**Java Collections: Internal Working**

**ArrayList**

ArrayList is a Resizable-array implementation of the List interface.

Internally ArrayList class uses an array of Object class to store its elements.

When initializing an ArrayList you can provide initial capacity then the array would be of the size provided as initial capacity.

If initial capacity is not specified then default capacity is used to create an array. Default capacity is 10.

When an element is added to an ArrayList it first verifies whether it can accommodate the new element or it needs to grow, in case capacity has to be increased then the new capacity is calculated which is 50% more than the old capacity and the array is increased by that much capacity.

When elements are removed from an ArrayList space created by the removal of an element has to be filled in the underlying array. That is done by shifting any subsequent elements to the left.

ArrayList works on the principle of creating an array and adding elements to it.

ArrayList class has a member variable elementData which is an Object array;

Object [] elementData;

When we do **List l = new ArrayList ();**

the array elementData is initialized with a size of 10.

**add (E element)**

When a new element is added the capacity of the array elementData is checked and if it is completely filled that is all element 10 are filled a new array is created with a new capacity by using Arrays.copyOf. If the elementData array is not exhausted the new element is added in the array.

So adding an element in a array may take more time as a completely new array needs to be created with greater capacity and the data in the old array is transferred into the new array.

**add (index i, E element)**

On adding an element at a particular index in ArrayList, ArrayList checks if an element is already present at that index. If no than the element passed in add () is added at that index, otherwise a new array is created with the index kept vacant and the remaining element shifted to right.

For Eg:

List<Integer> l = new ArrayList<Integer> ();

l.add(1);

l.add(2);

l.add(1,3);

l.add(4);

for(int i:l){

System.out.println(i);

}

Output

1

3

2

4

Here above we are trying to add 3 and position 1, since position 1 already has value '2'. A new array is created with value at index 1 kept vacant and the remaining elements are shifted to right. Than the element 3 is added at index 1.

**get (int index)**

The element present at that index in that array is returned. This is very fast.

When to use ArrayList When the requirement is fetch data frequently and adding data is one time activity. When not to use ArrayList When the list is updated frequently to understand ArrayList in more detail follow link Custom Array List. Here a custom Arraylist is created

With basic add and get operations.

**set (int index, Object element)**

index - index of the element to replace

element - element to be stored at the specified position

Replaces the element at the specified position in this list with the specified element

If the index is out of range (index < 0 || index >= size()) the this method throws

IndexOutOfBoundsException

How ArrayList Works Internally in Java

ArrayList arguably would be the most used collection along with the HashMap. Many of us programmers whip up code everyday which contains atleast one of these data structures to hold objects. I have already discussed how HashMap works internally in Java, in this post I'll try to explain how ArrayList internally works in Java.

As most of us would already be knowing that ArrayList is a Resizable-array implementation of the List interface i.e. ArrayList grows dynamically as the elements are added to it. So let's see how ArrayList is internally implemented; what is the backing data structure for an ArrayList, how it grows dynamically and ensures that there is always room to add elements. Because of all these side questions it is also a very important Java Collections interview question.

Note - Code of ArrayList used here for reference is from Java 8.

Where does ArrayList store elements

Basic data structure used by ArrayList to store objects is an array of Object class, which is defined as follows -

transient Object[] elementData;

I am sure many of you would be thinking why transient and how about serializing an ArrayList then?

ArrayList provides its own version of readObject and writeObject methods so no problem in serializing an ArrayList and that is the reason, I think, of making this Object array as transient.

What happens when ArrayList is created

ArrayList class provides 3 constructors to create an array list.

public ArrayList(int initialCapacity) - When this constructor is used we can provide some initial capacity rather than depending on the default capacity as defined in the ArrayList class.

