**COLLECTION**

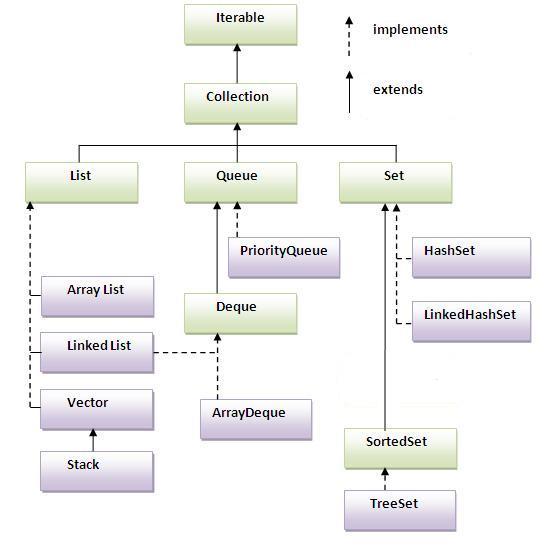
**Collections in java** is a framework that provides an architecture to store and manipulate the group of objects.

Frameworks

– Reusable semi-complete application

-Main body and algorithm

Collection represents a single unit of objects i.e. a group.



**Iterator interface**

Iterator interface provides the facility of iterating the elements in forward direction only

There are only three methods in the Iterator interface. They are:

1. **public boolean hasNext()** it returns true if iterator has more elements.
2. **public object next()** it returns the element and moves the cursor pointer to the next element.
3. **public void remove()** it removes the last elements returned by the iterator. It is rarely used.

# ArrayList

* Java ArrayList class uses a dynamic array for storing the elements.It extends AbstractList class and implements List interface.
* Java ArrayList class can contain duplicate elements.
* Java ArrayList class maintains insertion order.
* Java ArrayList class is non synchronized.
* Java ArrayList allows random access because array works at the index basis.
* In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

### Two ways to iterate the elements of collection in java

ArrayList<String> al=new ArrayList<String>();//creating arraylist

  al.add("chennai");//adding object in arraylist

  al.add("delhi");

  al.add("pune");

  al.add("noida");

Iterator itr=al.iterator();//getting Iterator from arraylist to traverse

the elements

while(itr.hasNext()){

   System.out.println(itr.next());

for(String obj:al)

    System.out.println(obj);

### User-defined class objects in Java ArrayList

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.\*;

public class TestCollection3{

public static void main(String args[]){

 //Creating user-defined class objects

  Student s1=new Student(101,"chennai",23);

  Student s2=new Student(102,"delhi",21);

  Student s3=new Student(103,"pune",25);

ArrayList<Student> al=new ArrayList<Student>();

  al.add(s1);//adding Student class object

 al.add(s2);

  al.add(s3);

  Iterator itr=al.iterator();

  //traversing elements of ArrayList object

  while(itr.hasNext()){

    Student st=(Student)itr.next();

    System.out.println(st.rollno+" "+st.name+" "+st.age);

  }

 }

}

### How to convert ArrayList to Array

### class ArrayListToArray {

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

### ArrayList al = new ArrayList();

### al.add(1);

### al.add(2);

### al.add(3);

### al.add(4);

### System.out.println("Contents of al: " + al);

### Object ia[] = al.toArray();

### int sum = 0;

### for(int i=0; i<ia.length; i++)

### sum += ((Integer) ia[i]).intValue();

### System.out.println("Sum is: " + sum);

### }

### }

### How to convert LinkedList to Array

### public class Main {       public void convertLinkedListToArray() {          List<String> theList = new LinkedList<String>();         theList.add("Apples");         theList.add("Bananas");         theList.add("Oranges");         theList.add("Grapes");                   String[] fruits = theList.toArray(new String[theList.size()]);           for (int i = 0; i < fruits.length; i++) {             System.out.println(fruits[i]);         }      }     public static void main(String[] args) {         new Main().convertLinkedListToArray();     }

### How to convert Array to ArrayList

### import java.util.\*;

### class Hello

### {

### public static void main(String[] args)

### {

### List myList = new ArrayList();

### String[] myStringArray = new String[] {"Java", "is", "Cool"};

### Collections.addAll(myList, myStringArray);

### System.out.println(myList);

### }

### }

**Sorting an array of primitives**

**import** java.util.Arrays;

**public** **class** Te {

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

// **TODO** Auto-generated method stub

**int**[] numbers = {4, 9, 1, 3, 2, 8, 7, 0, 6, 5};

System.***out***.println(Arrays.*toString*(numbers));

java.util.Arrays.*sort*(numbers);

System.***out***.println(Arrays.*toString*(numbers));

}

}

String[] fruits = {"Orange", "Grape", "Apple", "Lemon", "Banana"};

 System.out.println("Before sorting: " + Arrays.toString(fruits));

 Arrays.sort(fruits);

 System.out.println("After sorting: " + Arrays.toString(fruits));

**Sorting an array into reverse order**

String[] fruits = {"Orange", "Grape", "Apple", "Lemon", "Banana"};

