

Collections

1

Collections in Java

- Is a framework that provides an architecture to store and manipulate the group of objects.
- Operations performed on a data such as searching, sorting, insertion, deletion etc. can be performed by Java Collections
- Provides many interfaces and classes

Collection Interface

3

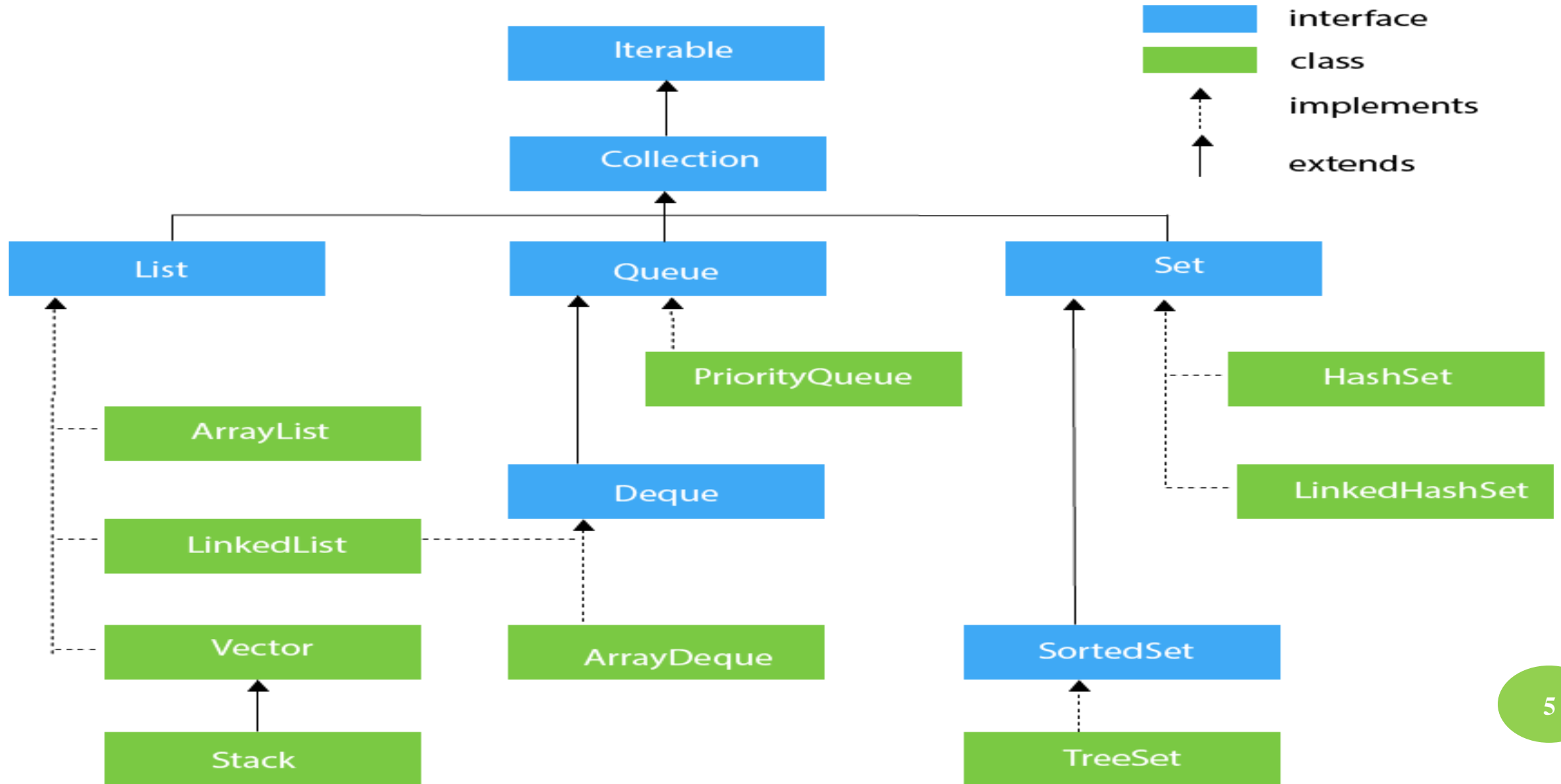
Collection Interface

- Collection interface extends the Iterable interface
- must be implemented by any class that defines a collection.
- general declaration is

interface Collection < E >

E specifies the type of object that collection will hold

Hierarchy Of Collection Framework



Commonly used methods interface

Methods	int size()
boolean add(E obj)	void clear()
boolean addAll(Collection C)	Object[] toArray()
boolean remove(Object obj)	boolean retainAll(Collection c)
boolean removeAll(Collection C)	Iterator iterator()
boolean contains(Object obj)	boolean equals(Object obj)
boolean isEmpty()	

The List Interface

- Extends the **Collection** Interface,
- Defines storage as sequence of elements.
- Following is its general declaration

interface List < E >

- Allows random access and insertion based on position.
- Allows Duplicate elements.

Method defined by List Interface

Methods	Description
Object get(int index)	Returns object stored at the specified index
Object set(int index, E obj)	Stores object at the specified index in the calling collection
int indexOf(Object obj)	Returns index of first occurrence of obj in the collection
int lastIndexOf(Object obj)	Returns index of last occurrence of obj in the collection
List subList(int start, int end)	Returns a list containing elements between start and end index in the collection

Contd..

Methods	Description
<code>void add(int index, E obj)</code>	Inserts obj into the invoking list at the index passed in index
<code>boolean addAll(int index, Collection<E> c)</code>	Inserts all elements of c into the invoking list at the index passed in index

The Set Interface

- Extends **Collection** interface
- Doesn't allow insertion of duplicate elements.
- general declaration is,

interface Set < E >

- doesn't define any method of its own.
- has two sub interfaces,

1. **SortedSet**
2. **NavigableSet.**

1. SortedSet

- interface extends **Set** interface
- Arranges added elements in an ascending order.

2. NavigableSet.

- interface extends **SortedSet** interface,
- allows retrieval of elements based on the closest match to a given value or values.

The Methods Defined by SortedSet

Method	Description
<code>Comparator<? super E> comparator()</code>	Returns the invoking sorted set's comparator. If the natural ordering is used for this set, <code>null</code> is returned.
<code>E first()</code>	Returns the first element in the invoking sorted set.
<code>SortedSet<E> headSet(E end)</code>	Returns a SortedSet containing those elements less than <i>end</i> that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set.
<code>E last()</code>	Returns the last element in the invoking sorted set.
<code>SortedSet<E> subSet(E start, E end)</code>	Returns a SortedSet that includes those elements between <i>start</i> and <i>end</i> -1. Elements in the returned collection are also referenced by the invoking object.
<code>SortedSet<E> tailSet(E start)</code>	Returns a SortedSet that contains those elements greater than or equal to <i>start</i> that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.

The Methods Defined by NavigableSet

Method	Description
<code>E ceiling(E obj)</code>	Searches the set for the smallest element e such that $e \geq obj$. If such an element is found, it is returned. Otherwise, null is returned.
<code>Iterator<E> descendingIterator()</code>	Returns an iterator that moves from the greatest to least. In other words, it returns a reverse iterator.
<code>NavigableSet<E> descendingSet()</code>	Returns a NavigableSet that is the reverse of the invoking set. The resulting set is backed by the invoking set.
<code>E floor(E obj)</code>	Searches the set for the largest element e such that $e \leq obj$. If such an element is found, it is returned. Otherwise, null is returned.

