq.add(20);
        pq.add(500);
        pq.add(5);
        pq.add(50);
        pq.add(200);
        pq.offer(1000);
        System.out.println(pq);
        pq.remove();
        pq.remove();
        pq.remove();
        System.out.println(pq.peek());
        System.out.println(pq);
    }
}
```

## ArrayDeque Demo



```
import java.util.*;
public class DequeExample {
public static void main(String[] args) {
    Deque<String> deque=new ArrayDeque<String>();
    deque.offer("arvind");
    deque.offer("vimal");
    deque.add("mukul");
    deque.offerFirst("jai");
    System.out.println("After offerFirst Traversal...");
    for(String s:deque){
        System.out.println(s);
    }
    //deque.poll();
    //deque.pollFirst();//it is same as poll()
    deque.pollLast();
    System.out.println("After pollLast() Traversal...");
    for(String s:deque){
        System.out.println(s);
    }
}
}
```



```
After offerFirst Traversal...
jai
arvind
vimal
mukul
After pollLast() Traversal...
jai
arvind
vimal
```

## ArrayDeque with
## User Defined class - Book

```java
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {
   this.id = id;
   this.name = name;
   this.author = author;
   this.publisher = publisher;
   this.quantity = quantity;
}
}
```

```java
public class ArrayDequeExample {
public static void main(String[] args) {
   Deque<Book> set=new ArrayDeque<Book>();
   //Creating Books
   Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
   Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
   Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
   //Adding Books to Deque
   set.add(b1);
   set.add(b2);
   set.add(b3);
   //Traversing ArrayDeque
   for(Book b:set){
   System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
   }
}
}
```

```
101 Let us C Yashwant Kanetkar BPB 8
102 Data Communications & Networking Forouzan Mc Graw Hill 4
103 Operating System Galvin Wiley 6
```

# Key Interfaces of Collection Framework

## 7. 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.

✓Map is used to represent a group of individual objects as **key-value pairs** (Ex: id - name)

✓Both key and value are objects only

✓Duplicate keys are not allowed, but values can be duplicated

✓The Map interface includes methods for **basic operations:** put(), get(), remove(), containsKey(), containsValue(), size(), and empty()

✓**Bulk operations:** (putAll() and clear(), and collection views (such as keySet(), entrySet(), and values()).

## Collection Views

The Collection view methods allow a Map to be viewed as a Collection in these three ways:

**keySet** — the Set of keys contained in the Map.
**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.

## Maps

A map is a collection whose elements have two parts: a key and a value.

The combination of a key and a value is called a mapping.

The map stores the mappings based on the key part of the mapping, in a way similar to how a Set collection stores its elements.

The map uses keys to quickly locate associated values.

| | |
|---|---|
| put(key : K, value : V) : V | Adds a mapping that associates V with K, and returns the value previously associated with K. Returns null if there was no value associated with K. |
| remove(key : Object) : V | Removes the mapping associated with the given key from the map, and returns the associated value. If there is not such mapping, returns null. |
| size() : int | Returns the number of mappings in the map. |
| values() : Collection<V> | Returns a collection consisting of all values stored in the map. |

| | |
|---|---|
| clear() : void | Removes all elements from the map. |
| containsValue(value: Object):boolean | Returns true if the map contains a mapping with the given value. |
| containsKey(key : Object) : boolean | Returns true if the map contains a mapping with the given key. |
| get(key : Object) : V | Returns the value associated with the specified key, or returns null if there is no such value. |
| isEmpty() : boolean | Returns true if the key contains no mappings. |
| keySet() : Set<K> | Returns the set of all keys stored in the map. |

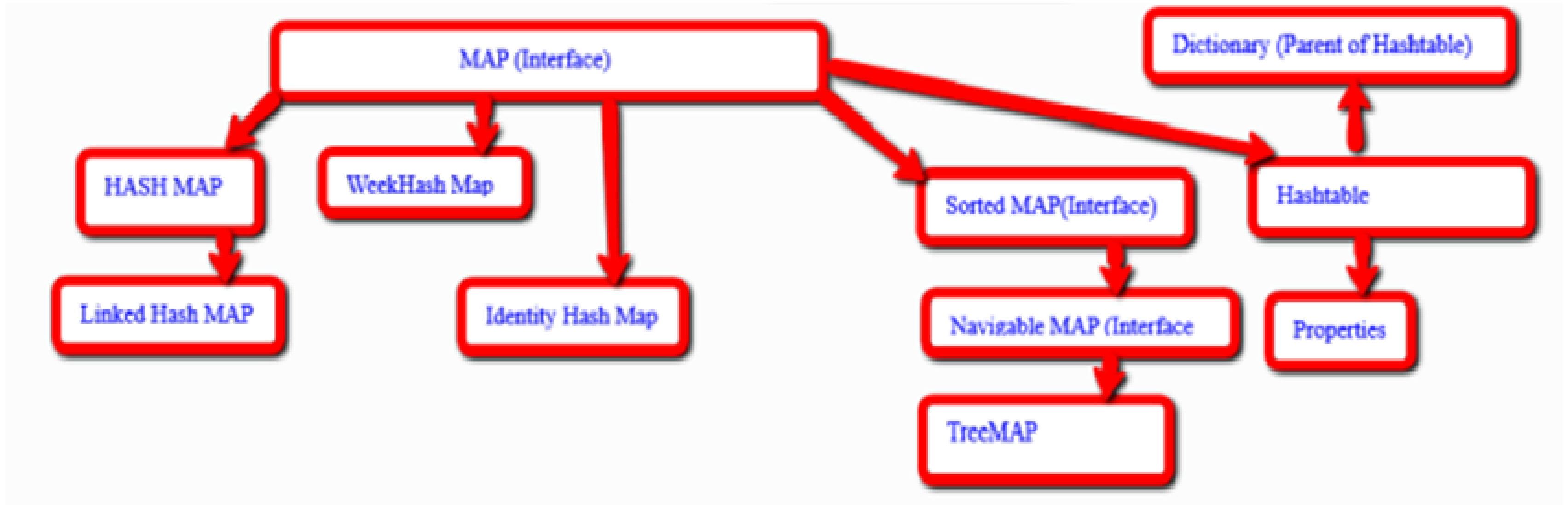
## Concrete Map Classes

Maps store keys with attached values. The keys are stored as sets.

- HashMap stores keys according to their hash codes, just like HashSet stores its elements.
- LinkedHashMap is a HashMap that can iterate over the keys in insertion order (order in which mappings were inserted) or in access order (order of last access).
- TreeMap stores mappings according to the natural order of the keys, or according to an order specified by a Comparator.

# Key Interfaces of Collection Framework

## 7. Map (Interface):



❖Map is not child interface of Collection
❖Hashtable, Properties and Dictionary are the legacy classes

## 8. SortedMap (Interface):

✓It's a child interface of Map
✓Used to represent a group of individual objects as **key-value** pairs according to some **sorting order**
✓Sorting should be done only based on **keys** but not on values

## 9. NavigableMap (Interface):

✓It's a child interface of SortedMap
✓Defines several methods for **navigation** purpose

Below example shows how to read add elements from HashMap. The method keySet() returns all key entries as a set object. Iterating through each key, we can get corresponding value object.

