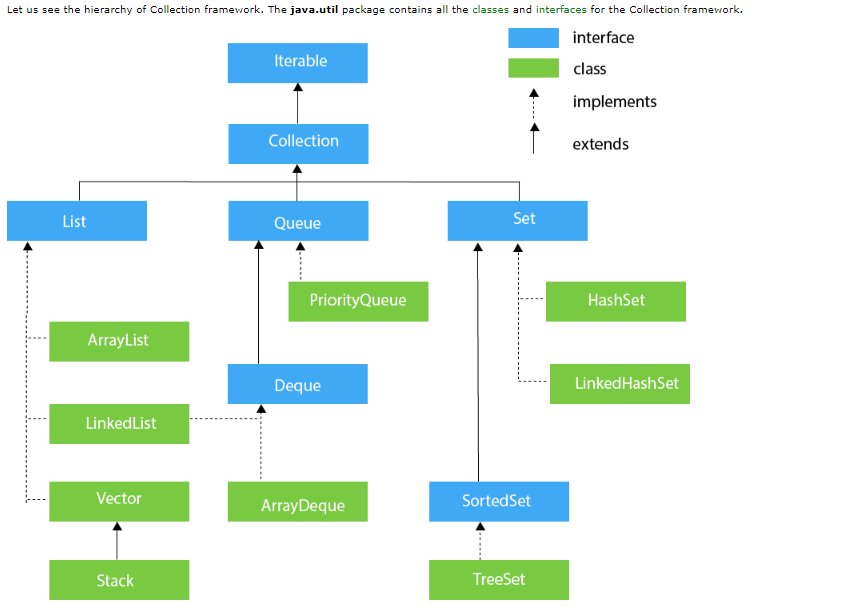
Collection Framework

<https://www.javatpoint.com/working-of-hashmap-in-java>

* What are the two ways to iterate the elements of a collection?
* What is the difference between ArrayList and LinkedList classes in collection framework?
* What is the difference between ArrayList and Vector classes in collection framework?
* What is the difference between HashSet and HashMap classes in collection framework?
* What is the difference between HashMap and Hashtable class?
* What is the difference between Iterator and Enumeration interface in collection framework?
* How can we sort the elements of an object? What is the difference between Comparable and Comparator interfaces?
* What does the hashcode() method?
* What is the difference between Java collection and Java collections?



### **Methods of Collection interface**

There are many methods declared in the Collection interface. They are as follows:

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean add(E e) | It is used to insert an element in this collection. |
| 2 | public boolean addAll(Collection<? extends E> c) | It is used to insert the specified collection elements in the invoking collection. |
| 3 | public boolean remove(Object element) | It is used to delete an element from the collection. |
| 4 | public boolean removeAll(Collection<?> c) | It is used to delete all the elements of the specified collection from the invoking collection. |
| 5 | default boolean removeIf(Predicate<? super E> filter) | It is used to delete all the elements of the collection that satisfy the specified predicate. |
| 6 | public boolean retainAll(Collection<?> c) | It is used to delete all the elements of invoking collection except the specified collection. |
| 7 | public int size() | It returns the total number of elements in the collection. |
| 8 | public void clear() | It removes the total number of elements from the collection. |
| 9 | public boolean contains(Object element) | It is used to search an element. |
| 10 | public boolean containsAll(Collection<?> c) | It is used to search the specified collection in the collection. |
| 11 | public Iterator iterator() | It returns an iterator. |
| 12 | public Object[] toArray() | It converts collection into array. |
| 13 | public <T> T[] toArray(T[] a) | It converts collection into array. Here, the runtime type of the returned array is that of the specified array. |
| 14 | public boolean isEmpty() | It checks if collection is empty. |
| 15 | default Stream<E> parallelStream() | It returns a possibly parallel Stream with the collection as its source. |
| 16 | default Stream<E> stream() | It returns a sequential Stream with the collection as its source. |
| 17 | default Spliterator<E> spliterator() | It generates a Spliterator over the specified elements in the collection. |
| 18 | public boolean equals(Object element) | It matches two collections. |
| 19 | public int hashCode() | It returns the hash code number of the collection. |