E higher(E obj)	Searches the set for the largest element e such that $e > obj$. If such an element is found, it is returned. Otherwise, null is returned.
E lower(E obj)	Searches the set for the largest element e such that $e < obj$. If such an element is found, it is returned. Otherwise, null is returned.
E pollFirst()	Returns the first element, removing the element in the process. Because the set is sorted, this is the element with the least value. null is returned if the set is empty.
E pollLast()	Returns the last element, removing the element in the process. Because the set is sorted, this is the element with the greatest value. null is returned if the set is empty.

Queue Interface

- Extends **collection** interface
- Defines behaviour of queue, that is first-in, first-out.
- Queue is an ordered of the homogeneous group of elements in which new elements are added at one end(rear) and elements are removed from the other end(front).
- General declaration is,

interface Queue< E >

Methods defined by Queue

Methods	Description
Object poll()	removes element at the head of the queue and returns null if queue is empty
Object remove()	removes element at the head of the queue and throws NoSuchElementException if queue is empty
Object peek()	returns the element at the head of the queue without removing it. Returns null if queue is empty
Object element()	same as peek(), but throws NoSuchElementException if queue is empty
boolean offer(E obj)	Adds object to queue.

Deque Interface

- Extends **Queue** interface
- Implements behaviour of a double-ended queue.
- Collection of elements in which elements can be inserted and removed from either end. i.e, it supports insertion and removal at both ends of an object of a class that implements it.
- General declaration is:

interface Deque< E >

Method Defined by Deque

- ❖ void addFirst(E obj)
- ❖ void addLast(E obj)
- ❖ Iterator<E> descendingIterator()
- ❖ E getFirst()
- ❖ E getLast()
- ❖ boolean offerFirst(E obj)
- ❖ boolean offerLast(E obj).
- ❖ E peekFirst()
- ❖ E peekLast()
- ❖ E pollFirst()
- ❖ E pollLast()
- ❖ E pop()
- ❖ E removeFirst()
- ❖ Boolean removeFirstOccurrence(Object obj)
- ❖ E removeLast()
- ❖ Boolean removeLastOccurrence(Object obj)

The Collections classes

Collections Classes

- ❖ Collections classes implement the collection interfaces.
- ❖ Defined in `java.util` package

The Collection classes

❖ AbstractCollection

❖ AbstractList

❖ AbstractQueue

❖ AbstractSequentialList

❖ LinkedList

❖ ArrayList

❖ ArrayDeque

❖ AbstractSet

❖ EnumSet

❖ HashSet

❖ LinkedHashSet

❖ PriorityQueue

❖ TreeSet

List

- ArrayList Class
- LinkedList class

ArrayList Class

- Extends AbstractList class
- Implements the List interface.
- ArrayList has this declaration:

class ArrayList<E>

- **ArrayList has three constructors**
 1. ArrayList()
 2. ArrayList(Collection C)
 3. ArrayList(int capacity)

Contd...

- Created with an initial capacity of 10. Once ArrayList is reached its maximum capacity, a new ArrayList is created with

$$\text{new capacity} = (\text{current capacity} * 3/2) + 1 = 10 * 3/2 + 1 = 16.$$

- Contain Duplicate elements
- Maintains the insertion order.
- Allows random access because it works on the index basis

Example to show collections are heterogeneous

```
public class AddEx
```

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

```
List al=new ArrayList(); //Here is no use of generic. No type safety. So, we can add both integer and string
```

```
al.add(10);
```

```
al.add(20);
```

```
al.add(30);
```

```
al.add(40);
```

```
al.add("abcd");
```

```
al.add(4, 35);
```

```
al.add(5, 45);
```

```
System.out.println("Elements after adding :-"+al); } }
```

Elements after adding :-

[10, 20, 30, 40, 35, 45, abcd]

Example for null insertion

```
ArrayList al=new ArrayList();
```

```
al.add("A");
```

```
al.add("B");
```

```
al.add(20);
```

```
al.add("A");
```

```
al.add(null);
```

```
System.out.println(al);
```

Output:

[A, B, 20, A, null]

Example of ArrayList

```
class Demo {  
    public static void main(String args[])  
    {  
        ArrayList< String> al = new ArrayList< String>();  
        al.add("abhilash");  
        al.add("adharsh");  
        al.add("avinash");  
        system.out.println(al);  
    } }  
}
```

Output:

abhilash

adharsh

avinash

```
class ArrayListDemo {  
    public static void main(String args[]) {  
  
        ArrayList<String> al = new ArrayList<String>();  
        System.out.println("Initial size of al: " +  
            al.size());  
  
        al.add("C");  
        al.add("A");  
        al.add("E");  
        al.add("B");  
        al.add("D");  
        al.add("F");  
        al.add(1, "A2");  
        al.add("F");  
        System.out.println("Size of al after additions: " +  
            al.size());  
        System.out.println("Contents of al: " + al);  
  
        al.remove("F");  
        al.remove(2);
```

```
        System.out.println("Size of al after deletions: " +  
            al.size());  
        System.out.println("Contents of al: " + al);  
        al.clear();  
        System.out.println("Contents of al: " + al);  
    }  
}
```

Initial size of al: 0

Size of al after additions: 8

Contents of al: [C, A2, A, E, B, D, F, F]

Size of al after deletions: 6

Contents of al: [C, A2, E, B, D, F]

Contents of al: []

Example of addAll method

```
ArrayList<String> al=new ArrayList<String>();
```

```
al.add("Ajay");
```

```
al.add("Hariharan");
```

```
al.add("Karthik");
```

```
ArrayList<String> al2=new ArrayList<String>();
```

```
al2.add("Amith");
```

```
al2.add("kiran");
```

```
al.addAll(al2);
```

```
System.out.println( al);
```

Ajay
Hariharan
Karthik
Amith
Kiran

Example of removeAll() method

```
ArrayList<String> al=new ArrayList<String>();
```

```
al.add("Ajay");
```

```
al.add("Karthik");
```

```
al.add("Rakshith");
```

```
ArrayList<String> al2=new ArrayList<String>();
```

```
al2.add("Subramanya");
```

```
al2.add("Ajay");
```

```
al.removeAll(al2);
```

```
System.out.println(al);
```

Output
Karthik
Rakshith

To find the Index of any particular element

```
ArrayList<String>=new ArrayList<>String;  
al.add("AA");  
al.add("BB");  
al.add("CC");  
al.add("DD");  
al.add("EE");  
al.add("FF");  
  
System.out.println("Index of CC: "+al.indexOf("CC"));  
System.out.println("Index of FF: "+al.indexOf("FF"));
```

Output:

Index of CC: 2

Index of FF: 5

Getting Array From An ArrayList

- **toArray()** method is used to get an array containing all the contents of the ArrayList

Example for obtaining array from an ArrayList

```
class ArrayListToArray {  
    public static void main(String args[]) {  
  
        ArrayList<Integer> al = new  
        ArrayList<Integer>();  
  
        al.add(1);  
  
        al.add(2);  
  
        al.add(3);  
  
        al.add(4);  
  
        System.out.println("Contents of al: " + al);  
  
        Integer ia[] = al.toArray (new  
        Integer[al.size()]);
```

```
        int sum = 0;  
        for(int i : ia)  
            sum += i;  
  
        System.out.println("Sum is: " + sum); } }
```

Contents of al: 1, 2, 3, 4
Sum is: 10

LinkedList class

- Extends AbstractSequentialList
- Implements List, Deque and Queue interface.
- LinkedList has two constructors.

