**<https://facingissuesonit.com/2019/10/15/java-collection-framework-introduction/>**

[**https://www.guru99.com/java-collections-interview-questions-answers.html**](https://www.guru99.com/java-collections-interview-questions-answers.html)

<https://www.sanfoundry.com/java-questions-answers-freshers-experienced/>

## When to use List, Set and Map in Java?

1) If you do not want to have duplicate values in the database then Set should be your first choice as all of its classes do not allow duplicates.  
2) If there is a need of frequent search operations based on the index values then List (ArrayList) is a better choice.  
3) If there is a need of maintaining the insertion order then also the List is a preferred collection interface.  
4) If the requirement is to have the key & value mappings in the database then Map is your best bet.

The **spliterator()** method of Java HashSet class is used to creates a late-binding and fail-fast Spliterator over the elements in this set.

What is difference between fail fast and fail safe iterator?

The Major difference between Fail Fast and Fail Safe iterator is that the **Fail Safe does not throw any ConcurrentModificationException in modifying the object during the iteration process, contrary to fail fast, which throws an exception in such scenarios**.

The ConcurrentModificationException occurs when an object is tried to be modified concurrently when it is not permissible. This exception usually comes when one is working with **Java Collection classes**.

This exception may be thrown by methods that have detected concurrent modification of an object when such modification is not permissible.

For example, it is not generally permissible for one thread to modify a Collection while another thread is iterating over it. In general, the results of the iteration are undefined under these cirdrcumstances. Some Iterator implementations (including those of all the general purpose collection implementations provided by the JRE) may choose to throw this exception if this behavior is detected. Iterators that do this are known as *fail-fast* iterators, as they fail quickly and cleanly, rather that risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Java’s ConcurrentModificationException? There are two basic approaches:

1. Do not make any changes to a collection while an Iterator loops through it.
2. If you can’t stop the underlying collection from being modified during iteration, create a clone of the target data structure and iterate through the clone
3. 1) The main difference between intermediate and terminal operations is that intermediate operations return a stream as a result and terminal operations return non-stream values like primitive or object or collection or may not return anything.
4. 2) As intermediate operations return another stream as a result, they can be chained together to form a pipeline of operations. Terminal operations can not be chained together.
5. 3) Pipeline of operations may contain any number of intermediate operations, but there has to be only one terminal operation, that too at the end of pipeline.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. no.** | **Key** | **final** | **Finally** | **finalize** |
| 1. | Definition | final is the keyword and access modifier which is used to apply restrictions on a class, method or variable. | finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not. | finalize is the method in Java which is used to perform clean up processing just before object is garbage collected. |
| 2. | Applicable to | Final keyword is used with the classes, methods and variables. | Finally block is always related to the try and catch block in exception handling. | finalize() method is used with the objects. |
| 3. | Functionality | (1) Once declared, final variable becomes constant and cannot be modified. (2) final method cannot be overridden by sub class. (3) final class cannot be inherited. | (1) finally block runs the important code even if exception occurs or not. (2) finally block cleans up all the resources used in try block | finalize method performs the cleaning activities with respect to the object before its destruction. |
| 4. | Execution | Final method is executed only when we call it. | Finally block is executed as soon as the try-catch block is executed.  It's execution is not dependant on the exception. | finalize method is executed just before the object is destroyed. |

|  |  |
| --- | --- |
| Checked Exception | Unchecked Exception |
| Checked exceptions occur at compile time. | Unchecked exceptions occur at runtime. |
| The compiler checks a checked exception. | The compiler does not check these types of exceptions. |
| These types of exceptions can be handled at the time of compilation. | These types of exceptions cannot be a catch or handle at the time of compilation, because they get generated by the mistakes in the program. |
| They are the sub-class of the exception class. | They are runtime exceptions and hence are not a part of the Exception class. |
| Here, the JVM needs the exception to catch and handle. | Here, the JVM does not require the exception to catch and handle. |
| Examples of Checked exceptions:   * File Not Found Exception * No Such Field Exception * Interrupted Exception * No Such Method Exception * Class Not Found Exception | Examples of Unchecked Exceptions:   * No Such Element Exception * Undeclared Throwable Exception * Empty Stack Exception * Arithmetic Exception * Null Pointer Exception * Array Index Out of Bounds Exception * Security Exception |

|  |  |  |
| --- | --- | --- |
| **No.** | **Method Overloading** | **Method Overriding** |
| 1) | Method overloading is used *to increase the readability* of the program. | Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class. |
| 2) | Method overloading is performed *within class*. | Method overriding occurs *in two classes* that have IS-A (inheritance) relationship. |
| 3) | In case of method overloading, *parameter must be different*. | In case of method overriding, *parameter must be same*. |
| 4) | Method overloading is the example of *compile time polymorphism*. | Method overriding is the example of *run time polymorphism*. |
| 5) | In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter. | *Return type must be same or covariant* in method overriding. |

**Arrays vs Collections** : In Java an **Array** represents group of elements and a **Collection** also represents group of elements, then *if already Arrays have the same feature why collections ?*let’s see what is an Array exactly.

