

ArrayList

ArrayList is a class which implements List interface. It is widely used because of the functionality and flexibility it offers. Most of the developers choose ArrayList over Array as it's a very good alternative of traditional java arrays.

The issue with arrays is that they are of fixed length so if it is full we cannot add any more elements to it, likewise if there are number of elements gets removed from it the memory consumption would be the same as it doesn't shrink. On the other ArrayList can dynamically grow and shrink as per the need. Apart from these benefits ArrayList class enables us to use predefined methods of it which makes our task easy. Let's see the ArrayList example first then we will discuss it's methods and their usage.

ArrayList Example in Java

Methods of ArrayList class

In the above example we have used methods such as add and remove. However there are number of methods available which can be used directly using object of ArrayList class. Let's discuss few of the important methods.

1) add(Object o): This method adds an object o to the arraylist.

```
obj.add("hello");
```

This statement would add a string hello in the arraylist at last position.

2) add(int index, Object o): It adds the object o to the array list at the given index.

```
obj.add(2, "bye");
```

It will add the string bye to the 2nd index (3rd position as the array list starts with index 0) of array list.

3) remove(Object o): Removes the object o from the ArrayList.

```
obj.remove("Chaitanya");
```

This statement will remove the string "Chaitanya" from the ArrayList.

4) remove(int index): Removes element from a given index.

```
obj.remove(3);
```

It would remove the element of index 3 (4th element of the list – List starts with 0).

5) set(int index, Object o): Used for updating an element. It replaces the element present at the specified index with the object o.

```
obj.set(2, "Tom");
```

It would replace the 3rd element (index =2 is 3rd element) with the value Tom.

6) int indexOf(Object o): Gives the index of the object o. If the element is not found in the list then this method returns the value -1.

```
int pos = obj.indexOf("Tom");
```

This would give the index (position) of the string Tom in the list.

7) Object get(int index): It returns the object of list which is present at the specified index.

```
String str= obj.get(2);
```

Function get would return the string stored at 3rd position (index 2) and would be assigned to the string "str". We have stored the returned value in string variable because in our example we have defined the ArrayList is of String type. If you are having integer array list then the returned value should be stored in an integer variable.

8) int size(): It gives the size of the ArrayList – Number of elements of the list.

```
int numberofitems = obj.size();
```

9) boolean contains(Object o): It checks whether the given object o is present in the array list if its there then it returns true else it returns false.

```
obj.contains("Steve");
```

It would return true if the string "Steve" is present in the list else we would get false.

10) clear(): It is used for removing all the elements of the array list in one go. The below code will remove all the elements of ArrayList whose object is obj.

```
obj.clear();
```

LinkedList

LinkedList is an implementation of List interface. Earlier we learnt about ArrayList class which also implements List Interface. In this tutorial we will see an example of LinkedList along with brief description of it's methods.

Example of LinkedList in Java

Methods of LinkedList class:

For all the examples in the below methods, consider llistobj as a reference for `LinkedList<String>`.

```
LinkedList<String> llistobj = new LinkedList<String>();
```

1) boolean add(Object item): It adds the item at the end of the list.

```
llistobj.add("Hello");
```

It would add the string “Hello” at the end of the linked list.

2) void add(int index, Object item): It adds an item at the given index of the the list.

```
llistobj.add(2, "bye");
```

This will add the string “bye” at the 3rd position(2 index is 3rd position as index starts with 0).

3) boolean addAll(Collection c): It adds all the elements of the specified collection c to the list. It throws NullPointerException if the specified collection is null. Consider the below example –

```
LinkedList<String> llistobj = new LinkedList<String>();  
ArrayList<String> arraylist= new ArrayList<String>();  
arraylist.add("String1");  
arraylist.add("String2");  
llistobj.addAll(arraylist);
```

This piece of code would add all the elements of ArrayList to the LinkedList.

4) boolean addAll(int index, Collection c): It adds all the elements of collection c to the list starting from a give index in the list. It throws NullPointerException if the collection c is null and IndexOutOfBoundsException when the specified index is out of the range.

```
llistobj.add(5, arraylist);
```

It would add all the elements of the ArrayList to the LinkedList starting from position 6 (index 5).

5) void addFirst(Object item): It adds the item (or element) at the first position in the list.

```
llistobj.addFirst("text");
```

It would add the string “text” at the beginning of the list.