1. LinkedList()

2. LinkedList(Collection C)

- Can be used as List, stack or Queue as it implements all the related interfaces
- Dynamic in nature i.e it allocates memory when required

Contd..

- Can contain duplicate elements
- Manipulation is fast because no shifting needs to be occurred.

Methods of Java LinkedList

- ❖ **void add(int index, Object element)** :It is used to insert the specified element at the specified position index in a list.
- ❖ **void addFirst(Object o)**: It is used to insert the given element at the beginning of a list.
- ❖ **void addLast(Object o)**:It is used to append the given element to the end of a list.
- ❖ **int size()**: It is used to return the number of elements in a list
- ❖ **boolean add(Object o)**: It is used to append the specified element to the end of a list.

- ❖ **boolean contains(Object o):** It is used to return true if the list contains a specified element.
- ❖ **boolean remove(Object o):** It is used to remove the first occurrence of the specified element in a list.
- ❖ **Object getFirst():** It is used to return the first element in a list.
- ❖ **Object getLast():** It is used to return the last element in a list.
- ❖ **int indexOf(Object o):** It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element.
- ❖ **int lastIndexOf(Object o):** It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element.

Example of LinkedList

```
LinkedList<String>l1=new  
LinkedList<String>();  
    l1.add("A");  
    l1.add("B");  
    l1.addLast("C");  
    l1.addFirst("D");  
    l1.add(2, "E");  
    l1.add("F");  
    l1.add("G");
```

```
System.out.println("Linked list : " + l1);  
l1.remove("B");  
l1.remove(3);  
l1.removeFirst();  
l1.removeLast();  
System.out.println("Linked list after deletion: " +  
l1);
```

Output:

Linked list : [D, A, E, B, C, F, G]

Linked list after deletion: [A, E, F]

Contd...

```
boolean status = ll.contains("E");  
if(status)  
    System.out.println("List contains the element 'E' ");  
else  
    System.out.println("List doesn't contain the element 'E'");  
int size = ll.size();  
System.out.println("Size of linked list = " + size);  
System.out.println("Element returned by get() : " + ll.get(2));  
ll.set(2, "Y");  
System.out.println("Linked list after change : " + ll);
```

Output:

List contains the element 'E'

Size of linked list = 3

Element returned by get() : F

Linked list after change : [A, E, Y]

When we go for List?

- Used when we want to allow or store duplicate elements.
- Can be used when we want to store null elements.
- When we want to preserve my insertion order, we should go for list.

HashSet class

- Extends **AbstractSet** class
- Implements the **Set** interface.
- Has three constructors.
 1. HashSet()
 2. HashSet(Collection C)
 3. HashSet(int capacity)
- HashSet stores the elements by using a mechanism called **hashing**..
- Contains only unique elements.

Contd...

- Does not maintain any order of elements
- HashSet allows null values however if you insert more than one nulls it would still return only one null value.

Example for HashSet

```
class Demo {  
    public static void main(String args[])  
    {  
        HashSet< String> hs = new HashSet<  
String>();  
        hs.add("B");  
        hs.add("A");  
        hs.add("D");
```

```
        hs.add("E");  
        hs.add("C");  
        hs.add("F");  
        hs.add("A");  
        System.out.println(hs);    } }
```

Output:

[D, E, F, A, B, C]

HashSet allows null values however if you insert more than one nulls it would still return only one null value.

```
HashSet<String> hset = new  
HashSet<String>();  
hset.add("Apple");  
hset.add("Mango");  
hset.add("Grapes");  
hset.add("Orange");  
hset.add("Fig");  
hset.add("Apple");  
hset.add("Mango");  
  
//Addition of null values
```

```
hset.add(null);  
hset.add(null);  
System.out.println(hset);  
} }
```

Output:
null
Mango
Grapes
Apple
Orange
Fig

LinkedHashSet Class

- Extends **HashSet** class
- Maintains a linked list of entries in the set.
- Stores elements in the order in which elements are inserted i.e it maintains the insertion order

Example for LinkedHashSet

```
class Demo {  
    public static void main(String args[]) {  
        LinkedHashSet< String> hs = new LinkedHashSet< String>();  
        hs.add("likith");  
        hs.add("dekshit");  
        hs.add("deepak");  
        hs.add("kusal");  
        hs.add("bagesh");  
        hs.add("asif");  
        System.out.println(hs); } }
```

Output:

likith
dekshit
deepak
kusal
bagesh
asif

Example for LinkedHashSet

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

```
// LinkedHashSet of String Type
```

```
LinkedHashSet<String> lhset = new  
LinkedHashSet<String>();
```

```
lhset.add("Z");  
lhset.add("PQ");  
lhset.add("N");  
lhset.add("O");  
lhset.add("KK");  
lhset.add("FGH");  
System.out.println(lhset);
```

```
// LinkedHashSet of Integer Type
```

```
LinkedHashSet<Integer> lhset2 = new  
LinkedHashSet<Integer>();
```

```
lhset2.add(99);  
lhset2.add(7);  
lhset2.add(0);  
lhset2.add(67);  
lhset2.add(89);  
lhset2.add(66);  
System.out.println(lhset2); } }
```

Output:

**[Z, PQ, N, O, KK, FGH]
[99, 7, 0, 67, 89, 66]**

TreeSet Class

- Extends **AbstractSet** class and implements the **NavigableSet** interface.
- Stores the elements in ascending order.
- Uses a Tree structure to store elements.
- Contains unique elements only like HashSet.
- Access and retrieval times are quite fast.
- Has four Constructors.
 1. **TreeSet()**
 2. **TreeSet(Collection *C*)**
 3. **TreeSet(Comparator *comp*)**
 4. **TreeSet(SortedSet *ss*)**

Example for TreeSet

```
class Demo {  
  
    public static void main(String args[]){  
  
        TreeSet<String> al=new TreeSet<String>();  
  
        al.add("sachin");  
  
        al.add("manoj");  
  
        al.add("sachin");  
  
        al.add("shashi");  
  
        System.out.println(al); } }
```

Output:

manoj, sachin, shashi

PriorityQueue Class

- Extends the **AbstractQueue** class, implements the Queue interface
- Provides the facility of using queue.
- PriorityQueue has six constructors.
 1. PriorityQueue()
 2. PriorityQueue(int capacity)
 3. PriorityQueue(int capacity, Comparator comp)
 4. PriorityQueue(Collection c)
 5. PriorityQueue(PriorityQueue c)
 6. PriorityQueue(SortedSet c)

Contd...

- Priority queue does not permit null elements
- Duplicate elements are *allowed* to be stored in a PriorityQueue.
- Capacity grows automatically as elements are added.
- Starting capacity is 11.



```
class TestCollection12{ public static void  
main(String args[]){  
PriorityQueue<String> queue=new  
PriorityQueue<String>();  
    queue.add("sachin");  
    queue.add("bagesh");  
    queue.add("chetan");  
    queue.add("karthik");  
    queue.add("arjun");  
    queue.add("advith");  
    queue.add("arya");  
    queue.add("manoj");
```

```
System.out.println("head:"+queue.element());  
System.out.println("head:"+queue.peek());  
System.out.println("iteratingthe queue elements:");  
System.out.println(queue);  
queue.remove();  
queue.poll();  
System.out.println("after removing two elements:");  
System.out.println(queue);} }
```

Output for PriorityQueue

head:advith

head:advith

Iterating the queue elements:

[advith, bagesh, arjun, manoj, karthik, chetan, arya, sachin]

after removing two elements:

[arya, bagesh, chetan, manoj, karthik, sachin]

ArrayDeque Class

- Extends **AbstractCollection** and implements the **Deque** interface.
- Creates a dynamic array and has no capacity restrictions.
- Defines the following constructors:
 1. `ArrayDeque()`
 2. `ArrayDeque(int size)`
 3. `ArrayDeque(Collection c)`
- Starting capacity is 16.

