1. How HashMap Works Internally?

To understand this first we should know internal structure of the HashMap.

HashMap stores data in from of key-value pair. Each key-value pair is stored in Object of Entry<K,V> class.

Let’s snippet of Entry class-

static class Entry<K,V> implements Map.Entry<K,V>

{

final K key;

V value;

Entry<K,V> next;

int hash;

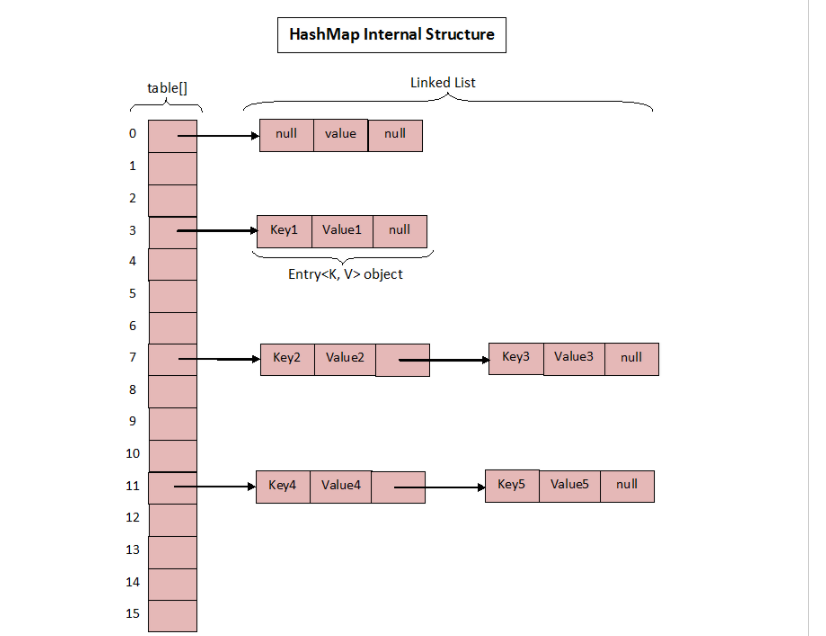
}

* Key- It stores key of the Object and it is final.
* Value- It stores value for the key.
* **Next- It holds pointer to next key-value pair. This attribute makes the key-value pairs stored as a linked list.**
* Hash – It stores hash code for the key.

These Entry Objects are stored in an array called table[]. Initial and default size of the array is 16.

Note- The table can be resized but length must be power of 2.

Below image describes the internal structure of the HashMap.



Now Entry will be stored in these 16 buckets/index. But which Entry will be placed where, decided by Hashing Principle.

**What is Hashing Principle?**

Hashing is an Algorithm to find unique integer value representation of an Object.

* Hashing Algorithm said to be the best if it returns the same hash code each time it is called on the same object. **Two objects can have same hash code in Java**.
* This hash code is passed in indexFor() method to calculate the index in table[] array.
* HashMap has its own method hash() to calculate the hash code. It is different from hashCode() method of Object class.
* For a **null** Object hash() method of HashMap returns 0 while hashCode() method of Object class throws NullPointerException.

String str=**null**;

System.***out***.println(str.hashCode());

Throws NullPointerException.

**How put() method works?**

* First it calculates hash Code for the key using hash() method.
* It calls indexFor() method by passing the hash code and length of the table. This method returns the index where this Entry will be stored.

**static** **int** indexFor(**int** h, **int** length) {

**return** h & (length-1);

* It checks whether the key is null or not. If it is null, calls putForNullKey() method which will return 0. The entry will be stored at table[0] because hash code for **null** is 0.
* If not null calculates the hash code and passes it to indexFor() method to calculate the index where new Entry will be stored.
* After getting the index, it checks all keys (in same bucket there may be more than one key present ) present in the linked list at that index ( or bucket). If the key is already present in the linked list, it replaces the old value with new value.
* If the key is not present in the linked list, it appends the specified key-value pair at the end of the linked list.

Let’s an Example-

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

hm.put(**null**, "sanjay");

hm.put("01", "sanjay");

hm.put("01", "Rai");//Since key 01 is already present, sanjay will be replaced by Rai

System.***out***.println("Map Size : "+hm.size());

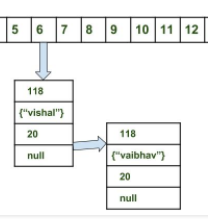
System.***out***.println("Elements in HashMap : "+hm);