 Arrays.sort(fruits);

System.out.println("Alphabetical order: " + Arrays.toString(fruits));

 Arrays.sort(fruits, Collections.reverseOrder());

System.out.println("Reverse-alphabetical order: " + Arrays.toString(fruits));

**Sorting a specified range of an array**

int[] numbers = {4, 9, 1, 3, 2, 8, 7, 0, 6, 5};

 System.out.println("Before sorting: " + Arrays.toString(numbers));

 Arrays.sort(numbers, 0, 5);

 System.out.println("Sorted a half: " + Arrays.toString(numbers));

Output:-

Before sorting: [4, 9, 1, 3, 2, 8, 7, 0, 6, 5]

Sorted a half: [1, 2, 3, 4, 9, 8, 7, 0, 6, 5]

### Example of addAll(Collection c) method

// similar to unionall

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

  al.add("chennai");

  al.add("delhi");

  al.add("pune");

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

  al2.add("noida");

  al2.add("guntur");

  al.addAll(al2);

  Iterator itr=al.iterator();

  while(itr.hasNext()){

   System.out.println(itr.next());

### Example of removeAll() method

//retains all items in first list & removes the common items

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

  al.add("chennai");

  al.add("delhi");

  al.add("pune");

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

  al2.add("chennai");

  al2.add("noida");

  al.removeAll(al2);

  System.out.println("iterating the elements after removing the elements of al2...");

  Iterator itr=al.iterator();

  while(itr.hasNext()){

   System.out.println(itr.next());

  }

Output:- delhi pune

### Example of retainAll() method

### //removes all items from current list & retains only the common //element

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

  al.add("chennai");

  al.add("delhi");

  al.add("pune");

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

  al2.add("chennai");

  al2.add("noida");

 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:-chennai

# LinkedList class

* Java LinkedList class uses doubly linked list to store the elements. It extends the AbstractList class and implements List and Deque interfaces.
* Java LinkedList class can contain duplicate elements.
* Java LinkedList class maintains insertion order.
* Java LinkedList class is non synchronized.
* In Java LinkedList class, manipulation is fast because no shifting needs to be occurred.
* Java LinkedList class can be used as list, stack or queue.

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses **dynamic array** to store the elements. | LinkedList internally uses **doubly linked list** to store the elements. |
| 2) Manipulation with ArrayList is **slow** because it internally uses array. If any element is removed from the array, all the bits are shifted in memory. | Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory. |
| 3) ArrayList class can **act as a list** only because it implements List only. | LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces. |
| 4) ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |

## ListIterator Interface

ListIterator Interface is used to traverse the element in backward and forward direction.

**Commonly used methods of ListIterator Interface:**

1. public boolean hasNext();
2. public Object next();
3. public boolean hasPrevious();
4. public Object previous();

### Example of ListIterator Interface:

public boolean hasNext();

public Object next();

public boolean hasPrevious();

public Object previous();

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

al.add("chennai");

al.add("delhi");

al.add("pune");

al.add(1,"noida");

System.out.println("element at 2nd position: "+al.get(2));

ListIterator<String> itr=al.listIterator();

System.out.println("traversing elements in forward direction...");

while(itr.hasNext()){

System.out.println(itr.next());

 }

System.out.println("traversing elements in backward direction...");

while(itr.hasPrevious()){

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

 }

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| 1) ArrayList is **not synchronized**. | Vector is **synchronized**. |
| 2) ArrayList **increments 50%** of current array size if number of element exceeds from its capacity. | Vector **increments 100%** means doubles the array size if total number of element exceeds than its capacity. |
| 3) ArrayList is **not a legacy** class, it is introduced in JDK 1.2. | Vector is a **legacy** class. |
| 4) ArrayList is **fast** because it is non-synchronized. | Vector is **slow** because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object. |
| 5) ArrayList uses **Iterator** interface to traverse the elements. | Vector uses **Enumeration** interface to traverse the elements. But it can use Iterator also. |

**Vector**

Vector<String> v=new Vector<String>();//creating vector

  v.add("umesh");//method of Collection

  v.addElement("irfan");//method of Vector

  v.addElement("kumar");

  //traversing elements using Enumeration

  Enumeration e=v.elements();

  while(e.hasMoreElements()){

   System.out.println(e.nextElement());

**Stack**

**Stack** is a subclass of **Vector** that implements a standard last-in, first-out stack. **Stack** only defines the default constructor, which creates an empty stack. **Stack** includes all the methods defined by **Vector**, and adds several of its own.

To put an object on the top of the stack, call **push()**. To remove and return the top element, call **pop()**. An **EmptyStackException** is thrown if you call **pop( )** when the invoking stack is empty. You can use **peek( )** to return, but not remove, the top object.