What is an Array in Java ?

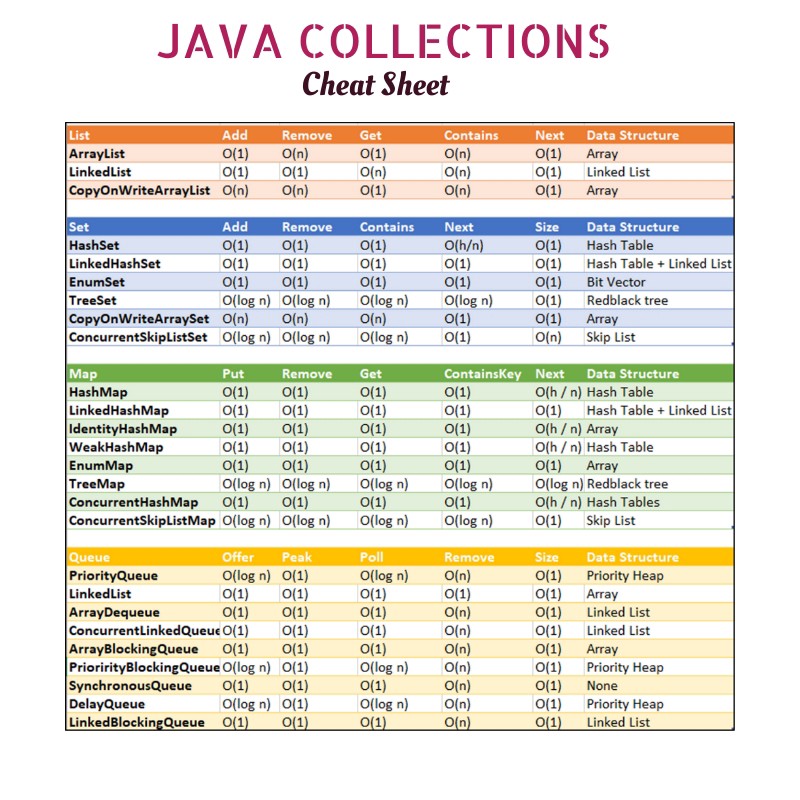
1. An Array is collection of *indexed* and *fixed* number of *homogeneous* (same type) elements.
2. Indexed : Arrays are stored elements in index based. The stored element position starts from ‘*zero’ and second element position ‘1′ and so on.*
3. For example if we take String array to represent Student names [“Satish”, ”Sunil”, ”Suresh”, ”Ravi”] position of “Satish” element is ‘0‘ , position of “Sunil” is 1, position of “Ravi” is 3.
4. Fixed : When we are defining array we have to specify the size of array. Example : String[] *students* = new String[10]; . Once we created an Array we can’t increase the size, size is fixed in arrays.
5. Homogeneous : Arrays elements are homogeneous, it means once we created array we can store same type of elements. For example in point 4 student array allow to store only String type elements, because the array type is String.