As example -

List<String> myList = new ArrayList<String>(7); Code in the ArrayList class is as -

public ArrayList(int initialCapacity) {

if (initialCapacity > 0) {

this.elementData = new Object[initialCapacity];

} else if (initialCapacity == 0) {

this.elementData = EMPTY\_ELEMENTDATA;

} else {

throw new IllegalArgumentException("Illegal Capacity: "+

initialCapacity);

}

}

Where EMPTY\_ELEMENTDATA is defined as -

private static final Object[] EMPTY\_ELEMENTDATA = {};

It is easy to see here, that if provided capacity is greater than zero then the elementData array will be created with that capacity, in case provided capacity is zero then elementData array is initialized with an empty Object array. In that case ArrayList will grow when first element is added.

public ArrayList() - In case default constructor is used i.e. ArrayList is created like -

myList = new ArrayList();

Code in the ArrayList class is as -

public ArrayList() {

this.elementData = DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA;

}

Where DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA is defined as

/\*\*

\* Shared empty array instance used for default sized empty instances. We

\* distinguish this from EMPTY\_ELEMENTDATA to know how much to inflate when

\* first element is added.

\*/

private static final Object[] DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA = {};

So you can see initially it will be initialized with an empty array, it will grow only when first element is added to the list.

public ArrayList(Collection<? extends E> c) - If we want to construct a list containing the elements of the specified collection we can use this constructor. In this constructor implementation checks for the length of the collection passed as parameter, if length is greater than zero then Arrays.copyOf method is used to copy the collection to the elementData array.

elementData = Arrays.copyOf(elementData, size, Object[].class);

How does ArrayList grow dynamically

When we add an element to an ArrayList it first verifies whether it has that much capacity in the array to store new element or not, in case there is not then the new capacity is calculated which is 50% more than the old capacity and the array is increased by that much capacity (Actually uses Arrays.copyOf which returns the original array increased to the new length).

Code in the ArrayList implementation is like this-

public boolean add(E e) {

ensureCapacityInternal(size + 1); // Increments modCount!!

elementData[size++] = e;

return true;

}

private void ensureCapacityInternal(int minCapacity) {

if (elementData == DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA) {

minCapacity = Math.max(DEFAULT\_CAPACITY, minCapacity);

}

ensureExplicitCapacity(minCapacity);

}

Where DEFAULT\_CAPACITY is defined as -

private static final int DEFAULT\_CAPACITY = 10;

private void ensureExplicitCapacity(int minCapacity) {

modCount++;

// overflow-conscious code

if (minCapacity - elementData.length > 0)

grow(minCapacity);

}

You can see here it is determined if there is a need to increase the size of the array, if yes then grow method is called.

private void grow(int minCapacity) {

// overflow-conscious code

int oldCapacity = elementData.length;

int newCapacity = oldCapacity + (oldCapacity >> 1);

if (newCapacity - minCapacity < 0)

newCapacity = minCapacity;

if (newCapacity - MAX\_ARRAY\_SIZE > 0)

newCapacity = hugeCapacity(minCapacity);

// minCapacity is usually close to size, so this is a win:

elementData = Arrays.copyOf(elementData, newCapacity);

}

Note that till Java 6 the new capacity calculation used to be like this -

int newCapacity = (oldCapacity \* 3)/2 + 1;

Which is changed in Java 7 to use right shift operator. With right shift operator also it will grow by 50% of old capacity.

Let's see it with the help of a small program

public class Test {

public static void main(String args[]) {

int a = 10;

System.out.println(a>>1);

}

}

Output

5

If the default capacity was 10 then

int newCapacity = oldCapacity + (oldCapacity >> 1);

will return 15.

What happens when element is removed

When elements are removed from an ArrayList using either remove(int i) (i.e using index) or remove(Object o), gap created by the removal of an element has to be filled in the underlying array. That is done by Shifting any subsequent elements to the left (subtracts one from their indices). System.arrayCopy method is used for that.

System.arraycopy(elementData, index+1, elementData, index, numMoved);

Here index+1 is the source position and index is the destination position. Since element at the position index is removed so elements starting from index+1 are copied to destination starting from index.

**HashMap**

How a HashMap works internally has become a popular question in almost all the interview. As almost everybody knows how to use a HashMap or the [**difference between HashMap and Hashtable**](http://www.javainterviewpoint.com/differences-betwen-hashmap-hashtable/). However, many people fail when the question is "How does a HashMap work internally?"