6) void addLast(Object item): It inserts the specified item at the end of the list.

```
llistobj.addLast("Chaitanya");
```

This statement will add a string “Chaitanya” at the end position of the linked list.

7) void clear(): It removes all the elements of a list.

```
llistobj.clear();
```

8) Object clone(): It returns the copy of the list.

For e.g. My linkedList has four items: text1, text2, text3 and text4.

```
Object str= llistobj.clone();  
System.out.println(str);
```

Output: The output of above code would be:

```
[text1, text2, text3, text4]
```

9) boolean contains(Object item): It checks whether the given item is present in the list or not. If the item is present then it returns true else false.

```
boolean var = llistobj.contains("TestString");
```

It will check whether the string "TestString" exist in the list or not.

10) Object get(int index): It returns the item of the specified index from the list.

```
Object var = llistobj.get(2);
```

It will fetch the 3rd item from the list.

11) Object getFirst(): It fetches the first item from the list.

```
Object var = llistobj.getFirst();
```

12) Object getLast(): It fetches the last item from the list.

```
Object var= llistobj.getLast();
```

13) int indexOf(Object item): It returns the index of the specified item.

```
llistobj.indexOf("bye");
```

14) int lastIndexOf(Object item): It returns the index of last occurrence of the specified element.

```
int pos = llistobj.lastIndexOf("hello");
```

integer variable pos will be having the index of last occurrence of string "hello".

15) Object poll(): It returns and removes the first item of the list.

```
Object o = llistobj.poll();
```

16) Object pollFirst(): same as poll() method. Removes the first item of the list.

```
Object o = llistobj.pollFirst();
```

17) Object pollLast(): It returns and removes the last element of the list.

```
Object o = llistobj.pollLast();
```

18) Object remove(): It removes the first element of the list.

```
llistobj.remove();
```

19) Object remove(int index): It removes the item from the list which is present at the specified index.

```
llistobj.remove(4);
```

It will remove the 5th element from the list.

20) Object remove(Object obj): It removes the specified object from the list.

```
llistobj.remove("Test Item");
```

21) Object removeFirst(): It removes the first item from the list.

```
llistobj.removeFirst();
```

22) Object removeLast(): It removes the last item of the list.

```
llistobj.removeLast();
```

23) Object removeFirstOccurrence(Object item): It removes the first occurrence of the specified item.

```
llistobj.removeFirstOccurrence("text");
```

It will remove the first occurrence of the string "text" from the list.

24) Object removeLastOccurrence(Object item): It removes the last occurrence of the given element.

```
llistobj.removeLastOccurrence("String1");
```

It will remove the last occurrence of string "String1".

25) Object set(int index, Object item): It updates the item of specified index with the give value.

```
llistobj.set(2, "Test");
```

It will update the 3rd element with the string "Test".

26) int size(): It returns the number of elements of the list.

```
l1.size();
```

ArrayList vs LinkedList

ArrayList and LinkedList both implements List interface and their methods and results are almost identical. However there are few differences between them which make one better over another depending on the requirement.

ArrayList Vs LinkedList

1) Search: ArrayList search operation is pretty fast compared to the LinkedList search operation. `get(int index)` in ArrayList gives the performance of $O(1)$ while LinkedList performance is $O(n)$.

Reason: ArrayList maintains index based system for its elements as it uses array data structure implicitly which makes it faster for searching an element in the list. On the other side LinkedList implements doubly linked list which requires the traversal through all the elements for searching an element.

2) Deletion: LinkedList remove operation gives $O(1)$ performance while ArrayList gives variable performance: $O(n)$ in worst case (while removing first element) and $O(1)$ in best case (While removing last element).

Conclusion: LinkedList element deletion is faster compared to ArrayList.

Reason: LinkedList's each element maintains two pointers (addresses) which points to the both neighbor elements in the list. Hence removal only requires change in the pointer location in the two neighbor nodes (elements) of the node which is going to be removed. While In ArrayList all the elements need to be shifted to fill out the space created by removed element.

3) Inserts Performance: LinkedList add method gives $O(1)$ performance while ArrayList gives $O(n)$ in worst case. Reason is same as explained for remove.

4) Memory Overhead: ArrayList maintains indexes and element data while LinkedList maintains element data and two pointers for neighbor nodes hence the memory consumption is high in LinkedList comparatively.

There are few **similarities** between these classes which are as follows:

Both ArrayList and LinkedList are implementation of List interface.

