Internals of Java Data Structures

**About ArrayList**

**How to get the capacity of ArrayList**

//How to get the capcity of ArrayList using reflection

**static** **int** getCapacity(ArrayList<?> list) **throws** Exception {

Field dataField = ArrayList.**class**.getDeclaredField("elementData");

dataField.setAccessible(**true**);

**return** ((Object[]) dataField.get(list)).length;

}

**How will you get deep copy and shallow of ArrayList**

Shallow copy can be obtained using list.clone() method.

Deep copy can be achieved using serializing and deserializing

ByteArrayOutputStream bos = **new** ByteArrayOutputStream();

ObjectOutputStream o = **new** ObjectOutputStream(bos);

o.writeObject(list);

**byte**[] yourBytes = bos.toByteArray();

InputStream in = **new** ByteArrayInputStream(yourBytes);

ObjectInputStream oin = **new** ObjectInputStream(in);

ArrayList<Employee> deepCopyList = (ArrayList<Employee>) oin.readObject();

**How arrayList grows internally**

**As per JDK 7**

**int oldCapacity = elementData.length;**

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

**elementData = Arrays.copyOf(elementData, newCapacity);**

public void ensureCapacity(int minCapacity) {

modCount++;

int oldCapacity = elementData.length;

if (minCapacity > oldCapacity) {

Object oldData[] = elementData;

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

if (newCapacity < minCapacity)

newCapacity = minCapacity;

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

elementData = Arrays.copyOf(elementData, newCapacity);

}

}

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);

}

**How hashcode is calculated in case of ArrayList ?**

The code is given below. It is the part of AbtractList.

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

**int** hashCode = 1;

**for** (E e : **this**)

hashCode = 31\*hashCode + (e==**null** ? 0 : e.hashCode());

**return** hashCode;

}

**Basically it calculates the hashcode of each object present inside the List**.

**How equals method works for ArrayList**

The code is given below. It is the part of AbtractList.

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

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

**return** **true**;

**if** (!(o **instanceof** List))

**return** **false**;

ListIterator<E> e1 = listIterator();

ListIterator e2 = ((List) o).listIterator();

**while** (e1.hasNext() && e2.hasNext()) {

E o1 = e1.next();

Object o2 = e2.next();

**if** (!(o1==**null** ? o2==**null** : o1.equals(o2)))

**return** **false**;

}

**return** !(e1.hasNext() || e2.hasNext());

}

**What is difference between System.arrayCopy and Arrays.copyOf ?**

The difference is that Arrays.copyOf does not only copy elements, it also creates a new array. System.arrayCopy copies into an existing array. Arrays.copyOf also internally uses System.arrayCopy.

**How Serialization and Deserialization happens inside ArrayList**

The code is given below.

**private** **void** writeObject(java.io.ObjectOutputStream s)

**throws** java.io.IOException{

// Write out element count, and any hidden stuff

**int** expectedModCount = modCount;

s.defaultWriteObject();

// Write out size as capacity for behavioural compatibility with clone()

s.writeInt(size);

// Write out all elements in the proper order.

**for** (**int** i=0; i<size; i++) {

s.writeObject(elementData[i]);

}

**if** (modCount != expectedModCount) {

**throw** **new** ConcurrentModificationException();

}

}

/\*\*

\* Reconstitute the <tt>ArrayList</tt> instance from a stream (that is,

\* deserialize it).

\*/

**private** **void** readObject(java.io.ObjectInputStream s)

**throws** java.io.IOException, ClassNotFoundException {

elementData = ***EMPTY\_ELEMENTDATA***;

// Read in size, and any hidden stuff

s.defaultReadObject();

// Read in capacity

s.readInt(); // ignored

**if** (size > 0) {

// be like clone(), allocate array based upon size not capacity

ensureCapacityInternal(size);

Object[] a = elementData;

// Read in all elements in the proper order.

**for** (**int** i=0; i<size; i++) {

a[i] = s.readObject();

}

}

}

Example of Encapsulation. ArrayList maintains the following buffer statement.

/\*\*

\* The array buffer into which the elements of the ArrayList are stored.

\* The capacity of the ArrayList is the length of this array buffer. Any

\* empty ArrayList with elementData == EMPTY\_ELEMENTDATA will be expanded to

\* DEFAULT\_CAPACITY when the first element is added.

\*/

**private** **transient** Object[] elementData;

There is not set() or get() method for this to ensure security.

**How contains() method works in ArrayList ? In case of null value, how ArrayList’s contains method behaves ?**

The code snippet is given below.