The **empty()** method returns **true** if nothing is on the stack. The **search()** method determines whether an object exists on the stack, and returns the number of pops that are required to bring it to the top of the stack. Here is an example that creates a stack, pushes several **Integer** objects onto it, and then pops them off again:

// Demonstrate the Stack class.   
import java.util.\*;   
class StackDemo {   
static void showpush(Stack st, int a) {   
st.push(new Integer(a));   
System.out.println("push(" + a + ")");   
System.out.println("stack: " + st);   
}   
static void showpop(Stack st) {   
System.out.print("pop -> ");   
Integer a = (Integer) st.pop();   
System.out.println(a);   
System.out.println("stack: " + st);   
}   
public static void main(String args[]) {   
Stack st = new Stack();   
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");   
}   
}   
}

The following is the output produced by the program; notice how the exception handler for **EmptyStackException** is caught so that you can gracefully handle a stack underflow:

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

**Queue**

**General-purpose Queue implementations:**

**LinkedList**: this class implements both List and Deque interface, thus having hybrid characteristics and behaviors of list and queue. Consider using a LinkedList when you want fast adding and fast removing elements at both ends, plus accessing elements by index.

**PriorityQueue**: this queue orders elements according to their natural ordering, or by a Comparator provided at construction time. Consider using a PriorityQueue when you want to take advantages of natural ordering and fast adding elements to the tail and fast removing elements at the head of the queue.

**ArrayDeque**: a simple implementation of the Deque interface. Consider using an ArrayDeque when you want to utilize features of a double ended queue without list-based ones (simpler than a LinkedList).

Basically, Queue provides three primary types of operations which differentiate a queue from others:

* Insert: adds an element to the tail of the queue.
* Remove: removes the element at the head of the queue.
* Examine: returns, but does not remove, the element at the head of the queue.

And for each type of operation, there are two versions:

* The first version throws an exception if the operation fails, e.g. could not add element when the queue is full.
* The second version returns a special value (either null or false, depending on the operation).

The following table summarizes the main operations of the Queue interface:

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

Queue<String> queueNames = new LinkedList<>();

queueNames.add("Dale");

queueNames.add("Bob");

queueNames.add("Frank");

queueNames.add("Alice");

queueNames.add("Eric");

queueNames.add("Cole");

queueNames.add("John");

for (String name : queueNames) {

    System.out.println(name);

}

Dale

Bob

Frank

Alice

Eric

Cole

John

**Queue example using PriorityQueue**

Queue<String> queueNames = new PriorityQueue<>();

queueNames.add("Dale");

queueNames.add("Bob");

queueNames.add("Frank");

queueNames.add("Alice");

queueNames.add("Eric");

queueNames.add("Cole");

queueNames.add("John");

queueNames.forEach(name -> System.out.println(name));

Alice

Bob

Cole

Dale

Eric

Frank

John

(the elements are sorted in the alphabetic order (natural ordering of Strings))

**PriorityQueue example**

java.util.PriorityQueue has been introduced in JDK 1.5. PriorityQueue maintains the element ordering on the basis of java Comparable and Comparator. When PriorityQueue orders the elements on the basis of Comparable implemented by element class, it is known as natural ordering and this is the PriorityQueue default ordering.

see the use of PriorityQueue with the methods add(), poll() and remove() etc.

add() method adds the element in PriorityQueue.

poll() method fetches the element from head and removes it.

remove() method removes the specified element.

import java.util.PriorityQueue;

public class PriorityQueueTest {

public static void main(String... args ){

PriorityQueue<String> pq=new PriorityQueue<String>();

pq.add("A");

pq.add("B");

pq.add("C");

pq.add("D");

System.out.println("After Using add method.");

for(String s:pq){

System.out.println(s);

}

System.out.println("After Using poll method.");

// Retrieves and removes the head of this queue, or returns null if this queue is empty.

pq.poll();

for(String s:pq){

System.out.println(s);

}

System.out.println("After Using remove method.");

// Removes a single instance of the specified element from this queue, if it is present

pq.remove("D");

for(String s:pq){

System.out.println(s);

}

}

}

Output:-

After Using add method.

A

B

C

D

After Using poll method.

B

D

C

After Using remove method.

B

C

**Queue** is a list where insertion is done at one end and removal is done at the other end.   
  
**Dequeue** is a list where every end supports insertion and removal.  
With this feature, it is possible to use the dequeue as a list and a stack at the same time as required by the application.   
  
**Priority queue** does not have any ends.  
In a priority queue, elements can be inserted in any order but removal of the elements is in a sorted order.  
Due to this behavior, a priority queue can be used to sort the elements.

**Deque or Double Ended Queue or Deck**

The java.util.Deque interface is a subtype of the java.util.Queue interface.

In this type of queue you can add and remove elements from both the ends

and hence it is abbreviated as “Double Ended Queue” and pronounced

as “deck” in short.

We can choose between the following Deque implementations of Java Collections.