They both maintain the elements insertion order which means while displaying ArrayList and LinkedList elements the result set would be having the same order in which the elements got inserted into the List.

Both these classes are non-synchronized and can be made synchronized explicitly by using `Collections.synchronizedList` method.

The iterator and listIterator returned by these classes are fail-fast (if list is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove or add methods, the iterator will throw a ConcurrentModificationException).

When to use LinkedList and when to use ArrayList?

1) As explained above the insert and remove operations give good performance ($O(1)$) in LinkedList compared to ArrayList($O(n)$). Hence if there is a requirement of frequent addition and deletion in application then LinkedList is a best choice.

2) Search (get method) operations are fast in ArrayList ($O(1)$) but not in LinkedList ($O(n)$) so If there are less add and remove operations and more search operations requirement, ArrayList would be your best bet.

VECTOR

Vector implements List Interface. Like ArrayList it also maintains insertion order but it is rarely used in non-thread environment as it is synchronized and due to which it gives poor performance in searching, adding, delete and update of its elements.

Three ways to create vector class object:

Method 1:

```
Vector vec = new Vector();
```

It creates an empty Vector with the default initial capacity of 10. It means the Vector will be re-sized when the 11th elements needs to be inserted into the Vector. Note: By default vector doubles its size. i.e. In this case the Vector size would remain 10 till 10 insertions and once we try to insert the 11th element It would become 20 (double of default capacity 10).

Method 2:

```
Syntax: Vector object= new Vector(int initialCapacity)
```

```
Vector vec = new Vector(3);
```

It will create a Vector of initial capacity of 3.

Method 3:

```
Syntax:
```

```
Vector object= new vector(int initialcapacity, capacityIncrement)
```

```
Vector vec= new Vector(4, 6)
```

Here we have provided two arguments. The initial capacity is 4 and capacityIncrement is 6. It means upon insertion of 5th element the size would be 10 ($4+6$) and on 11th insertion it would be 16($10+6$).

Important methods of Vector Class:

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.

```
/*Display Vector elements*/
Enumeration en = vec.elements();
System.out.println("\nElements are:");
while(en.hasMoreElements())
    System.out.print(en.nextElement() + " ");
```

iterator()

```
ListIterator litr = vector.listIterator();
System.out.println("Traversing in Forward Direction:");
while(litr.hasNext())
{
    System.out.println(litr.next());
}
```

listIterator()

```
System.out.println("\nTraversing in Backward Direction:");
while(litr.hasPrevious())
{
    System.out.println(litr.previous());
}
```

Converting Vector to List

```
// Step4: Converting Vector to List
List<String> list = Collections.list(vector.elements());
```

Converting Vector to ArrayList


```
//Converting Vector to ArrayList
ArrayList<String> arraylist = new ArrayList<String>(vector);
```

Vector to String array

```
//Converting Vector to String Array
String[] array = vector.toArray(new String[vector.size()]);
```

HashSet

This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element. This class is not synchronized. However it can be synchronized explicitly like this: Set s = Collections.synchronizedSet(new HashSet(...));

Points to Note about HashSet:

- HashSet doesn't maintain any order, the elements would be returned in any random order.
- HashSet doesn't allow duplicates. If you try to add a duplicate element in HashSet, the old value would be overwritten.
- HashSet allows null values however if you insert more than one nulls it would still return only one null value.
- HashSet is non-synchronized.
- The iterator returned by this class is fail-fast which means iterator would throw ConcurrentModificationException if HashSet has been modified after creation of iterator, by any means except iterator's own remove method.

HashSet Methods:

boolean add(Element e): It adds the element e to the list.

void clear(): It removes all the elements from the list.

Object clone(): This method returns a shallow copy of the HashSet.

boolean contains(Object o): It checks whether the specified Object o is present in the list or not. If the object has been found it returns true else false.

boolean isEmpty(): Returns true if there is no element present in the Set.

int size(): It gives the number of elements of a Set.

boolean remove(Object o): It removes the specified Object o from the Set.

- 1) Using Iterator
- 2) Without using Iterator

TreeSet

TreeSet is similar to HashSet except that it sorts the elements in the ascending order while HashSet doesn't maintain any order. TreeSet allows null element but like HashSet it doesn't allow. Like most of the other collection classes this class is also not synchronized, however it can be synchronized explicitly like this: `SortedSet s = Collections.synchronizedSortedSet(new TreeSet(...));`

HashSet vs TreeSet

- 1) HashSet gives better performance (faster) than TreeSet for the operations like add, remove, contains, size etc. HashSet offers constant time cost while TreeSet offers $\log(n)$ time cost for such operations.
- 2) HashSet does not maintain any order of elements while TreeSet elements are sorted in ascending order by default.