//Output

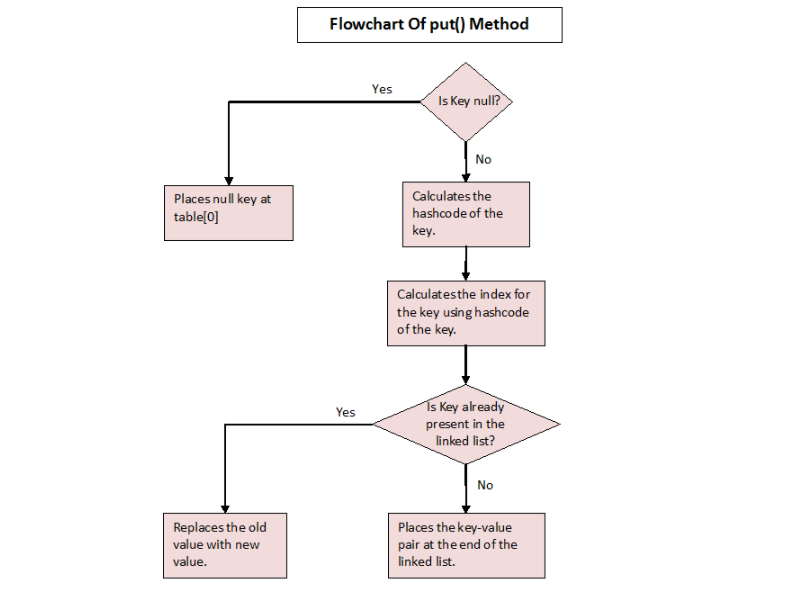
Map Size : 2

Elements in HashMap : {**null**=sanjay, 01=Rai}

* What happens when two Objects have the same hash code. Both will be stored in the same bucket in form of LinkedList. Entry class has attribute next which points the next Object. Below is the diagram. For Vishal and Vaibhav hash code is same.



Let’s understand the same using a flowchart.



**Note - Capacity of each bucket may differ - capacity = number of buckets \* load factor.**

**How get() method works?**

* First checks whether specified key is null or not. If the key is null, it calls getForNullKey() method.
* If the key is not null, hash code of the specified key is calculated.
* indexFor() method is used to find out the index of the specified key in the table[] array.
* After getting index, it will iterate though linked list at that position and checks for the key using equals() method. If the key is found, it returns the value associated with it. otherwise returns null.

What happens during get() method when two objects have the same hash code?

If both the Objects have same hash code then they will be stored at the same index. In our example Vishal and Vaibhav are stored at index 6.

Here Keys.equals() method will be called to identify the correct key and corresponding value will be returned.

**How null key is handled in HashMap? Since equals() and hashCode() are used to store and retrieve values, how does it work in case of the null key?**

The null key is handled specially in HashMap, there are two separate methods for that putForNullKey(V value) and getForNullKey().

**Why do we do Hashing?**

Using this a huge string can be represented as small fixed length value which helps in indexing and searching.

**What is change in Java 8 for HashMap?**

In Java 8 when too many unequal keys produces same hashcode i.e number of elements in a buckets grows beyond a certain limit (TREEIFY\_THRESHOLD=8) content of that bucket switches from LinkedList to Balance Tree which improves the performance.

1. How to Create Immutable Class and Object in Java?

Let’s know what is mutable Object?

If you can modify the Object data and result is stored in the same Object then it is called mutable Object.

Below is the Example-

**package** com.practice.commons;

**import** com.practice.support.Student;

**public** **class** MutableObject {

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

Student student=**new** Student(100, "Sanjay Rai");

System.***out***.println("Object Address Before Modification= "+student);

/\*Modifying the Student Object Data\*/

student.setId(101);

System.***out***.println("Object Address After Modification= "+student);

}

}

Output-

Object Address Before Modification= com.practice.support.Student@70dea4e

Object Address After Modification= com.practice.support.Student@70dea4e

If you have an Object from one class and data cannot be modified, even if you modify result is not stored in the same object i.e a new Object is created ( memory address is changed )is called Immutable Object.

It is quite popular in Java to create Immutable objects to deal with Concurrency.

**In Java all wrapper, String and File classes are immutable**. Now Let’s create user defined Immutable Class.

Rules-

* Declare class as final so that it cannot be extended and subclass cannot override the methods.
* Declare all fields private and final.
* Don’t provide setter methods.
* When exposing methods which modify the state of the class, you must always return a new instance of the class. (student.getAge().setYear(1991);
* Initialize all the fields via a constructor performing deep copy.
* If class holds another mutable object-
* Inside the constructor use a clone copy of the passed argument.
* Always return cloned copy (cloneAge) never return the real object instance (age).

Let’s see an example which follows steps till 3.