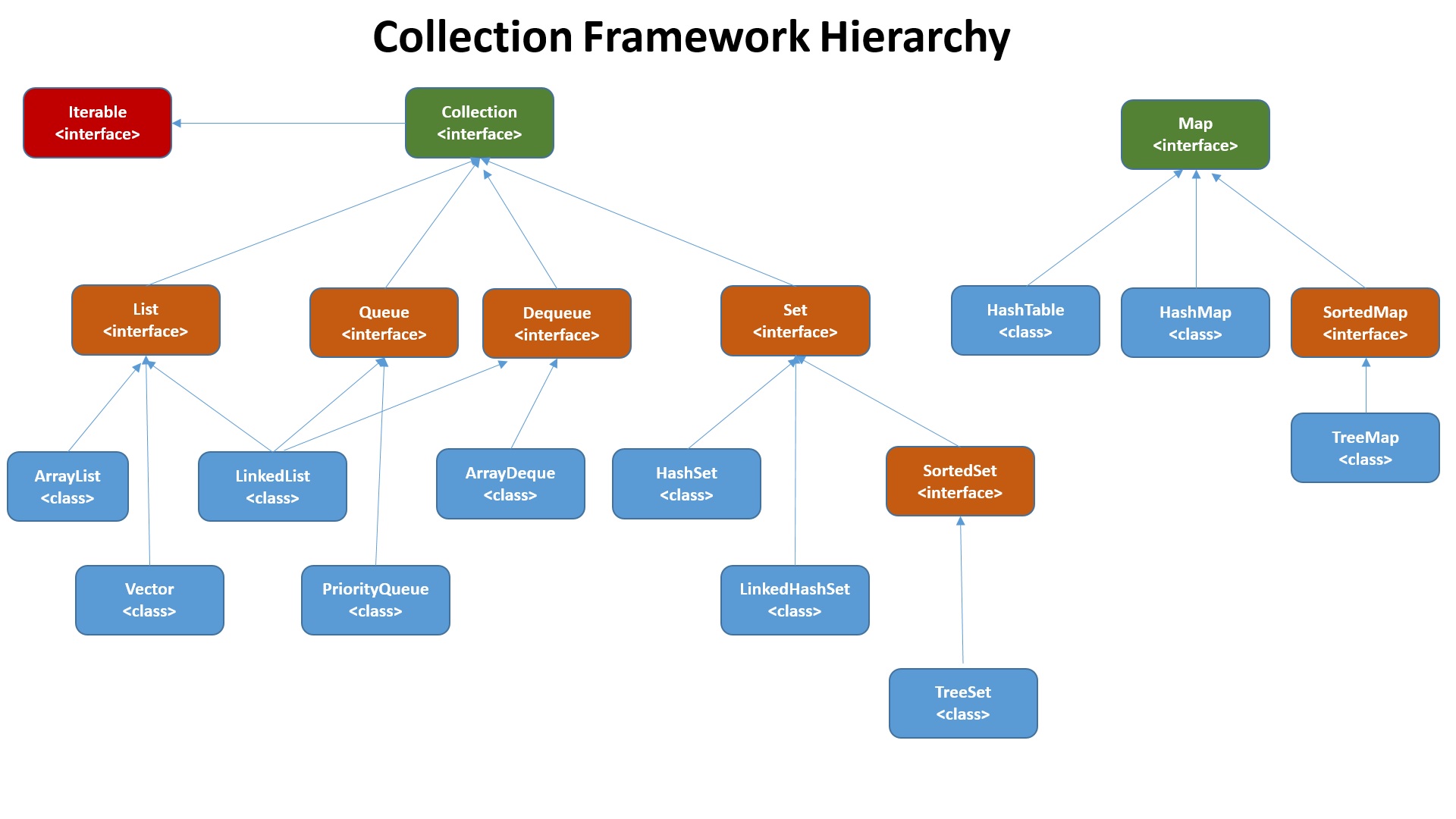
**Arrays vs Collections :**

| **Arrays** | **Collections** |
| --- | --- |
| 1. Size is fixed | 1. Size is not fixed, size is growable |
| 2. Can hold both primitive types(boolean, byte, short, int, long …etc) and object types. | 2. Can hold only object types |
| 3. There is no underlying data structures in arrays. The array itself used as data structure in java. | 3. Every Collection class there is underlying data structure. |
| 4. There is no utility methods in arrays | 4. Every Collection provides utility methods |

**Why collections ?**

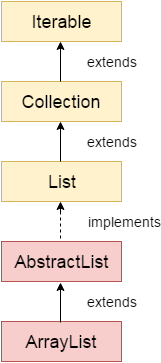
1. Arrays are fixed in size, in development always it’s not possible to define size of an array. We need to increase size based on requirement. Collections are not fixed in size, they are growable (size will be increased when it’s reached max size).
2. Arrays can only holds one type of elements. Collections can hold different type of elements.
3. There is no default method support from arrays for sorting, searching, retrieving etc… but collections have default utility method support, it’s handy for for developer. It will reduce the coding time.