Similarities:

- 1) Both HashSet and TreeSet does not hold duplicate elements, which means both of these are duplicate free.
- 2) If you want a sorted Set then it is better to add elements to HashSet and then convert it into TreeSet rather than creating a TreeSet and adding elements to it.
- 3) Both of these classes are non-synchronized that means they are not thread-safe and should be synchronized explicitly when there is a need of thread-safe operations.

Map

A Map is an object that maps keys to values. A map cannot contain duplicate keys. There are three main implementations of Map interfaces: HashMap, TreeMap, and LinkedHashMap.

HashMap: it makes no guarantees concerning the order of iteration

TreeMap: It stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashMap.

LinkedHashMap: It orders its elements based on the order in which they were inserted into the set (insertion-order).

HashMap

HashMap maintains key and value pairs and often denoted as `HashMap<Key, Value>` or `HashMap<K, V>`. HashMap implements Map interface. HashMap is similar to Hashtable with

two exceptions – HashMap methods are unsynchronized and it allows null key and null values unlike Hashtable. It is used for maintaining key and value mapping.

It is not an ordered collection which means it does not return the keys and values in the same order in which they have been inserted into the HashMap. It neither does any kind of sorting to the stored keys and Values. You must need to import java.util.HashMap or its super class in order to use the HashMap class and methods.

HashMap Class Methods

void clear(): It removes all the key and value pairs from the specified Map.

Object clone(): It returns a copy of all the mappings of a map and used for cloning them into another map.

boolean containsKey(Object key): It is a boolean function which returns true or false based on whether the specified key is found in the map.

boolean containsValue(Object Value): Similar to containsKey() method, however it looks for the specified value instead of key.

Value get(Object key): It returns the value for the specified key.

boolean isEmpty(): It checks whether the map is empty. If there are no key-value mapping present in the map then this function returns true else false.

Set keySet(): It returns the Set of the keys fetched from the map.

value put(Key k, Value v): Inserts key value mapping into the map. Used in the above example.

int size(): Returns the size of the map – Number of key-value mappings.

Collection values(): It returns a collection of values of map.

Value remove(Object key): It removes the key-value pair for the specified key. Used in the above example.

void putAll(Map m): Copies all the elements of a map to the another specified map.

ArrayList vs HashMap in Java

1) Implementation: ArrayList implements List Interface while HashMap is an implementation of Map interface. List and Map are two entirely different collection interfaces.

2) Memory consumption: ArrayList stores the element's value alone and internally maintains the indexes for each element.

```
ArrayList<String> arraylist = new ArrayList<String>();  
//String value is stored in array list  
arraylist.add("Test String");
```

HashMap stores key & value pair. For each value there must be a key associated in HashMap. That clearly shows that memory consumption is high in HashMap compared to the ArrayList.

```
HashMap<Integer, String> hmap= new HashMap<Integer, String>();  
//String value stored along with the key value in hash map
```

```
hmap.put(123, "Test String");
```

3) Order: ArrayList maintains the insertion order while HashMap doesn't. Which means ArrayList returns the list items in the same order in which they got inserted into the list. On the other side HashMap doesn't maintain any order, the returned key-values pairs are not sorted in any kind of order.

4) Duplicates: ArrayList allows duplicate elements but HashMap doesn't allow duplicate keys (It does allow duplicate values).

5) Nulls: ArrayList can have any number of null elements. HashMap allows one null key and any number of null values.

6) get method: In ArrayList we can get the element by specifying the index of it. In HashMap the elements is being fetched by specifying the corresponding key.

HashMap vs Hashtable

What is the Difference between HashMap and Hashtable? This is one of the frequently asked interview questions for Java/J2EE professionals. HashMap and Hashtable both classes implements java.util.Map interface, however there are differences in the way they work and their usage. Here we will discuss the differences between these classes.

HashMap vs Hashtable

1) HashMap is non-synchronized. This means if it's used in multithread environment then more than one thread can access and process the HashMap simultaneously.

