###### Collections

A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data.

**What Is a Collections Framework?**

A *collections framework* is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

* **Interfaces**: These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
* **Implementations:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.

**Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be *polymorphic*: that is, the same method can be used on many different implementations of the appropriate collection interface.

In essence algorithms are reusable functionality.

**How to make an Object or Class Immutable?**

1.   Make the class final

2.   Make all members final, set them explicitly, in a static block, or in the constructor

3.   Make all members private

4.   No Methods that modify state

5.   Be extremely careful to limit access to mutable members (remember the field may be final but the object can still be mutable. ie private final Date is StillMutable). You should make defensive copies in these cases.

**Benefits of the Java Collection Framework:**

* Reduces programming effort.
* Increases program speed and quality.
* Allows interoperability among unrelated APIs.
* Reduces effort to learn and to use new APIs.
* Reduces effort to design new APIs.

**Interfaces:**

The core collection Interfaces



A Set is a special kind of Collection, SortedSet is a special kind of Set, and so forth.

**Note:**

* Map is not a true Collection.
* All core Collection interfaces are generic.

**Set:**

HashSet implements Set interface which does not allow duplicate value. It is not synchronized and is not thread safe.

* s1.retainAll(s2) - transforms s1 into the intersection of s1 and s2. (The intersection of two sets is the set containing only the elements common to both sets.)
* s1.addAll(2) - transforms s1 into the union of s1 and s2.
* s1.containsAll(s2) – returns true is s2 is subset of s1.(s2 is a subset of s1 if set s1 contains all the elements of s2.)
* s1.removeAll(s2) - transforms s1 into the set difference of s1 and s2. (For example, the set difference of s1 minus s2 is the set containing all of the elements found in s1 but not in s2.)

How HashSet works internally in Java?

HashSet uses HashMap internally to store its objects. Whenever you create a HashSet object, one HashMap object associated with it is also created. This HashMap object is used to store the elements you enter in the HashSet. The elements you add into the HashSet are **Stored** as Keys in HashMap object. The value associated with those keys will be a constant.

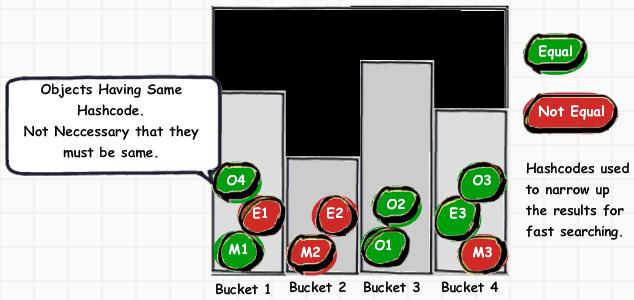
Every constructor of HashSet class internally creates one HashMap object.

What is the need to Override Hashcode() and equals() method?

* If two objects are same then they must return same value in hashcode() and equals() method whenever invoked.
* It is not necessary that two different objects must have different hashcode values. It might be possible that they share common hash bucket.

As the question comes into your mind that equals() method is used to compare objects that they are having same value or not but **why should we override the hashcode method?**

The answer to the question is for the hash technique based data structures like HashMap and HashTable.



As you see in the above diagram that every object is placed in Hash bucket depending on the hashcode they have. It is not necessary that every different object must have different hashcode. **Hashcode is used to narrow the search result.** When we try to insert any key in HashMap first it checks whether any other object present with same hashcode and if yes then it checks for equals() method. If two objects are same them HashMap will not add that key instead it will replace the old value by the new one.