```
import java.util.HashMap;
import java.util.Set;

public class MyHashMapRead {
   public static void main(String a[]){
      HashMap<String, String> hm = new HashMap<String, String>();
      //add key-value pair to hashmap
      hm.put("first", "FIRST INSERTED");
      hm.put("second", "SECOND INSERTED");
      hm.put("third","THIRD INSERTED");
      System.out.println(hm);
      Set<String> keys = hm.keySet();
      for(String key: keys){
         System.out.println("Value of "+key+" is: "+hm.get(key));
      }
   }
}
```

Output:

```
{second=SECOND INSERTED, third=THIRD INSERTED, first=FIRST INSERTED}
Value of second is: SECOND INSERTED
Value of third is: THIRD INSERTED
Value of first is: FIRST INSERTED
```

Below example shows how to find whether specified value exists or not. By using containsValue() method you can find out the value existance.

```
import java.util.HashMap;

public class MyHashMapValueSearch {

   public static void main(String a[]){
      HashMap<String, String> hm = new HashMap<String, String>();
      //add key-value pair to hashmap
      hm.put("first", "FIRST INSERTED");
      hm.put("second", "SECOND INSERTED");
      hm.put("third","THIRD INSERTED");
      System.out.println(hm);
      if(hm.containsValue("SECOND INSERTED")){
         System.out.println("The hashmap contains value SECOND INSERTED");
      } else {
         System.out.println("The hashmap does not contains value SECOND INSERTED");
      }
      if(hm.containsValue("first")){
         System.out.println("The hashmap contains value first");
      } else {
         System.out.println("The hashmap does not contains value first");
      }
   }
}
```

Output:

```
{second=SECOND INSERTED, third=THIRD INSERTED, first=FIRST INSERTED}
The hashmap contains value SECOND INSERTED
The hashmap does not contains value first
```

## HashMap Demo

```
import java.util.*;

public class HashMapDemo {
    public static void main(String[] args) {
        HashMap m=new HashMap();
        m.put("sadhu",5400);
        m.put("simi",5400);
        m.put("sharan",6600);
        m.put("pramod",7600);
        System.out.println(m);
        m.put("nag",5400);
        m.put("bsrk",5400);
        System.out.println(m);
```

```
//Set s=m.keySet();
        //System.out.println(s);

        Collection s1=m.keySet();
        System.out.println(s1);
        Collection c=m.values();
        System.out.println(c);

        Set s2=m.entrySet();
        Iterator itr=s2.iterator();
        while(itr.hasNext()){
        Map.Entry m1=(Map.Entry)itr.next();
        System.out.println(m1.getKey()+"__"+m1.getValue());
        }
        }
    }
```

Below example shows how to get all key-value pair as Entry objects. Entry class provides getter methods to access key-value details. The method entrySet() provides all entries as set object.

```
import java.util.HashMap;
import java.util.Map.Entry;
import java.util.Set;

public class MyHashMapEntrySet {

   public static void main(String a[]){
      HashMap<String, String> hm = new HashMap<String, String>();
      //add key-value pair to hashmap
      hm.put("first", "FIRST INSERTED");
      hm.put("second", "SECOND INSERTED");
      hm.put("third","THIRD INSERTED");
      System.out.println(hm);
      //getting value for the given key from hashmap
      Set<Entry<String, String>> entires = hm.entrySet();
      for(Entry<String,String> ent:entires){
         System.out.println(ent.getKey()+" ==> "+ent.getValue());
      }
   }
}
```

Output:

```
{second=SECOND INSERTED, third=THIRD INSERTED, first=FIRST INSERTED}
second ==> SECOND INSERTED
third ==> THIRD INSERTED
```

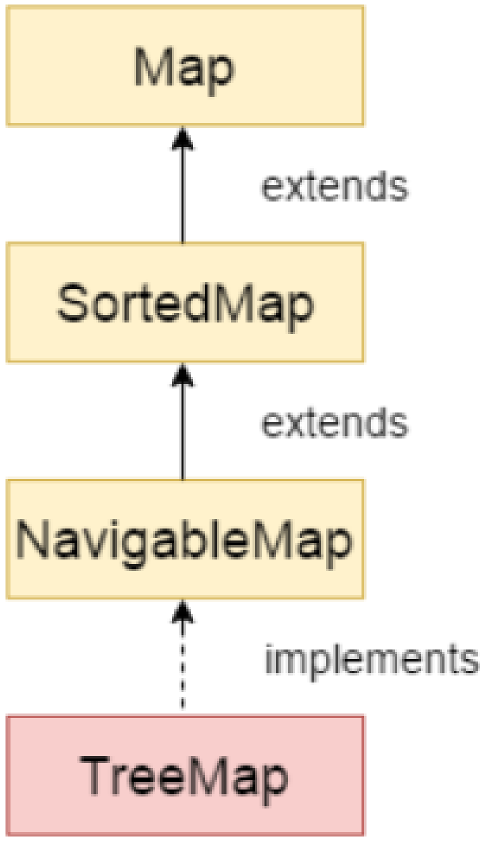
Below example shows how to find whether specified key exists or not. By using containsKey() method you can find out the key existance.

```
import java.util.HashMap;

public class MyHashMapKeySearch {

    public static void main(String a[]){
        HashMap<String, String> hm = new HashMap<String, String>();
        //add key-value pair to hashmap
        hm.put("first", "FIRST INSERTED");
        hm.put("second", "SECOND INSERTED");
        hm.put("third","THIRD INSERTED");
        System.out.println(hm);
        if(hm.containsKey("first")){
            System.out.println("The hashmap contains key first");
        } else {
            System.out.println("The hashmap does not contains key first");
        }
        if(hm.containsKey("fifth")){
            System.out.println("The hashmap contains key fifth");
        } else {
            System.out.println("The hashmap does not contains key fifth");
        }
    }
}
```

Output:

```
{second=SECOND INSERTED, third=THIRD INSERTED, first=FIRST INSERTED}
The hashmap contains key first
The hashmap does not contains key fifth
```

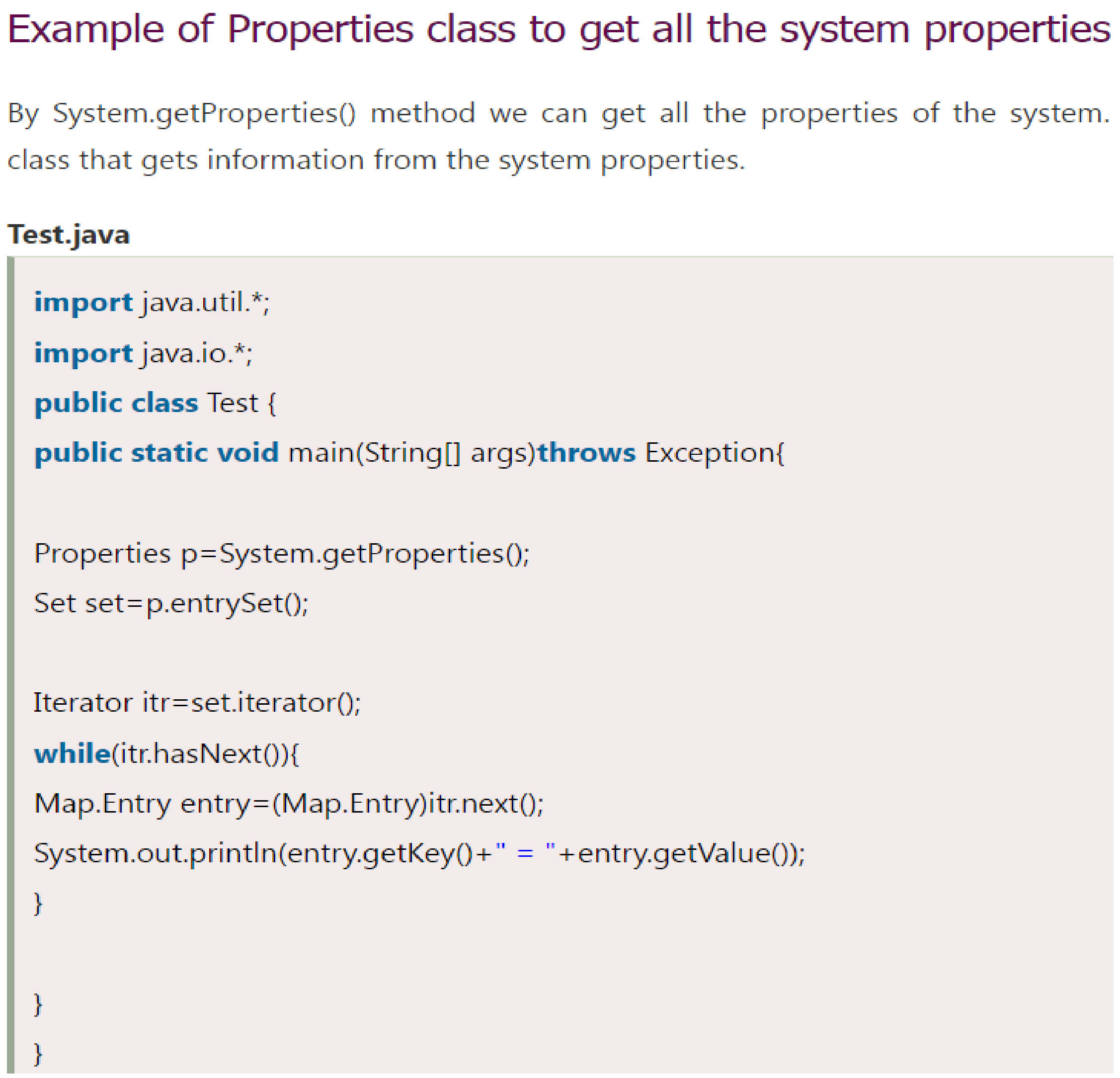
# TreeMap Demo



```
import java.util.*;

public class TreeMapDemo {
    public static void main(String[] args) {

        TreeMap m=new TreeMap(new MyComparator());
        m.put(105,"asdf");
        m.put(101, "pqr");
        m.put(102, "abc");
        m.put(103, "xyz");
        m.put(104, "mno");
        System.out.println(m);
    }
}
class MyComparator implements Comparator{
    public int compare(Object o1, Object o2){
        String s1=o1.toString();
        String s2=o2.toString();
        return s2.compareTo(s1);
    }
}
```

## IdentityHashMap Demo

```
import java.util.*;
public class IdentityHashMapDemo {

    public static void main(String[] args) {

        IdentityHashMap m=new IdentityHashMap();
        Integer i1=new Integer(10);
        Integer i2=new Integer(10);
        m.put(i1,"sadhu");
        m.put(i2,"sreeni");
        System.out.println(m);
    }
}
```



### Example of Properties class to get all the system properties

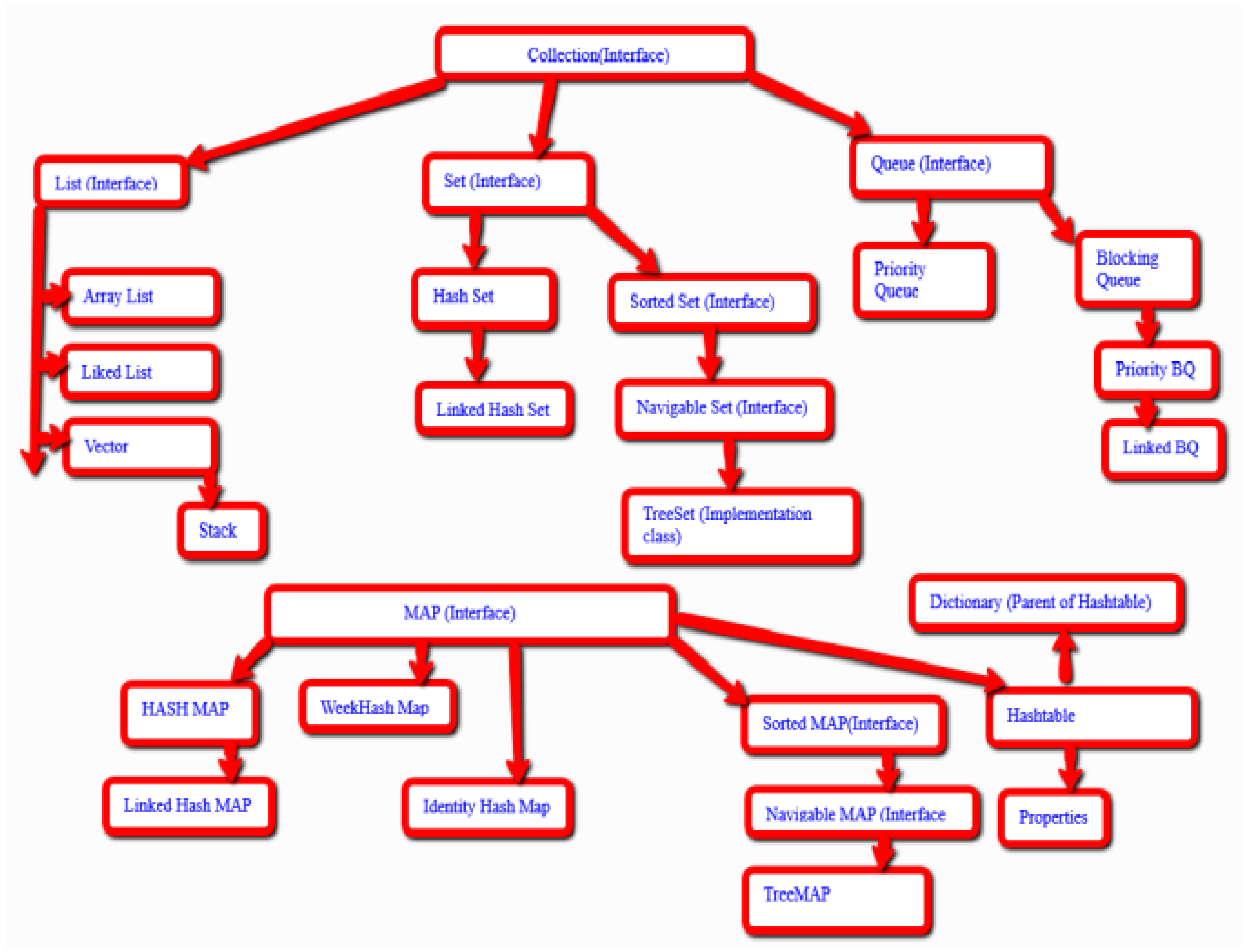
By System.getProperties() method we can get all the properties of the system. class that gets information from the system properties.

**Test.java**