Hashtable is synchronized. It ensures that no more than one thread can access the Hashtable at a given moment of time. The thread which works on Hashtable acquires a lock on it to make the other threads wait till its work gets completed.

2) HashMap allows one null key and any number of null values.

Hashtable doesn't allow null keys and null values.

3) HashMap implementation LinkedHashMap maintains the insertion order and TreeMap sorts the mappings based on the ascending order of keys.

Hashtable doesn't guarantee any kind of order. It doesn't maintain the mappings in any particular order.

4) Initially Hashtable was not the part of collection framework it has been made a collection framework member later after being retrofitted to implement the Map interface.

HashMap implements Map interface and is a part of collection framework since the beginning.

5) Another difference between these classes is that the Iterator of the HashMap is a fail-fast and it throws ConcurrentModificationException if any other Thread modifies the map structurally by

adding or removing any element except iterator's own remove() method. In Simple words fail-fast means: When calling iterator.next(), if any modification has been made between the moment the iterator was created and the moment next() is called, a ConcurrentModificationException is immediately thrown.

Enumerator for the Hashtable is not fail-fast.

For e.g.

HashMap:

```
HashMap hm= new HashMap();
....
....
Set keys = hm.keySet();
for (Object key : keys) {
    //it will throw the ConcurrentModificationException here
    hm.put(object & value pair here);
}
```

Hashtable:

```
Hashtable ht= new Hashtable();
....
.....
Enumeration keys = ht.keys();
for (Enumeration en = ht.elements() ; en.hasMoreElements() ; en.nextElement()) {
    //No exception would be thrown here
    ht.put(key & value pair here);
}
```

When to use HashMap and Hashtable?

1) As stated above the main difference between HashMap & Hashtable is synchronization. If there is a need of thread-safe operation then Hashtable can be used as all its methods are synchronized but it's a legacy class and should be avoided as there is nothing about it, which cannot be done by HashMap. For multi-thread environment I would recommend you to use ConcurrentHashMap (Almost similar to Hashtable) or even you can make the HashMap synchronized explicitly (Read here..).

2) Synchronized operation gives poor performance so it should be avoided until unless required. Hence for non-thread environment HashMap should be used without any doubt.

HashSet vs HashMap

HashSet	HashMap
---------	---------

HashSet class implements the Set interface	HashMap class implements the Map interface
In HashSet we store objects(elements or values) e.g. If we have a HashSet of string elements then it could depict a set of HashSet elements: {"Hello", "Hi", "Bye", "Run"}	HashMap is used for storing key & value pairs. In short it maintains the mapping of key & value (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This is how you could represent HashMap elements if it has integer key and value of String type: e.g. {1->"Hello", 2->"Hi", 3->"Bye", 4->"Run"}
HashSet does not allow duplicate elements that means you can not store duplicate values in HashSet.	HashMap does not allow duplicate keys however it allows to have duplicate values.
HashSet permits to have a single null value.	HashMap permits single null key and any number of null values.

Similarities:

1) Both HashMap and HashSet are not synchronized which means they are not suitable for thread-safe operations until unless synchronized explicitly. This is how you can synchronize them explicitly:

HashSet:

```
Set s = Collections.synchronizedSet(new HashSet(...));
```

HashMap:

```
Map m = Collections.synchronizedMap(new HashMap(...));
```

2) Both of these classes do not guarantee that the order of their elements will remain constant over time.

3) If you look at the source code of HashSet then you may find that it is backed up by a HashMap. So basically it internally uses a HashMap for all of its operations.

4) They both provide constant time performance for basic operations such as adding, removing element etc.

Java Enum

An enum is a special type of data type which is basically a collection (set) of constants. In this tutorial we will learn how to use enums in Java and what are the possible scenarios where we can use them.

This is how we define Enum

```
public enum Directions{  
    EAST,  
    WEST,  
    NORTH,  
    SOUTH  
}
```

Here we have a variable Directions of enum type, which is a collection of four constants EAST, WEST, NORTH and SOUTH.

How to assign value to a enum type?

```
Directions dir = Directions.NORTH;
```

The variable dir is of type Directions (that is a enum type). This variable can take any value, out of the possible four values (EAST, WEST, NORTH, SOUTH). In this case it is set to NORTH.