**What will happen if I don’t override the hashcode method?**

If the object does not implement hashcode() method and used as key we will not get object back.

package com.G2.Collections;

import java.util.HashMap;

class Movie {

private String name, actor;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getActor() {

return actor;

}

public String getActor() {

return actor;

}

public void setActor(String actor) {

this.actor = actor;

}

public int getReleaseYr() {

return releaseYr;

}

public void setReleaseYr(int releaseYr) {

this.releaseYr = releaseYr;

}

private int releaseYr;

}

public class HashMapDemo {

public static void main(String[] args) {

Movie m = new Movie();

m.setActor("Akshay");

m.setName("Thank You");

m.setReleaseYr(2011);

Movie m1 = new Movie();

m1.setActor("Akshay");

m1.setName("Khiladi");

m1.setReleaseYr(1993);

Movie m2 = new Movie();

m2.setActor("Akshay");

m2.setName("Taskvir");

m2.setReleaseYr(2010);

Movie m3 = new Movie();

m3.setActor("Akshay");

m3.setName("Taskvir");

m3.setReleaseYr(2010);

HashMap<Movie, String> map = new HashMap<Movie, String>();

map.put(m, "ThankYou");

map.put(m1, "Khiladi");

map.put(m2, "Tasvir");

map.put(m3, "Duplicate Tasvir");

//Iterate over HashMap

for (Movie mm : map.keySet()) {

System.out.println(map.get(mm).toString());

}

Movie m4 = new Movie();

m4.setActor("Akshay");

m4.setName("Taskvir");

m4.setReleaseYr(2010);

/\* We are trying to retrieve m2, by creating object m4 with exact values as of m2, However Hashcode method is not implemented and that why we are not able to get Object m2 \*/

if(map.get(m4) == null ){

System.out.println("----------------");

System.out.println("Object not found");

System.out.println("----------------");

}else{

System.out.println(map.get(m4).toString());

}

}

}

**Output:**

Khiladi

Tasvir

ThankYou

Duplicate Tasvir

—————-

Object not found

—————-

As you can see in above program:

1. Duplicate objects are added in HashMap as a key (Because we have not overriden the hashcode and equals method)
2. We are not able to get back object from map (Because hashcode is not implemented).

**Override only hashcode():**

MyClass first = new MyClass("a","first");

MyClass second = new MyClass("a","second");

If you only override hashcode then when you call myMap.put(first, somevalue) it takes first, calculates its hashcode and stores it in a given bucket. Then when you call myMap.put(second, someothervalue) it should replace first with the second because they are equal.

But the problem is that equals was not redefined, so when the map hases second and iterates through the bucket looking if there is an object k such that second.equals(k) is true it won’t find any as second.equals(first) will be false.

**Overrides only equals():**

If only equals is overridden, then when you call myMap.put(first, somevalue) first will hash to some bucket and when you call myMap.put(second, someOtherValue) it will hash to some other bucket (as they have different hashcode). So, although they are equal, as they don’t have hash to the same bucket, the map can’t realize it and both of them stay in the map.

**TreeSet():**

Constructs a new, empty tree set, sorted according to the natural ordering of its elements.

**TreeSet(Collection<? Extends E> c)**

Constructs a new tree set containing the elements in the specified collection, sorted according to the natural ordering of its elements.

**TreeSet(Comparator<? Super E> comparator)**

Constructs a new, empty tree set, sorted according to the specified comparator.

**TreeSet(SortedSet<E> s)**

Constructs a new tree set containing the same elements and using the same ordering as the specified sorted set.

**10 Equals and HashCode Interview Questions in Java.**

**List:**

List is an ordered Collection (sometimes called a sequence). List may contain duplicate elements. In addition to the operations inherited from Collection, the List interface includes operations for the following:

* Positional Access – manipulates elements based on their numeric position in the list. This includes methods such as get, add, addAll, and remove
* Search - Search for a specified object in the list and returns its numerical position. Search method includes indexOf and lastIndexOf.
* Iteration – Extends Iterator semantics to take advantage of the list’s sequential nature. The listIterator provides this behavior.
* Range-view - The sublist method performs arbitrary range operations on the list.

The Java platform contains two general-purpose List implementations. ArrayList, which is usually the better-performing implementation, and LinkedList which offers better performance under certain circumstances.

**ArrayList trimToSize() method:**

trimToSize() method is used for memory optimization. It trims the capacity of ArrayList to the current list size. For e.g. An ArrayList is having capacity of 15 but there are only 5 elements in it, calling trimToSize() method on this ArrayList would change the capacity from 15 to 5.

**How to synchronize ArrayList in Java?**

As we are aware that ArrayList is not synchronized and should not be used in multi-thread environment without explicit synchronization.