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

**return** indexOf(o) >= 0;

}

**public** **int** indexOf(Object o) {

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

**for** (**int** i = 0; i < size; i++)

**if** (elementData[i]==**null**)

**return** i;

} **else** {

**for** (**int** i = 0; i < size; i++)

**if** (o.equals(elementData[i]))

**return** i;

}

**return** -1;

}

**As you see from the above code, if the object is null or the ArrayList contains many null object, the first null object is obtained**, besides it checks the object’s equality. It means we have to override the equals method for the objects to store in the ArrayList.

If do not override equals method, list.contains() return false as shown below.

ArrayList<Employee> list = **new** ArrayList<Employee>();

Employee emp1 = **new** Employee("A");

emp1.setAge(11);

list.add(emp1);

Employee emp2 = **new** Employee("B");

emp2.setAge(12);

list.add(emp2);

Employee emp3 = **new** Employee("A");

emp3.setAge(11);

System.***out***.println(list.contains(emp3));// It returns false

To make the above statement true, we have to override the equals() method. This is the same thing for LinkedList.

**\*\*** **In case LinkedList, the add() methods adds the object in the last of the LinkedList as shown from the code. Appends the specified element to the end of this list.**

**public** **boolean** add(E e) {

linkLast(e);

**return** **true**;

}

**How peek() method works in LinkedList ?**

The code is given below. Retrieves, but does not remove, the head (first element) of this list.

**public** E peek() {

**final** Node<E> f = first;

**return** (f == **null**) ? **null** : f.item;

}

**How element() method works in LinkedList ?**

The code is given below. Retrieves, but does not remove, the head (first element) of this list.

**public** E element() {

**return** getFirst();

}

**How remove() works in LinkedList ?**

The code is given below. Retrieves and removes the head (first element) of this list.

**public** E remove() {

**return** removeFirst();

}

**How offer() method works in LinkedList ?**

The code is given below. Adds the specified element as the tail (last element) of this list.

**public** **boolean** offer(E e) {

**return** add(e);

}

**How clone() method works in LinkedList ?**

The code is given below. Returns a shallow copy of LinkedList.

**public** Object clone() {

LinkedList<E> clone = superClone();

// Put clone into "virgin" state

clone.first = clone.last = **null**;

clone.size = 0;

clone.modCount = 0;

// Initialize clone with our elements

**for** (Node<E> x = first; x != **null**; x = x.next)

clone.add(x.item);

**return** clone;

}

@SuppressWarnings("unchecked")

**private** LinkedList<E> superClone() {

**try** {

**return** (LinkedList<E>) **super**.clone();

} **catch** (CloneNotSupportedException e) {

**throw** **new** InternalError();

}

}

**How Serialization and Deserialization happens in LinkedList?**

The code is given below.

**private** **void** writeObject(java.io.ObjectOutputStream s)

**throws** java.io.IOException {

// Write out any hidden serialization magic

s.defaultWriteObject();

// Write out size

s.writeInt(size);

// Write out all elements in the proper order.

**for** (Node<E> x = first; x != **null**; x = x.next)

s.writeObject(x.item);

}

@SuppressWarnings("unchecked")

**private** **void** readObject(java.io.ObjectInputStream s)

**throws** java.io.IOException, ClassNotFoundException {

// Read in any hidden serialization magic

s.defaultReadObject();

// Read in size

**int** size = s.readInt();

// Read in all elements in the proper order.

**for** (**int** i = 0; i < size; i++)

linkLast((E)s.readObject());

}

**\*\* The hashhcode() and equals() method of LinkedList are same as that of AbstractList which are used in ArrayList.**

**Internals of HashSet**

HashSet internally uses HashMap as the code is given below.

**private** **transient** HashMap<E,Object> map;

**private** **static** **final** Object ***PRESENT*** = **new** Object();

The internal HashMap used “PRESENT” as seen above as a dummy object.

In case of LinkedHashSet, the constructor is given below.

**HashSet(int initialCapacity, float loadFactor, boolean dummy) {**

**map = new LinkedHashMap<>(initialCapacity, loadFactor);**

**}**

**The question may be how LinkedHashSet works internally. The answer is LinkedHashSet internally uses LinkedHashMap.**

**The following methods are given about how HashSet internally works.**

**To get size :: map.size();**

**To check hashSet is empty : map.isEmpty();**

**To check whether hashset contains an object : map.containsKey(o);**

**To add object to hashset : map.put(e, *PRESENT*)==null;**

**How Serialization and Deserialization works in HashSet ?**

**The code is given below.**

**private** **void** writeObject(java.io.ObjectOutputStream s)

**throws** java.io.IOException {

// Write out any hidden serialization magic

s.defaultWriteObject();

// Write out HashMap capacity and load factor

s.writeInt(map.capacity());

s.writeFloat(map.loadFactor());

// Write out size

s.writeInt(map.size());

// Write out all elements in the proper order.