Use of Enum types in if-else statements

This is how we can use an enum variable in a if-else logic.

```
/* You can assign any value here out of  
 * EAST, WEST, NORTH, SOUTH. Just for the  
 * sake of example, I'm assigning to NORTH  
 */
```

```
Directions dir = Directions.NORTH;
```

```
if(dir == Directions.EAST) {  
    // Do something. Write your logic  
} else if(dir == Directions.WEST) {  
    // Do something else  
} else if(dir == Directions.NORTH) {  
    // Do something  
} else {  
    /* Do Something. Write logic for  
     * the remaining constant SOUTH  
     */  
}
```

Important points to Note:

- 1) While defining Enums, the constants should be declared first, prior to any fields or methods.
- 2) When there are fields and methods declared inside Enum, the list of enum constants must end with a semicolon(;).

How HashMap works in Java

Most common interview questions are "How HashMap works in java", "How get and put method of HashMap work internally". Here I am trying to explain internal functionality with an easy example. Rather than going through theory, we will start with example first, so that you will get better understanding and then we will see how get and put function work in java.

Lets take a very simple example. I have a Country class, we are going to use Country class object as key and its capital name(string) as value. Below example will help you to understand, how these key value pair will be stored in hashmap.

1. Country.java

```
public class Country {  
  
    String name;  
    long population;  
  
    public Country(String name, long population) {  
        super();  
        this.name = name;  
        this.population = population;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public long getPopulation() {  
        return population;  
    }  
    public void setPopulation(long population) {  
        this.population = population;  
    }  
}
```

// If length of name in country object is even then return 31(any random number) and if odd then return 95(any random number).

// This is not a good practice to generate hashCode as below method but I am doing so to give better and easy understanding of hashmap.

```
@Override  
public int hashCode() {  
    if(this.name.length()%2==0)  
        return 31;  
    else  
        return 95;  
}  
@Override  
public boolean equals(Object obj) {
```



```

Country other = (Country) obj;
if (name.equalsIgnoreCase((other.name)))
    return true;
return false;
}

}

```

If you want to understand more about hashCode and equals method of object, you may refer hashCode() and equals() method in java

2. HashMapStructure.java(main class)

```

import java.util.HashMap;
import java.util.Iterator;

public class HashMapStructure {

    public static void main(String[] args) {

        Country india=new Country("India",1000);
        Country japan=new Country("Japan",10000);

        Country france=new Country("France",2000);
        Country russia=new Country("Russia",20000);

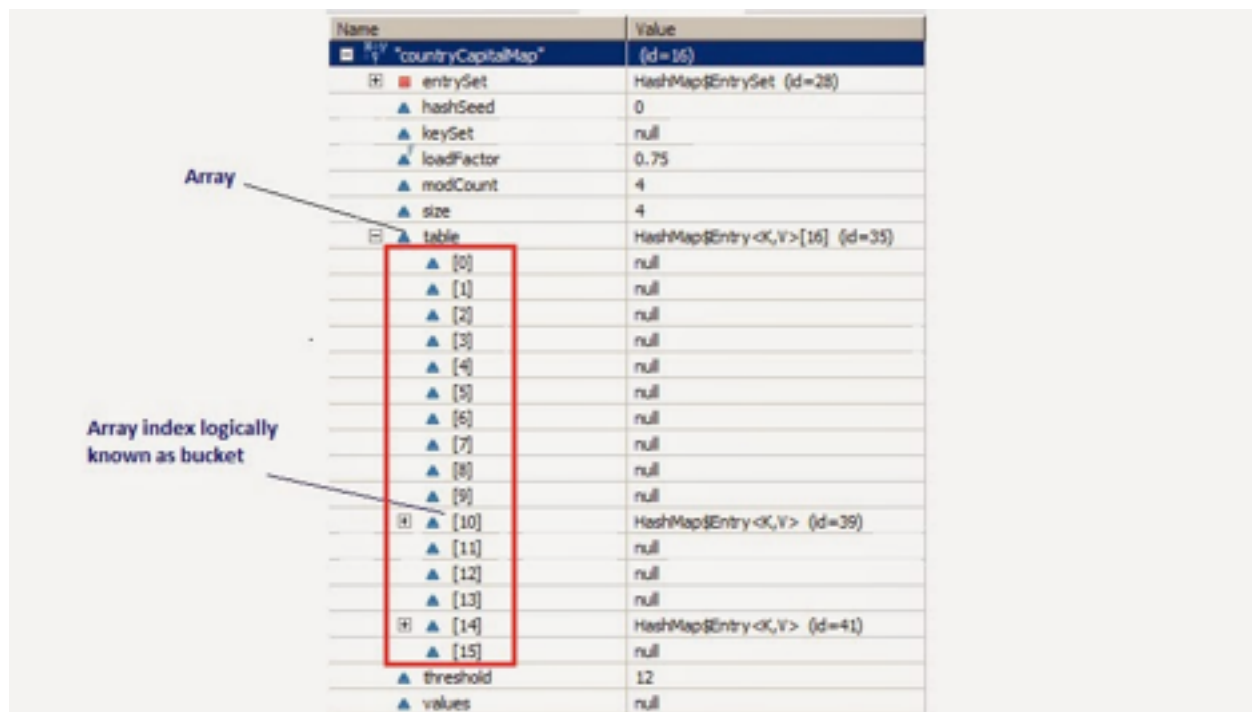
        HashMap<Country, String> countryCapitalMap=new HashMap<Country,String>();
        countryCapitalMap.put(india,"Delhi");
        countryCapitalMap.put(japan,"Tokyo");
        countryCapitalMap.put(france,"Paris");
        countryCapitalMap.put(russia,"Moscow");

        Iterator<Country> countryCapitalIter=countryCapitalMap.keySet().iterator();//put debug
point at this line
        while(countryCapitalIter.hasNext())
        {
            Country countryObj=countryCapitalIter.next();
            String capital=countryCapitalMap.get(countryObj);
            System.out.println(countryObj.getName()+"----"+capital);
        }
    }

}