� java.util.LinkedList

� java.util.ArrayDeque

Examples of creating queue instance

Queue queue1 = new LinkedList();

Queue queue2 = new ArrayDeque();

import java.util.Deque;

import java.util.Iterator;

import java.util.LinkedList;

import java.util.NoSuchElementException;

import java.util.Queue;

public class DequeExample {

public static void main(String[] args)

{

//Creating a queue object through LinkedList

Deque deque = new LinkedList();

//Adding elements to the deque

deque.add("Java");

//addFirst() adds the element to the head of the deque

deque.addFirst("jQuery");

//addFirst() adds the element to the tail of the deque

deque.addLast("HTML5");

//offer() adds the elements to the deque and returns a boolean

deque.offer("AngualarJS");

//offerFirst() adds the element to the head of the deque and returns a boolean

deque.offerFirst("NodeJS");

//offerFirst() adds the element to the tail of the deque and returns a boolean

deque.offerLast("Javascript");

System.out.println("Elements of the deque"+deque);

//getFirst() Will retrive the head of the deque

System.out.println("First element of the deque before removal:"+deque.getFirst());

//The removeFirst() &pop() method will remove the first element of the queue

deque.removeFirst();

deque.pop();

//peekFirst() Will retrive the head of the deque

System.out.println("First element of the deque after removal:"+deque.peekFirst());

//getLast() Will retrive the tail of the deque

System.out.println("Last element of the deque before removal:"+deque.getLast());

//The removeLast() method will remove the tail element of the queue

deque.removeLast();

//peekLast() Will retrive the tail of the deque

System.out.println("Last element of the deque after removal:"+deque.peekLast());

// Iterate through the queue elements.

System.out.println("Normal Iteration");

Iterator it1 = deque.iterator();

while (it1.hasNext()) {

System.out.println(" "+ it1.next());

}

// Reverse order iterator

Iterator it2 = deque.descendingIterator();

System.out.println("Reversed Iteration");

while (it2.hasNext()) {

System.out.println(" "+ it2.next());

}

}

}

Output

Elements of the deque[NodeJS, jQuery, Java, HTML5, AngualarJS, Javascript]

First element of the deque before removal:NodeJS

First element of the deque after removal:Java

Last element of the deque before removal:Javascript

Last element of the deque after removal:AngualarJS

Standard Iterator

Java

HTML5

AngualarJS

Reverse Iterator

AngualarJS

HTML5

Java

C

**Example of ArrayDeque in Java**

1. ArrayDeque implements Deque interface and ArrayDeque are available from jdk1.6.

2. Deque is that queue which allows insert and remove of elements from both sides.

3. ArrayDeque is not thread safe. ArrayDeque allows unlimited insertion of elements.

4. ArrayDeque can be used as queue and stack both. When it is used as stack, it is faster than stack and when it is used as

queue, it is faster than linkedlist.

5. Iterator object of ArrayDeque allows removal and modification of ArrayDeque elements.

import java.util.ArrayDeque;

import java.util.Iterator;

public class ArrayDequeTest {

public static void main(String... args){

ArrayDeque<String> aq= new ArrayDeque<String>();

aq.add("A");

aq.add("B");

aq.add("C");

//offerFirst-adds elements at the front of the ArrayDeque

aq.offerFirst("D");

//offerLast inserts the element at the last of ArrayDeque

aq.offerLast("E");

Iterator<String> itr= aq.iterator();

while(itr.hasNext()){

System.out.println(itr.next());

}

}

}

**Output**

D

A

B

C

E

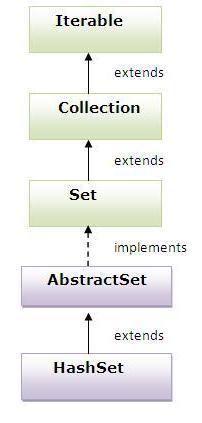
# HashSet

* uses hashtable to store the elements.It extends AbstractSet class and implements Set interface.
* contains unique elements only.

## Difference between List and Set:

## List can contain duplicate elements whereas Set contains unique elements only.

## Hierarchy of HashSet class:



HashSet<String> al=new HashSet<String>();

  al.add("chennai");

  al.add("delhi");

  al.add("pune");

  al.add("noida");

  Iterator<String> itr=al.iterator();

  while(itr.hasNext()){

   System.out.println(itr.next());

  }

HashSet using containsAll() method

String elements[] = { "A", "B", "C", "D", "E" };

Set set = **new** HashSet(Arrays.*asList*(elements));

elements = **new** String[] { "A", "B", "C" };

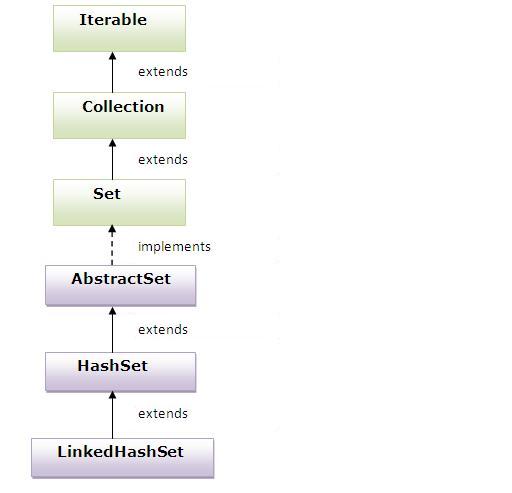
Set set2 = **new** HashSet(Arrays.*asList*(elements));

System.***out***.println(set.containsAll(set2));

# LinkedHashSet

* contains unique elements only like HashSet. It extends HashSet class and implements Set interface.
* maintains insertion order.