Student.java

**package** com.practice.support;

**public** **final** **class** Student {

**private** **final** **int** id;

**private** **final** String name;

**private** **final** Age age;

**public** Student(**int** id, String name,Age age) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.age=age;

}

**public** **int** getId() {

**return** id;

}

**public** String getName() {

**return** name;

}

**public** Age getAge()

{

**return** age;

}

}

ImmutableObject.java

**package** com.practice.commons;

**import** com.practice.support.Age;

**import** com.practice.support.Student;

**public** **class** ImmutableObject {

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

Age age=**new** Age();

age.setDay(10);

age.setMonth(06);

age.setYear(1993);

Student student=**new** Student(100, "Sanjay Rai", age);

System.***out***.println("Age Before modification - "+student.getAge().getYear());

student.getAge().setYear(1991);

System.***out***.println("Age After Modification - "+student.getAge().getYear());

}

}

Output-

Age Before modification - 1993

Age After Modification - 1991

Now Implement step 4 and 5

Change getAge like

**public** Age getAge()

{

Age cloneAge = **new** Age();

cloneAge.setDay(**this**.age.getDay());

cloneAge.setMonth(**this**.age.getMonth());

cloneAge.setYear(**this**.age.getYear());

**return** cloneAge;

}

Now output looks like-

Age Before modification - 1993

Age After Modification – 1993

Immutable Objects are of two types

First does not allow modification-

Integer i=new Integer(10);

Here Integer does not provides any method to modify so it is immutable. Same is applicable for File class also.

Second allows modification, but after modification new object is created example is String class.

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

String str="Sanjay";

str=str.concat(" Rai");

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

}

In Heap there is already an Object containing Sanjay after concat new Object Sanjay Rai is formed.

More Reference - <http://www.informit.com/articles/article.aspx?p=20530>

1. How to Combine , get common and get unique elements from two lists?

Combine – Use union – addAll() method

Duplicate/Common – use intersection – retainAll() method

Unique – use removeAll() method

Complete Code Snippet

List<Integer> list1 = Arrays.*asList*(1, 2, 3, 4);

List<Integer> list2 = Arrays.*asList*(2, 3, 4, 6, 7);

// Prepare a union

List<Integer> union = **new** ArrayList<Integer>(list1);

union.addAll(list2);

System.***out***.println("Combination of Lists- "+union);

// Prepare an intersection

List<Integer> intersection = **new** ArrayList<Integer>(list1);

intersection.retainAll(list2);

System.***out***.println(" Duplicate / Common elements- "+intersection);

// Subtract the intersection from the union

union.removeAll(intersection);

// Print the result

System.***out***.print("Unique elements- ");

**for** (Integer n : union) {

System.***out***.print(n+" ");

Output-

Combination of Lists- [1, 2, 3, 4, 2, 3, 4, 6, 7]

Duplicate / Common elements- [2, 3, 4]

Unique elements- 1 6 7

1. Comparable vs Comparator

/\*Sorting a Primitive Data Type\*/

**int** intArr[]={12,5,45,23,13,89,20};

Arrays.*sort*(intArr);

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

/\*Sorting String Data Type\*/

String strArr[]={"Java","Spring","Hibernate","Jsp","Angular","webservice"};

Arrays.*sort*(strArr);

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

/\*Sorting a List\*/

List techList=Arrays.*asList*("Java","Spring","Hibernate","Jsp","Angular","webservice");

Collections.*sort*(techList);

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

Default order of sorting is ascending.

Output-

[5, 12, 13, 20, 23, 45, 89]

[Angular, Hibernate, Java, Jsp, Spring, webservice]

[Angular, Hibernate, Java, Jsp, Spring, webservice]

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

**Comparable and Comparator**

**Comparable-**

* Present in java.util.lang package
* Calling method Collections.sort(list) method.
* Uses sorting method – int compareTo(Object o1, Object o2)

it is used for natural ordering means object itself must know how it is to be ordered. ( on what attribute it is to be ordered is defined in compareTo() method)

While implementing it class must implement Comparable interface

Employee.java

**package** com.practice.comparable\_comparator;

**public** **class** Employee **implements** Comparable<Employee> {

**private** **int** empId;

**private** String name;

**private** **double** salary;

**private** **int** age;

**public** **int** getEmpId() {

**return** empId;

}

**public** **void** setEmpId(**int** empId) {

**this**.empId = empId;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **double** getSalary() {

**return** salary;

}

**public** **void** setSalary(**double** salary) {

**this**.salary = salary;

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

**public** Employee(**int** empId, String name, **double** salary, **int** age) {

**super**();

**this**.empId = empId;

**this**.name = name;

**this**.salary = salary;

**this**.age = age;

}

@Override

**public** String toString() {

**return** "Employee [empId=" + empId + ", name=" + name + ", salary=" + salary + ", age=" + age + "]";

}

@Override

**public** **int** compareTo(Employee emp) {

**return** **this**.age-emp.age;//Ascending order

emp.age-this.age//Descending order

}

}

ComparableAndComparableApp.java

**package** com.practice.comparable\_comparator;

**import** java.util.Arrays;

**import** java.util.Collections;

**import** java.util.List;

**public** **class** ComparableAndComparableApp {

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

/\*Sorting list of Employee Objects\*/

Employee emp4=**new** Employee(102, "Satya", 35000,30);

