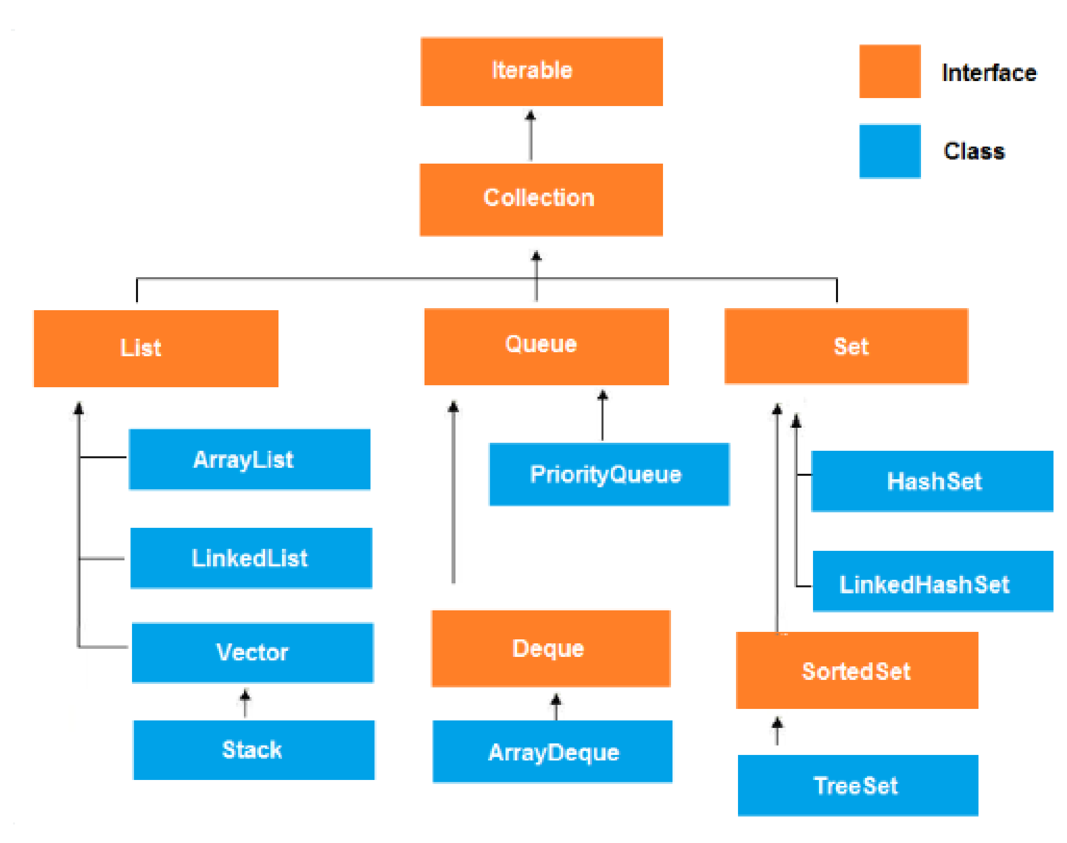
JAVA COLLECTIONS

JAVA COLLECTIONS

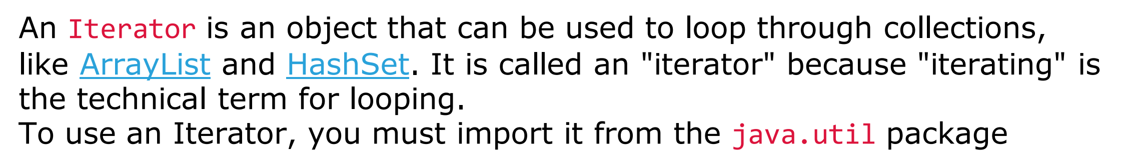
By Ayala Berkovich



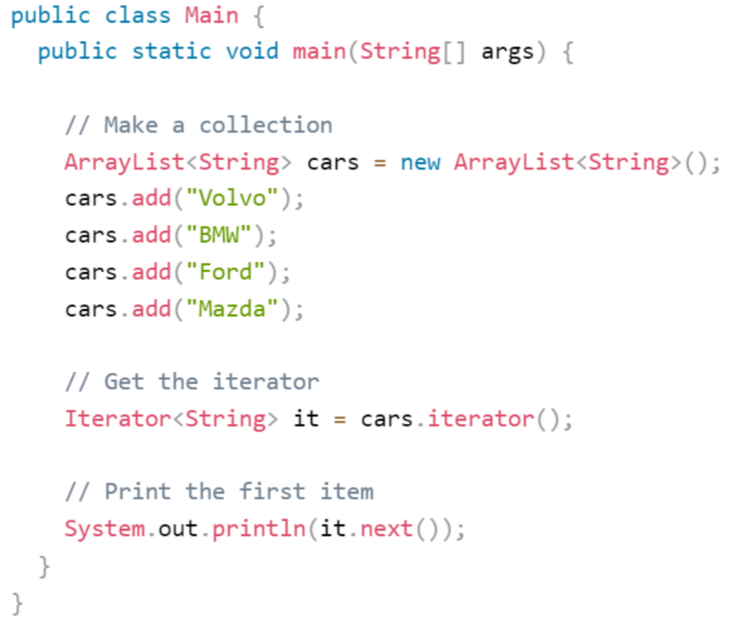
Iterable is a super interface to **Collection**, so any class (such as Set or List…) that implements Collection also implements Iterable.
Has just one method:

Iterator<T> iterator()

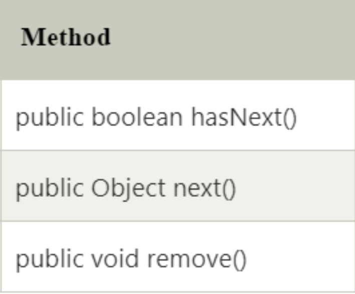
Returns an iterator over a set of elements of type T



An `Iterator` is an object that can be used to loop through collections, like ArrayList and HashSet. It is called an "iterator" because "iterating" is the technical term for looping.
To use an Iterator, you must import it from the `java.util` package



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

    // Make a collection
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");

    // Get the iterator
    Iterator<String> it = cars.iterator();