```

Now put debug point at line 23 and right click on project->debug as-> java application. Program will stop execution at line 23 then right click on countryCapitalMap then select watch. You will be able to see structure as below.



Name	Value
countryCapitalMap	(id=16)
entrySet	HashMap\$EntrySet (id=28)
hashSeed	0
keySet	null
loadFactor	0.75
modCount	4
size	4
table	HashMap\$Entry<K,V>[16] (id=35)
[0]	null
[1]	null
[2]	null
[3]	null
[4]	null
[5]	null
[6]	null
[7]	null
[8]	null
[9]	null
[10]	HashMap\$Entry<K,V> (id=39)
[11]	null
[12]	null
[13]	null
[14]	HashMap\$Entry<K,V> (id=41)
[15]	null
threshold	12
values	null

Now From above diagram, you can observe following points

There is an Entry[] array called table which has size 16.

This table stores Entry class's object. HashMap class has a inner class called Entry. This Entry have key value as instance variable. Lets see structure of entry class Entry Structure.

```

static class Entry implements Map.Entry
{
    final K key;
    V value;
    Entry next;
    final int hash;
    ...//More code goes here
}

```

Whenever we try to put any key value pair in hashmap, Entry class object is instantiated for key value and that object will be stored in above mentioned Entry[] (table). Now you must be wondering, where will above created Entry object get stored (exact position in table). The answer is, hash code is calculated for a key by calling hashCode() method. This hashcode is used to calculate index for above Entry[] table.

Now, If you see at array index 10 in above diagram, It has an Entry object named HashMap\$Entry.

We have put 4 key-values in hashmap but it seems to have only 2!!!! This is because if two objects have same hashcode, they will be stored at same index. Now question arises how? It stores objects in a form of LinkedList (logically).

So how hashcode of above country key-value pairs are calculated.

HashCode for Japan = 95 as its length is odd.

HashCode for India = 95 as its length is odd

HashCode for Russia = 31 as its length is even.

HashCode for France = 31 as its length is even.

0	null
10	HashMap\$Entry<K,V> (id=33)
hash	90
key	Country (id=37)
name	"Japan" (id=45)
population	10000
next	HashMap\$Entry<K,V> (id=39)
hash	90
key	Country (id=47)
next	null
value	"Delhi" (id=48)
value	"Tokyo" (id=40)
11	null
12	null
13	null
14	HashMap\$Entry<K,V> (id=35)
15	null

Entry<Country(Japan),Tokyo> ->
next=Entry<Country(India),Delhi>

If two objects have same hashcode then they will be stored in a form of linkedlist at that index

hashCode() and equals() in Java

These methods can be found in the Object class and hence available to all java classes. Using these two methods, an object can be stored or retrieved from a Hashtable, HashMap or HashSet.

```
hashCode()  
equals()
```

hashCode():

You might know if you put entry in HashMap, first hashCode is calculated and this hashCode used to find bucket(index) where this entry will get stored in hashMap. You can read more at [How hashMap works in java](#). What if you don't override hashCode method, it will return integer representation of memory address.

equals():

You have to override equals method, when you want to define equality between two object. If you don't override this method, it will check for reference equality(==) i.e. if two reference refers to same object or not

Lets override default implementation of hashCode() and equals():

You don't have to always override these methods, but lets say you want to define equality of country object based on name, then you need to override equals method and if you are overriding equals method, you should override hashCode method too. Below example will make it clear.