Employee emp1=**new** Employee(101, "Sanjay", 26000,25);

Employee emp2=**new** Employee(104, "Uuvraj", 44000,35);

Employee emp3=**new** Employee(103, "Deepayen", 30000,22);

List empList=Arrays.*asList*(emp1,emp2,emp3,emp4);

/\* Collections.sort() method is applicable only for Objects if they implement Comparable interface otherwise compilation issue\*/

Collections.*sort*(empList);

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

}

}

Output- (Sorted based on age)

30000.0-103-Deepayen-22

26000.0-101-Sanjay-25

35000.0-102-Satya-30

44000.0-104-Uuvraj-35

**Comparator Interface**

* Present in java.util.Comparator
* Uses calling method Collection.sort(list,comparator).
* Sorting method – int compare(Object o1, Object o2)

Using comparable object can be sorted based on single attribute but using comparator interface we can create as many as number of attributes comparators externally.

Steps-

* Create Comparator class
* Create new instance of it.
* Pass it in Collections.sort() method.

Employee.java

Exactly same as previous

NameCompare.java

**package** com.practice.comparable\_comparator;

**import** java.util.Comparator;

**public** **class** NameCompare **implements** Comparator<Employee> {

@Override

**public** **int** compare(Employee emp1, Employee emp2) {

**return** emp1.getName().compareTo(emp2.getName());

}

}

SalaryCompare.java

**package** com.practice.comparable\_comparator;

**import** java.util.Comparator;

**public** **class** SalaryCompare **implements** Comparator<Employee> {

@Override

**public** **int** compare(Employee emp1, Employee emp2) {

**if**(emp1.getSalary()<emp2.getSalary())

**return** -1;

**if**(emp1.getSalary()>emp2.getSalary())

**return** 1;

**else**

**return** 0;

}

}

ComparableAndComparableApp.java

**package** com.practice.comparable\_comparator;

**import** java.util.Arrays;

**import** java.util.Collections;

**import** java.util.List;

**public** **class** ComparableAndComparableApp {

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

/\*Sorting list of Employee Objects\*/

Employee emp4=**new** Employee(102, "Satya", 35000,30);

Employee emp1=**new** Employee(101, "Sanjay", 26000,25);

Employee emp2=**new** Employee(104, "Uuvraj", 44000,35);

Employee emp3=**new** Employee(103, "Deepayen", 30000,22);

List<Employee> empList=Arrays.*asList*(emp1,emp2,emp3,emp4);

/\*Sorting based on salary\*/

SalaryCompare salaryCompare=**new** SalaryCompare();

Collections.*sort*(empList, salaryCompare);

System.***out***.println("----------------Sorting Based on Salary------------------------");

**for**(Employee emp: empList)

{

System.***out***.println(emp.getSalary()+"-"+emp.getEmpId()+"-"+emp.getName()+"-"+emp.getAge());

}

System.***out***.println("----------------Sorting Based Name------------------------");

/\*Sorting based on name\*/

NameCompare nameCompare=**new** NameCompare();

Collections.*sort*(empList, nameCompare);

**for**(Employee emp: empList)

{

System.***out***.println(emp.getSalary()+"-"+emp.getEmpId()+"-"+emp.getName()+"-"+emp.getAge());

}

}

}

Output-

----------------Sorting Based on Salary------------------------

26000.0-101-Sanjay-25

30000.0-103-Deepayen-22

35000.0-102-Satya-30

44000.0-104-Uuvraj-35

----------------Sorting Based Name------------------------

30000.0-103-Deepayen-22

26000.0-101-Sanjay-25

35000.0-102-Satya-30

44000.0-104-Uuvraj-35

Here the sorting order is ascending by default

How to make the same in descending?

Just need to change the compare() method.

For Salary

@Override

**public** **int** compare(Employee emp1, Employee emp2) {

**if**(emp1.getSalary()<emp2.getSalary())

**return** 1;

**if**(emp1.getSalary()>emp2.getSalary())

**return** -1;

**else**

**return** 0;

}

For name

@Override

**public** **int** compare(Employee emp1, Employee emp2) {

**return** emp2.getName().compareTo(emp1.getName());

}

Note- If any class implements Comparable interface in Java then collection of that object either List or Array can be sorted automatically by using Collections.sort() or Arrays.sort() method and objects will be sorted based on there natural order defined by CompareTo method

**Difference Between Comparable and Comparator**

Comparable provides single sorting sequence as sorting technique is defined inside bean class while Comparator provides multiple sequence technique as we define as number of comparator as number of elements present in bean class.