There are two ways to synchronize explicitly:

1. Using Collections.synchronizedList() method.
2. Using thread-safe variant of ArrayList: CopyOnWriteArrayList

While using Collections.synchronizedList() Iterator using should be in Synchronized block.

**How to Sort ArrayList in descending order?**

We are using Collections.reverseOrder() method along with Collections.sort() in order to sort the list in decreasing order.

Ex: Collections.sort(arrayList, Collections.reverseOrder())

**How to initialize ArrayList?**

1. Initialization using Arrays.asList()

ArrayList<Type> obj = new ArrayList<Type>(

Arrays.asList(Object o1, Object o2, Object o3, ....so on));

1. Anonymous inner class method to initialize ArrayList

ArrayList<T> obj = new ArrayList<T>(){{

add(Object o1);

add(Object o2);

add(Object o3);

...

...

}};

1. Normal way of ArrayList initialization
2. Use Collections.ncopies

ArrayList<T> obj = new ArrayList<T>(Collections.nCopies(count, element));

**How ArrayList works internally?**

Characteristics of AL:

* ArrayList is growable arrays. It dynamically grows its size.
* It is going to maintain the Insertion order.
* Allow duplicates
* It is index based.

By default, 10 is the size of AL.

Logic to increase size of AL: (capacity\*3)/2+1.

**LinkedList in Java:**

Linked Lists are a very common way of storing arrays of data. The major benefit of linked lists is that you do not specify a fixed size for your list. The more elements you add to the chain, the bigger the chain gets.

Within the LinkedList implementation there is a **private class Node** which provides the structure for a node in a doubly linked list. It has item variable for holding the value and two reference to Node class itself for connecting to next and previous nodes.

private static class Node<E> {

E item;

Node<E> next;

Node<E> prev;

Node(Node<E> prev, E element, Node<E> next) {

this.item = element;

this.next = next;

this.prev = prev;

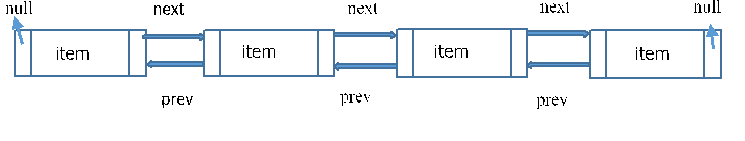
}

}

**Graphical representation of Linked List with nodes:**

Here is a graphical representation of a linked list to help you better visualize how actually a node looks like and how it connects with other nodes through next and prev references.

Since it is storing references for both next and previous nodes that is why it is a doubly linked list implementation.



* Java LL class uses doubly linked list to store the elements. It extends the AbstractList class and implements List and Deque interfaces.
* Java LL class can contain duplicate elements.
* Java LL class maintains insertion order.
* Java LL class is not synchronized.
* In Java LL class, manipulation is fast because no shifting needs to be occurred.
* Java LL class can be used as List, stack or queue.

**Converting LinkedList to array using toArray():**

LinkedList<String> linkedList = new LinkedList<String>();

String [] array = linkedList.toArray(new String[linkedList.size]);

**java.util.HashMap:**

* Map implementation that satisfies most of the basic use cases.
* Not “thread-safe”.
* Iteration not guaranteed in insertion order.
* Need to use “synchronized” operations when manipulating from multiple threads (concurrent adds/removes/iterations)
* Most enterprise applications populate the map once and the read many times from many threads. Given this HashMap suffices for all such scenarios, without any worries of performance overheads.

**How HashMap works internally?**

Maps are associated Arrays, that’s let you store in key-value associations.

HashMap is one of the implementations of the Map. HashMap is known as HashMap because; it is based on technique called Hashing.

Hashing is transforming a large object into short or fixed length presentation.

Helps in faster indexing and search.

In Java every object has a method public in hashcode () that will return the hash value for a given object.

If 2 objects are equal, two objects should have same hash code as well. That means, it’s very important to have robust hash code implementation in your classes.

**Why equals and hash code are important?**

Hash code is used in storing values into HashMap. If Hash code is not consistent, we will not able to see, the values. Lookup is not proper.