    // Print the first item
    System.out.println(it.next());
  }
}
```



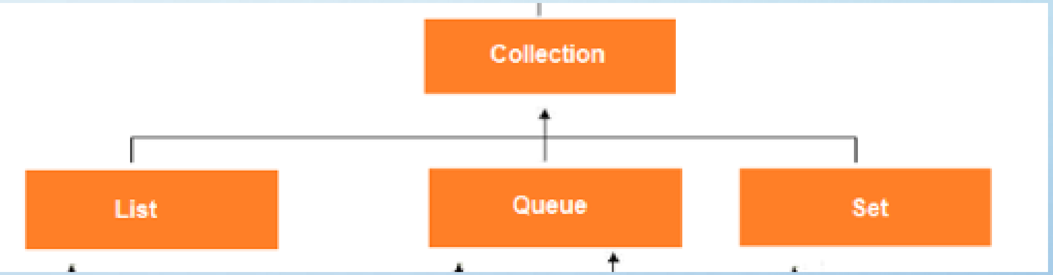
Method
public boolean hasNext()
public Object next()
public void remove()

By Ayala Berkovich

JAVA COLLECTIONS

The **Collection** interface contains methods that perform basic operations, such as :

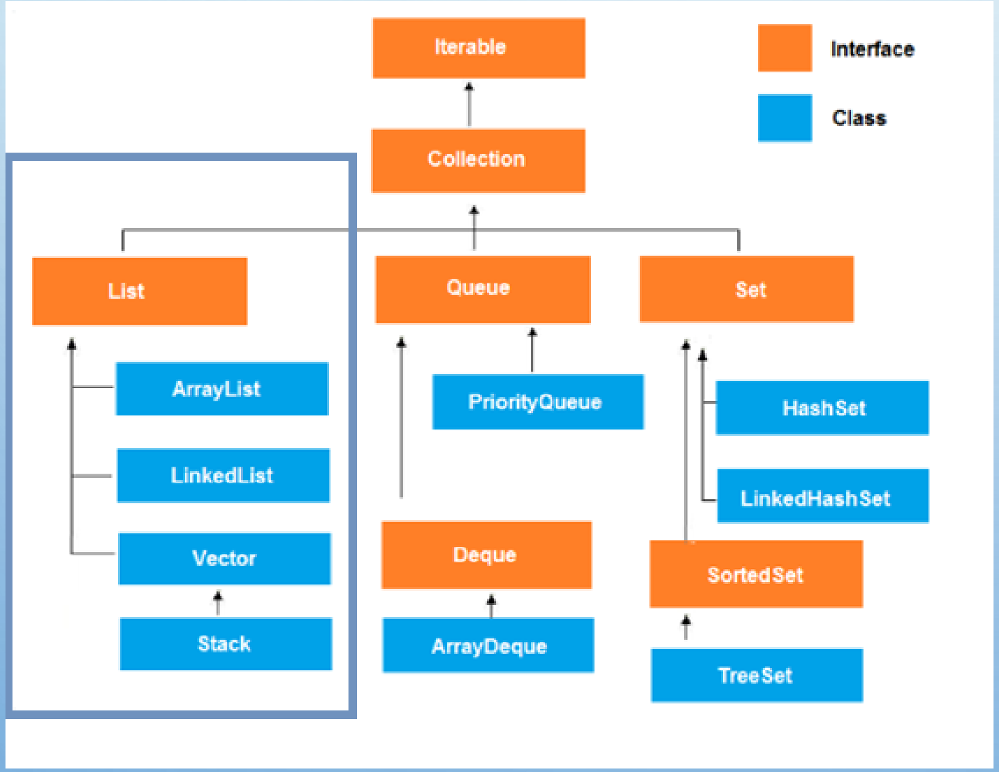
- int size(),
- boolean isEmpty(),
- boolean contains(Object element),
- boolean add(E element)
- boolean remove(Object element)
- boolean containsAll(Collection<?> c)
- boolean addAll(Collection<? extends E> c)
- boolean removeAll(Collection<?> c)
- boolean retainAll(Collection<?> c)
- void clear()



By Ayala Berkovich

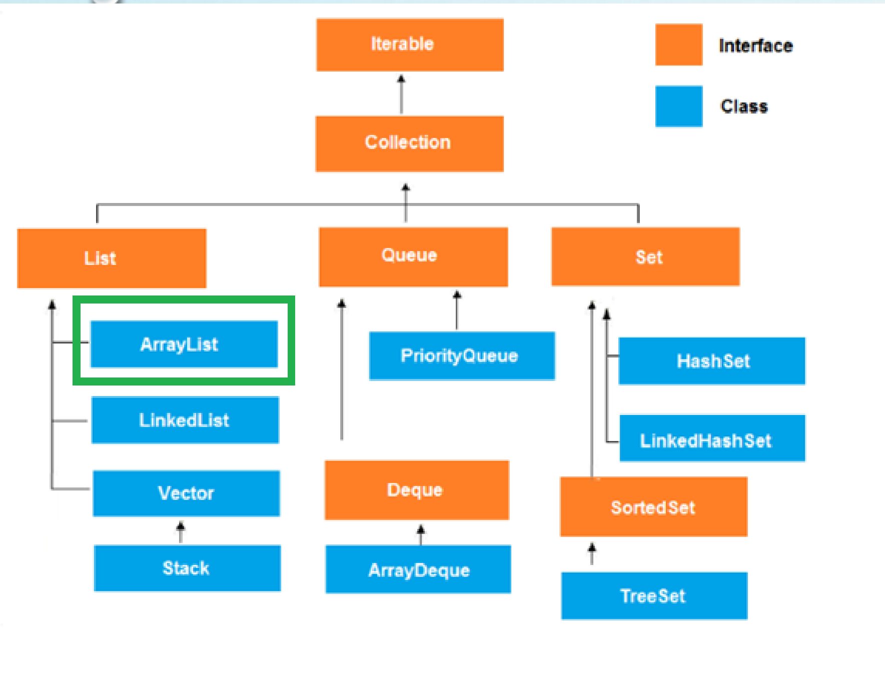
JAVA COLLECTIONS

LIST INTERFACE IS THE CHILD INTERFACE OF COLLECTION INTERFACE. WHERE WE CAN STORE THE ORDERED COLLECTION OF OBJECTS. LIST INTERFACE IS IMPLEMENTED BY THE CLASSES **ARRAYLIST, LINKEDLIST**, **VECTOR**, AND **STACK**.



By Ayala Berkovich

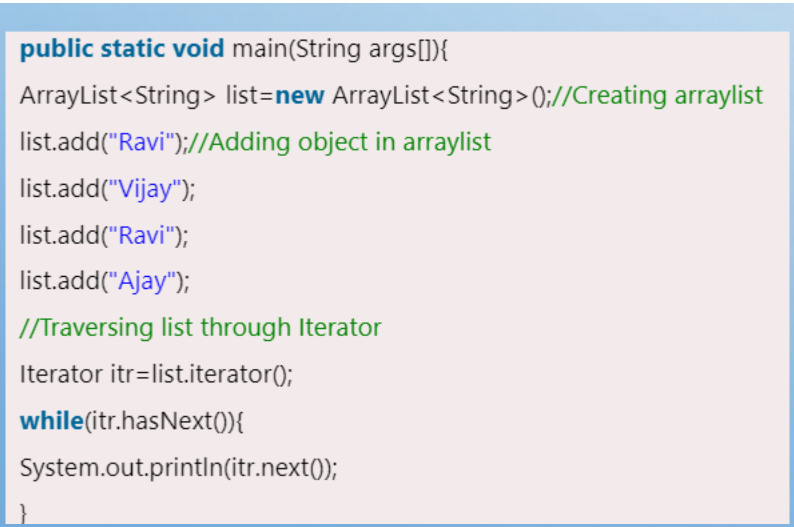
JAVA COLLECTIONS



The **ArrayList** class implements the **List** interface. It uses a dynamic array to **store the duplicate element of different data types**.

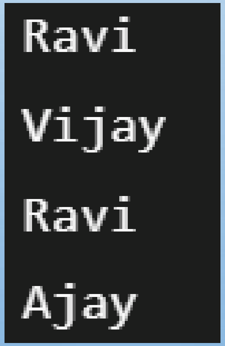
The **ArrayList** class is **non-synchronized -** two or more threads can access the methods of that particular class at any given time. StringBuilder is an example of a non-synchronized class..

The elements stored in the **ArrayList** class can be randomly accessed, opposite is sequential access (**LinkedList**) that you must move through the items of structure



```
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();//Creating arraylist
list.add("Ravi");//Adding object in arraylist
list.add("Vijay");
list.add("Ravi");
list.add("Ajay");
//Traversing list through Iterator
Iterator itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
```

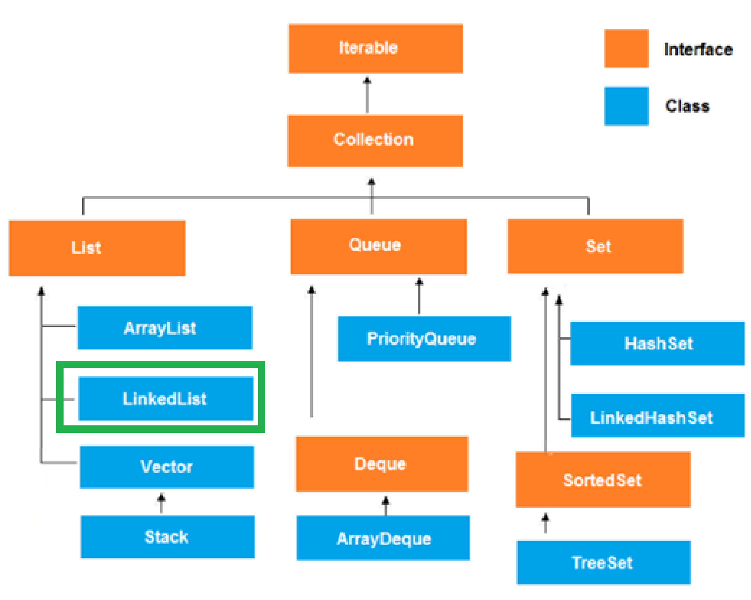
Output



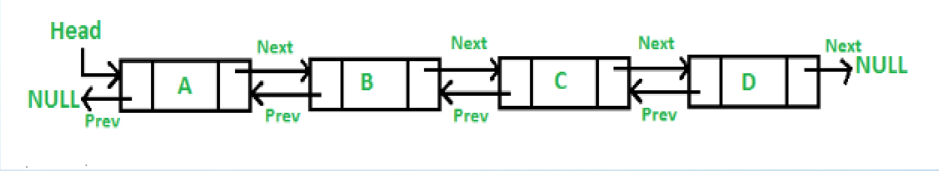
```
Ravi
Vijay
Ravi
Ajay
```

By Ayala Berkovich

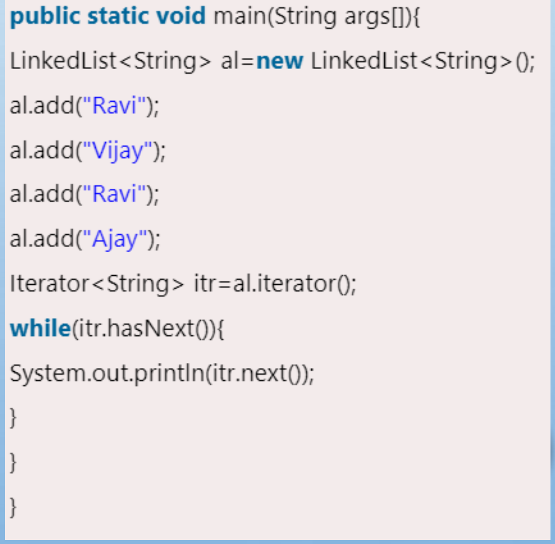
JAVA COLLECTIONS



LinkedList implements the Collection interface.
It uses a doubly linked list internally to store the elements.

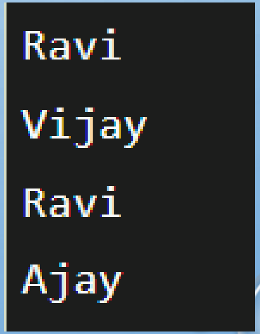


It can store the duplicate elements.
It maintains the insertion order and is **not synchronized**.