Contd..

ArrayDeque can be used as a **stack** (LIFO) as well as a **queue** (FIFO)

ArrayDeque is faster than the Stack class when used as a stack and faster than the LinkedList class when used as a queue.



Example for ArrayDeque Class As Stack

```
ArrayDeque<String> adq = new ArrayDeque<String>();
```

```
adq.push("A");
```

```
adq.push("B");
```

```
adq.push("D");
```

```
adq.push("E");
```

```
adq.push("F");
```

```
System.out.print("Popping the stack: ");
```

```
while(adq.peek() != null)
```

```
System.out.print(adq.pop() + " ");
```

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

Output:

Popping the stack:

F E D B A

ArrayDeque As Queue

```
ArrayDeque<String> arrayDeque = new ArrayDeque<String>();
```

```
    arrayDeque.offer("One");
```

```
    arrayDeque.offer("Two");
```

```
    arrayDeque.offer("Three");
```

```
    arrayDeque.offer("Four");
```

```
    arrayDeque.offer("Five");
```

```
    System.out.println(arrayDeque);
```

```
    System.out.println(arrayDeque.poll());
```

```
    System.out.println(arrayDeque.poll());
```

Output:

[One, Two, Three, Four,
Five]

One

Two



Accessing a Collection

Accessing a Collection

➤ Three possible ways to cycle through the elements of any collection.

1. Using Iterator interface
2. Using ListIterator interface
3. Using for-each loop

Iterator

- Used to traverse a collection object elements one by one.
- Is available since Java 1.2 Collection Framework.
- Is applicable for all Collection classes. So it is also known as Universal Java Cursor.
- It supports both READ and REMOVE Operations



Methods of Iterator Interface

No.	Method	Description
1	<code>public boolean hasNext()</code>	It returns true if iterator has more elements.
2	<code>public Object next()</code>	It returns the element and moves the cursor pointer to the next element.
3	<code>public void remove()</code>	It removes the last elements returned by the iterator. It is rarely used.

Steps to use an Iterator

- Obtain an iterator to the start of the collection by calling the collection's `iterator()` method.
- Set up a loop that makes a call to `hasNext()` method. Make the loop iterate as long as `hasNext()` method returns true.
- Within the loop, obtain each element by calling `next()` method

Accessing Elements Using Iterator

- Iterator Interface is used to traverse a list in forward direction
- Enabling to remove or modify the elements of the collection.
- Each collection classes provide **iterator()** method to return an iterator.

Example

```
class IteratorDemo
{
    public static void main(String args[])
    {
        ArrayList< String> ar = new ArrayList<
String>();
        ar.add("ab");
        ar.add("bc");
        ar.add("cd");
        ar.add("de");
        Iterator it = ar.iterator();
```

```
while(it.hasNext())
{
    System.out.println(it.next()+" ");
}
```

Output

ab
bc
cd
de

Limitations of Iterator

- ❖ Only forward direction iterating is possible.

ListIterator

- used to traverse a list in both **forward** and **backward** direction.
- Available to only those collections that implements the List Interface.

Methods of ListIterator

Method	Description
void add(E obj)	Inserts obj into the list in front of the element that will be returned by the next call to next() method.
boolean hasNext()	Returns true if there is a next element. Otherwise, returns false.
boolean hasPrevious()	Returns true if there is a previous element. Otherwise, returns false.
Object next()	Returns the next element. A NoSuchElementException is thrown if there is not a next element.
int nextIndex()	Returns the index of the next element. If there is not a next element, returns the size of the list.

Contd...

Object previous()	Returns the previous element. A NoSuchElementException is thrown if there is not a previous element.
int previousIndex()	Returns the index of the previous element. If there is not a previous element, returns -1.
void remove()	Removes the current element from the list. An IllegalStateException is thrown if remove() method is called before next() or previous() method is invoked.
void set(E obj)	Assigns obj to the current element. This is the element last returned by a call to either next() or previous() method

Example

```
ArrayList< String> ar = new ArrayList< String>();
```

```
ar.add("ab");
```

```
ar.add("bc");
```

```
ar.add("cd");
```

```
ar.add("de");
```

```
ListIterator<String> litr = ar.listIterator();
```

```
while(litr.hasNext())
```

```
{   System.out.print(litr.next()+" ");   }
```

```
while(litr.hasPrevious())
```

```
{   System.out.print(litr.previous()+" "); }
```

Output:

ab bc cd de

de cd bc ab

Limitations of ListIterator

- It is the most powerful iterator but it is only applicable for List implemented classes, so it is not a universal iterator.

Using for-each loop

- for-each version of for loop can also be used for traversing the elements of a collection.
- Only be used if we don't want to modify the contents of a collection and we don't want any **reverse** access.
- Cycle through any collection of object that implements Iterable interface

Example

```
class ForEachDemo {  
    public static void main(String args[])  
    {  
        LinkedList< String> ls = new LinkedList< String>();  
        ls.add("a");  
        ls.add("b");  
        ls.add("c");  
        ls.add("d");  
        for(String str : ls)  
        {  
            System.out.print(str+" ");  
        }  
    }  
}
```

Output:

a b c d

Storing User-Defined Classes in Collections

```
class Address {
String name;
String street;
String city;
String state;
String code;

Address(String n, String s, String c,
String st, String cd) {
name = n;
street = s;
city = c;
state = st;
code = cd;
}

public String toString() {
return name + "\n" + street + "\n" +
city + " " + state + " " + code;
} }
```

```
class MailList {
public static void main(String args[]) {
LinkedList<Address> ml = new
LinkedList<Address>();

ml.add(new Address("Sachin", "11
Malleshwaram",
"Urbana", "KA", "61801"));
ml.add(new Address("Sumanth",
"Hebbala", "Hydrabad", "AP", "61853"));
ml.add(new Address("Shreyas", "Kanchi",
"Chennai", "TA", "61820"));

for(Address element : ml)
System.out.println(element + "\n");
System.out.println();
} }
```