| [List](https://www.geeksforgeeks.org/list-interface-java-examples/) | [Set](https://www.geeksforgeeks.org/set-in-java/) | [Map](https://www.geeksforgeeks.org/map-interface-java-examples/) |
| --- | --- | --- |
| The list interface allows duplicate elements | Set does not allow duplicate elements. | The map does not allow duplicate elements |
| The list maintains insertion order. | Set do not maintain any insertion order. | The map also does not maintain any insertion order. |
| We can add any number of null values. | But in set almost only one null value. | The map allows a single null key at most and any number of null values. |
| List implementation classes are [Array List](https://www.geeksforgeeks.org/arraylist-in-java/), [LinkedList](https://www.geeksforgeeks.org/linked-list-in-java/). | Set implementation classes are [HashSet](https://www.geeksforgeeks.org/hashset-in-java/), [LinkedHashSet](https://www.geeksforgeeks.org/linkedhashset-in-java-with-examples/), and [TreeSet](https://www.geeksforgeeks.org/treeset-in-java-with-examples/). | Map implementation classes are [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/), [HashTable](https://www.geeksforgeeks.org/hashtable-in-java/), [TreeMap](https://www.geeksforgeeks.org/treemap-in-java/), [ConcurrentHashMap](https://www.geeksforgeeks.org/concurrenthashmap-in-java/), and [LinkedHashMap](https://www.geeksforgeeks.org/linkedhashmap-class-java-examples/). |
| The list provides get() method to get the element at a specified index. | Set does not provide get method to get the elements at a specified index | The map does not  provide get method to get the elements at a specified index |
| If you need to access the elements frequently by using the index then we can use the list | If you want to create a collection of unique elements then we can use set | If you want to store the data in the form of key/value pair then we can use the map. |
| To traverse the list elements by using Listlterator. | Iterator can be used traverse the set elements | Through keyset, value, and entry set. |

ArrayList



Java **ArrayList** class uses a dynamic [*array*](https://www.javatpoint.com/array-in-java) for storing the elements. It is like an array, but there is no size limit. We can add or remove elements anytime. So, it is much more flexible than the traditional array. It is found in the java.util package. It is like the Vector in C++.

The ArrayList in Java can have the duplicate elements also. It implements the List interface so we can use all the methods of the List interface here. The ArrayList maintains the insertion order internally.

It inherits the AbstractList class and implements [List interface](https://www.javatpoint.com/java-list).

The important points about the Java ArrayList class are:

* Java ArrayList class can contain duplicate elements.
* Java ArrayList class maintains insertion order.
* Java ArrayList class is non [synchronized](https://www.javatpoint.com/synchronization-in-java).
* Java ArrayList allows random access because the array works on an index basis.
* In ArrayList, manipulation is a little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.
* We can not create an array list of the primitive types, such as int, float, char, etc. It is required to use the required wrapper class in such cases. For example

**23. Suppose we want to add an element in the middle of list. Which list implementation will provide you better performance? ArrayList or LinkedList?**

Answer: For the above scenario, LinkedList is a better choice because in the case of LinkedList, when we add an element at the specified position, internally, a node is created and only two links are changed.

But in the case of ArrayList, a lot of shifting is done in the memory when we add an element in the middle of the list or anywhere, except at the end.

So, LinkedList gives faster performance when we add an element in the middle of list.

**24. Both ArrayList and LinkedList provide get() method to retrieve an element at the specified position from the list. Which one is faster, ArrayList or LinkedList?**

Answer: ArrayList’s get() method is faster than LinkedList’s get() because LinkedList does not implement Random Access Interface.

Due to which it will traverse from the beginning or ending over the list until it reaches the index specified.

**2) How do you increase the current capacity of an ArrayList?**

**ensureCapacity() method** is used to increase the current capacity of an ArrayList. However, capacity of an ArrayList is automatically increased when we try to add more elements than the current capacity. To manually increase the current capacity, ensureCapacity() method is used.

Sort()

* Whenever we need to sort the values in a collection, this “sort” method transfers control to the compare method in the class.
* The compare method then returns some values based on the comparison.
* It returns 0 if both the objects are equal.
* This returns 1 if the first object is greater than the second.
* It returns -1 if the second object is greater than the first.
* Using these values the function decides whether to swap the values for the sorting process.