```
public static void main(String args[]){
LinkedList<String> al=new LinkedList<String>();
al.add("Ravi");
al.add("Vijay");
al.add("Ravi");
al.add("Ajay");
Iterator<String> itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

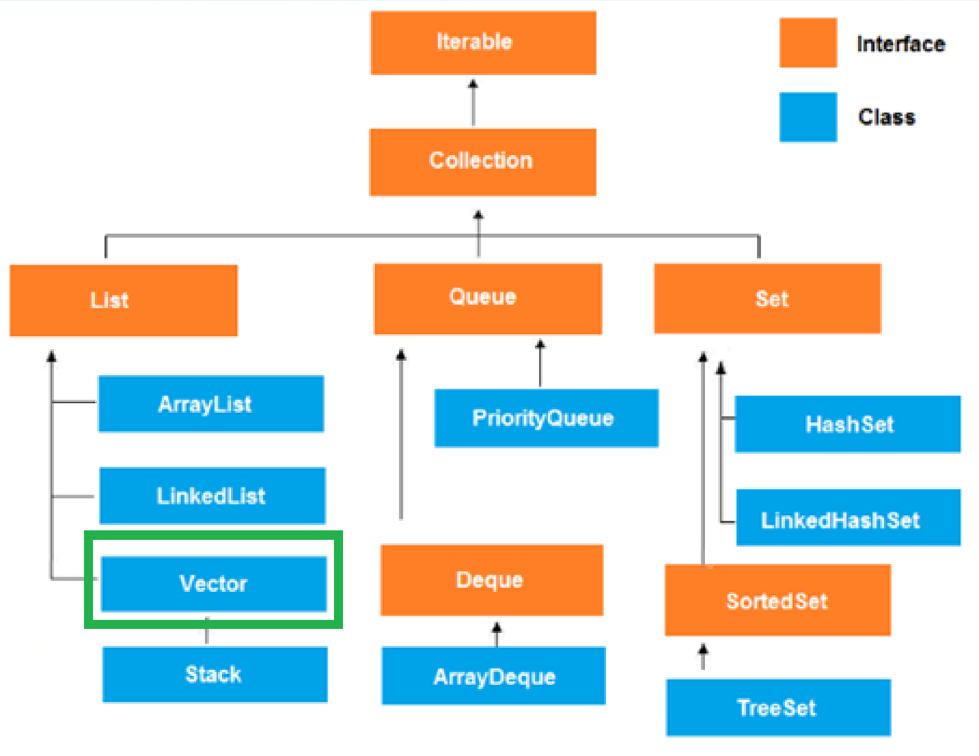
Output



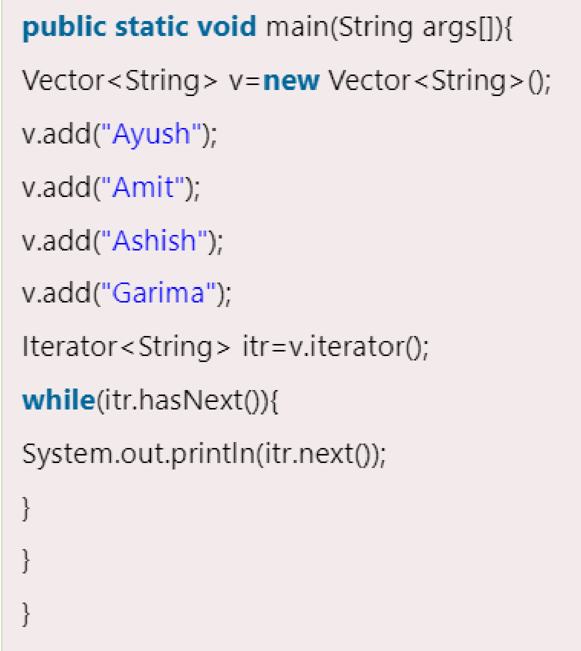
```
Ravi
Vijay
Ravi
Ajay
```

By Ayala Berkovich

JAVA COLLECTIONS

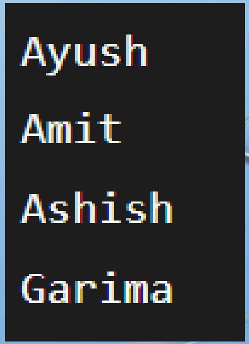


Vector uses a dynamic array to store the data elements.
It is similar to **ArrayList**.
But , It **is synchronized** and contains many methods that are not the part of Collection framework.



```
public static void main(String args[]){
Vector<String> v=new Vector<String>();
v.add("Ayush");
v.add("Amit");
v.add("Ashish");
v.add("Garima");
Iterator<String> itr=v.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

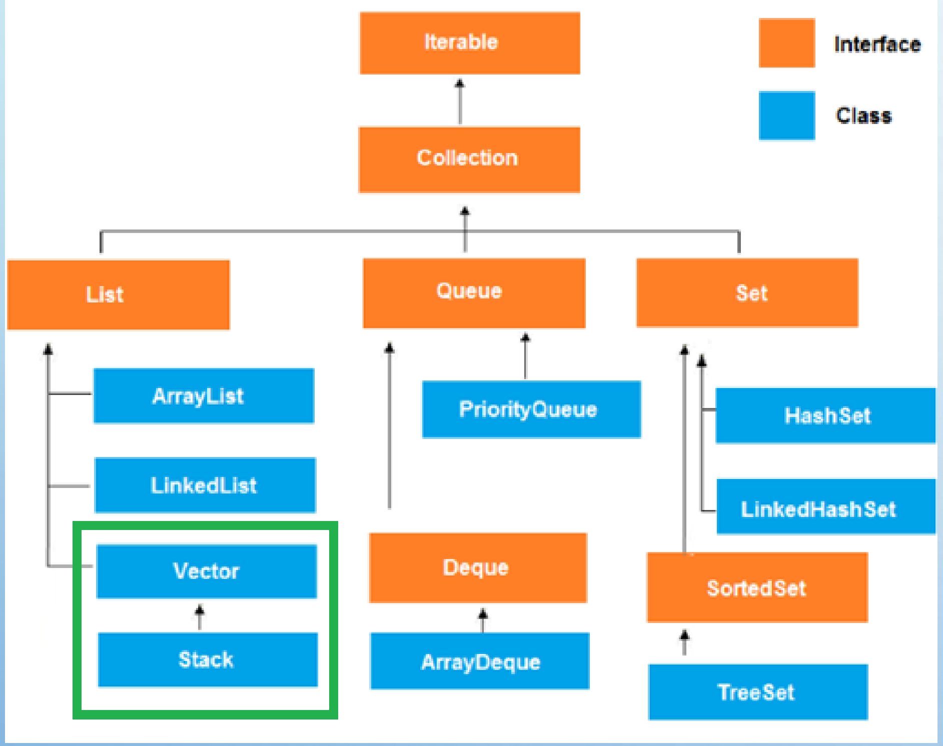
Output



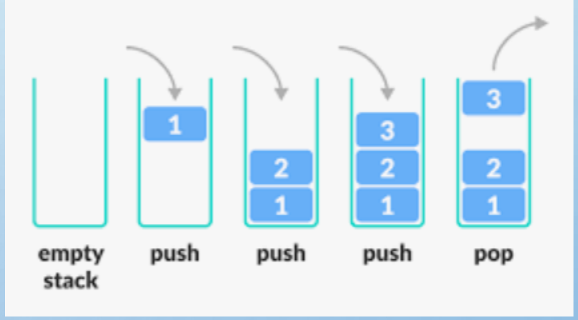
```
Ayush
Amit
Ashish
Garima
```

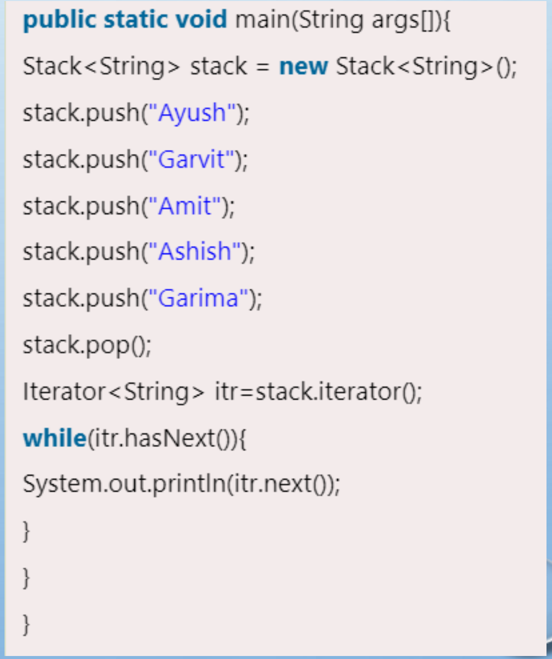
By Ayala Berkovich

JAVA COLLECTIONS



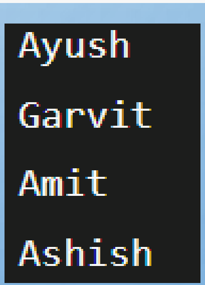
The **stack** is the subclass of **Vector**.
It implements the **last-in-first-out(LIFO)** data structure.
The stack contains all of the methods of **Vector** class and also provides its methods like:
boolean push(), boolean peek()- retrieve or fetch the first element of the Stack, boolean push(object o), which defines its properties.