```
import java.util.*;
import java.io.*;
public class Test {
public static void main(String[] args)throws Exception{

Properties p=System.getProperties();
Set set=p.entrySet();

Iterator itr=set.iterator();
while(itr.hasNext()){
Map.Entry entry=(Map.Entry)itr.next();
System.out.println(entry.getKey()+" = "+entry.getValue());
}


}
}
```

# Collection Framework - Summary

Collection Framework

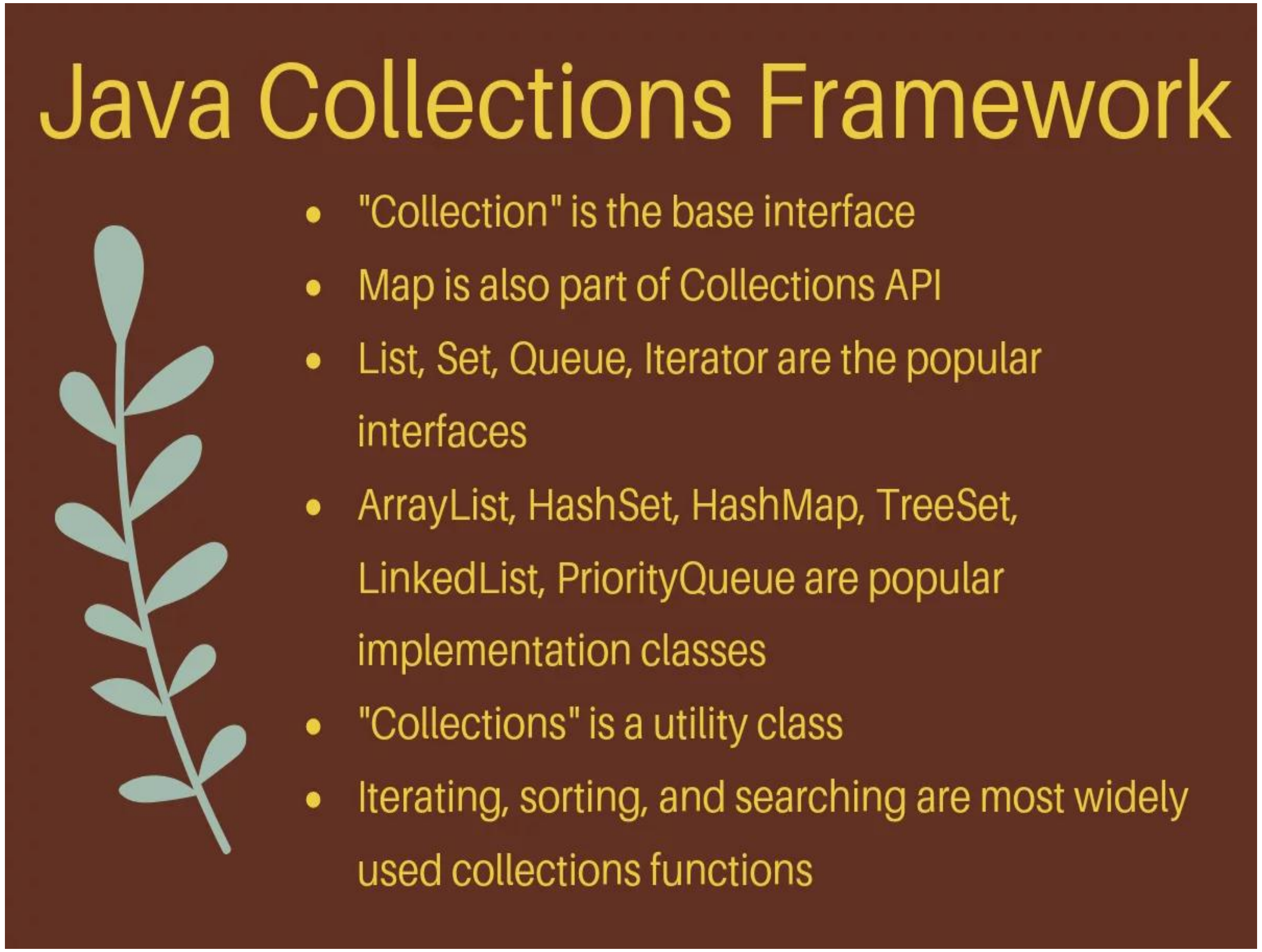


## Collection Framework : and more..

**Utility Classes:**
1. Arrays - *applies for arrays*
2. Collections - *A List 'l' may be sorted as follows:*
   *Collections. Sort(l);*

**Cursors (Iterators):**
1.Enumeration - *for legacy classes*
2.Interator - *Universal iterator*
3.ListIterator – *list types*

**Interfaces (for Sorting):**
1. Comparable – natural sorting order
2. Comparator - any other order



Java Collections Framework

- "Collection" is the base interface
- Map is also part of Collections API
- List, Set, Queue, Iterator are the popular interfaces
- ArrayList, HashSet, HashMap, TreeSet, LinkedList, PriorityQueue are popular implementation classes
- "Collections" is a utility class
- Iterating, sorting, and searching are most widely used collections functions

## Summary of Collection Framework Interfaces

**The Java Collections Framework hierarchy consists of two distinct interface trees:**

➢**The first tree** starts with the **Collection** interface, which provides for the basic functionality used by all collections, such as add and remove methods.