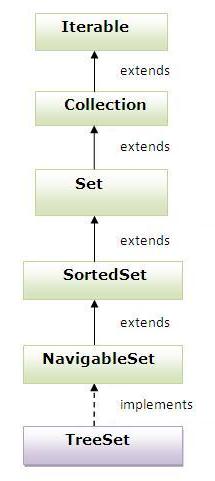
## Hierarchy of LinkedHashSet class



# TreeSet

* contains unique elements only like HashSet. The TreeSet class implements NavigableSet interface that extends the SortedSet interface.
* maintains ascending order.

## Hierarchy of TreeSet class:



**Stack**

**import java.util.\*;**

**class StackDemo {**

**static void showpush(Stack st, int a) {**

**st.push(new Integer(a));**

**System.out.println("push(" + a + ")");**

**System.out.println("stack: " + st);**

**}**

**static void showpop(Stack st) {**

**System.out.print("pop -> ");**

**Integer a = (Integer) st.pop();**

**System.out.println(a);**

**System.out.println("stack: " + st);**

**}**

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

**Stack st = new Stack();**

**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");**

**}**

**}**

**}**

# Map Interface

A map contains values on the basis of key i.e. key and value pair. Each key and value pair is known as an entry. Map contains only unique keys.

Map is useful if you have to search, update or delete elements on the basis of key.

## Entry Interface

Entry is the subinterface of Map. So we will be accessed it by Map.Entry name. It provides methods to get key and value.

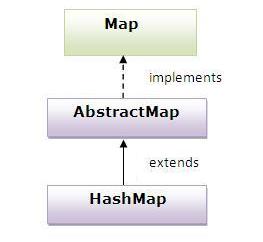
### Methods of Map.Entry interface

1. **public Object getKey():** is used to obtain key.
2. **public Object getValue():**is used to obtain value.

# HashMap

* A HashMap contains values based on the key. It implements the Map interface and extends AbstractMap class.
* It contains only unique elements.
* It may have one null key and multiple null values.
* It maintains no order.

## Hierarchy of HashMap class:



HashMap<Integer,String> hm=new HashMap<Integer,String>();

   hm.put(100,"chennai");

  hm.put(101,"delhi");

  hm.put(102,"pune");

  for(Map.Entry m:hm.entrySet()){

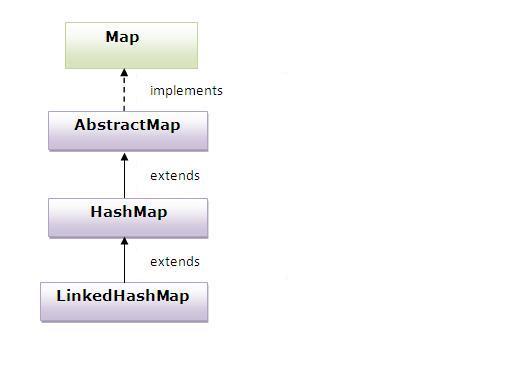
   System.out.println(m.getKey()+" "+m.getValue());

  }

# LinkedHashMap

* A LinkedHashMap contains values based on the key. It implements the Map interface and extends HashMap class.
* It contains only unique elements.
* It may have one null key and multiple null values.
* It is same as HashMap instead maintains insertion order.

## Hierarchy of LinkedHashMap class:



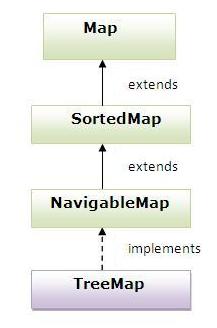
# TreeMap

* A TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
* It contains only unique elements.
* It cannot have null key but can have multiple null values.
* It is same as HashMap instead maintains ascending order.

**Difference between HashMap and TreeMap:**

|  |  |
| --- | --- |
| 1) HashMap is can contain one null key. | TreeMap can not contain any null key. |
| 2) HashMap maintains no order. | TreeMap maintains ascending order. |

## Hierarchy of TreeMap class:



# Hashtable

* A Hashtable is an array of list.Each list is known as a bucket.The position of bucket is identified by calling the hashcode() method.A Hashtable contains values based on the key. It implements the Map interface and extends Dictionary class.
* It contains only unique elements.
* It may have not have any null key or value.
* It is synchronized.