```
public static void main(String args[]){
Stack<String> stack = new Stack<String>();
stack.push("Ayush");
stack.push("Garvit");
stack.push("Amit");
stack.push("Ashish");
stack.push("Garima");
stack.pop();
Iterator<String> itr=stack.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

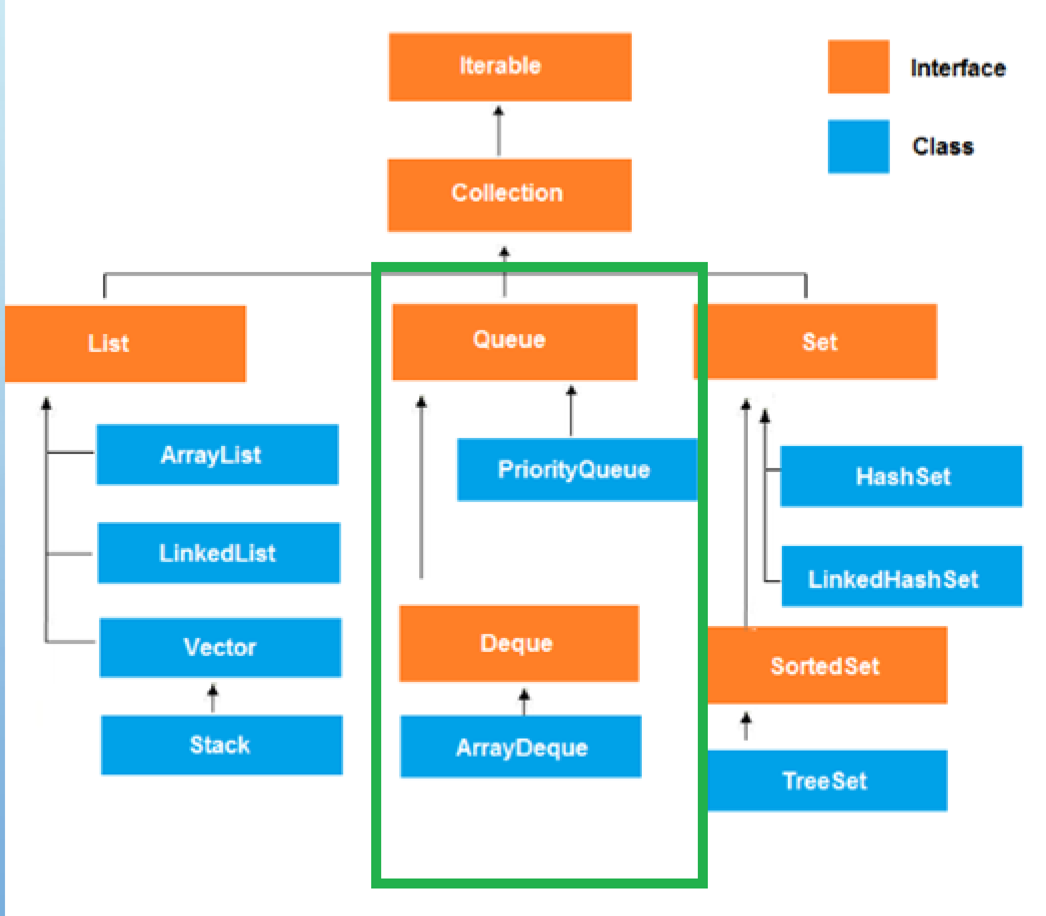
Output



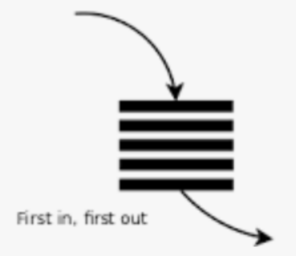
```
Ayush
Garvit
Amit
Ashish
```

By Ayala Berkovich

JAVA COLLECTIONS



Queue interface maintains the first-in-first-out(**FIFO**) order.

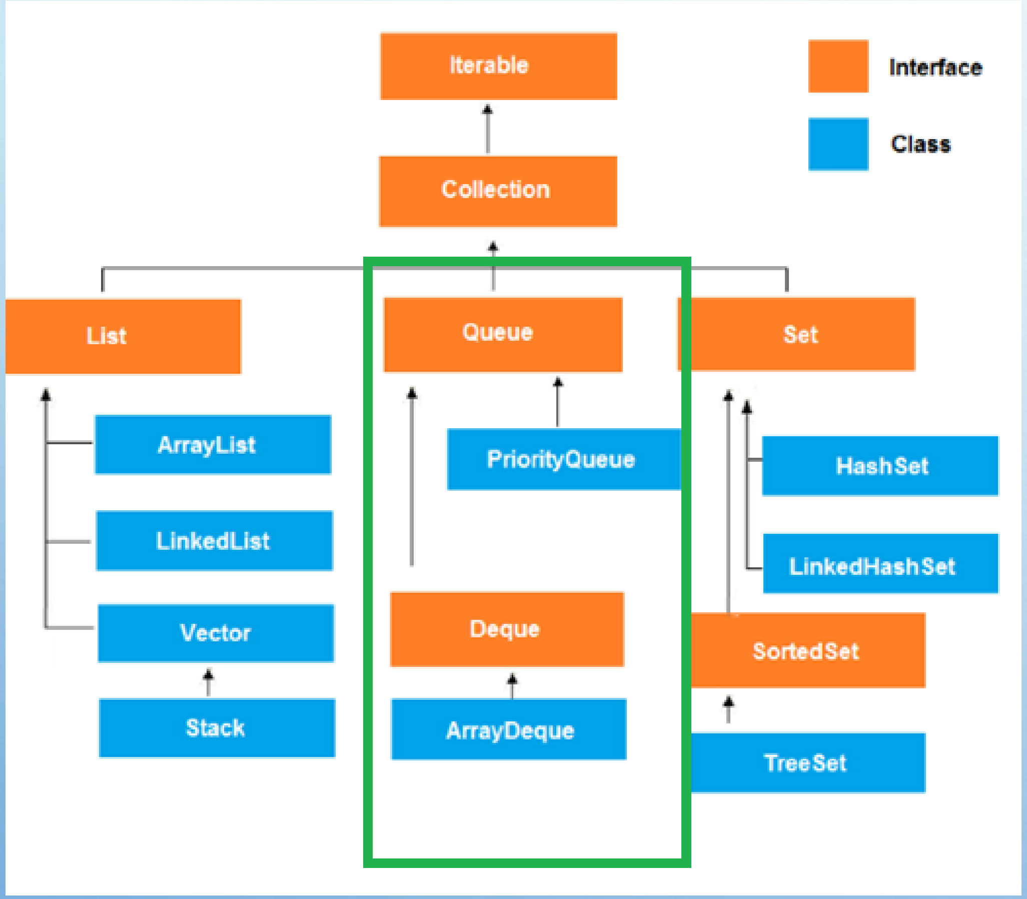


There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.
Queue interface can be instantiated as:

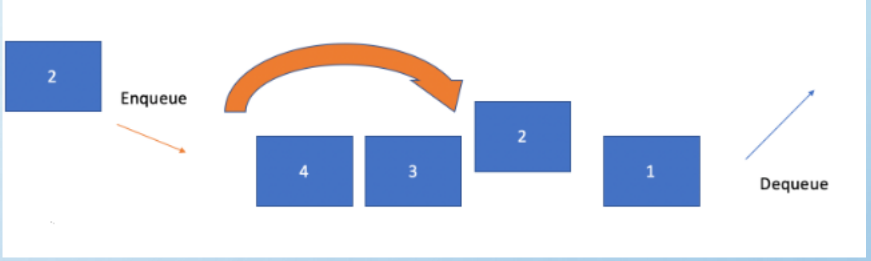
1.Queue<String> q1 = **new** PriorityQueue();
2.Queue<String> q2 = **new** ArrayDeque();

By Ayala Berkovich

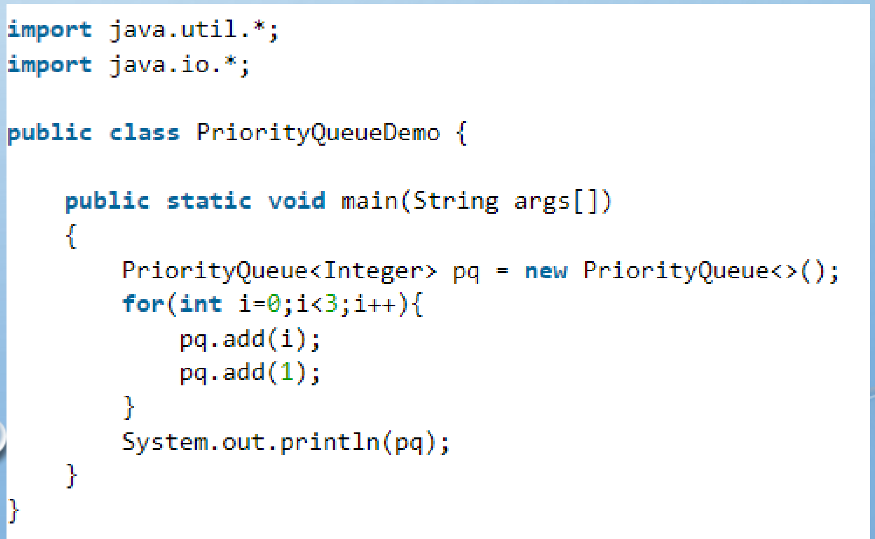
JAVA COLLECTIONS



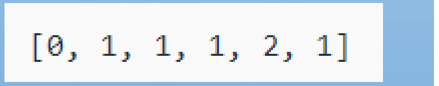
The PriorityQueue class implements the **Queue** interface.
It holds the elements or objects which are to **be processed by their priorities**.



PriorityQueue **doesn't allow null values** to be stored in the queue.