➢Its sub interfaces — Set, List, and Queue — provide for more specialized collections.

➢The Set interface does not allow duplicate elements. This can be useful for storing collections such as a deck of cards or student records. The Set interface has a subinterface, SortedSet, that provides for ordering of elements in the set.

➢The List interface provides for an ordered collection, for situations in which you need precise control over where each element is inserted. You can retrieve elements from a List by their exact position

➢The Queue interface enables additional insertion, extraction, and inspection operations. Elements in a Queue are typically ordered in on a FIFO basis.

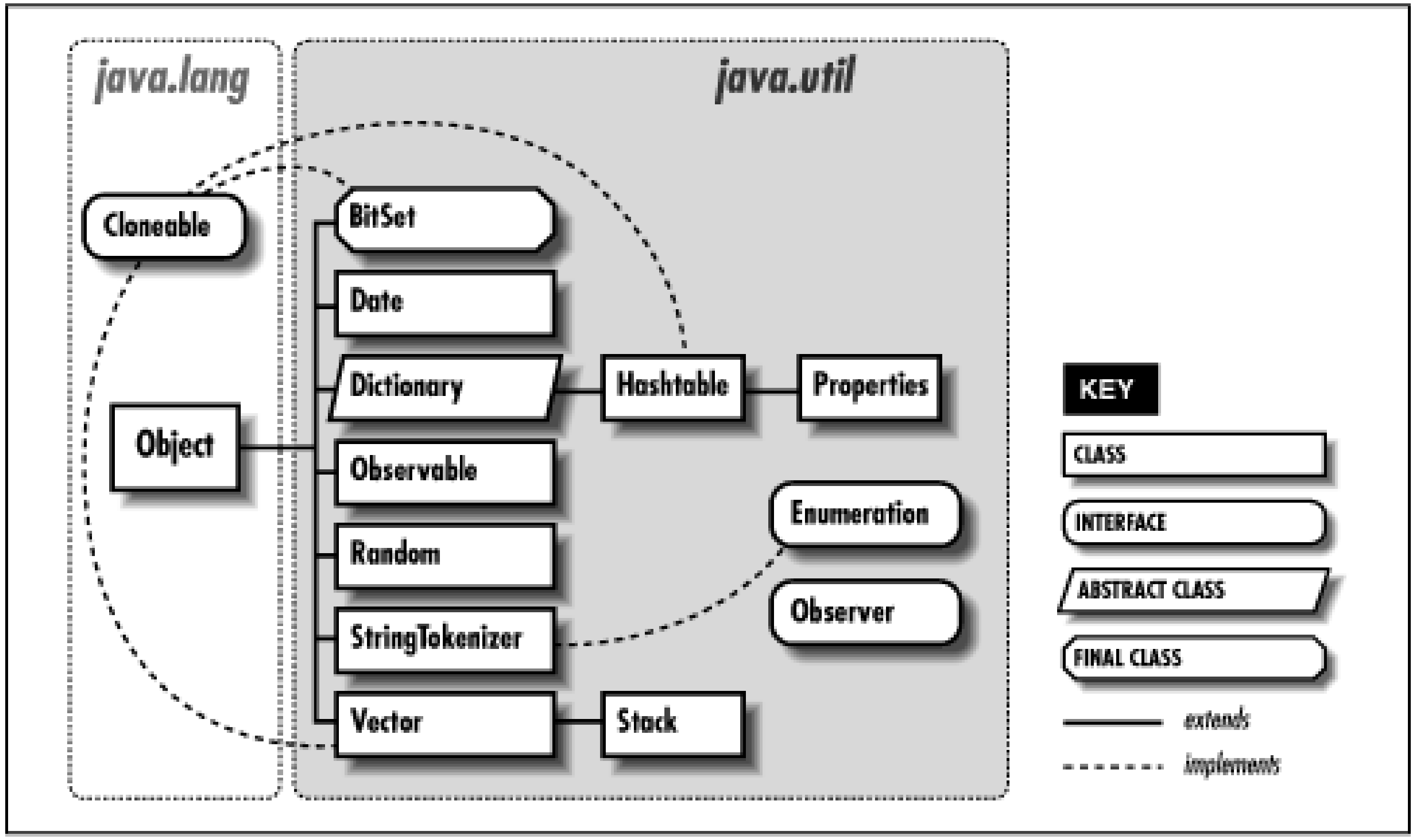
➢The Deque interface enables insertion, deletion, and inspection operations at both the ends. Elements in a Deque can be used in both LIFO and FIFO.

**The second tree** starts with the **Map** interface, which maps keys and values similar to a Hashtable.
Map's subinterface, SortedMap, maintains its key-value pairs in ascending order or in an order specified by a Comparator.

These interfaces allow collections to be manipulated independently of the details of their representation.

# Other Utility Classes:

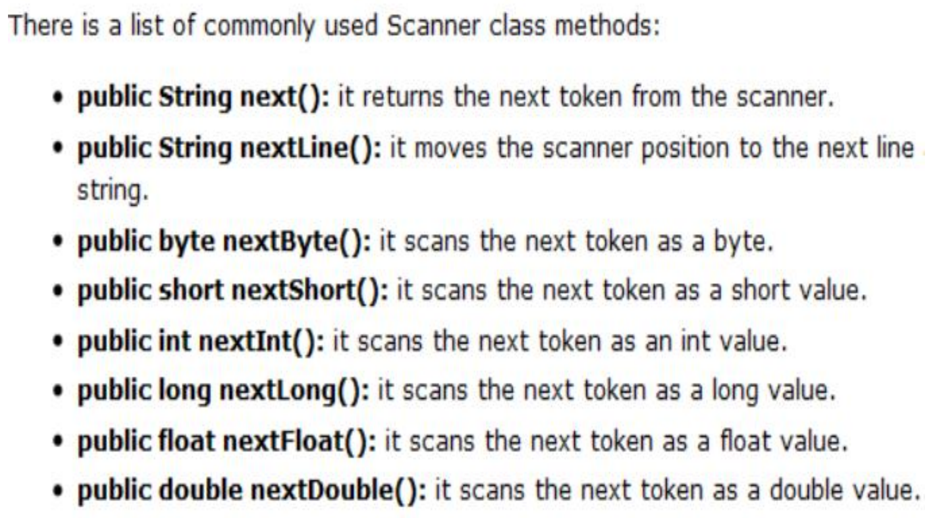
- ✓ **Date**
- ✓ **Scanner**
- ✓ **Calendar**
- ✓ **Formatter**
- ✓ **StringTokenizer**
- ✓ **BitSet**
- ✓ **Random**



## Date class

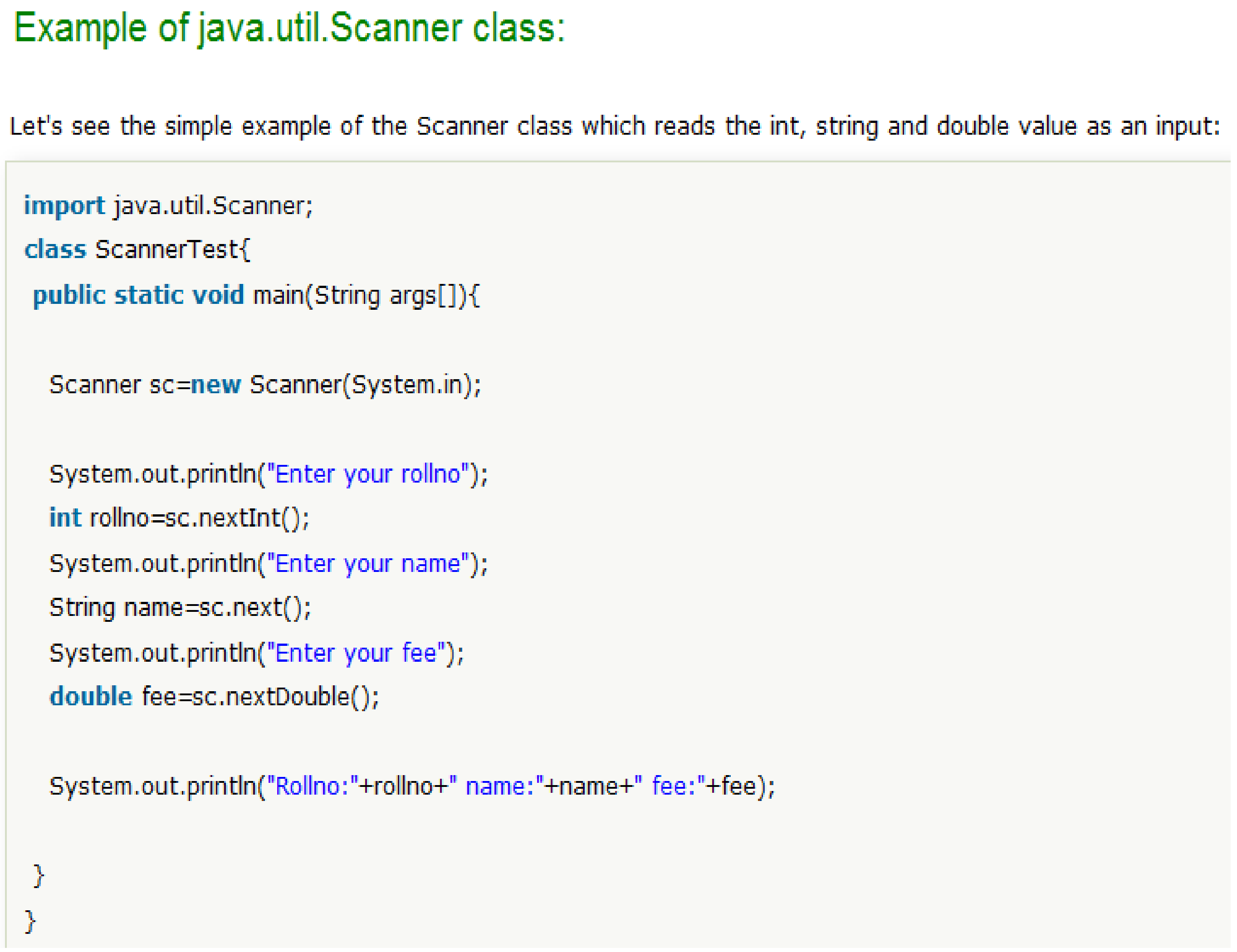
The **java.util.Date** class represents a specific instant in time, with millisecond precision