HashMap uses internally LinkedList. When 2 Objects have same hashcode(index), elements are stored in LinkedList.

HashMap default size is 16. Ranges from 0-15.

HashMap allows null key, which always goes to index 0 as hash of null is “0”.

Remove all mappings from HashMap - .clear()

**How to synchronize HashMap?**

HashMap is non-synchronized collection class. If we need to perform thread-safe operations on it then we must need to synchronize it explicitly.

In order to synchronize the map we have to use Collections.synchronizedMap(hashmap) it returns a thread-safe map backed up by the specified HashMap.

**Important Note:** Iterator should be used in a synchronized block even if we have synchronized the HashMap explicitly

**What has changed in Java 8?**

In Java 8, when we have too many unequal keys which gives same hash code (index) - when the number of items in a hash bucket grows beyond certain threshold(TREEIFY\_THRESHOLD = 8), content of that buckets switches from using linked list of Entry objects to a balanced tree.

This theoretically improves the worst-case performance from O(n) to O(log n).

Balanced search tree, where left nodes have lesser weight (HashCode or Comparable result) for the keys involved.

If both the hashcodes have same value, then compares on Key. Bigger key goes right and smaller key goes left.

**Map.Entry Interface:**

Map.Entry interface helps us iterating a Map class such as HashMap and TreeMap etc.

**A TALE OF A BUNCH OF MAPS!**

**java.util.HashTable:**

* Legacy associate array implementation; Inducted to collection family by implementing Map interface later. Where in Java first release we had only HashTable, This is the time, and collections did not exist.
* All methods are “thread-safe”.
* There is “synchronized” keyword on each public method such as (put, get, remove etc.)
* Overhead in an environment where Map is initialized once and the read by multiple threads.
* Not good, because once we will write into the HashTable, and read always from that table. If synchronized, read has to till another thread completes reading.

**How to sort HashTable in Java?**

There are ways to sort Hashtable using Collections.list and Collection.sort, however best thing is using LinkedHashMap and TreeMap.

There are 2 best ways to sort Hashtable:

1. Use LinkedHashMap: When you want to preserve the insertion order.
2. Use TreeMap: When you want to sort the key-value pairs.

**java.util.LinkedHashMap:**

* Very similar to HashMap
* Iteration is guaranteed in insertion order
* Maintains separate doubly linked list of all entries that is kept in entry insertion order
* Can be used in use-cases where hash map like behavior is needed at the same time order of insertion has to be preserved.

**java.util.TreeMap:**

* Implementation of SortedMap and NavigableMap interfaces.
* Iteration is guaranteed in “natural sorted” order of keys.
* Either the keys should implement “Comparable” interface(if not exception will be thrown: ClassCastException); if not provide we need to provide an explicit Comparator in the constructor.
* Red-black tree based implementation. NavigableMap interface provides methods that can return closes match to the key(floorEntry()…)

**java.util.EnumMap:**

* EnumMap<K extends Enum<K>,V>
* For use with Enum type keys. All the keys in an enum map must come from a single enum type that is specified, explicitly or implicitly, when the map is created.
* Null keys are not permitted in EnumMap. And not synchronized.
* Iterator does not fail-fast.

**java.util.concurrent.ConcurrentHashMap:**

* Supports full concurrency during retrieval. Means, retrieval operations do not block even if adds are running concurrently.
* Reads can happen fast, while writes require a lock is acquired at granular level, whole table is not locked only segments are locked. So a very rare chance of read waiting on write to complete.
* Iterations do not throw concurrent modification exception (with in same thread).
* Can be used in cases where a lot of concurrent addition happens followed by or concurrent reads later on.
* Null key is not allowed.

**Iterator:**

Iterator is an object that implements either the Iterator or the ListIterator interface.

Iterator enables you to cycle through collection, obtaining or removing elements.

ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements.

Each of the collection classes provides and iterator() method that returns an iterator to the start of collection. By using this iterator object, you can access each element in the collection, one element at a time.

In general, below are the steps to follow:

1. Obtain an iterator to the start of collection by calling the Collection’s iterator() method.
2. Set up a loop that makes a call to hasNext(). Have the loop iterate as long as hasNext() returns true.
3. Within the loop, obtain each element by calling next().