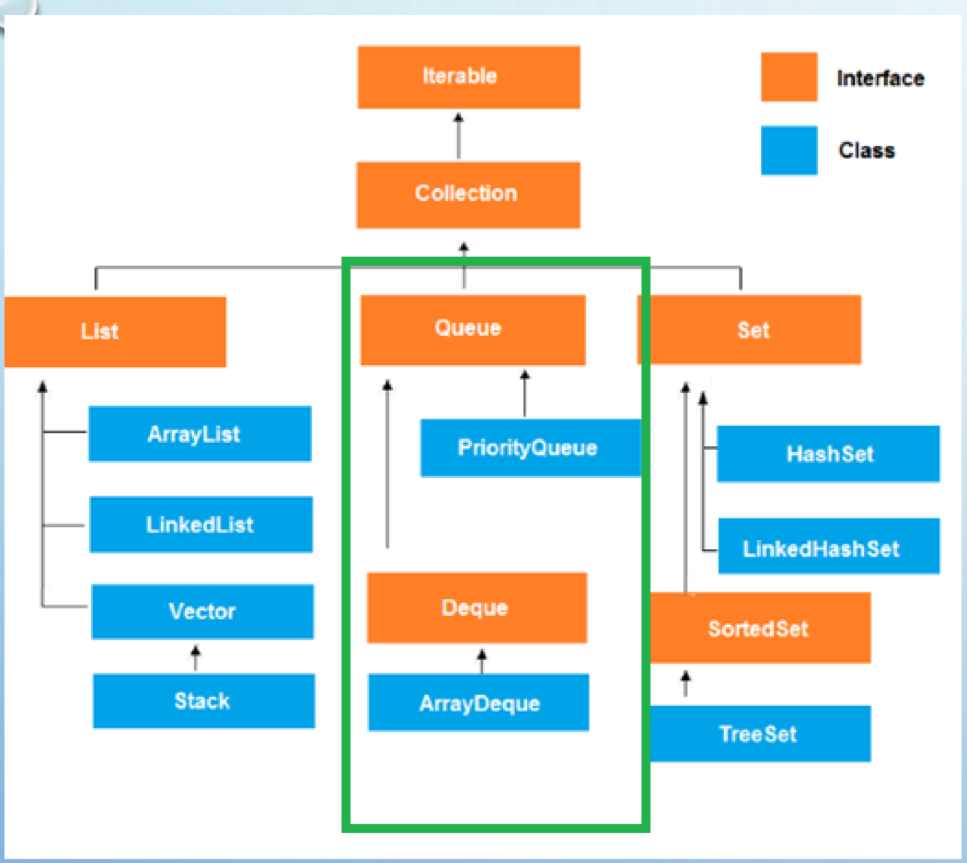
```
import java.util.*;
import java.io.*;

public class PriorityQueueDemo {

    public static void main(String args[])
    {
        PriorityQueue<Integer> pq = new PriorityQueue<>();
        for(int i=0;i<3;i++){
            pq.add(i);
            pq.add(1);
        }
        System.out.println(pq);
    }
}
```



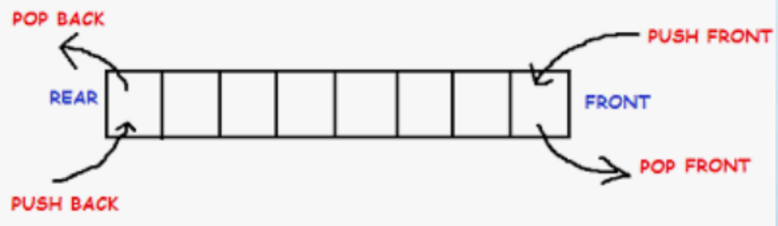
```
[0, 1, 1, 1, 2, 1]
```

By Ayala Berkovich

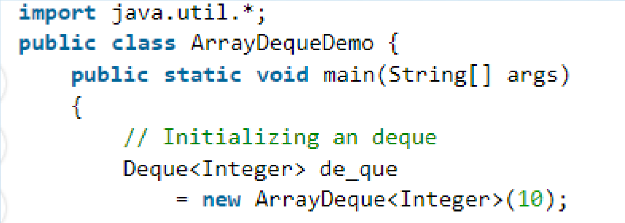
JAVA COLLECTIONS



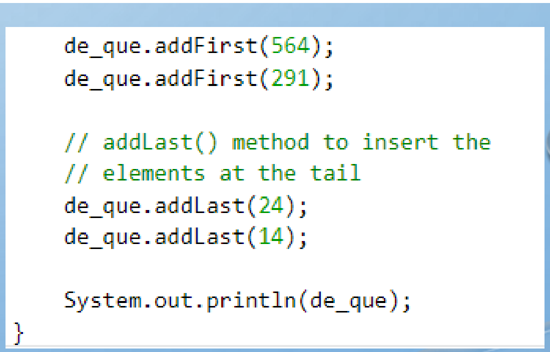
Deque interface extends the **Queue** interface.
In **Deque**, we **can remove and add the elements from both the side**.