```
import java.util.Date; // explicit import
//import java.sql.*; // implicit import
public class Test {

    public static void main(String[] args) {
        Date d=new Date();
        System.out.println(d);
    }
}
```



There is a list of commonly used Scanner class methods:

- **public String next():** it returns the next token from the scanner.
- **public String nextLine():** it moves the scanner position to the next line string.
- **public byte nextByte():** it scans the next token as a byte.
- **public short nextShort():** it scans the next token as a short value.
- **public int nextInt():** it scans the next token as an int value.
- **public long nextLong():** it scans the next token as a long value.
- **public float nextFloat():** it scans the next token as a float value.
- **public double nextDouble():** it scans the next token as a double value.



### Example of java.util.Scanner class:

Let's see the simple example of the Scanner class which reads the int, string and double value as an input:

```
import java.util.Scanner;
class ScannerTest{
 public static void main(String args[]){

   Scanner sc=new Scanner(System.in);

   System.out.println("Enter your rollno");
   int rollno=sc.nextInt();
   System.out.println("Enter your name");
   String name=sc.next();
   System.out.println("Enter your fee");
   double fee=sc.nextDouble();

   System.out.println("Rollno:"+rollno+" name:"+name+" fee:"+fee);

 }
}
```

The java.util.Scanner class is a simple text scanner which can parse primitive types and strings using regular expression

## StringTokenizer

The **java.util.StringTokenizer** class allows you to break a String into tokens. It is simple way to break a String. It is a legacy class of Java

```
import java.util.StringTokenizer;
public class Simple{
 public static void main(String args[]){
   StringTokenizer st = new StringTokenizer("my name is khan"," ");
     while (st.hasMoreTokens()) {
         System.out.println(st.nextToken());
     }
   }
}
```

**Output:**

```
my
name
is
khan
```

```
import java.util.StringTokenizer;
public class StringTokenizer3
{
 /* Driver Code */
 public static void main(String args[])
 {
   /* StringTokenizer object */
   StringTokenizer st = new StringTokenizer("Hello Everyone Have a nice day"," ");
         /* Prints the number of tokens present in the String */
         System.out.println("Total number of Tokens: "+st.countTokens());
 }
}
```

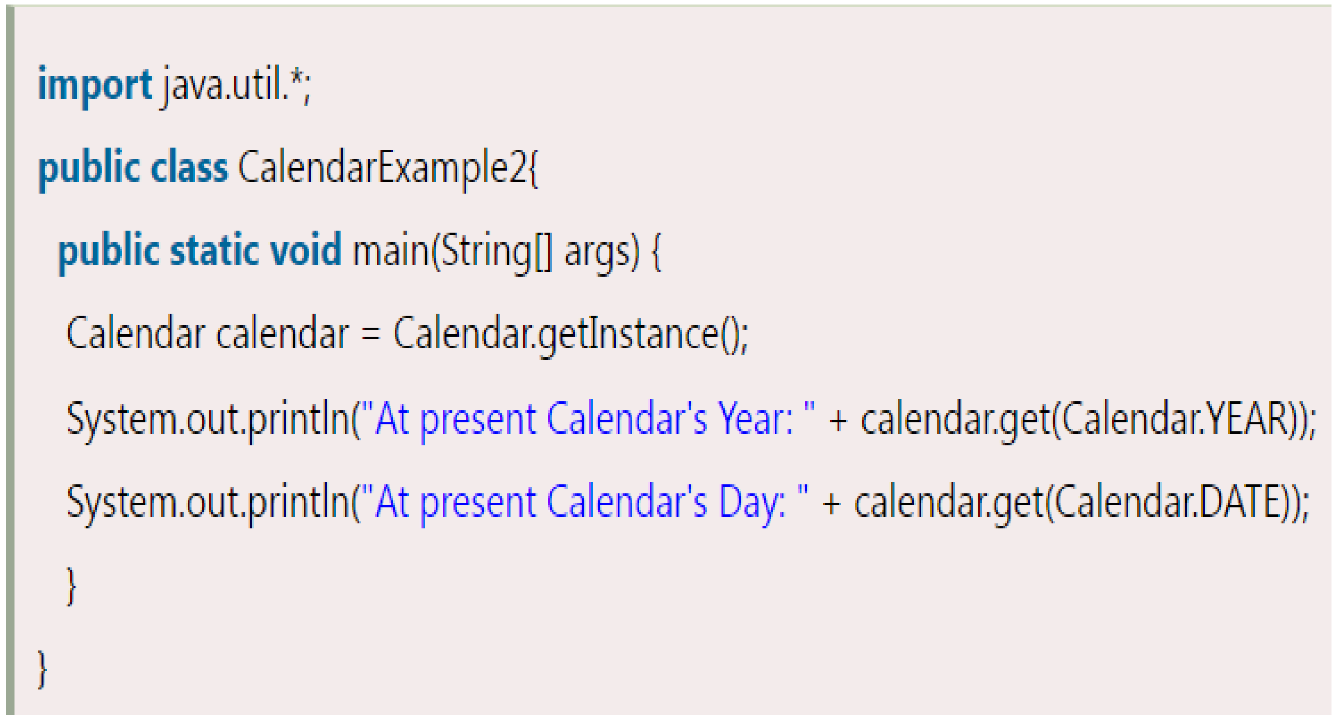
**Output:**