The output from the program is shown here

Sachin

11 Malleshwaram

Urbana KA 61801

Sumanth

1142 Hebbala

Hydrabad AP 61853

Shreyas

867 kanchi

Chennai TA 61820

Working with Maps

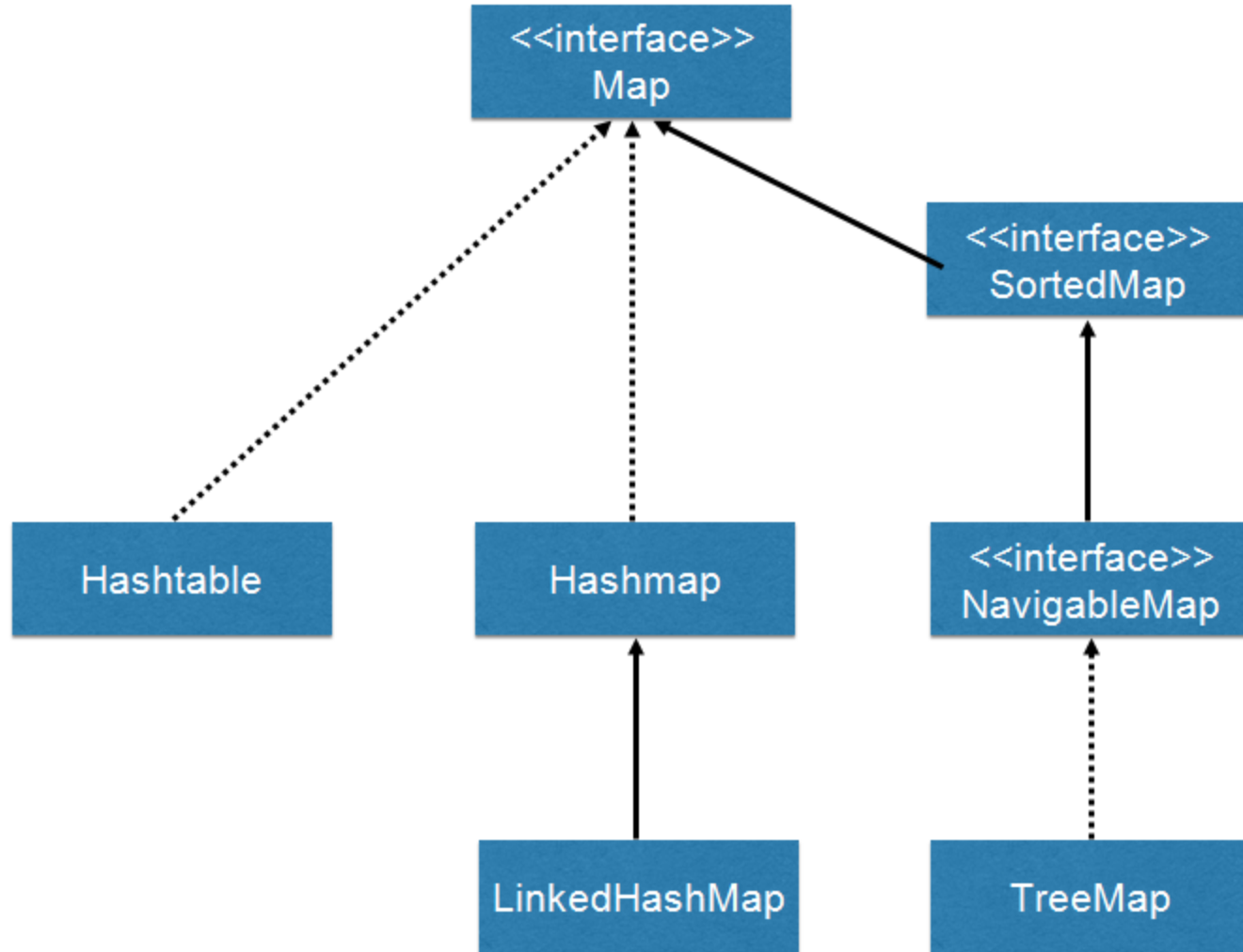
Map Interface

- Stores data in key and value association.
- Both key and values are objects.
- Key must be unique but the values can be duplicate.
- May have one null key and multiple null values
- **Map** is declared as shown here:

interface Map<K, V>

Here, **K** specifies the type of keys, and **V** specifies the type of values.

Map Interface



.....→ implements
——→ extends

Methods Defined by Map

Method	Description
<code>void clear()</code>	Removes all key/value pairs from the invoking map.
<code>boolean containsKey(Object k)</code>	Returns true if the invoking map contains k as a key. Otherwise, returns false.
<code>boolean containsValue(Object v)</code>	Returns true if the map contains v as a value. Otherwise, returns false.
<code>Set<Map.Entry<K, V>> entrySet()</code>	Returns a Set that contains the entries in the map. The set contains objects of type Map.Entry. Thus, this method provides a set-view of the invoking map.
<code>boolean equals(Object obj)</code>	Returns true if obj is a Map and contains the same entries. Otherwise, returns false.
<code>V get(Object k)</code>	Returns the value associated with the key k. Returns null if the key is not found.
<code>int hashCode()</code>	Returns the hash code for the invoking map.

Methods Defined by Map

Method	Description
Set<K> keySet()	Returns a Set that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.
V put(K k, V v)	Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are k and v, respectively. Returns null if the key did not already exist. Otherwise, the previous value linked to the key is returned.
void putAll(Map<K, V> m)	Puts all the entries from m into this map.
V remove(Object k)	Removes the entry whose key equals k.
int size()	Returns the number of key/value pairs in the map.
Collection<V> values()	Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

SortedMap Interface

- **SortedMap** interface extends **Map**.
- Ensures that the entries are maintained in ascending order based on the keys.
- **SortedMap** is declared as shown here:

```
interface SortedMap<K, V>
```

Methods Defined by SortedMap

Method	Description
K firstKey()	Returns the first key in the invoking map.
SortedMap<K, V> headMap(K end)	Returns a sorted map for those map entries with keys that are less than end.
K lastKey()	Returns the last key in the invoking map.
SortedMap<K, V> subMap(K start, K end)	Returns a map containing those entries with keys that are greater than or equal to start and less than end.
SortedMap<K, V> tailMap(K start)	Returns a map containing those entries with keys that are greater than or equal to start.

NavigableMap Interface

- Extends **SortedMap**
- Behavior of a map that supports the retrieval of entries based on the closest match to a given key or keys.
- General declaration:

interface NavigableMap<K,V>

Methods defined by NavigableMap

Method	Description
Map.Entry<K,V> ceilingEntry(K obj)	Searches the map for the smallest key k such that $k \leq \text{obj}$. If such a key is found, its entry is returned. Otherwise, null is returned.
NavigableSet<K> descendingKeySet()	Returns a NavigableSet that contains the keys in the invoking map in reverse order. Thus, it returns a reverse set-view of the keys. The resulting set is backed by the map.
K higherKey(K obj)	Searches the set for the largest key k such that $k > \text{obj}$. If such a key is found, it is returned. Otherwise, null is returned.
K lowerKey(K obj)	Searches the set for the largest key k such that $k < \text{obj}$. If such a key is found, it is returned. Otherwise, null is returned.
Map.Entry<K,V> pollFirstEntry()	Returns the first entry, removing the entry in the process. Because the map is sorted, this is the entry with the least key value. null is returned if the map is empty.
Map.Entry<K,V> pollLastEntry()	Returns the last entry, removing the entry in the process. Because the map is sorted, this is the entry with the greatest key value. null is returned if the map is empty.

Map.Entry Interface

- Enables to work with a map entry
- **Map.Entry** is declared like this:

```
interface Map.Entry<K, V>
```

Methods Defined by Map.Entry

Method	Description
boolean equals(Object obj)	Returns true if obj is a Map.Entry whose key and value are equal to that of the invoking object.
K getKey()	Returns the key for this map entry.
V getValue()	Returns the value for this map entry.
int hashCode()	Returns the hash code for this map entry.
V setValue(V v)	Sets the value for this map entry to v. A ClassCastException is thrown if v is not the correct type for the map. An IllegalArgumentException is thrown if there is a problem with v. A NullPointerException is thrown if v is null and the map does not permit null keys. An UnsupportedOperationException is thrown if the map cannot be changed.