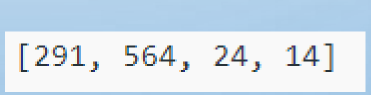
Deque can be instantiated as:
Deque d = **new** ArrayDeque();



```
import java.util.*;
public class ArrayDequeDemo {
    public static void main(String[] args)
    {
        // Initializing an deque
        Deque<Integer> de_que
            = new ArrayDeque<Integer>(10);
```



```
        de_que.addFirst(564);
        de_que.addFirst(291);

        // addLast() method to insert the
        // elements at the tail
        de_que.addLast(24);
        de_que.addLast(14);

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



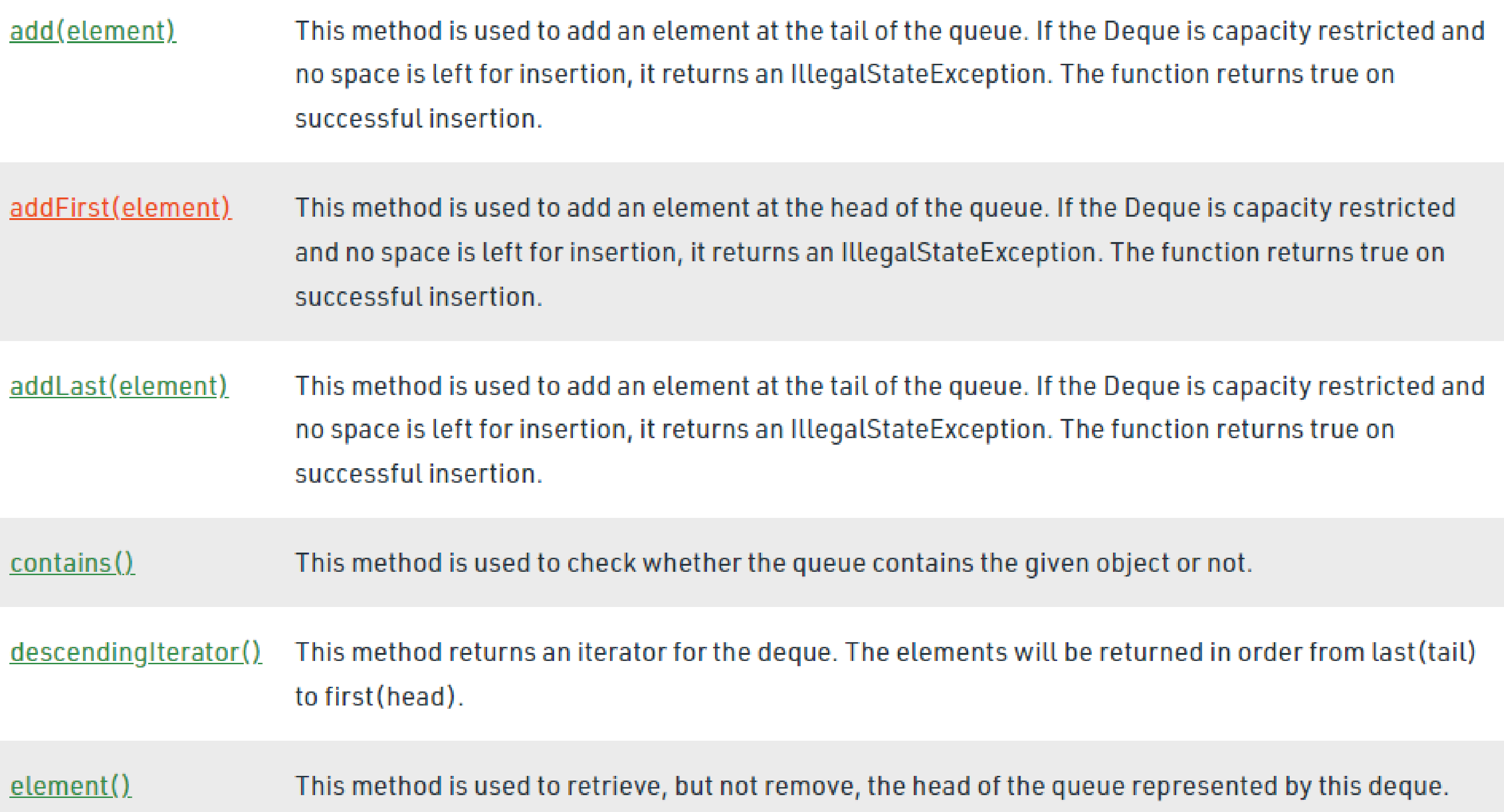
```
[291, 564, 24, 14]
```

Interface also provides us with the **poll(), pop(), pollFirst(), pollLast()** methods where **pop()** is used to remove and return the head of the deque. However, poll() is used because this offers the same functionality as **pop()** and doesnt return an exception when the deque is empty.

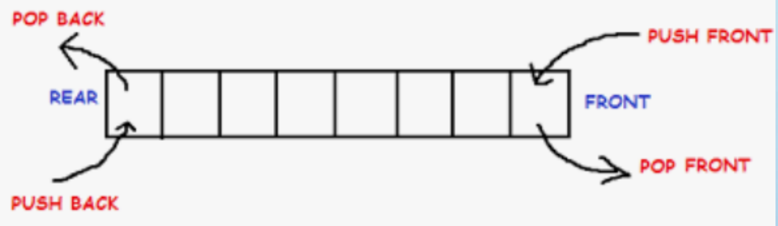
By Ayala Berkovich

JAVA COLLECTIONS

Deque functions:



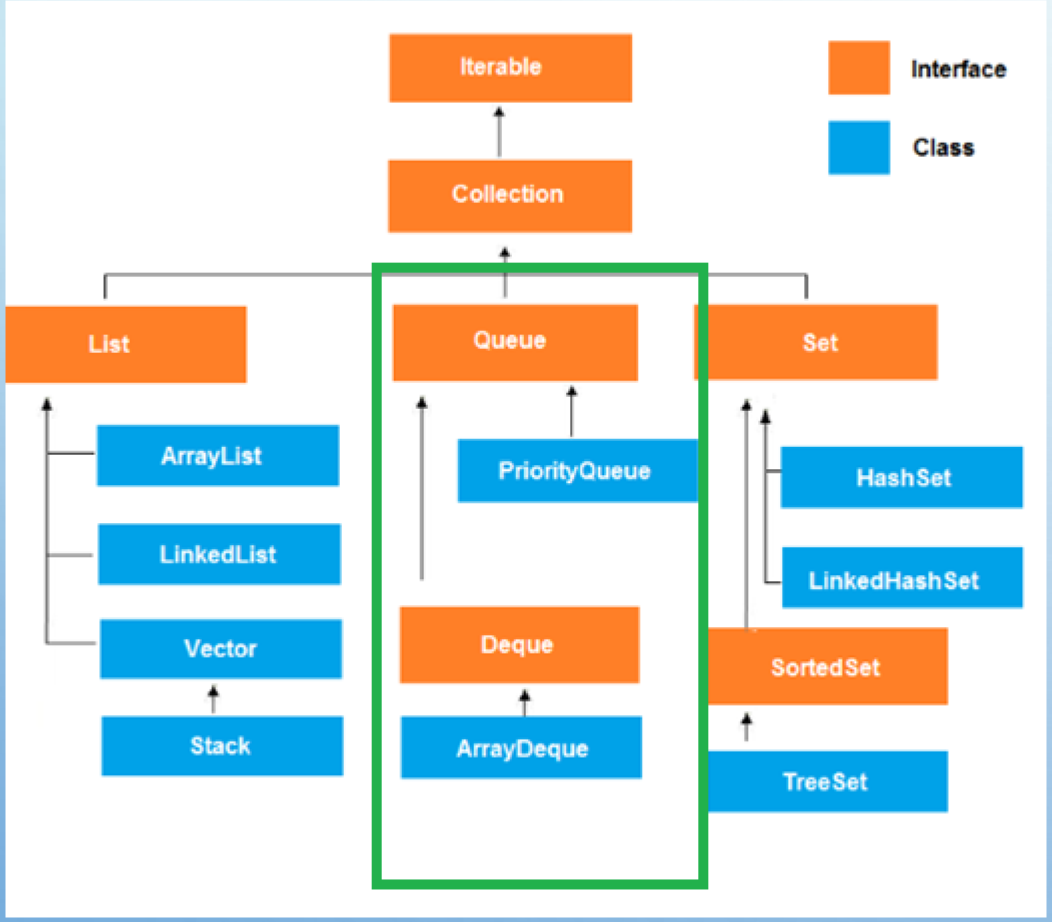
add(element)	This method is used to add an element at the tail of the queue. If the Deque is capacity restricted and no space is left for insertion, it returns an IllegalStateException. The function returns true on successful insertion.
addFirst(element)	This method is used to add an element at the head of the queue. If the Deque is capacity restricted and no space is left for insertion, it returns an IllegalStateException. The function returns true on successful insertion.
addLast(element)	This method is used to add an element at the tail of the queue. If the Deque is capacity restricted and no space is left for insertion, it returns an IllegalStateException. The function returns true on successful insertion.
contains()	This method is used to check whether the queue contains the given object or not.
descendingIterator()	This method returns an iterator for the deque. The elements will be returned in order from last(tail) to first(head).
element()	This method is used to retrieve, but not remove, the head of the queue represented by this deque.



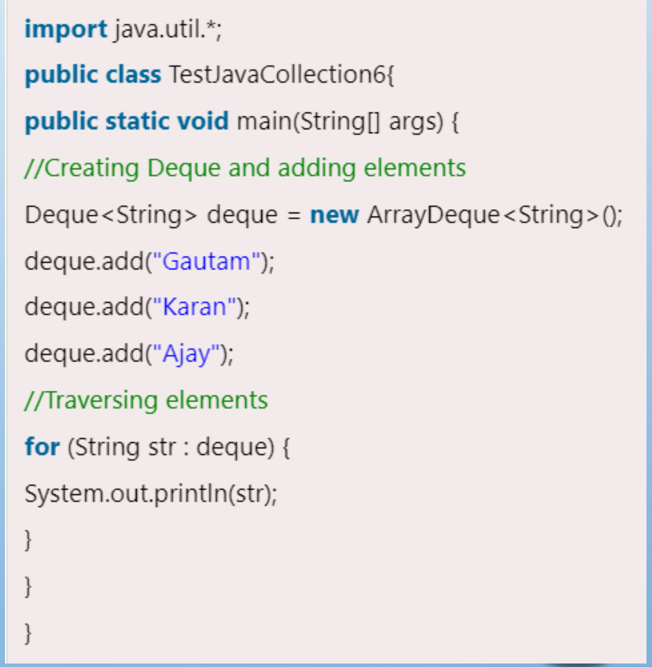
…

By Ayala Berkovich

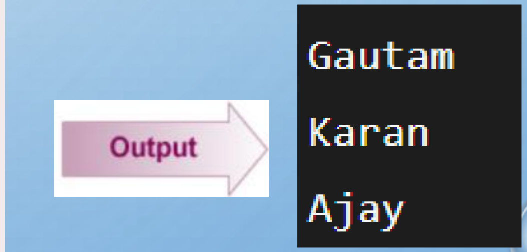
JAVA COLLECTIONS



ArrayDeque class implements the **Deque** interface.
It facilitates us to use the Deque so we can add or delete the elements from both the ends.
ArrayDeque is faster than **ArrayList** and **Stack**.
Consider the following example.



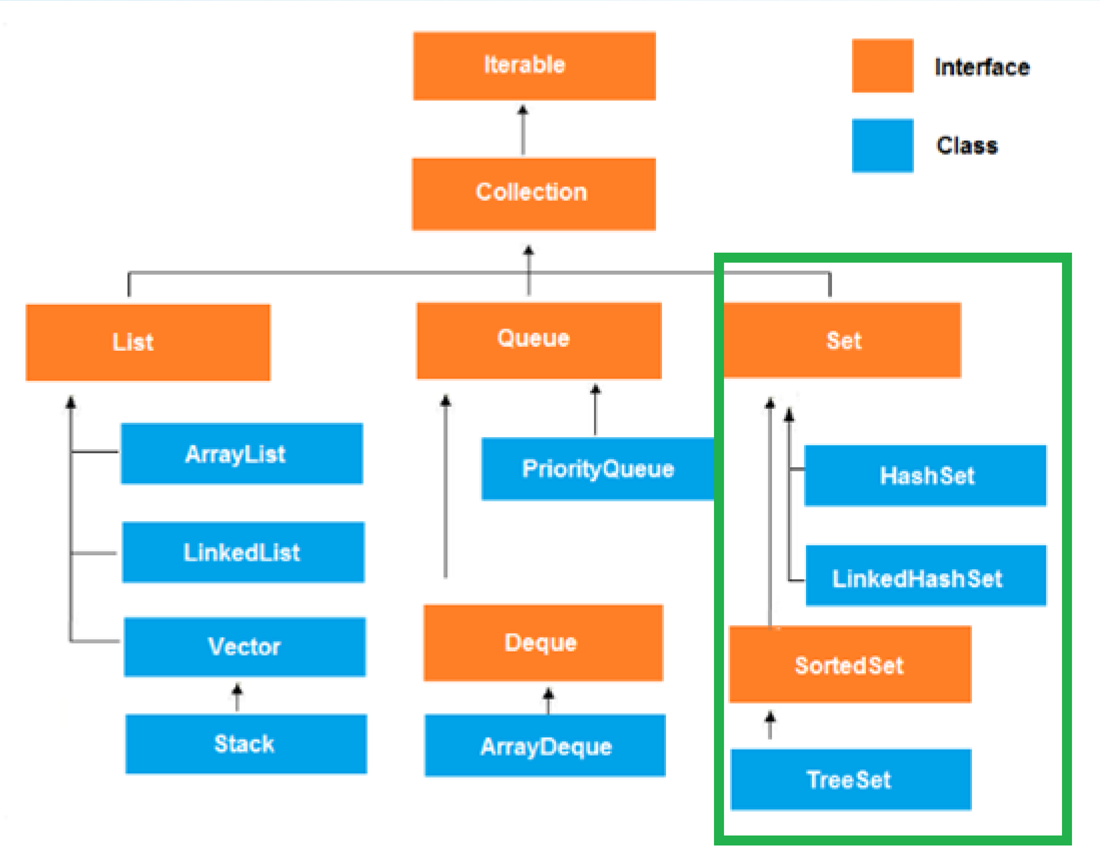
```java
import java.util.*;
public class TestJavaCollection6{
public static void main(String[] args) {
//Creating Deque and adding elements
Deque<String> deque = new ArrayDeque<String>();
deque.add("Gautam");
deque.add("Karan");
deque.add("Ajay");
//Traversing elements
for (String str : deque) {
System.out.println(str);
}
}
}
```