**for** (E e : map.keySet())

s.writeObject(e);

}

**private** **void** readObject(java.io.ObjectInputStream s)

**throws** java.io.IOException, ClassNotFoundException {

// Read in any hidden serialization magic

s.defaultReadObject();

// Read in HashMap capacity and load factor and create backing HashMap

**int** capacity = s.readInt();

**float** loadFactor = s.readFloat();

map = (((HashSet)**this**) **instanceof** LinkedHashSet ?

**new** LinkedHashMap<E,Object>(capacity, loadFactor) :

**new** HashMap<E,Object>(capacity, loadFactor));

// Read in size

**int** size = s.readInt();

// Read in all elements in the proper order.

**for** (**int** i=0; i<size; i++) {

E e = (E) s.readObject();

map.put(e, ***PRESENT***);

}

}

**How clone() method works in HashSet ?**

It returns the shallow copy of the HashSet.

**public** Object clone() {

**try** {

HashSet<E> newSet = (HashSet<E>) **super**.clone();

newSet.map = (HashMap<E, Object>) map.clone();

**return** newSet;

} **catch** (CloneNotSupportedException e) {

**throw** **new** InternalError();

}

}

**How equals() and hashcode() method works in HashSet ?**

The equals method is given below. These methods are part of AbstractSet

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

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

**return** **true**;

**if** (!(o **instanceof** Set))

**return** **false**;

Collection c = (Collection) o;

**if** (c.size() != size())

**return** **false**;

**try** {

**return** containsAll(c);

} **catch** (ClassCastException unused) {

**return** **false**;

} **catch** (NullPointerException unused) {

**return** **false**;

}

}

The hashcode method is given below.

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

**int** h = 0;

Iterator<E> i = iterator();

**while** (i.hasNext()) {

E obj = i.next();

**if** (obj != **null**)

h += obj.hashCode();

}

**return** h;

}

How TreeSet works internally

The code is given below.

**private** **transient** NavigableMap<E,Object> m;

**private** **static** **final** Object ***PRESENT*** = **new** Object();

**public** TreeSet() {

**this**(**new** TreeMap<E,Object>());

}

**public** TreeSet(Comparator<? **super** E> comparator) {

**this**(**new** TreeMap<>(comparator));

}

From the above code it is obvious that TreeSet internally uses TreeMap.

\*\* The equals(), clone(), hashcode() methods work like HashSet.

**How LinkedHashSet internally works ?**

LinkedHashSet internally uses HashSet, methods and constructors are inherited.

\*\* TreeSet internally uses TreeMap which has been developed on the base line of Red and Black Tree algorithm.

**How HashMap put() method works internally ?**

Always remember that, the initial capacity and threashold in case of HashMap is 16, 2 << 4.

The load factor is 0.75f.

The constructor for HashMap is given below.

public HashMap() {

this(DEFAULT\_INITIAL\_CAPACITY, DEFAULT\_LOAD\_FACTOR);

}

The source code for put method given below.

**public** V put(K key, V value) {

**if** (table == ***EMPTY\_TABLE***) {

**inflateTable(threshold);**

}

**if** (key == **null**)

**return** putForNullKey(value);

**int hash = hash(key);**

**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**;

}

**void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {

**if** ((size >= threshold) && (**null** != table[bucketIndex])) {

**resize(2 \* table.length);**

hash = (**null** != key) ? hash(key) : 0;

bucketIndex = *indexFor*(hash, table.length);

}

createEntry(hash, key, value, bucketIndex);

}

**void** createEntry(**int** hash, K key, V value, **int** bucketIndex) {

Entry<K,V> e = table[bucketIndex];

table[bucketIndex] = **new** Entry<>(hash, key, value, e);

size++;

}

Steps of Put Method

1. Get the hashcode of the Key
2. Calculate the hash value of the hashcode
3. Find the index of the bucket using the following function

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

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

}

1. Iterate the entire bucket and check whether key is already present or not, if present, simply modify the value or update the value.
2. If the key is not present, check the bucket has been filled up or not.
3. If the bucket has been filled up, increase the size of the Entry object array as double, the code is given below.