Map Classes

HashMap class

- Extends **AbstractMap** and implements **Map** interface.
- Uses a **hashtable** to store the map.
- Allows the execution time of `get()` and `put()` to remain same
- Does not maintain order of its element.
- HashMap has four constructor.
 1. `HashMap()`
 2. `HashMap(Map m)`
 3. `HashMap(int capacity)`
 4. `HashMap(int capacity, float fillratio)`

Example

```
class HashMapDemo {  
    public static void main(String args[])  
    {  
        HashMap< String, Integer> hm = new  
        HashMap< String, Integer>();  
        hm.put("a", 100);  
        hm.put("b", 200);  
        hm.put("c", 300);  
        hm.put("d", 400);  
        Set< Map.Entry< String, Integer> > st =
```

```
hm.entrySet();  
    for(Map.Entry<String,Integer> me:st)  
    {  
        System.out.print(me.getKey()+":");  
        System.out.println(me.getValue());    } }
```

output

b 200

c 300

a 100

d 400

```
HashMap<String, Integer> map = new HashMap<>();

    print(map);
    map.put("vishal", 10);
    map.put("sachin", 30);
    map.put("vaibhav", 20);

    System.out.println("Size of map is:- " + map.size());

    System.out.println(map);
    if (map.containsKey("vishal"))
    {
Integer a = map.get("vishal");
System.out.println("value for key \"vishal\" is:- " + a);
    }

    map.clear();
    System.out.println(map);
}

{
```

```
        if (map.isEmpty())
        {
            System.out.println("map is empty");
        }
    else
    {
        System.out.println(map);
    }
}
```



Output:

map is empty Size of map is:- 3

{ vaibhav=20, vishal=10, sachin=30 }

value for key "vishal" is:- 10

map is empty



HashMap example to remove() elements

```
HashMap<Integer,String> map=new HashMap<Integer,  
String>();  
  
    map.put(100,"Amit");  
  
    map.put(101,"Vijay");  
  
    map.put(102,"Rahul");  
  
    map.put(103, "Gaurav");  
  
System.out.println("Initial list of elements: "+map);  
  
//key-based removal  
  
map.remove(100);  
  
System.out.println("Updated list of elements: "+map);
```

```
map.remove(101);  
  
System.out.println("Updated list of elements: "+map);  
  
//key-value pair based removal  
  
map.remove(102, "Rahul");  
  
System.out.println("Updated list of elements: "+map);
```

Initial list of elements: { 100=Amit, 101=Vijay, 102=Rahul, 103=Gaurav }

Updated list of elements: { 101=Vijay, 102=Rahul, 103=Gaurav }

Updated list of elements: { 102=Rahul, 103=Gaurav }

Updated list of elements: { 103=Gaurav }



TreeMap Class

- Extends AbstractMap and implements the NavigableMap interface.
- It creates maps stored in a tree structure.
- A TreeMap provides an efficient means of storing key/value pairs in sorted order and allows rapid retrieval.
- TreeMap is a generic class

➤ **class TreeMap<K, V>**

- The following **TreeMap** constructors are defined:

1. `TreeMap()`
2. `TreeMap(Comparator comp)`
3. `TreeMap(Map m)`
4. `TreeMap(SortedMap sm)`

Contd....

- Java TreeMap contains only unique elements.
- Java TreeMap cannot have a null key but can have multiple null values.
- Java TreeMap maintains ascending order



Example

```
class Demo {  
    public static void main(String args[]) {  
  
        TreeMap< String,Integer> tm = new TreeMap<  
String,Integer>();  
  
        tm.put("a", 100);  
        tm.put("b", 200);  
        tm.put("c", 300);  
        tm.put("d", 400);  
        Set< Map.Entry < String,Integer> > st =  
tm.entrySet();
```

```
for(Map.Entry me:st)  
{  
    System.out.print(me.getKey()+":");  
    System.out.println(me.getValue()); }  
} }
```


Output

a 100

b 200

c 300

d 400

Java TreeMap Example: SortedMap

```
SortedMap<Integer,String> map=new TreeMap<Integer,String>();
```

```
map.put(100,"Amit");
```

```
map.put(102,"Ravi");
```

```
map.put(101,"Vijay");
```

```
map.put(103,"Rahul");
```

```
//Returns key-value pairs whose keys are less than the specified key.
```

```
System.out.println("headMap: "+map.headMap(102));
```

```
//Returns key-value pairs whose keys are greater than or equal to the specified key.
```

```
System.out.println("tailMap: "+map.tailMap(102));
```

```
//Returns key-value pairs exists in between the specified key.
```

```
System.out.println("subMap: "+map.subMap(100, 102));
```

headMap: { 100=Amit, 101=Vijay }

tailMap: { 102=Ravi, 103=Rahul }

subMap: { 100=Amit, 101=Vijay }



LinkedHashMap Class

- Extends HashMap class.
- Maintains a order of insertion.
- Defines the following constructor
 1. LinkedHashMap()
 2. LinkedHashMap(Map< ? extends k, ? extends V> m)
 3. LinkedHashMap(int capacity)
 4. LinkedHashMap(int capacity, float fillratio)
 5. LinkedHashMap(int capacity, float fillratio, boolean order)

```
LinkedHashMap<Integer, String> map = new LinkedHashMap<Integer, String>();
```

```
map.put(100,"Amit");
```

```
map.put(101,"Vijay");
```

```
map.put(102,"Rahul");
```

```
//Fetching key
```

```
System.out.println("Keys: "+map.keySet());
```

```
//Fetching value
```

```
System.out.println("Values: "+map.values());
```

```
//Fetching key-value pair
```

```
System.out.println("Key-Value pairs: "+map.entrySet());
```

Keys: [100, 101, 102]

Values: [Amit, Vijay, Rahul]

Key-Value pairs: [100=Amit, 101=Vijay, 102=Rahul]



IdentityHashMap Class

- Extends **AbstractMap** implements the **Map** interface
- IdentityHashMap has this declaration:

```
class IdentityHashMap<K, V>
```

EnumMap Class

- Extends **AbstractMap** and implements **Map**.
- It is specifically for use with keys of an **enum** type.
- The declaration:

```
class EnumMap<K extends Enum<K>, V>
```

Comparators

Comparator Interface

- Java Comparator interface is used to order the objects of user-defined class.
- Gives the ability to decide how elements will be sorted and stored within collection and map.
- Comparator is a generic interface that has this declaration:

interface Comparator<T>

- The **Comparator** interface defines two methods: **compare()** and **equals()**.

Contd..,

- Comparator Interface defines **compare()** method.

int compare(T *obj1*, T *obj2*)

- returns 0 if two objects are equal.
- returns a positive value if object1 is greater than object2. Otherwise a negative value is returned.
- can throw a **ClassCastException** if the type of object are not compatible for comparison.
- **equals()** method, shown here, tests whether an object equals the invoking comparator:

boolean equals(Object *obj*)


```
class MyComp implements Comparator<String>
{
    public int compare(String a, String b)
    {
        String aStr, bStr;
        aStr = a;
        bStr = b;

        // Reverse the comparison.
        return bStr.compareTo(aStr);
    } }
```

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

```
        TreeSet<String> ts = new
        TreeSet<String>(new MyComp());

        ts.add("C");

        ts.add("A");

        ts.add("B");

        ts.add("E");

        ts.add("F");

        ts.add("D");

        for(String element : ts)

            System.out.print(element + " ");

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

Output:
F E D C B A

Why Generic Collections?

- **Java Generics** programming is introduced in JDK 5 to deal with type-safe objects.
- Before generics, we can store any type of objects in collection i.e. **non-generic**.
- Now generics, forces the java programmer to **store specific type** of objects.

Advantage of Java Generics

- **Type-safety :** We can hold only a single type of objects in generics. It doesn't allow to store other objects.
- **Type casting is not required:** There is no need to typecast the object.
- **Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

Example that uses non-generics code