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses a **dynamic array** to store the elements. | LinkedList internally uses a **doubly linked list** to store the elements. |
| 2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the other elements are shifted in memory. | Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory. |
| 3) An ArrayList class can **act as a list** only because it implements List only. | LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces. |
| 4) ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |
| 5) The memory location for the elements of an ArrayList is contiguous. | The location for the elements of a linked list is not contagious. |
| 6) Generally, when an ArrayList is initialized, a default capacity of 10 is assigned to the ArrayList. | There is no case of default capacity in a LinkedList. In LinkedList, an empty list is created when a LinkedList is initialized. |
| 7) To be precise, an ArrayList is a resizable array. | LinkedList implements the doubly linked list of the list interface. |

# **Java HashSet**



Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

* HashSet stores the elements by using a mechanism called **hashing.**
* HashSet contains unique elements only.
* HashSet allows null value.
* HashSet class is non synchronized.
* HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
* HashSet is the best approach for search operations.
* The initial default capacity of HashSet is 16, and the load factor is 0.75.

Now as you can see that whenever we create a HashSet, it internally creates a [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/) and if we insert an element into this HashSet using add() method, it actually call put() method on internally created HashMap object with element you have specified as it’s key and constant Object called **“PRESENT”** as it’s value. So we can say that **a Set achieves uniqueness internally through HashMap**. Now the whole story comes around [how a HashMap and put() method internally works](https://www.geeksforgeeks.org/internal-working-of-hashmap-java/).

As we know in a [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/) each key is unique and when we call put(Key, Value) method, it returns the previous value associated with key, or null if there was no mapping for key. So in add() method we check the return value of map.put(key, value) method with null value.

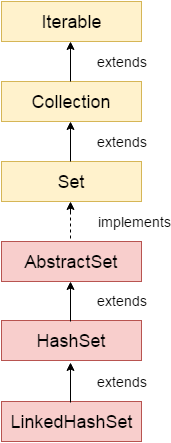
1. If map.put(key, value) returns null, then the statement “map.put(e, PRESENT) == null” will return true and element is added to the HashSet(internally HashMap).
2. If map.put(key, value) returns old value of the key, then the statement “map.put(e, PRESENT) == null” will return false and element is not added to the HashSet(internally HashMap).

# **Java LinkedHashSet Class**

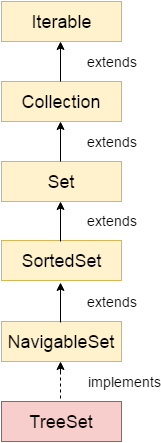
Java LinkedHashSet class is a Hashtable and Linked list implementation of the Set interface. It inherits the HashSet class and implements the Set interface.

The important points about the Java LinkedHashSet class are:

* Java LinkedHashSet class contains unique elements only like HashSet.
* Java LinkedHashSet class provides all optional set operations and permits null elements.
* Java LinkedHashSet class is non-synchronized.
* Java LinkedHashSet class maintains insertion order.



# **Java TreeSet class**



Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.

* Java TreeSet class contains unique elements only like HashSet.
* Java TreeSet class access and retrieval times are quiet fast.
* Java TreeSet class doesn't allow null element.
* Java TreeSet class is non synchronized.
* Java TreeSet class maintains ascending order.

TreeSet is basically an implementation of a self-balancing binary search tree like a [Red-Black Tree](https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/). Therefore operations like add, remove, and search takes O(log(N)) time. The reason is that in a self-balancing tree, it is made sure that the height of the tree is always O(log(N)) for all the operations. Therefore, this is considered as one of the most efficient data structures in order to store the huge sorted data and perform operations on it. However, operations like printing N elements in the sorted order take O(N) time.

# **Java Queue Interface**

The interface Queue is available in the java.util package and does extend the Collection interface. It is used to keep the elements that are processed in the First In First Out (FIFO) manner. It is an ordered list of objects, where insertion of elements occurs at the end of the list, and removal of elements occur at the beginning of the list.

## PriorityQueue Class

PriorityQueue is also class that is defined in the collection framework that gives us a way for processing the objects on the basis of priority. It is already described that the insertion and deletion of objects follows FIFO pattern in the Java queue. However, sometimes the elements of the queue are needed to be processed according to the priority, that's where a PriorityQueue comes into action.

# **Java Deque Interface**

Java Deque Interface is a linear collection that supports element insertion and removal at both ends. Deque is an acronym for **"double ended queue".**

ArrayDeque class

The ArrayDeque class provides the facility of using deque and resizable-array. It inherits AbstractCollection class and implements the Deque interface.

The important points about ArrayDeque class are:

* Unlike Queue, we can add or remove elements from both sides.
* Null elements are not allowed in the ArrayDeque.
* ArrayDeque is not thread safe, in the absence of external synchronization.
* ArrayDeque has no capacity restrictions.
* ArrayDeque is faster than LinkedList and Stack.

**Java Map Interface**

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.

A Map is useful if you have to search, update or delete elements on the basis of a key.