For the collections that implement List, you can also obtain an iterator by calling ListIterator.

**Difference between HashMap and HashTable:**

|  |  |
| --- | --- |
| HashMap | HashTable |
| It is not synchronized. | It is synchronized. |
| Allows null values and null keys | Doesn’t allow null key or value |
| HashMap is introduced in JDK 1.2 | HashTable is legacy class |
| Collections.synchronizedMap() is used to synchronize | By default it is synchronized. |
| HashMap is fast | HashTable is slow |
| HashMap is traversed by Iterator | HashTable is traversed by Enumerator and Iterator |
| Iterator in HashMap is fail-fast | Iterator in HashTable is not fail-fast |
| HashMap inherits AbstractMap Class | HashTable inherits Dictionary class. |

**Difference between HashSet and HashMap:**

|  |  |
| --- | --- |
| HashSet | HashMap |
| HashSet class implements Set Interface | HashMap implements Map interface |
| In HashSet we store objects | HashMap is used for storing keys and value pairs. In short it maintains the mapping of key and value. |
| Duplicate elements are not allowed. | Duplicate keys are not allowed. |
| HashSet permits to have a single null value. | HashMap permits single null key and any number of null values. |

**Difference between HashSet, TreeSet and LinkedHashSet:**

TreeSet, HashSet and LinkedHashSet in Java are 3 Set implementation in collection framework and like many others they are also used to store objects.

|  |  |  |
| --- | --- | --- |
| HashSet | TreeSet | LinkedHashSet |
| Main Feature: General purpose collection for storing object | Main Feature: Sorting | Main Feature: Preserve insertion Order. |
| Internally uses HashMap for implementation | Is a Sorted set implementation, which allows keeping elements in the sorted order defined either by Comparator or Comparable interface | LinkedHashSet uses doubly linked list |
| Not allow to store duplicates | Not allow to store duplicates | Not allow to store duplicates |
| Not thread-safe | Not thread-safe | Not thread-safe |
| Fail-fast iterator | Fail-fast iterator | Fail-fast iterator |
| Performance and Speed: Fast | Performance and Speed: bit slower since, it has sorting | Performance and Speed: Next to HashSet. |
| Null is allowed | Null is not allowed | Null is allowed |
| Uses equals() method for comparison | Uses compareTo() method to maintain ordering | Uses equals() method for comparison |

**Iterator v/s ListIterator:**

|  |  |
| --- | --- |
| Iterator | ListIterator |
| Iterator used for traversing List and Set | ListIterator used to traverse only List. |
| We can traverse only in forward direction | We can traverse in both the directions (forward and backward) |
| We cannot obtain indexed while using iterator | We can obtain indexes at any point of time traversing a list using ListIterator. The methods nextIndex() and previousIndex() are used for this purpose |
| We cannot add element to the collection while traversing it using Iterator, throws ConcurrentModificationException. | We can add element at any point of time while traversing a list. |
| We cannot replace the existing element value when using iterator | By using set(E e) method of ListIterator we can replace the element returned by next() or previous() methods |

**ArrayList v/s Vector:**

|  |  |
| --- | --- |
| ArrayList | Vector |
| Is not synchronized | Is synchronized |
| Performance: High, because; not thread-safe | Performance: Slow, because; thread-safe |
| Automatic Increase in Capacity: It increases the size by (initialCapacity\*3)/2+1 | Automatic Increase in Capacity: Doubles the size of an array. |
| Set Increment Size: AL does not define the increment size | Set Increment Size: Vector defines the increment size. |
| Enumerator: Only uses Iterator | Enumerator: Other than HashTable, Vector is the only other class uses Enumerator |
| Introdcution in Java: AL was introduced in Java version 1.2 | Introdcution in Java: It was since very first version of jdk. |
|  |  |