```
class OldStyle {  
    public static void main(String args[])  
    {  
        ArrayList list = new ArrayList();  
        list.add("one");  
        list.add("two");  
        list.add("three");  
        list.add("four");  
        Iterator itr = list.iterator();  
        while(itr.hasNext()) {  
            /*To retrieve an element, an explicit type cast is needed because the collection stores only  
            Object.*/  
  
            String str = (String) itr.next();  
            System.out.println(str + " is " + str.length() + " chars long.");  
        } } }
```

Example to demonstrate we can add any type of object in non generic

```
class Demo{  
  
    public static void main(string args[]){  
  
        ArrayList list = new ArrayList();  
        list.add("abc");  
        list.add(5);                //OK  
        for(Object obj : list)  
        {  
            //type casting leading to ClassCastException  
            at runtime  
            String str=(String) obj;  
            Systemout.println(str);  
        }  
    }  
}
```

**Above code compiles fine but throws
ClassCastException at runtime because we are
trying to cast Object in the list to String whereas
one of the element is of type Integer.**

Legacy Classes

- legacy classes and interface were redesign by JDK 5 to support Generics.
- Following are the legacy classes defined by **java.util** package
 - ❖ Dictionary
 - ❖ HashTable
 - ❖ Properties
 - ❖ Stack
 - ❖ Vector
 - ❖ one legacy interface called **Enumeration**

Enumeration interface

- Defines the methods by which we can *enumerate* (obtain one at a time) the elements in a collection of objects
- Used by several methods defined by the legacy classes (such as Vector and Properties)It has this declaration:

interface Enumeration<E>

- where E specifies the type of element being enumerated
- Enumeration specifies the following two methods:
 1. **boolean hasMoreElements()**
 2. **E nextElement()**

Vector Class

- Implements a dynamic array. It is similar to ArrayList
- Vector is declared like this:

class Vector<E>

- Vector constructors:
 1. Vector()
 2. Vector(int *size*)
 3. Vector(int *size*, int *incr*)
 4. Vector(Collection<? extends E> *c*)

Contd...

- Maintain insertion order
- default initial capacity is 10.
- vector doubles its size. i.e. the Vector size would remain 10 till 10 insertions and once we try to insert the 11th element It would become 20
- Allows null values



Method in vector

- **void addElement(Object element):** It inserts the element at the end of the Vector.
- **int capacity():** This method returns the current capacity of the vector.
- **int size():** It returns the current size of the vector.
- **void setSize(int size):** It changes the existing size with the specified size.
- **boolean contains(Object element):** This method checks whether the specified element is present in the Vector. If the element is been found it returns true else false.
- **boolean containsAll(Collection c):** It returns true if all the elements of collection c are present in the Vector.
- **Object elementAt(int index):** It returns the element present at the specified location in Vector.

- **Object firstElement():** It is used for getting the first element of the vector.
- **Object lastElement():** Returns the last element of the array.
- **Object get(int index):** Returns the element at the specified index.
- **boolean isEmpty():** This method returns true if Vector doesn't have any element.
- **boolean removeElement(Object element):** Removes the specified element from vector.
- **boolean removeAll(Collection c):** It Removes all those elements from vector which are present in the Collection c.
- **void setElementAt(Object element, int index):** It updates the element of specified index with the given element.

```
Vector<Integer> v = new Vector<Integer>(3, 2);
System.out.println("Initial size: " + v.size());
System.out.println("Initial capacity: " +
v.capacity());
v.addElement(1);
v.addElement(2);
v.addElement(3);
v.addElement(4);
System.out.println("Capacity after four additions: " +
v.capacity());
v.addElement(5);
System.out.println("Current capacity: " +
v.capacity());
v.addElement(6);
v.addElement(7);
System.out.println("Current capacity: " +
v.capacity());
v.addElement(9);
v.addElement(10);
System.out.println("Current capacity: " +
v.capacity());
v.addElement(11);
```

```
v.addElement(12);
System.out.println("First element: " + v.firstElement());
System.out.println("Last element: " + v.lastElement());
if(v.contains(3))
System.out.println("Vector contains 3.");

// Enumerate the elements in the vector.
Enumeration vEnum = v.elements();
System.out.println("\nElements in vector:");
while(vEnum.hasMoreElements())
System.out.print(vEnum.nextElement() + " ");
System.out.println();
} }
```

Output

Initial size: 0

Initial capacity: 3

Capacity after four additions: 5

Current capacity: 5

Current capacity: 7

Current capacity: 9

First element: 1

Last element: 12

Vector contains 3.

Elements in vector:


1 2 3 4 5 6 7 9 10 11 12

```
Vector<String> vec = new Vector<String>(4);
//Adding elements to a vector
vec.add("Tiger");
vec.add("Lion");
vec.add("Dog");
vec.add("Elephant");
//Check size and capacity
System.out.println("Size is: "+vec.size());
System.out.println("Default capacity is: “
+vec.capacity());
//Display Vector elements
System.out.println("Vector element is: "+vec);
vec.addElement("Rat");
vec.addElement("Cat");
vec.addElement("Deer");
//Again check size and capacity after two insertions

System.out.println("Size after addition: "+vec.size());
System.out.println("Capacity after addition is: "+ve
c.capacity());
//Display Vector elements again
```

```
System.out.println("Elements are: "+vec);
//Checking if Tiger is present or not in this vector

if(vec.contains("Tiger"))
{
    System.out.println("Tiger is present at the index “
+vec.indexOf("Tiger"));
}
else
{
    System.out.println("Tiger is not present in the list.");
}
//Get the first element
System.out.println("The first animal of the vector is = “
+vec.firstElement());
//Get the last element
System.out.println("The last animal of the vector is = “
+vec.lastElement());
```



Output

Size is: 4

Default capacity is: 4

Vector element is: [Tiger, Lion, Dog, Elephant]

Size after addition: 7 Capacity after addition is: 8

Elements are: [Tiger, Lion, Dog, Elephant, Rat, Cat, Deer]

Tiger is present at the index 0

The first animal of the vector is = Tiger

The last animal of the vector is = Deer



Stack

- **Stack** implements a standard last-in, first-out stack
- **Stack** was declared as shown here:

```
class Stack<E>
```

Here, **E** specifies the type of element stored in the stack

- Stack class allow to store Heterogeneous elements.
- Stack work on Last in First out (LIFO) manner.
- Stack allow to store duplicate values.
- Initial 10 memory location is create whenever object of stack is created and it is re-sizable.
- Stack also organizes the data in the form of cells like Vector.
- Stack is one of the sub-class of Vector.



Methods in Stack class

- Object push(*Object element*) : Pushes an element on the top of the stack.
- Object pop() : Removes and returns the top element of the stack. An 'EmptyStackException' exception is thrown if we call pop() when the invoking stack is empty.
- Object peek() : Returns the element on the top of the stack, but does not remove it.
- boolean empty() : It returns true if nothing is on the top of the stack. Else, returns false.
- int search(*Object element*) : It determines whether an object exists in the stack. If the element is found, it returns the position of the element from the top of the stack. Else, it returns -1.