```
Gautam
Karan
Ajay
```

By Ayala Berkovich

JAVA COLLECTIONS



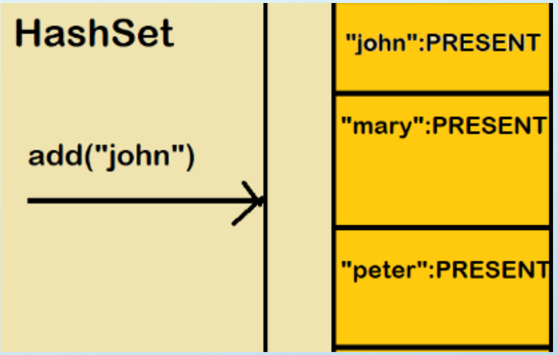
Set Interface in Java is present in **java.util** package.
It extends the **Collection interface**.
It represents the **unordered set of elements which doesn't allow us to store the duplicate items**.
We can store at most one null value in Set.
Set is implemented by **HashSet**, **LinkedHashSet**, and **TreeSet**.

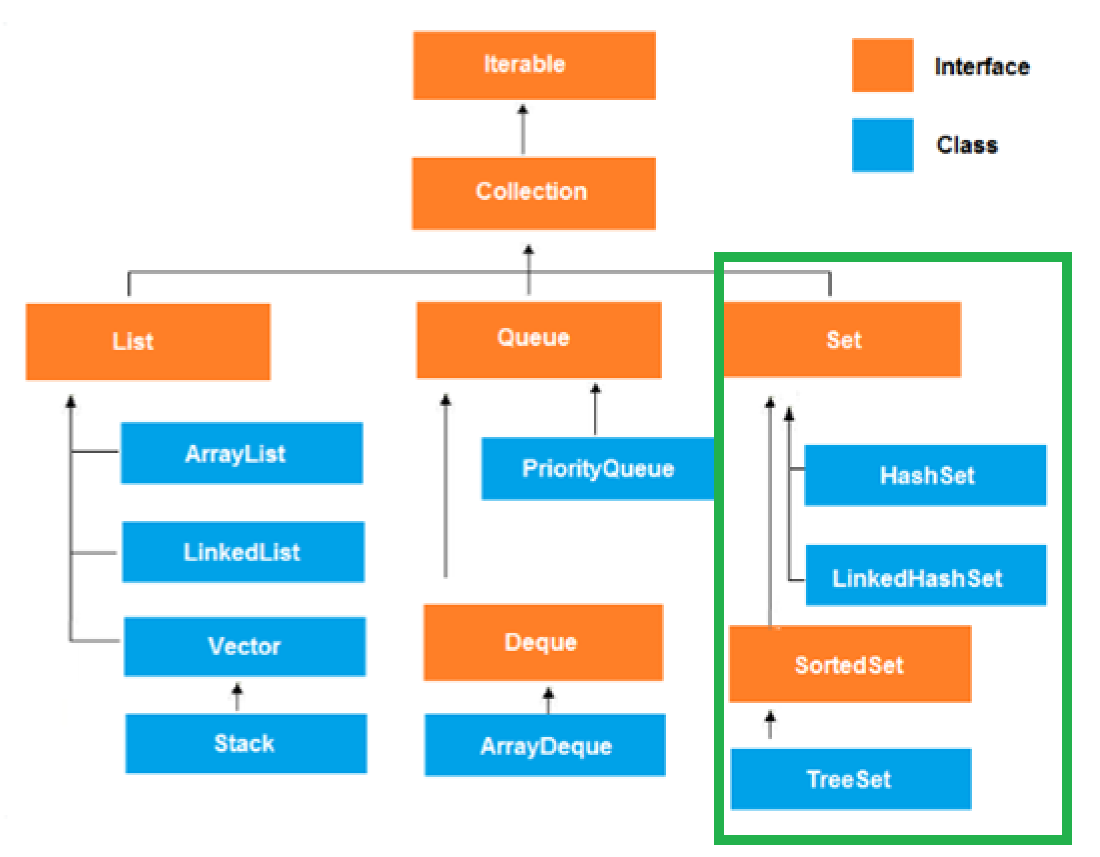
1.Set<data-type> s1 = **new** HashSet<data-type>();
2.Set<data-type> s2 = **new** LinkedHashSet<data-type>();
3.Set<data-type> s3 = **new** TreeSet<data-type>();

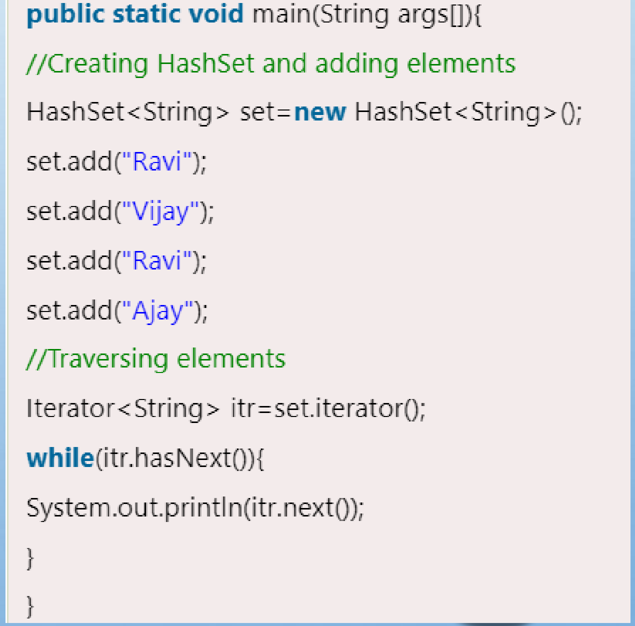
By Ayala Berkovich

JAVA COLLECTIONS

HashSet class implements **Set Interface**.
It represents the collection that uses a **hash table for storage**.
It contains **unique** items. **HashSet** allows only one null key.

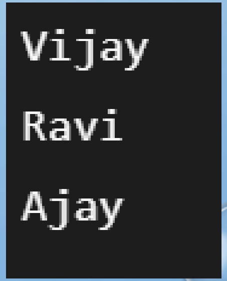






```
public static void main(String args[]){
//Creating HashSet and adding elements
HashSet<String> set=new HashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//Traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
```

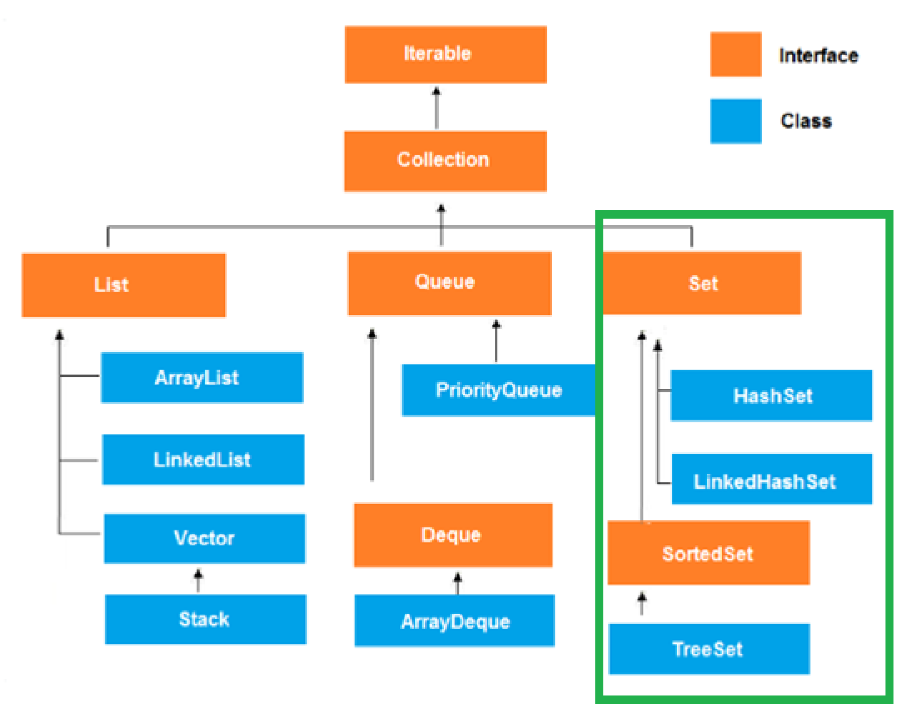
Output



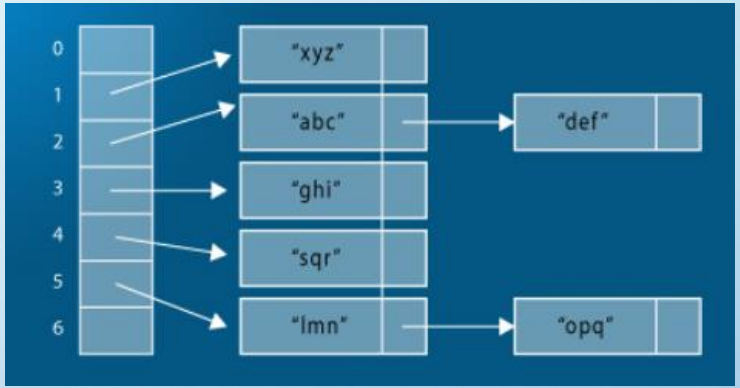
```
Vijay
Ravi
Ajay
```

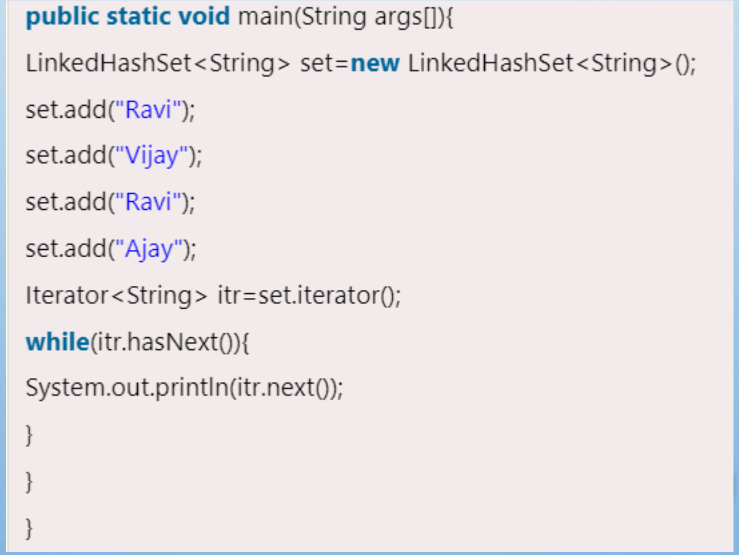
By Ayala Berkovich

JAVA COLLECTIONS



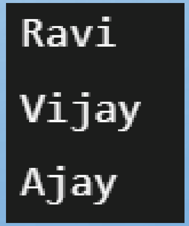
LinkedHashSet class represents **the LinkedList** implementation of **Set** Interface.
It extends the HashSet class and implements Set interface.
Like **HashSet**, It also contains unique elements