```
Total number of Tokens: 6
```

## Calendar class

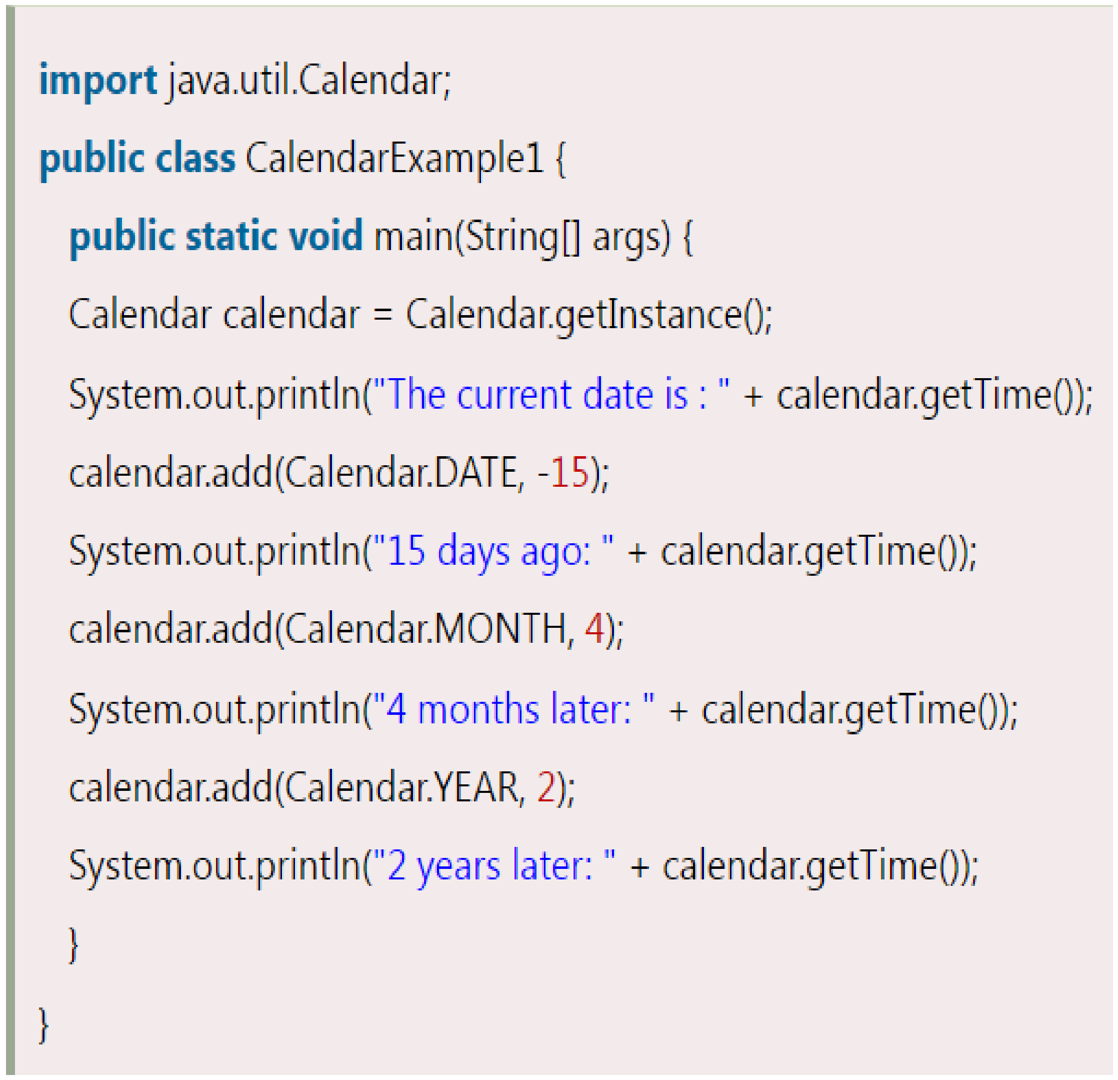
Java Calendar class is an abstract class that provides methods for converting date between a specific instant in time and a set of calendar fields such as MONTH, YEAR, HOUR, etc. It inherits Object class and implements the Comparable interface.

### Java Calendar Class Example: get()



```
import java.util.*;
public class CalendarExample2{
  public static void main(String[] args) {
   Calendar calendar = Calendar.getInstance();
   System.out.println("At present Calendar's Year: " + calendar.get(Calendar.YEAR));
   System.out.println("At present Calendar's Day: " + calendar.get(Calendar.DATE));
   }
}
```

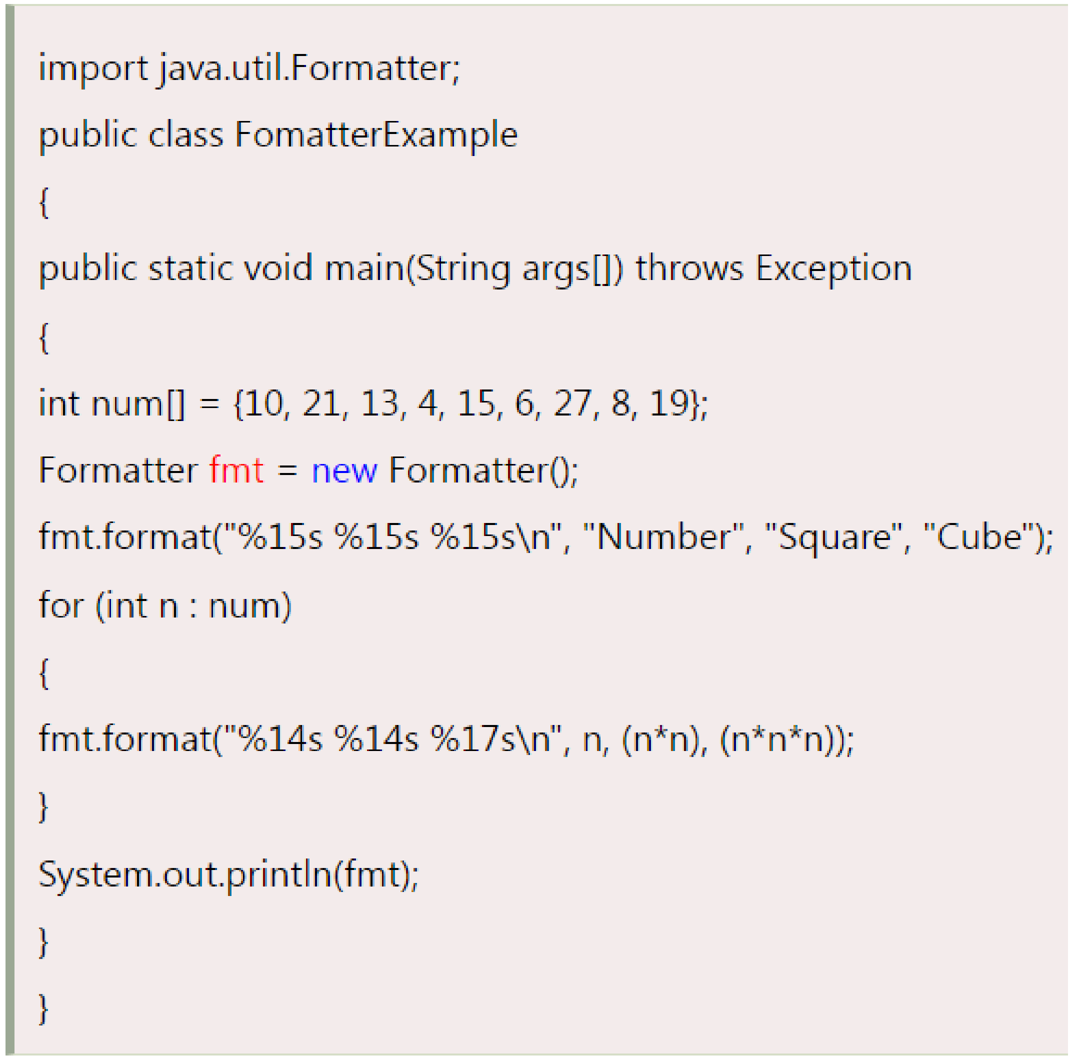
### Java Calendar Class Example