**void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {

**if** ((size >= threshold) && (**null** != table[bucketIndex])) {

**resize(2 \* table.length);**

hash = (**null** != key) ? hash(key) : 0;

bucketIndex = *indexFor*(hash, table.length);

}

createEntry(hash, key, value, bucketIndex);

}

**void resize(int newCapacity)** {

**Entry[] oldTable = table;**

**int oldCapacity = oldTable.length;**

**if (oldCapacity == *MAXIMUM\_CAPACITY*) {**

**threshold = Integer.*MAX\_VALUE*;**

**return;**

**}**

**Entry[] newTable = new Entry[newCapacity];**

**transfer(newTable, initHashSeedAsNeeded(newCapacity));**

table = newTable;

**threshold = (int)Math.*min*(newCapacity \* loadFactor, *MAXIMUM\_CAPACITY* + 1);**

}

Let see how it works

* Suppose initial capacity is 16.
* Initial capacity and threshold are same ie 16. Refer to the constructor of hashmap , it

**threshold = initialCapacity;**

**this.loadFactor = loadFactor; // ie 0.75**

* If the size of threshold is greater than or equal to 16 and bucket index has a value, increase the size to double. As it is obvious from the below code.

**if ((size >= threshold) && (null != table[bucketIndex])) {**

**resize(2 \* table.length);**

**In the above case, resize( 2 \* 16 ), so size = 16**

* Now calculate the old capacity which is nothing but the length of the bucket ie

**int oldCapacity = oldTable.length;**

* Check if the old capacity is greater than Maximum Size ie

**static** **final** **int** ***MAXIMUM\_CAPACITY*** = 1 << 30; //**1073741824**

* A new array of size 32 will be created.
* Now new threshold value will be calculated as

**threshold = (int)Math.*min*(newCapacity \* loadFactor, *MAXIMUM\_CAPACITY* + 1);**

**threshold = (int)Math.*min*(32 \* 0.75, 1073741824 + 1);**

**threshold = (int)Math.*min*(32 \* 0.75, 1073741824 + 1);**

**threshold = (int) Math.min(24, 1073741824 + 1);**

**threshold = 24; //Finally**

How hashmap reacts when you pass the your capacity in hashmap ? How capacity is calculated ?

Let us take an example below.

**HashMap hm = new HashMap(7);**

Now in this case, the initial capacity will be 7, load factor will be 0.75. The code is given below.

**public** HashMap(**int** initialCapacity) {

**this**(initialCapacity, ***DEFAULT\_LOAD\_FACTOR***);

}

**public** HashMap(**int** initialCapacity, **float** loadFactor) {

**if** (initialCapacity < 0)

**throw** **new** IllegalArgumentException("Illegal initial capacity: " +

initialCapacity);

**if** (initialCapacity > ***MAXIMUM\_CAPACITY***)

initialCapacity = ***MAXIMUM\_CAPACITY***;

**if** (loadFactor <= 0 || Float.*isNaN*(loadFactor))

**throw** **new** IllegalArgumentException("Illegal load factor: " +

loadFactor);

**this**.loadFactor = loadFactor;

threshold = initialCapacity;

init();

}

Now in this case the growth of HashMap is bit different. Let us see below. Now again see the put method.

**public** V put(K key, V value) {

**if (table == *EMPTY\_TABLE*) {**

**inflateTable(threshold);**

**}**

**if** (key == **null**)

**return** putForNullKey(value);

**int** hash = hash(key);

**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**;

}

Let us see the inflateTable() method.

**private** **void** inflateTable(**int** toSize) {

// Find a power of 2 >= toSize

**int capacity = *roundUpToPowerOf2*(toSize);**

**threshold = (int) Math.*min*(capacity \* loadFactor, *MAXIMUM\_CAPACITY* + 1);**

**table = new Entry[capacity];**

initHashSeedAsNeeded(capacity);

}

The above method is used to create initial array of Entry object. Let us see how threshold is calculated.

**int capacity = *roundUpToPowerOf2*(toSize); The method is given below.**

**private** **static** **int** roundUpToPowerOf2(**int** number) {

// assert number >= 0 : "number must be non-negative";

**return** number >= ***MAXIMUM\_CAPACITY***

? ***MAXIMUM\_CAPACITY***

: (number > 1) ? Integer.*highestOneBit*((number - 1) << 1) : 1;

}

What is Integer.*highestOneBit*((number - 1) << 1) : 1; The code is given below.