### **Iterator interface**

|  |
| --- |
| Iterator interface provides the facility of iterating the elements in a forward direction only. |

#### **Methods of Iterator interface**

There are only three methods in the Iterator interface. They are:

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean hasNext() | It returns true if the iterator has more elements otherwise it returns false. |
| 2 | public Object next() | It returns the element and moves the cursor pointer to the next element. |
| 3 | public void remove() | It removes the last elements returned by the iterator. It is less used. |

## **Iterable Interface**

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,

1. Iterator<T> iterator()

It returns the iterator over the elements of type T.

## **Collection Interface**

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Some of the methods of Collection interface are Boolean add ( Object obj), Boolean addAll ( Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

## **List Interface**

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

1. List <data-type> list1= **new** ArrayList();
2. List <data-type> list2 = **new** LinkedList();
3. List <data-type> list3 = **new** Vector();
4. List <data-type> list4 = **new** Stack();

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

The classes that implement the List interface are given below.

## **ArrayList**

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

1. **import** java.util.\*;
2. **class** TestJavaCollection1{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist
5. list.add("Ravi");//Adding object in arraylist
6. list.add("Vijay");
7. list.add("Ravi");
8. list.add("Ajay");
9. //Traversing list through Iterator
10. Iterator itr=list.iterator();
11. **while**(itr.hasNext()){
12. System.out.println(itr.next());
13. }
14. }
15. }

Output:

Ravi

Vijay

Ravi

Ajay

## **LinkedList**

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection2{
3. **public** **static** **void** main(String args[]){
4. LinkedList<String> al=**new** LinkedList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ravi");
8. al.add("Ajay");
9. Iterator<String> itr=al.iterator();
10. **while**(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }

Output:

Ravi

Vijay

Ravi

Ajay

## **Vector**

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection3{
3. **public** **static** **void** main(String args[]){
4. Vector<String> v=**new** Vector<String>();
5. v.add("Ayush");
6. v.add("Amit");
7. v.add("Ashish");
8. v.add("Garima");
9. Iterator<String> itr=v.iterator();
10. **while**(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }

Output:

Ayush

Amit

Ashish

Garima

## **Stack**

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection4{
3. **public** **static** **void** main(String args[]){
4. Stack<String> stack = **new** Stack<String>();
5. stack.push("Ayush");
6. stack.push("Garvit");
7. stack.push("Amit");
8. stack.push("Ashish");
9. stack.push("Garima");
10. stack.pop();
11. Iterator<String> itr=stack.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

Output:

Ayush

Garvit

Amit

Ashish

## **Queue Interface**

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

1. Queue<String> q1 = **new** PriorityQueue();
2. Queue<String> q2 = **new** ArrayDeque();

There are various classes that implement the Queue interface, some of them are given below.

## **PriorityQueue**

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection5{
3. **public** **static** **void** main(String args[]){
4. PriorityQueue<String> queue=**new** PriorityQueue<String>();
5. queue.add("Amit Sharma");
6. queue.add("Vijay Raj");
7. queue.add("JaiShankar");
8. queue.add("Raj");
9. System.out.println("head:"+queue.element());
10. System.out.println("head:"+queue.peek());
11. System.out.println("iterating the queue elements:");
12. Iterator itr=queue.iterator();
13. **while**(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. queue.remove();
17. queue.poll();
18. System.out.println("after removing two elements:");
19. Iterator<String> itr2=queue.iterator();
20. **while**(itr2.hasNext()){
21. System.out.println(itr2.next());
22. }
23. }
24. }

Output:

head:Amit Sharma

head:Amit Sharma

iterating the queue elements:

Amit Sharma

Raj

JaiShankar

Vijay Raj

after removing two elements:

Raj

Vijay Raj

## **Deque Interface**

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

1. Deque d = **new** ArrayDeque();

## **ArrayDeque**

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection6{
3. **public** **static** **void** main(String[] args) {
4. //Creating Deque and adding elements
5. Deque<String> deque = **new** ArrayDeque<String>();
6. deque.add("Gautam");
7. deque.add("Karan");
8. deque.add("Ajay");
9. //Traversing elements
10. **for** (String str : deque) {
11. System.out.println(str);
12. }
13. }
14. }

Output:

Gautam

Karan

Ajay

## **Set Interface**

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

1. Set<data-type> s1 = **new** HashSet<data-type>();
2. Set<data-type> s2 = **new** LinkedHashSet<data-type>();
3. Set<data-type> s3 = **new** TreeSet<data-type>();

## **HashSet**

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection7{
3. **public** **static** **void** main(String args[]){
4. //Creating HashSet and adding elements
5. HashSet<String> set=**new** HashSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=set.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

Output:

Vijay

Ravi

Ajay

## **LinkedHashSet**

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection8{
3. **public** **static** **void** main(String args[]){
4. LinkedHashSet<String> set=**new** LinkedHashSet<String>();
5. set.add("Ravi");
6. set.add("Vijay");
7. set.add("Ravi");
8. set.add("Ajay");
9. Iterator<String> itr=set.iterator();
10. **while**(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }

Output:

Ravi

Vijay

Ajay

## **SortedSet Interface**

SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

1. SortedSet<data-type> set = **new** TreeSet();

## **TreeSet**

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

Consider the following example:

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection9{
3. **public** **static** **void** main(String args[]){
4. //Creating and adding elements
5. TreeSet<String> set=**new** TreeSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //traversing elements
11. Iterator<String> itr=set.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

Output:

Ajay

Ravi

Vijay

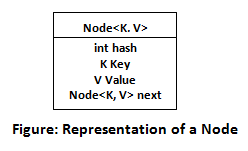
## Working of HashMap in Java

## **What is Hashing**

It is the process of converting an object into an integer value. The integer value helps in indexing and faster searches.

## **What is HashMap**

HashMap is a part of the Java collection framework. It uses a technique called Hashing. It implements the map interface. It stores the data in the pair of Key and Value. HashMap contains an array of the nodes, and the node is represented as a class. It uses an array and LinkedList data structure internally for storing Key and Value. There are four fields in HashMap.



Before understanding the internal working of HashMap, you must be aware of hashCode() and equals() method.

* **equals():** It checks the equality of two objects. It compares the Key, whether they are equal or not. It is a method of the Object class. It can be overridden. If you override the equals() method, then it is mandatory to override the hashCode() method.
* **hashCode():** This is the method of the object class. It returns the memory reference of the object in integer form. The value received from the method is used as the bucket number. The bucket number is the address of the element inside the map. Hash code of null Key is 0.
* **Buckets:** Array of the node is called buckets. Each node has a data structure like a LinkedList. More than one node can share the same bucket. It may be different in capacity.



## **Insert Key, Value pair in HashMap**

We use put() method to insert the Key and Value pair in the HashMap. The default size of HashMap is 16 (0 to 15).

### **Example**

In the following example, we want to insert three (Key, Value) pair in the HashMap.

1. HashMap<String, Integer> map = **new** HashMap<>();
2. map.put("Aman", 19);
3. map.put("Sunny", 29);
4. map.put("Ritesh", 39);

Let's see at which index the Key, value pair will be saved into HashMap. When we call the put() method, then it calculates the hash code of the Key "Aman." Suppose the hash code of "Aman" is 2657860. To store the Key in memory, we have to calculate the index.