# Difference between HashMap and Hashtable

HashMap and Hashtable both are used to store data in key and value form. Both are using hashing technique to store unique keys

|  |  |
| --- | --- |
| **HashMap** | **Hashtable** |
| 1) HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code. | Hashtable is **synchronized**. It is thread-safe and can be shared with many threads. |
| 2) HashMap **allows one null key and multiple null values**. | Hashtable **doesn't allow any null key or value**. |
| 3) HashMap is a **new class introduced in JDK 1.2**. | Hashtable is a **legacy class**. |
| 4) HashMap is **fast**. | Hashtable is **slow**. |
| 5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap); | Hashtable is internally synchronized and can't be unsynchronized. |
| 6) HashMap is **traversed by Iterator**. | Hashtable is **traversed by Enumerator and Iterator**. |
| 7) Iterator in HashMap is **fail-fast**. | Enumerator in Hashtable is **not fail-fast**. |
| 8) HashMap inherits **AbstractMap** class. | Hashtable inherits **Dictionary** class. |

Fail fast simply means that it will throw an exception when it identifies some change in the collection object while iterating through the collection.

Fail Safe Iterator makes copy of the internal data structure (object array) and iterates over the copied data structure. Will not throw exception

|  |  |
| --- | --- |
| **Comparable** | **Comparator** |
| 1) Comparable provides **single sorting sequence**. In other words, we can sort the collection on the basis of single element such as id or name or price etc. | Comparator provides **multiple sorting sequence**. In other words, we can sort the collection on the basis of multiple elements such as id, name and price etc. |
| 2) Comparable **affects the original class** i.e. actual class is modified. | Comparator **doesn't affect the original class** i.e. actual class is not modified. |
| 3) Comparable provides **compareTo() method** to sort elements. | Comparator provides **compare() method** to sort elements. |
| 4) Comparable is found in **java.lang** package. | Comparator is found in **java.util** package. |
| 5) We can sort the list elements of Comparable type by **Collections.sort(List)** method. | We can sort the list elements of Comparator type by **Collections.sort(List,Comparator)** method. |

## Iterator vs ListIterator

1) Iterator is used for traversing List and Set both.

We can use ListIterator to traverse List only, we cannot traverse Set using ListIterator.

2) We can traverse in only forward direction using Iterator.

Using ListIterator, we can traverse a List in both the directions (forward and Backward).

3) We cannot obtain indexes while using Iterator

We can obtain indexes at any point of time while traversing a list using ListIterator. The methods nextIndex() and previousIndex() are used for this purpose.

4) We cannot add element to collection while traversing it using Iterator, it throws ConcurrentModificationException when you try to do it.

We can add element at any point of time while traversing a list using ListIterator.

5) We cannot replace the existing element value when using Iterator.

By using set(E e) method of ListIterator we can replace the last element returned by next() or previous() methods.

6) Methods of Iterator:

* hasNext()
* next()
* remove()

Methods of ListIterator:

* add(E e)
* hasNext()
* hasPrevious()
* next()
* nextIndex()
* previous()
* previousIndex()
* remove()
* set(E e)

**Examples for Comparator**

****

**import** java.util.Comparator;

**import** java.util.PriorityQueue;

**public** **class** mytest {

**public** **static** **void** main(String... args ){

PriorityQueueComparator pqc=**new** PriorityQueueComparator();

PriorityQueue<String> pq=**new** PriorityQueue<String>(5,pqc);

pq.add("ABC");

pq.add("BD");

pq.add("ABCD");

**for**(String s:pq){

System.***out***.println(s);

}

}

}

**class** PriorityQueueComparator **implements** Comparator<String>{

**public** **int** compare(String s1, String s2) {

**if** (s1.length() < s2.length()) {

**return** -1;

}

**if** (s1.length() > s2.length()) {

**return** 1;

}

**return** 0;

}

}

Find the output.

BD

ABC

ABCD

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

public class MyArrayListSort {

    public static void main(String a[]){

        List<Empl> list = new ArrayList<Empl>();

        list.add(new Empl("Ram",3000));

        list.add(new Empl("John",6000));

        list.add(new Empl("Crish",2000));

        list.add(new Empl("Tom",2400));

        Collections.sort(list,new MySalaryComp());

        System.out.println("Sorted list entries: ");

        for(Empl e:list){

            System.out.println(e);

        }

    }

}

class MySalaryComp implements Comparator<Empl>{

    @Override

    public int compare(Empl e1, Empl e2) {

        if(e1.getSalary() < e2.getSalary()){

            return 1;

        } else {

            return -1;

        }

    }

}

class Empl{

    private String name;

    private int salary;

    public Empl(String n, int s){

        this.name = n;

        this.salary = s;

    }

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

    public int getSalary() {

        return salary;

    }

    public void setSalary(int salary) {

        this.salary = salary;

    }

    public String toString(){

        return "Name: "+this.name+"-- Salary: "+this.salary;

    }

}

**Example for Comparable & Comparator**

class Dog implements Comparator<Dog>, Comparable<Dog>

{

private String name;

private int age;

Dog(){ } Dog(String n, int a){ name = n; age = a; }

public String getDogName(){ return name; }

public int getDogAge(){ return age; }

// Overriding the compareTo method

public int compareTo(Dog d) // comparable

{

return (this.name).compareTo(d.name);

}

// Overriding the compare method to sort the age

public int compare(Dog d, Dog d1) // comparator

{

return d.age - d1.age;