Lets see with the help of example. We have a class called Country

1. Country.java

```
public class Country {  
  
    String name;  
    long population;  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public long getPopulation() {  
        return population;  
    }  
    public void setPopulation(long population) {  
        this.population = population;  
    }  
}
```

This country class have two basic attributes- name and population.

Now create a class called "EqualityCheckMain.java"

```
public class EqualityCheckMain {  
  
    public static void main(String[] args) {  
  
        Country india1=new Country();  
        india1.setName("India");  
        Country india2=new Country();  
        india2.setName("India");  
        System.out.println("Is india1 is equal to india2:" +india1.equals(india2));  
    }  
}
```

When you run above program, you will get following output

Is india1 is equal to india2:false

In above program, we have created two different objects and set their name attribute to "india". Because both references india1 and india2 are pointing to different object, as default implementation of equals check for ==,equals method is returning false. In real life, it should have return true because no two countries can have same name.

Now lets override equals and return true if two country's name are same.

Add this method to above country class:

```
@Override  
public boolean equals(Object obj) {  
    if (this == obj)  
        return true;  
    if (obj == null)  
        return false;  
    if (getClass() != obj.getClass())  
        return false;  
    Country other = (Country) obj;  
    if (name == null) {  
        if (other.name != null)  
            return false;  
    } else if (!name.equals(other.name))  
        return false;  
    return true;  
}
```

and now run EqualityCheckMain.java again

You will get following output:

Is india1 is equal to india2:true

Now this is because overridden equals method return true if two country have same name.

One thing to remember here, signature of equals method should be same as above.

Lets put this Country objects in hashmap:

Here we are going to use Country class object as key and its capital name(string) as value in HashMap.

```
import java.util.HashMap;
import java.util.Iterator;

public class HashMapEqualityCheckMain {

    public static void main(String[] args) {
        HashMap<Country,String> countryCapitalMap=new HashMap<Country,String>();
        Country india1=new Country();
        india1.setName("India");
        Country india2=new Country();
        india2.setName("India");

        countryCapitalMap.put(india1, "Delhi");
        countryCapitalMap.put(india2, "Delhi");

        Iterator<Country> countryCapitalIter=countryCapitalMap.keySet().iterator();
        while(countryCapitalIter.hasNext())
        {
            Country countryObj=countryCapitalIter.next();
            String capital=countryCapitalMap.get(countryObj);
            System.out.println("Capital of "+ countryObj.getName()+"----"+capital);
        }
    }
}
```

When you run above program, you will see following output:

Capital of India----Delhi

Capital of India----Delhi

Now you must be wondering even through two objects are equal why HashMap contains two key value pair instead of one. This is because First HashMap uses hashCode to find bucket for that key object, if hashCodes are same then only it checks for equals method and because hashCode for above two country objects uses default hashCode method, Both will have different memory address hence different hashCode.

Now lets override hashCode method.Add following method to Country class

```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((name == null) ? 0 : name.hashCode());
    return result;
}
```

Now run HashMapEqualityCheckMain.java again
You will see following output:

Capital of India----Delhi

So now hashCode for above two objects india1 and india2 are same, so Both will be point to same bucket,now equals method will be used to compare them which will return true. This is the reason java doc says "if you override equals() method then you must override hashCode() method"

hashCode() and equals() contracts:

equals():

The equals method implements an equivalence relation on non-null object references:

It is reflexive: for any non-null reference value x, x.equals(x) should return true.

It is symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.

It is transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.

It is consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.

For any non-null reference value x, x.equals(null) should return false.

hashCode():

Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.

If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.

It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct

integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

Key points to remember:

If you are overriding equals method then you should override hashCode() also.

If two objects are equal then they must have same hashCode.

If two objects have same hashCode then they may or may not be equal

Always use same attributes to generate equals and hashCode as in our case we have used name.