```
public static void main(String args[]){
LinkedHashSet<String> set=new LinkedHashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

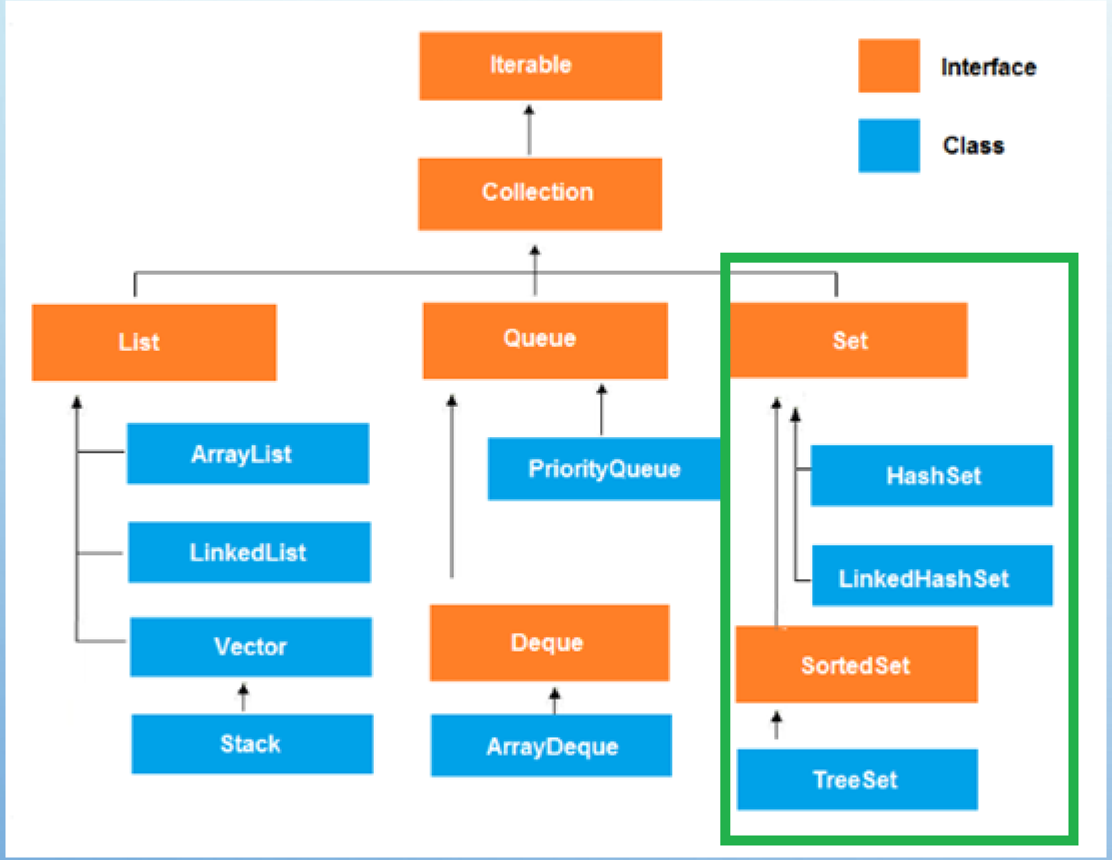
Output



```
Ravi
Vijay
Ajay
```

By Ayala Berkovich

JAVA COLLECTIONS

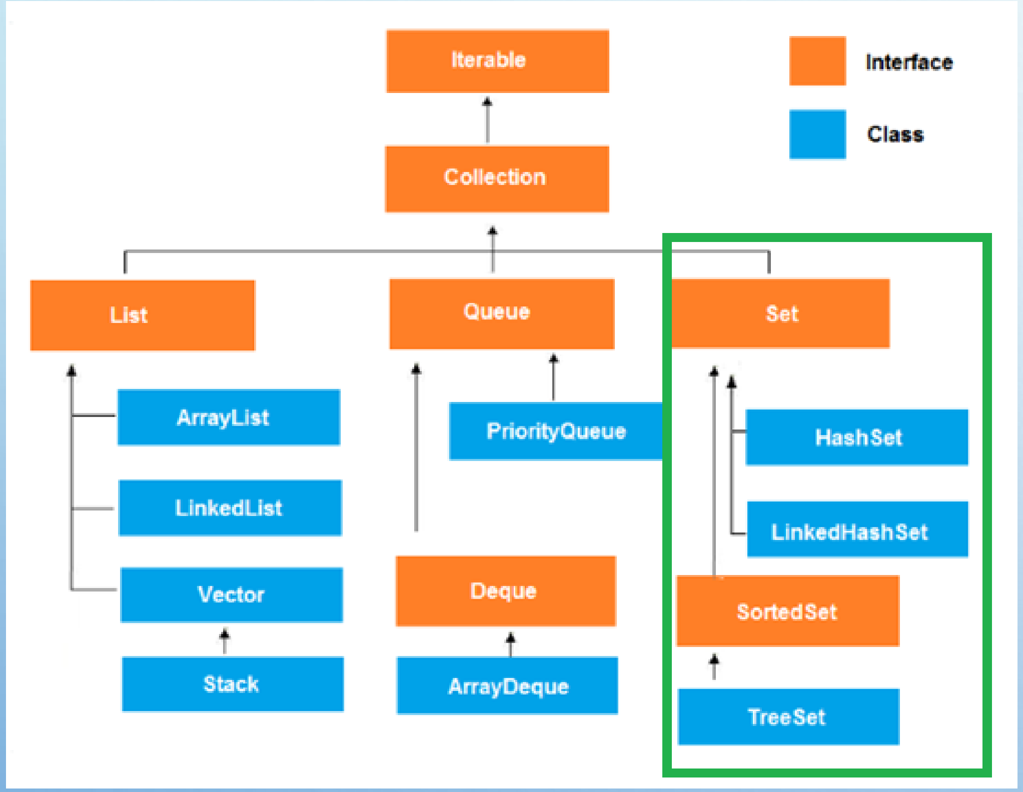


SortedSet is the alternate of **Set** interface that provides a total ordering on its elements.
The elements of the SortedSet are arranged in the increasing (ascending) order.
The SortedSet provides the additional methods that inhibit the natural ordering of the elements.
The SortedSet can be instantiated as:

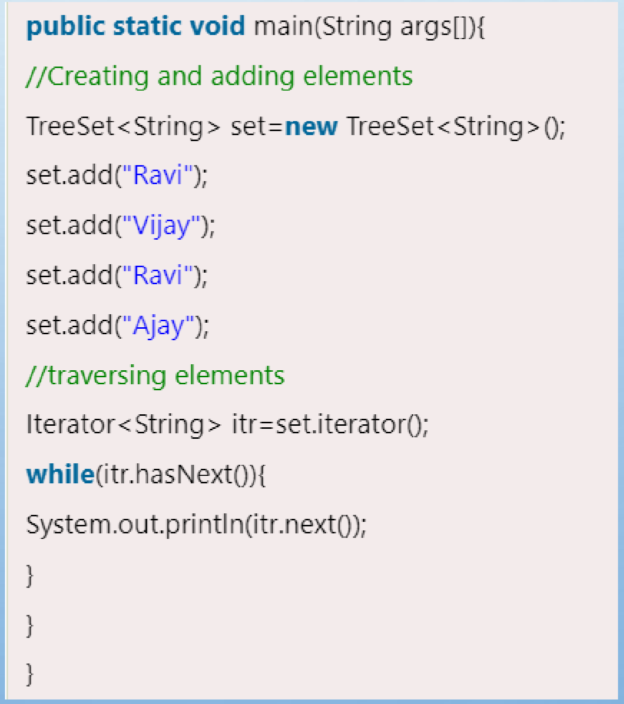
```
SortedSet<data-type> set = new TreeSet();
```

By Ayala Berkovich

JAVA COLLECTIONS

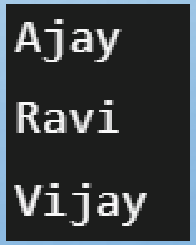


Java **TreeSet** class implements the **Set** interface that uses a tree for storage.
Like **HashSet**, **TreeSet** also contains **unique elements**.
Access and retrieval time of **TreeSet** is quite fast.
The elements in **TreeSet** stored in **ascending order(from little to big or by alef-bet order).**



```
public static void main(String args[]){
//Creating and adding elements
TreeSet<String> set=new TreeSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output



```
Ajay
Ravi
Vijay
```

JAVA COLLECTIONS

Build Algorithm and write code to check string to valid value of parenthesis:{}[]()

For example: if expression: **(1+{3+[4+5]+6}+7)** is valid?

JAVA COLLECTIONS

Build Algorithm and write code to:

Lottery, how to make a lottery of numbers from 1 to 100 that will never return a number that has already been?

By Ayala Berkovich