**public static int** highestOneBit(**int** i) {  
 *// HD, Figure 3-1* i |= (i >> 1);  
 i |= (i >> 2);  
 i |= (i >> 4);  
 i |= (i >> 8);  
 i |= (i >> 16);  
 **return** i - (i >>> 1);  
}

We need to understand about how the above method ***roundUpToPowerOf2*(toSize)** works. Let us consider a small java program and see how numbers are getting generated.

**static final int *MAXIMUM\_CAPACITY*** = 1 << 30;  
**public static int** roundUpToPowerOf2(**int** number) {  
 *// assert number >= 0 : "number must be non-negative";* **return** number >= ***MAXIMUM\_CAPACITY*** ? ***MAXIMUM\_CAPACITY*** : (number > 1) ? Integer.*highestOneBit*((number - 1) << 1) : 1;  
}

System.***out***.println(*roundUpToPowerOf2*(0)); *//1*System.***out***.println(*roundUpToPowerOf2*(1)); *//1*System.***out***.println(*roundUpToPowerOf2*(2)); *//2*System.***out***.println(*roundUpToPowerOf2*(3)); *//4*System.***out***.println(*roundUpToPowerOf2*(4)); *//4*System.***out***.println(*roundUpToPowerOf2*(5)); *//8*System.***out***.println(*roundUpToPowerOf2*(7)); *//8*System.***out***.println(*roundUpToPowerOf2*(9)); *//16*System.***out***.println(*roundUpToPowerOf2*(11)); *//16*System.***out***.println(*roundUpToPowerOf2*(15)); *//16*System.***out***.println(*roundUpToPowerOf2*(16)); *//16*System.***out***.println(*roundUpToPowerOf2*(19)); *//32*System.***out***.println(*roundUpToPowerOf2*(27)); *//32*System.***out***.println(*roundUpToPowerOf2*(32)); *//32*System.***out***.println(*roundUpToPowerOf2*(33)); *//64*System.***out***.println(*roundUpToPowerOf2*(48)); *//64*System.***out***.println(*roundUpToPowerOf2*(65)); *//128*

The mathematical trick is given below.

Choose any number and see how it falls in the below segment.

**… 512 256 128 64 32 16 8 4 2 1**

Let us number 65, it falls in between 64 and 128, so capacity will be 128.

If number is 16, capacity will 16

Now threashold calculation

**threshold = (int) Math.*min*(capacity \* loadFactor, *MAXIMUM\_CAPACITY* + 1);**

Let us consider, initial capacity is 5, it falls in between 4 and 8, so capacity will be 8.

initialCapacity = 5;

capacity = 8;

Threshold = (int) Math.min(8 \* 0.75 , ***MAXIMUM\_CAPACITY* + 1)**

Threshold = (int) Math.min(6 , ***MAXIMUM\_CAPACITY* + 1)**

Threshold = 6;

It means after 6 entries, the Entry object will grow and all the values will be copied.

**\*\* When the Entry array object is resized, again the hash value is calculated.**

**In case of LinkedHashMap, the following init() is overridden, but method is blank in case of HashMap. The purpose is to create a LinkedList in case of LinkedHashMap. LinkedHashMap extends HashMap.**

**private** **transient** Entry<K,V> header;

**void** init() {

header = **new** Entry<>(-1, **null**, **null**, **null**);

header.before = header.after = header;

}

**Besides the following methods have been overridden in LinkedHashMap.**

**void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {

**super**.addEntry(hash, key, value, bucketIndex);

// Remove eldest entry if instructed

Entry<K,V> eldest = header.after;

**if** (removeEldestEntry(eldest)) {

removeEntryForKey(eldest.key);

}

}

**void** createEntry(**int** hash, K key, V value, **int** bucketIndex) {

HashMap.Entry<K,V> old = table[bucketIndex];

Entry<K,V> e = **new** Entry<>(hash, key, value, old);

table[bucketIndex] = e;

e.addBefore(header);

size++;

}

**void** transfer(HashMap.Entry[] newTable, **boolean** rehash) {

**int** newCapacity = newTable.length;

**for** (Entry<K,V> e = header.after; e != header; e = e.after) {

**if** (rehash)

e.hash = (e.key == **null**) ? 0 : hash(e.key);

**int** index = *indexFor*(e.hash, newCapacity);

e.next = newTable[index];

newTable[index] = e;

}

}

TreeMap does not extend HashMap, it extends AbstractHashMap only.