}

}

public class Example

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

// Takes a list o Dog objects

List<Dog> list = new ArrayList<Dog>();

list.add(new Dog("Shaggy",3));

list.add(new Dog("Lacy",2));

list.add(new Dog("Roger",10));

list.add(new Dog("Tommy",4));

list.add(new Dog("Tammy",1));

Collections.sort(list);

// Sorts the array list for(Dog a: list)

//printing the sorted list of names

System.out.print(a.getDogName() + ", ");

// Sorts the array list using comparator

Collections.sort(list, new Dog());

System.out.println(" ");

for(Dog a: list)//printing the sorted list of ages System.out.print(a.getDogName() +" : "+ a.getDogAge() + ", "); } }

if you want to sort objects based on natural order then use Comparable in Java and if you want to sort on some other attribute of object then use Comparator in Java  
Lacy, Roger, Shaggy, Tammy, Tommy, // natural ordering (alphabhetical sorting or asciibetical sorting)

Tammy : 1, Lacy : 2, Shaggy : 3, Tommy : 4, Roger : 10, // totol ordering

**Map implementations**

**How to find the frequency key count in HashMap for the given String array?**

String[] splitted ={"if","if","if","else","else"};

Map<String, Integer> hm = **new** HashMap<String, Integer>();

**int** x;

**for** (**int** i = 0; i < splitted.length; i++) {

**if** (hm.containsKey(splitted[i])) {

x = hm.get(splitted[i]);

hm.put(splitted[i], x + 1);

} **else** {

hm.put(splitted[i], 1);

}

}

**for** (String key : hm.keySet()) {

System.***out***.println(key + " " + hm.get(key));

}

Output:-

else 2

if 3

**How to remove duplicate values from Hashmap**

Map<String,Object> mapValues = new HashMap<String,Object>(5);

mapValues.put("1", "TJ");

mapValues.put("2", "Arun");

mapValues.put("3", "TJ");

mapValues.put("4", "Venkat");

mapValues.put("5", "Arun");

// mapValues.put("5", "sree"); 🡪 this will overwrite Arun as sree

Collection<Object> list = mapValues.values();

for(Iterator<Object> itr = list.iterator(); itr.hasNext();)

{

if(Collections.frequency(list, itr.next())>1)

{

itr.remove();

}

}

System.out.println(list);

Output:- TJ Arun Venkat

**HashMap vs ConcurrentHashMap**

ConcurrentHashMap is thread-safe that is the code can be accessed by single thread at a time .      
     while HashMap is not thread-safe .

ConcurrentHashMap does not allow NULL values . So the key can not be null in  
     ConcurrentHashMap .While In HashMap there can only be one null key .

**How to synchronize HashMap**

**import** **java.util.\***;

**public** **class** **HashMapSynchronization** {

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

// create map

Map<String,String> map = **new** HashMap<String,String>();

// populate the map

map.put("1","ALIVE ");

map.put("2","IS");

map.put("3","AWESOME");

// create a synchronized map

Map<String,String> syncMap = Collections.synchronizedMap(map);

System.out.println("Synchronized map :"+syncMap);

}

}

**Hashing**

*Hashing* is an efficient technique for storing and retrieving data.

A common hashing scheme uses an array where each element is a list of items. The array elements are called ***buckets*.**

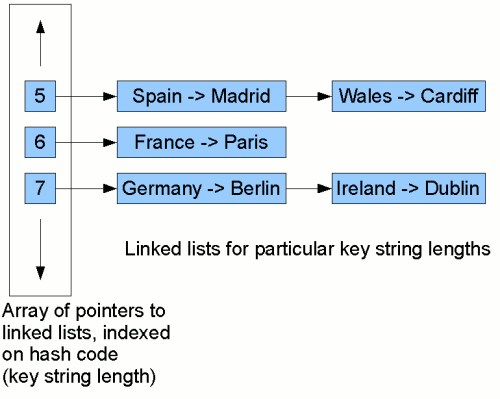
Converting an item to its array index is done by a ***hash function***.

The array index returned by the hash function is called the ***hash value***of the item

|  |  |
| --- | --- |
| **Example data to put in a map** | |
|  |
| Cuba | Havana |
| England | London |
| France | Paris |
| Spain | Madrid |
| Switzerland | Berne |

|  |  |  |
| --- | --- | --- |
| **Position (hash code = key length)** | **Keys array** | **Values array** |
| 1 |  |  |
| 2 |  |  |
| 3 |  |  |
| 4 | Cuba | Havana |
| 5 | Spain | Madrid |
| 6 | France | Paris |
| 7 | England | London |
| 8 |  |  |
| 9 |  |  |
| 10 |  |  |
| 11 | Switzerland | Berne |

we can solve the problem of **collisions** by having an array of (references to) **linked lists**2 rather than simply an array of keys/values



**1. If the objects are same returns same hashCode (reverse is not true)ie. same hashCode can have different objects**

class Hello

{

public static void main(String[] args)

{

A object1 = new A();

A object2 = object1;

// Compare objects and then compare their hash codes

if (object1.equals(object2))