Java Map Hierarchy

# **Java HashMap**

* Java HashMap contains values based on the key.
* Java HashMap contains only unique keys.
* Java HashMap may have one null key and multiple null values.
* Java HashMap is non synchronized.
* Java HashMap maintains no order.
* The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

### **Hierarchy of HashMap class**



**Hashing**

Hashing is a process of converting an object into integer form by using the method hashCode(). Its necessary to write hashCode() method properly for better performance of HashMap.

[**hashCode() method**](https://www.geeksforgeeks.org/equals-hashcode-methods-java/)

hashCode() method is used to get the hash Code of an object. hashCode() method of object class returns the memory reference of object in integer form. Definition of hashCode() method is public native hashCode(). It indicates the implementation of hashCode() is native because there is not any direct method in java to fetch the reference of object. It is possible to provide your own implementation of hashCode().   
In HashMap, hashCode() is used to calculate the bucket and therefore calculate the index. 

[**equals() method**](https://www.geeksforgeeks.org/equals-hashcode-methods-java/)

equals method is used to check that 2 objects are equal or not. This method is provided by Object class. You can override this in your class to provide your own implementation.   
HashMap uses equals() to compare the key whether they are equal or not. If equals() method return true, they are equal otherwise not equal. 

**Buckets**

A bucket is one element of HashMap array. It is used to store nodes. Two or more nodes can have the same bucket. In that case link list structure is used to connect the nodes. Buckets are different in capacity. A relation between bucket and capacity is as follows: 

capacity = number of buckets \* load factor

A single bucket can have more than one nodes, it depends on hashCode() method. The better your hashCode() method is, the better your buckets will be utilized. 

**Index Calculation in Hashmap**

Hash code of key may be large enough to create an array. hash code generated may be in the range of integer and if we create arrays for such a range, then it will easily cause outOfMemoryException. So we generate index to minimize the size of array. Basically following operation is performed to calculate index. 

index = hashCode(key) & (n-1).

where n is number of buckets or the size of array. In our example, I will consider n as default size that is 16.

**Why the above method is used to calculate the index**

Using a bitwise AND operator is similar to doing bit masking wherein only the lower bits of the hash integer are considered which in turn provides a very efficient method of calculating the modulus based on the length of the hashmap.

**Index Calculation in Hashmap**

Hash code of key may be large enough to create an array. hash code generated may be in the range of integer and if we create arrays for such a range, then it will easily cause outOfMemoryException. So we generate index to minimize the size of array. Basically following operation is performed to calculate index. 

index = hashCode(key) & (n-1).

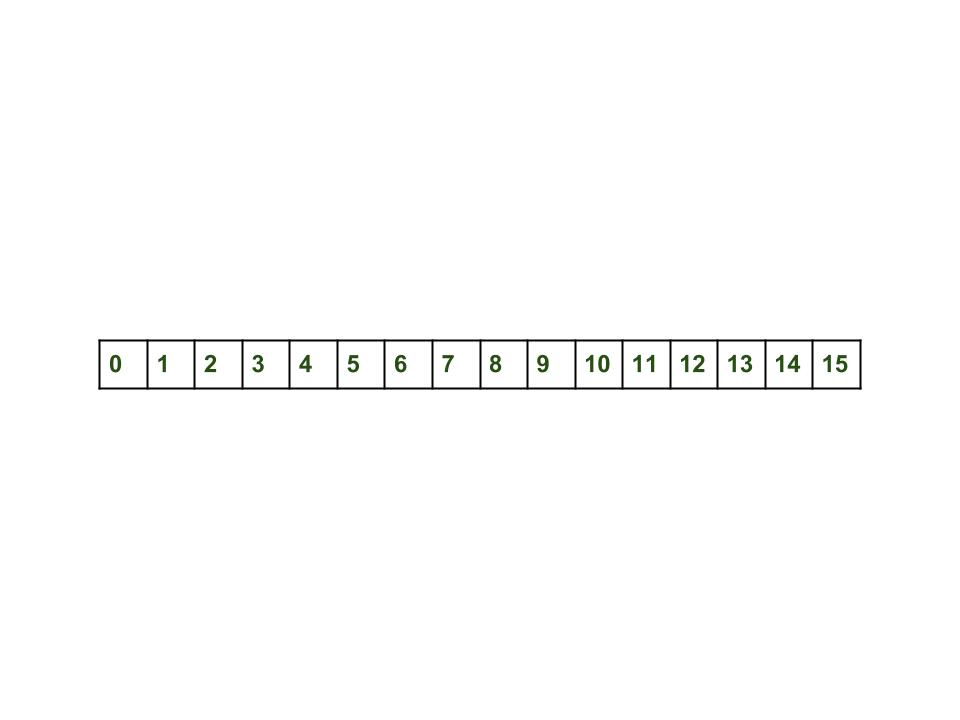
where n is number of buckets or the size of array. In our example, I will consider n as default size that is 16.