```
class StackDemo {  
    static void showpush(Stack<Integer> st, int a) {  
        st.push(a);  
        System.out.println("push(" + a + ")");  
        System.out.println("stack: " + st);  
    }  
    static void showpop(Stack<Integer> st) {  
        System.out.print("pop -> ");  
        Integer a = st.pop();  
        System.out.println(a);  
        System.out.println("stack: " + st);  
    }  
    public static void main(String args[]) {
```

```
        Stack<Integer> st = new Stack<Integer>();  
        System.out.println("stack: " + st);  
        showpush(st, 42);  
        showpush(st, 66);  
        showpush(st, 99);  
        showpop(st);  
        showpop(st);  
        showpop(st);  
        try {  
            showpop(st);  
        } catch (EmptyStackException e) {  
            System.out.println("empty stack");  
        }  
    } }
```

Output

stack: []

push(42)

stack: [42]

push(66)

stack: [42, 66]

push(99)

stack: [42, 66, 99]

pop -> 99

stack: [42, 66]

pop -> 66

stack: [42]

pop -> 42

stack: []

pop -> empty stack

```
public class StackBasicExample {  
    public static void main(String a[]){  
  
        Stack<Integer> stack = new Stack<>();  
        System.out.println("Empty stack : " + stack);  
        System.out.println("Empty stack : " +  
            stack.isEmpty());  
        stack.push(1001);  
        stack.push(1002);  
        stack.push(1003);  
        stack.push(1004);  
        System.out.println("Non-Empty stack : " +  
            stack);  
        System.out.println("Non-Empty stack: Pop  
            Operation : " + stack.pop());  
  
        System.out.println("Non-Empty stack : After  
        Pop Operation : " + stack);  
    }  
}
```

```
        System.out.println("Non-Empty stack : search()  
        Operation : " + stack.search(1002));  
        System.out.println("Non-Empty stack : " +  
            stack.isEmpty());  
    }  
}
```



Output:

Empty stack : []

Empty stack : true

Non-Empty stack : [1001, 1002, 1003, 1004]

Non-Empty stack: Pop Operation : 1004

Non-Empty stack : After Pop Operation : [1001, 1002, 1003]

Non-Empty stack : search() Operation : 2

Non-Empty stack : false



Dictionary

- Dictionary is an abstract class.
- It represents a key/value pair and operates much like Map.
- It is declared as shown here:

```
class Dictionary<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values

Abstract Methods Defined by Dictionary

Method	Purpose
Enumeration<V> elements()	Returns an enumeration of the values contained in the dictionary.
V get(Object key)	Returns the object that contains the value associated with key. If key is not in the dictionary, a null object is returned.
boolean isEmpty()	Returns true if the dictionary is empty, and returns false if it contains at least one key.
Enumeration<K> keys()	Returns an enumeration of the keys contained in the dictionary.
V put(K key, V value)	Inserts a key and its value into the dictionary. Returns null if key is not already in the dictionary; returns the previous value associated with key if key is already in the dictionary.
V remove(Object key)	Removes key and its value. Returns the value associated with key. If key is not in the dictionary, a null is returned.
int size()	Returns the number of entries in the dictionary.

Hashtable

- Like HashMap, Hashtable also stores key/value pair. However neither **keys** nor **values** can be **null**.
- It is declared like this:

class Hashtable<K, V>

- Here, **K** specifies the type of keys, and **V** specifies the type of values.
- **Hashtable** constructors are shown here:
 - Hashtable()
 - Hashtable(int *size*)
 - Hashtable(int *size*, float *fillRatio*)
 - Hashtable(Map *m*)

Legacy Methods Defined by Hashtable

Method	Description
<code>void clear()</code>	Resets and empties the hash table.
<code>Object clone()</code>	Returns a duplicate of the invoking object.
<code>boolean contains(Object value)</code>	Returns true if some value equal to value exists within the hash table. Returns false if the value isn't found.
<code>boolean containsKey(Object key)</code>	Returns true if some key equal to key exists within the hash table. Returns false if the key isn't found.
<code>boolean containsValue(Object value)</code>	Returns true if some value equal to value exists within the hash table. Returns false if the value isn't found.
<code>Enumeration<V> elements()</code>	Returns an enumeration of the values contained in the hash table.

V get(Object key)	Returns the object that contains the value associated with key. If key is not in the hash table, a null object is returned.
boolean isEmpty()	Returns true if the hash table is empty; returns false if it contains at least one key.
Enumeration<K> keys()	Returns an enumeration of the keys contained in the hash table.
V put(K key, V value)	Inserts a key and a value into the hash table. Returns null if key isn't already in the hash table; returns the previous value associated with key if key is already in the hash table.
void rehash()	Increases the size of the hash table
String toString()	Returns the string equivalent of a hash table.

```
class HTDemo {  
  
    public static void main(String args[]) {  
  
        Hashtable<String, Double> balance =  
        new Hashtable<String, Double>();  
  
        String str;  
  
        double bal;  
  
        balance.put("sachin", 3434.34);  
        balance.put("sumanth", 123.22);  
        balance.put("shreyas", 1378.00);  
        balance.put("karthik", 99.22);  
        balance.put("chetan", -19.08);  
  
        Enumeration<String> names = balance.keys();
```

```
        while(names.hasMoreElements()) {  
            str = names.nextElement();  
  
            System.out.println(str + ": " +  
            balance.get(str));  
        }  
  
        System.out.println();  
  
        bal = balance.get("sachin");  
        balance.put("sachin", bal+1000);  
  
        System.out.println("sachin new balance: " +  
        balance.get("sachin"));  
    }  
}
```

output

karthik: 99.22

chetan: -19.08

sachin: 3434.34

shreyas: 1378.0

sumanth: 123.22

sachin new balance: 4434.34

```
class HTDemo2 {  
    public static void main(String args[]) {  
        Hashtable<String, Double> balance =  
        new Hashtable<String, Double>();  
        String str;  
        double bal;  
        balance.put("manoj", 3434.34);  
        balance.put("sumanth", 123.22);  
        balance.put("shreyas", 1378.00);  
        balance.put("karthik", 99.22);  
        balance.put("cheatn", -19.08);  
  
        Set<String> set = balance.keySet();
```

```
        Iterator itr = set.iterator();  
        while(itr.hasNext()) {  
            str = itr.next();  
            System.out.println(str + ": " +  
            balance.get(str));  
        }  
        System.out.println();  
  
        bal = balance.get("manoj");  
        balance.put("manoj", bal+1000);  
        System.out.println("manoj new balance: " +  
        balance.get("manoj"));  
    } }
```

Properties

- Properties class extends Hashtable class.
- Used to maintain list of value in which both key and value are String
- Properties class define two constructor
 1. Properties()
 2. Properties(Properties *propDefault*)
- One useful capability of the **Properties** class is that we can specify a default property that will be returned if no value is associated with a certain key.

Methods

public String getProperty(String key): It returns value based on the key.

public String getProperty(String key, String defaultValue): It searches for the property with the specified key.

public void setProperty(String key, String value): It calls the put method of Hashtable.



```
class PropDemo {  
    public static void main(String args[]) {  
  
        Properties capitals = new Properties();  
        capitals.put("Karnataka", "Bengaluru");  
        capitals.put("TamilNadhu", "Chennai");  
        capitals.put("AndraPradesh", "Hyderabad");  
        capitals.put("Madhya Pradesh", "Bhopal");  
        capitals.put("Rajasthan", "Jaipur");  
  
        Set states = capitals.keySet();
```

```
        for(Object name : states)  
  
            System.out.println("The capital of " +  
                                name + " is " +  
                                capitals.getProperty((String)name) + ".");  
        System.out.println();  
  
        String str = capitals.getProperty("Kerala", "Not  
        Found");  
  
        System.out.println("The capital of Kerala is "  
                                + str + ".");  
    }  
}
```

output

The capital of Karnataka is Bengaluru

The capital of TamilNadhu is Chennai.

The capital of Rajasthan is Jaipur.

The capital of AndraPradesh is Hyderabad.

The capital of Madhya Pradesh is Bhopal.

The capital of Kerala is Not Found.