System.out.println("hash code 1 = " + object1.hashCode() +

", hashcode 2 = " + object2.hashCode());

// Compare hash codes and then compare objects

if (object1.hashCode() == object2.hashCode())

{

if (object1.equals(object2))

System.out.println("object1 equals object2");

else

System.out.println("object1 does not equal object2");

}

}

}

output:-

hash code 1 = 18426253, hashcode 2 = 18426253

object1 equals object2

**2.Same hashCode having different objects**

public class Test

{

public static void main(String[] args) {

Map<Integer, Integer> map=new HashMap<Integer, Integer>();

map.put(1, 11);

map.put(4, 11);

System.out.println(map.hashCode());

Map<Integer, Integer> map1=new HashMap<Integer, Integer>();

map1.put(1, 11);

map1.put(4, 11);

System.out.println(map1.hashCode());

if (map.equals(map1)) {

System.out.println("equal ");

}

}

}

**3.If the objects are different returns different hashCodes.**

class ArrayUtil

{

public static void main(String[] args)

{

A object1 = new A();

A object2 = new A();

// Compare objects and then compare their hash codes

if (!(object1.equals(object2)))

System.out.println("hash code 1 = " + object1.hashCode() +

", hashcode 2 = " + object2.hashCode());

// Compare hash codes and then compare objects

if (object1.hashCode() != object2.hashCode())

{

if (object1.equals(object2))

System.out.println("object1 equals object2");

else

System.out.println("object1 does not equal object2");

}

}

}

OUTPUT:-

hash code 1 = 16197143, hashcode 2 = 31843011

object1 does not equal object2

**why to override hashCode() & equals() method**

**import** java.util.Arrays;

**import** java.util.HashSet;

**import** java.util.Set;

**class** A {}

**public** **class** HashCodeExample {

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

// String implicitly overrides hashCode() & equals method

String s1 = **new** String("Hello");

String s2 = **new** String("Hello");

**if**(s1.hashCode()==s2.hashCode())

{

**if**(s1.equals(s2))

System.***out***.println("true"); // true

**else**

System.***out***.println("false");

}

Emp s3 = **new** Emp(100);

Emp s4 = **new** Emp(100);

// Assuming two employees having same ids

**if**(s3.hashCode()==s3.hashCode())

{

**if**(s3.equals(s4))

System.***out***.println("true");

**else**

System.***out***.println("false"); // false

// true after overriding hashCode() & equals() }

}}

**class** Emp

{

**int** id;

Emp(**int** id) { **this**.id=id; }

@Override

**public** **int** hashCode()

{

**final** **int** PRIME = 31;

**int** result = 1;

result = PRIME \* result + getId();

**return** result;

}

**private** **int** getId() {

**return** id;

}

**public** **boolean** equals(Object o) {

**if**(o == **null**)

{

**return** **false**;

}

**if** (o == **this**)

{

**return** **true**;

}

**if** (getClass() != o.getClass())

{

**return** **false**;

}

Emp e = (Emp) o;

**return** (**this**.getId() == e.getId());

}

}

**Program: How to avoid duplicate user defined objects in TreeSet?**

import java.util.Comparator;

import java.util.Set;

import java.util.TreeSet;

public class MyUserDuplicates {

    public static void main(String a[]){

        Set<Emp> ts = new TreeSet<Emp>(new EmpComp());

        ts.add(new Emp(201,"John",40000));

        ts.add(new Emp(302,"Krish",44500));

        ts.add(new Emp(146,"Tom",20000));

        ts.add(new Emp(543,"Abdul",10000));

        ts.add(new Emp(12,"Dinesh",50000));

        //adding duplicate entry

        ts.add(new Emp(146,"Tom",20000));

        //check duplicate entry is there or not

        for(Emp e:ts){

            System.out.println(e);

        }

    }

}

class EmpComp implements Comparator<Emp>{

    @Override

    public int compare(Emp e1, Emp e2) {

        if(e1.getEmpId() == e2.getEmpId()){

            return 0;

        } if(e1.getEmpId() < e2.getEmpId()){

            return 1;

        } else {

            return -1;

        }

    }

}

class Emp {

    private int empId;

    private String empName;

    private int empSal;

    public Emp(int id, String name, int sal){

        this.empId = id;

        this.empName = name;

        this.empSal = sal;

    }

    public int getEmpId() {

        return empId;

    }

    public void setEmpId(int empId) {

        this.empId = empId;

    }

    public String getEmpName() {

        return empName;

    }

    public void setEmpName(String empName) {

        this.empName = empName;

    }

    public int getEmpSal() {

        return empSal;

    }

    public void setEmpSal(int empSal) {

        this.empSal = empSal;

    }

    public String toString(){

        return empId+" : "+empName+" : "+empSal;

    }

}

TreeSet in descending order using comparator using anonymous class

TreeSet ts = new TreeSet(new Comparator<String>) {

public int compare(String s1, String s2)

{

s2.compareTo(s1);

}

};

ts.add("sam");

ts.add("ravi");

ts.add("amsha");

for(String s : ts)

System.out.println(s);

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_