So the answer to the question is that it works based **on the hashing principle,** but it is not as simple as it sounds. Hashing is the mechanism of assigning unique code to a variable or attribute using an algorithm to enable easy retrieval. A true hashing mechanism should always return the same hashCode() when it is applied to the same object.

Then comes the question, "How does hashing help in storing and retrieving the value in HashMap?" Many will say that the value will be stored in the bucket and retrieved using the key. If you think that is how it works then you are absolutely wrong. To prove it, let's take a look at the HashMap class:

/\*\*

\* The table, resized as necessary. Length MUST Always be a power of two.

\*/

transient Entry[] table;

So what is the use of Entry[] in a HashMap for? Because the HashMap stores the Objects as **Entry instances,**not as**key and value**

## What Is Entry Class?

HashMap has an inner class called an Entry Class which holds the key and values. And there is also something called next, which you will get to know a bit later.

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

{

final K key;

V value;

Entry<K,V> next;

final int hash;

........

}

You know that the HashMap stores the Entry instances in an array and not as key-value pairs. In order to store a value, you will use the put() method of the HashMap, so now let's dig into that and see how it works.

## How Does Put() Method Work Internally?

The Code [**implementation of the put() method**](https://dzone.com/articles/www.javainterviewpoint.com/) will look like this:

public V put(K key, V value)

{

if (key == null)

return putForNullKey(value);

int hash = hash(key.hashCode());

int i = indexFor(hash, table.length);

for (Entry<K,V> e = table[i]; e != null; e = e.next)

{

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

{

V oldValue = e.value;

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

modCount++;

addEntry(hash, key, value, i);

return null;

}

* First, it checks if the key given is null or not. If the given key is null, it will be stored in the zero position, as the hashcode of null will be zero.
* Then it applies the hashcode to the key **.hashCode()** by calling the hashcode method. In order to get the value within the limits of an array, the hash(key.hashCode()) is called, which performs some shifting operations on the hashcode.
* The **indexFor()** method is used to get the exact location to store the Entry object.
* Then comes the most important part what happens if two different object has the same hashcode ( eg : Aa,BB will have the same hashcode) and will it be stored in the same bucket. To handle this let's think of the LinkedList in data structure it will have a next attribute which will always point to the next object. The same way the next attribute in the Entry class points to the next object. Using this different objects with the same hashcode will be placed next to each other.
* In the case of the Collision, the HashMap checks for the value of the next attribute if it is **null** it inserts the Entry object in that location, if next attribute is not null then it keeps the loop running till next attribute is null then stores the Entry object there.

## How are Duplicate Keys Prevented in HashMap?

As we all know, HashMap doesn't allow duplicate keys, even though when we insert the same key with different values, only the latest value is returned.

import java.util.HashMap;

import java.util.Map;

public class HashMapEg

{

public static void main(String[] args)

{

Map map = new HashMap();

map.put(1,"sam");

map.put(1,"Ian");

map.put(1,"Scott");

map.put(null,"asdf");

System.out.println(map);

}

}

For the above code, you will get the output as  {null=asdf, 1=Scott} ,  as the values sam  and Ian  will be replaced by Scott. So, how does this happen?

All the Entry Objects in the LinkedList will have the same hashcode, but HashMap uses equals() . This method checks the equality, so if **key.equals(k)**is true, then it will replace the value object inside the Entry class and not the key. So this way it prevents the duplicate key from being inserted.

## How Does Get() Method Work Internally?

Almost the same logic as applied in the put() method will be used to retrieve the value.

public V get(Object key)

{

if (key == null)

return getForNullKey();

int hash = hash(key.hashCode());

for (Entry<K,V> e = table[indexFor(hash, table.length)];e != null;e = e.next)

{

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

return e.value;

}

return null;

}

* First, it gets the hash code of the key object, which is passed, and finds the bucket location.
* If the correct bucket is found, it returns the value (e.value)
* If no match is found, it returns null.

## What Happens If Two Keys Have the Same Hashcode?

The same collision resolution mechanism will be used here. **key.equals(k)**will check until it is true, and if it is true, it returns the value of it.

Hope this article clarifies the troublesome HashMap internal mechanism. Happy Learning !! :)