**ArrayList v/s HashMap:**

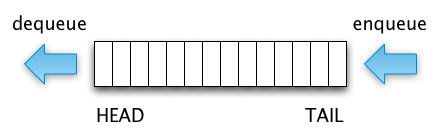
|  |  |  |
| --- | --- | --- |
|  | ArrayList (AL) | HashMap (HM) |
| Implementation | AL implements List interface | HM implements Map interface |
| Memory Consumption | AL stores values alone and internally maintains the indexes for each element | HashMap store key & value pair. For each value there must be a key associated in HM. Hence memory consumption is high compared to AL. |
| Order | AL maintains the insertion order. | HM does not maintain the insertion order. |
| Duplicates | AL allows duplicate elements. | HM does not allow duplicate keys(It does allow duplicate values) |
| Nulls | AL can have any number of Null elements. | HM allows only one Null key |
| Get method | AL can get the element by specifying the index. | HM can get the value by passing the key to the HM. |

**ArrayList v/s LinkedList:**

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| Since Array is index based DS searching and getting an element from Array with index is pretty fast. Array provides O(1) performance for get(index) method but remove is costly in AL as you need to rearrange all elements. | LL can’t provide Random or index based access and you need to iterate over linked list to retrieve any element which is of order O(n). |
| Insertions are bit slow compare to LL. Performance O(n) | Insertions are easy and fast in linkedlist because there is no risk of resizing the array and copying content to new array. Performance O(1). |
| Removal is slower | Removal is faster |
| ArrayList requires less memory to store | LL requires more memory |

**Queues:**

Queue is FIFO structure. Addition takes place **only** at the tail, and removal takes place **only** at the head.



**The basic operations are:**

* Enqueue (x): add an item at the tail
* dequeue: remove the item at the head
* peek: return the item at the head (without removing it)
* size: returns the number of items in the queue
* isEmpty: return whether the queue has no items

**Usage:**

* Lines of cars at a light
* Printer buffer
* Packets waiting at the router
* Simulation

**Queue Implementations:**

Being a Collection subtype all methods in the Collection interface are also available in the Queue interface.

Since Queue is an interface you need to instantiate a concrete implementation of the interface in order to use it. You can choose between the following Queue implementations in the Java Collection API:

1. java.util.LinkedList
2. java.util.PriorityQueue

LinkedList is a pretty standard queue implementation.

**Priority Queue:**

PriorityQueue stores its elements internally according to their natural order (if they implement Comparable), or according to a Comparator passed to the PriorityQueue.

For Example: Let’s say we have an application that generates stocks reports for daily trading session and it processes a lots of data and takes time to process it. So customers are sending request to the application that is actually getting queued but we want to process premium customer’s first and standard customers after them. So in this case PriorityQueue implementation in java can be really helpful.

The head of the priority queue is the least element based on the natural ordering or comparator based ordering, if there are multiple objects with same ordering, then it can poll any one of them randomly. When we poll the queue, it returns the head object from the queue.

Priority Queue does not all NULL values. We cannot create Priority Queue of Objects that is non-comparable.

PQ size is unbounded but we can specify the initial capacity at the time of creation. When we add elements to the PQ, it’s capacity grows automatically.

PQ is not thread safe, so provides PriorityBlockingQueue class that implements the BlockingQueueInterface to use in java multi-threading environment.

**Deque:**

The deque interface is a subtype of Queue interface. Deque is a linear collection that supports insertion and removal at both ends. Thus **Deque** is short for **Double Ended Queue.** This interface defines the methods to access elements at both the ends of the deque. Methods provided are insert, remove and examine the element.

LinkedList is pretty standard queue/deque implementation. ArrayDeque stores the elements internally in an Array. If the number of elements exceeds the space in the array, a new array is allocated., and all the elements moved over. In other words, the ArrayDeque grows a needed, even if it stores elements in an array.

**Comparator and Comparable in Java:**

Both interfaces used to sort collection elements.

|  |  |  |
| --- | --- | --- |
| No. | Comparable | Comparator |
| 1 | It provides single sorting sequence. We can sort the collection on the basis of single element | It provides multiple sorting sequence. We can sort the collection on the basis of multiple elements. |
| 2 | Provides compareTo() method | Provides compare() method. |
| 3 | It is found in java.lang | It is found in java.util |
| 4 | Collections.sort(List) | Collections.sort(List, Comparator) |
|  |  |  |