**Why the above method is used to calculate the index**

Using a bitwise AND operator is similar to doing bit masking wherein only the lower bits of the hash integer are considered which in turn provides a very efficient method of calculating the modulus based on the length of the hashmap.

* **Initially Empty hashMap:** Here, the hashmap is size is taken as 16.

HashMap map = new HashMap();



* **Initially Empty hashMap:** Here, the hashmap is size is taken as 16.
* HashMap map = new HashMap();

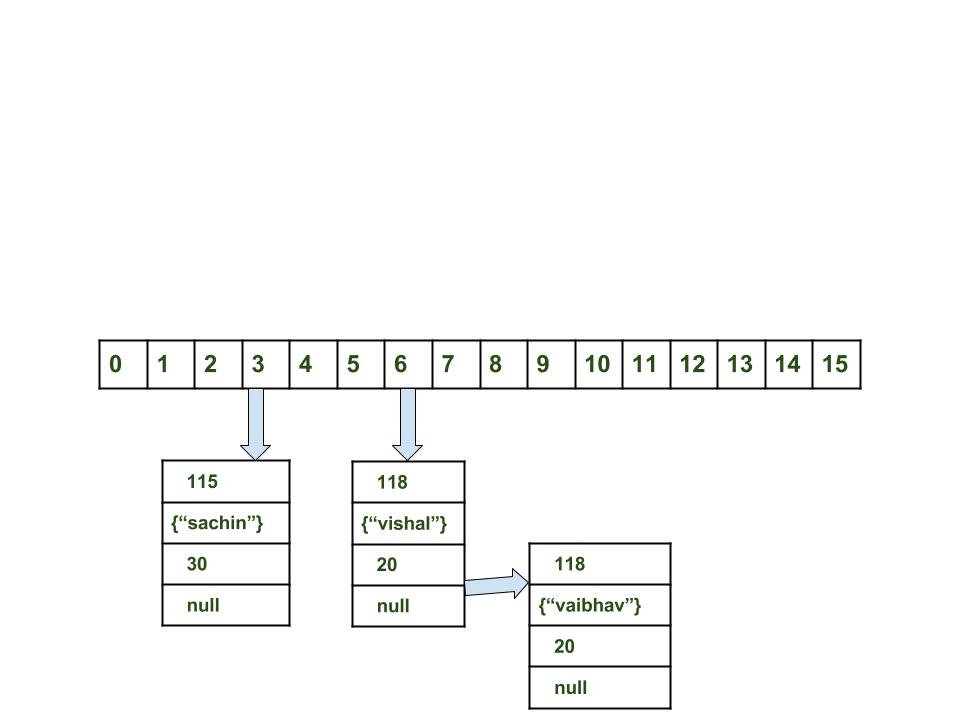
**Inserting Key-Value Pair:** Putting one key-value pair in above HashMap

* **Inserting Key-Value Pair:** Putting one key-value pair in above HashMap

map.put(new Key("vishal"), 20);

* **Steps:**
  1. Calculate hash code of Key {“vishal”}. It will be generated as 118.
  2. Calculate index by using index method it will be 6.
  3. Create a node object as :

1. Place this object at index 6 if no other object is presented there.
2. In this case a node object is **found at the index 6** – this is a case of collision.
3. In that case, check via hashCode() and equals() method that if both the keys are same.
4. If keys are same, replace the value with current value.
5. Otherwise connect this node object to the previous node object via linked list and both are stored at index 6.   
   Now HashMap becomes :



map.get(new Key("sachin"));