```
import java.util.Calendar;
public class CalendarExample1 {
   public static void main(String[] args) {
   Calendar calendar = Calendar.getInstance();
   System.out.println("The current date is : " + calendar.getTime());
   calendar.add(Calendar.DATE, -15);
   System.out.println("15 days ago: " + calendar.getTime());
   calendar.add(Calendar.MONTH, 4);
   System.out.println("4 months later: " + calendar.getTime());
   calendar.add(Calendar.YEAR, 2);
   System.out.println("2 years later: " + calendar.getTime());
   }
}
```

## Formatter class

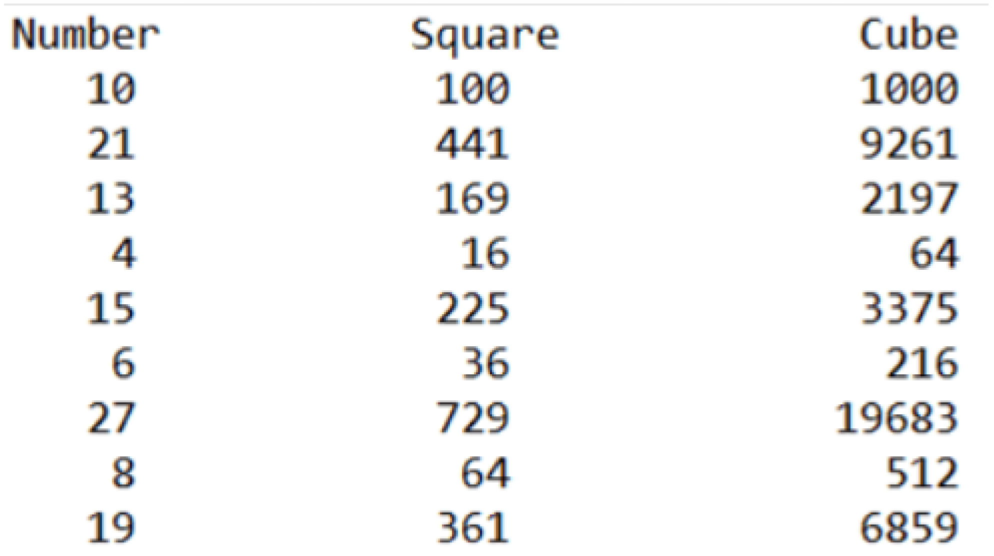
**FormatterExample.java**



```
import java.util.Formatter;
public class FomatterExample
{
public static void main(String args[]) throws Exception
{
int num[] = {10, 21, 13, 4, 15, 6, 27, 8, 19};
Formatter fmt = new Formatter();
fmt.format("%15s %15s %15s\n", "Number", "Square", "Cube");
for (int n : num)
{
fmt.format("%14s %14s %17s\n", n, (n*n), (n*n*n));
}
System.out.println(fmt);
}
}
```

The **java.util.Formatter** class provides support for layout justification and alignment, common formats for numeric, string, and date/time data, and locale-specific output.

Formatter uses format() method.



```
Number          Square           Cube
    10             100           1000
    21             441           9261
    13             169           2197
     4              16             64
    15             225           3375
     6              36            216
    27             729          19683
     8              64            512
    19             361           6859
```

## Random class

Java Random class is used to generate a stream of pseudorandom numbers.

```
import java.util.Random;
public class JavaRandomExample2 {
  public static void main(String[] args) {
    Random random = new Random();
    //return the next pseudorandom integer value
    System.out.println("Random Integer value : "+random.nextInt());
    // setting seed
    long seed =20;
    random.setSeed(seed);
    //value after setting seed
    System.out.println("Seed value : "+random.nextInt());
    //return the next pseudorandom long value
    Long val = random.nextLong();
    System.out.println("Random Long value : "+val);
  }
}
```

| Methods | Description |
|---|---|
| doubles() | Returns an unlimited stream of pseudorandom double values. |
| ints() | Returns an unlimited stream of pseudorandom int values. |
| longs() | Returns an unlimited stream of pseudorandom long values. |
| next() | Generates the next pseudorandom number. |
| nextBoolean() | Returns the next uniformly distributed pseudorandom boolean value from the random number generator's sequence |
| nextByte() | Generates random bytes and puts them into a specified byte array. |
| nextDouble() | Returns the next pseudorandom Double value between 0.0 and 1.0 from the random number generator's sequence |
| nextFloat() | Returns the next uniformly distributed pseudorandom Float value between 0.0 and 1.0 from this random number generator's sequence |

## BitSet Class

The Java **BitSet** class implements a vector of bits.
The BitSet grows automatically as more bits are needed.
The BitSet class comes under *java.util* package.
The BitSet class extends the Object class and provides the implementation of Serializable and Cloneable interfaces

Each component of bit set contains at least one Boolean value.
The contents of one BitSet may be changed by other BitSet using logical AND, logical OR and logical exclusive OR operations.
The index of bits of BitSet class is represented by positive integers.

```
import java.util.BitSet;
public class BitSetDemo {

  public static void main(String args[]) {
      BitSet bits1 = new BitSet(16);
      BitSet bits2 = new BitSet(16);

      // set some bits
      for(int i = 0; i < 16; i++) {
         if((i % 2) == 0) bits1.set(i);
         if((i % 5) != 0) bits2.set(i);
      }

      System.out.println("Initial pattern in bits1: ");
      System.out.println(bits1);
      System.out.println("\nInitial pattern in bits2: ");
      System.out.println(bits2);

      // AND bits
      bits2.and(bits1);
      System.out.println("\nbits2 AND bits1: ");
      System.out.println(bits2);

      // OR bits
      bits2.or(bits1);
      System.out.println("\nbits2 OR bits1: ");
      System.out.println(bits2);
```

```
Initial pattern in bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

Initial pattern in bits2:
{1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14}

bits2 AND bits1:
{2, 4, 6, 8, 12, 14}

bits2 OR bits1:
{0, 2, 4, 6, 8, 10, 12, 14}
```

# Questions?

Thank you!!

Sadhu Sreenivas, PTO
C-DAC Hyderabad
+91 9848 2524 98