### **Calculating Index**

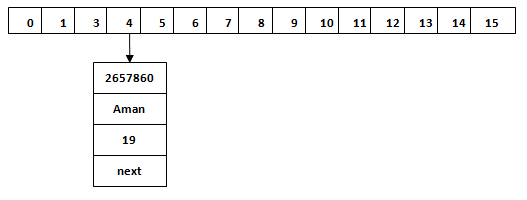
Index minimizes the size of the array. The Formula for calculating the index is:

1. Index = hashcode(Key) & (n-1)

Where n is the size of the array. Hence the index value for "Aman" is:

1. Index = 2657860 & (16-1) = 4

The value 4 is the computed index value where the Key and value will store in HashMap.

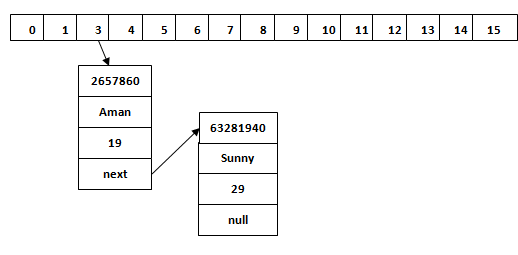


## **Hash Collision**

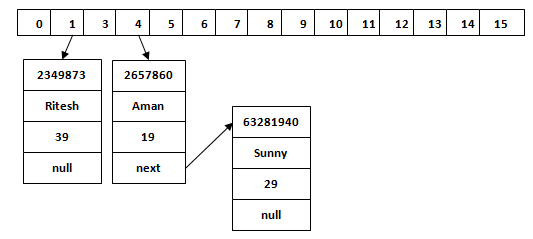
This is the case when the calculated index value is the same for two or more Keys. Let's calculate the hash code for another Key "Sunny." Suppose the hash code for "Sunny" is 63281940. To store the Key in the memory, we have to calculate index by using the index formula.

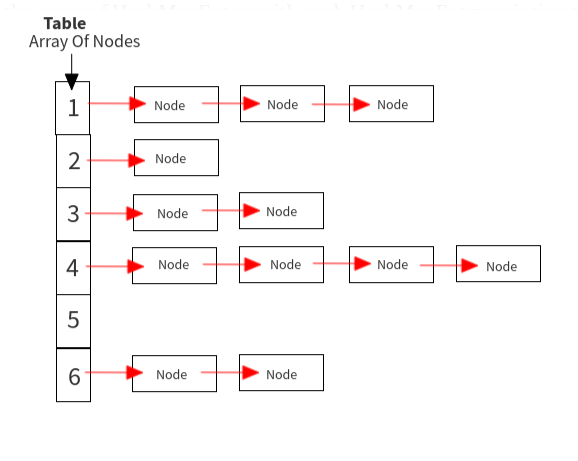
1. Index=63281940 & (16-1) = 4

The value 4 is the computed index value where the Key will be stored in HashMap. In this case, equals() method check that both Keys are equal or not. If Keys are same, replace the value with the current value. Otherwise, connect this node object to the existing node object through the LinkedList. Hence both Keys will be stored at index 4.



Similarly, we will store the Key "Ritesh." Suppose hash code for the Key is 2349873. The index value will be 1. Hence this Key will be stored at index 1.





## **get() method in HashMap**

get() method is used to get the value by its Key. It will not fetch the value if you don't know the Key. When get(K Key) method is called, it calculates the hash code of the Key.

Suppose we have to fetch the Key "Aman." The following method will be called.

1. map.get(**new** Key("Aman"));

It generates the hash code as 2657860. Now calculate the index value of 2657860 by using index formula. The index value will be 4, as we have calculated above. get() method search for the index value 4. It compares the first element Key with the given Key. If both keys are equal, then it returns the value else check for the next element in the node if it exists. In our scenario, it is found as the first element of the node and return the value 19.

Let's fetch another Key "Sunny."

The hash code of the Key "Sunny" is 63281940. The calculated index value of 63281940 is 4, as we have calculated for put() method. Go to index 4 of the array and compare the first element's Key with the given Key. It also compares Keys. In our scenario, the given Key is the second element, and the next of the node is null. It compares the second element Key with the specified Key and returns the value 29. It returns null if the next of the node is null.