**Steps:**

* 1. Calculate hash code of Key {“sachin”}. It will be generated as 115.
  2. Calculate index by using index method it will be 3.
  3. Go to index 3 of array and compare first element’s key with given key. If both are equals then return the value, otherwise check for next element if it exists.
  4. In our case it is found as first element and returned value is 30.
* **Steps:**
  1. Calculate hash code of Key {“vaibhav”}. It will be generated as 118.
  2. Calculate index by using index method it will be 6.
  3. Go to index 6 of array and compare first element’s key with given key. If both are equals then return the value, otherwise check for next element if it exists.
  4. In our case it is not found as first element and next of node object is not null.
  5. If next of node is null then return null.
  6. If next of node is not null traverse to the second element and repeat the process 3 until key is not found or next is not null.
  7. Time complexity is almost constant for put and get method until rehashing is not done.
  8. In case of collision, i.e. index of two or more nodes are same, nodes are joined by link list i.e. second node is referenced by first node and third by second and so on.
  9. If key given already exist in HashMap, the value is replaced with new value.
  10. hash code of null key is 0.
  11. When getting an object with its key, the linked list is traversed until the key matches or null is found on next field.

In Java 8, HashMap replaces linked list with a binary tree when the number of elements in a bucket reaches certain threshold. While converting the list to binary tree, hashcode is used as a branching variable.

If there are two different hashcodes in the same bucket, one is considered bigger and goes to the right of the tree and other one to the left. But when both the hashcodes are equal, HashMap assumes that the keys are comparable, and compares the key to determine the direction so that some order can be maintained. It is a good practice to make the keys of HashMap comparable.

HashMap is a powerful data structure in Java used to store the key-pair values. It maps a value by its associated key. It allows us to store the null values and null keys. It is a non-synchronized class of Java collection. Whereas, ConcurrentHashMap is introduced as an alternative to the HashMap. The [ConcurrentHashMap](https://www.javatpoint.com/java-concurrenthashmap) is a synchronized collection class.

The [HashMap](https://www.javatpoint.com/java-hashmap) is non-thread-safe and can not be used in a Concurrent multi-threaded environment. Comparatively, ConcurrentHashMap is a thread-safe and specially designed for use in multi-threaded and Concurrent environment.

In this section, we will see the difference between HashMap and ConcurrentHashMap based on several parameters such as thread-safety, synchronization, performance, uses, etc.

Below are some key differences between HashMap and ConcurrentHashMap:

* As discussed above, the HashMap is a non-synchronized and non-Thread safe, while the ConcurrentHashMap is a synchronized and Thread-safe collection class. Though the ConcurrentHashMap can not match the synchronization level of Hashtable, it performs well for most of the practical cases.
* The HashMap can be synchronized using the **Collection.syncronizedMap;** It returns a collection that is almost equal to Hashtable.
* The synchronized HashMap is less scalable than the ConcurrentHashMap.
* In the multi-threaded environment, The ConcurrentHashMap has improved performance than Synchronized HashMap.
* In the single-threaded environment, The HashMap is slightly better than ConcurrentHashMap.
* In HashMap, if one thread is iterating the object and the other thread wants to modify the objects, we will get a **ConcurrentModificationException** runtime exception. But, in ConcurrentHashMap, one thread can perform modification while the other thread is running.

Advantages of ConcurrentHashMap over HashMap

The advantages of using ConcurrentHashMap are as follows:

* It provides very high concurrency in a multi-threaded environment.
* The read operation can be very fast when the write operation is done with a lock.
* It provides No object-level Locking.
* It uses a multitude of locks.
* It allows other threads to iterate the objects when one thread is iterating.
* It is thread-safe, especially in the case of multi-threading.

# **Java LinkedHashMap class**



Java LinkedHashMap class is Hashtable and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.

### **Points to remember**

* Java LinkedHashMap contains values based on the key.
* Java LinkedHashMap contains unique elements.
* Java LinkedHashMap may have one null key and multiple null values.
* Java LinkedHashMap is non synchronized.
* Java LinkedHashMap maintains insertion order.
* The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

An object that implements the Enumeration interface generates a series of elements, one at a time. Successive calls to the nextElement method return successive elements of the series.

|  |  |
| --- | --- |
| **Iterator** | **ListIterator** |
| The Iterator can traverse the array elements in the forward direction. | ListIterator can traverse the array elements in backward as well as forward directions. |
| It can be used in Queue, List, and Set. | It can be used in List. |
| It can perform only remove operation. | It can perform add, remove, and set operation while traversing the collection. |

The difference between Collection and Collections are:

|  |  |
| --- | --- |
| **Collection** | **Collections** |
| The collection is an interface. | Collections is a class. |
| It represents a group of objects as a single entity. | It defines various utility methods for collection objects. |
| The collection is the root interface of the Java Collection framework. | Collections is a general utility class. |
| This interface is used to derive the collection data structures. | This class contains static methods to manipulate data structure. |