# **Collections in Java**

Any group of individual objects which are represented as a single unit is known as the collection of the objects. In Java, a separate framework named the “Collection Framework” has been defined in JDK 1.2 which holds all the collection classes and interface in it.

The Collection interface (**java.util.Collection**) and Map interface (**java.util.Map**) are the two main “root” interfaces of Java collection classes.

#### What is Collection in Java

A Collection represents a single unit of objects, i.e., a group.

Collection is a container that groups multiple elements into a single unit where each element is an object.

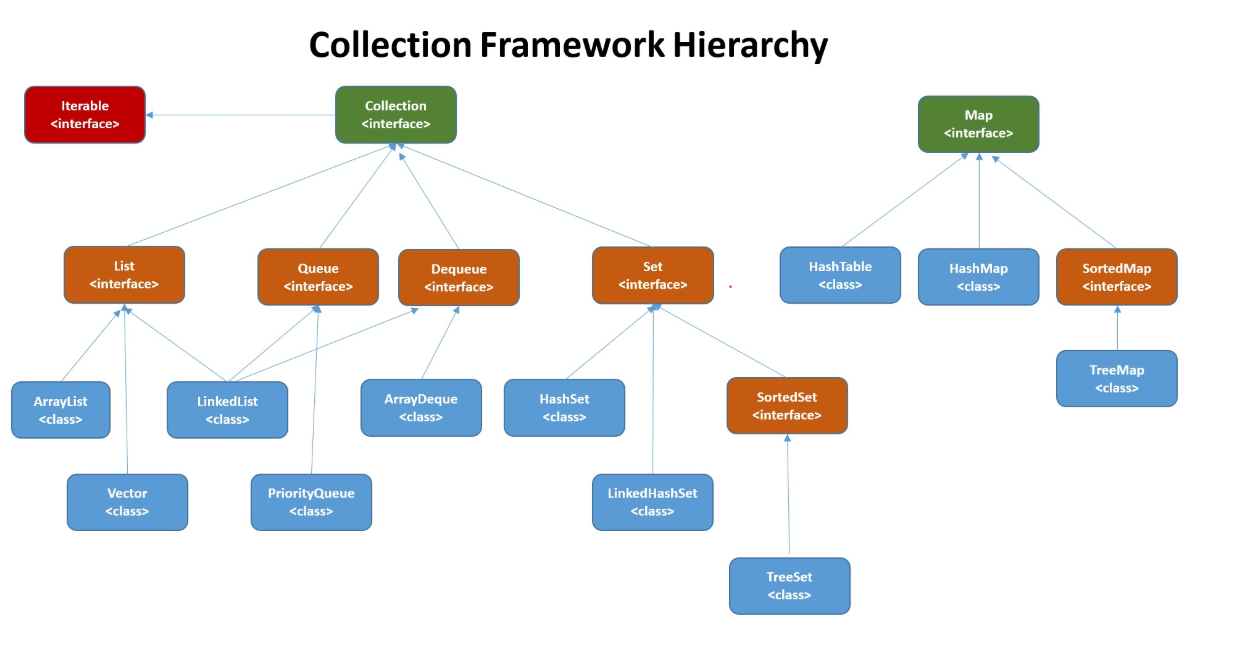
What is a framework in Java

* It provides readymade architecture.
* It represents a set of classes and interfaces.
* It is optional.

What is Collection framework

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm



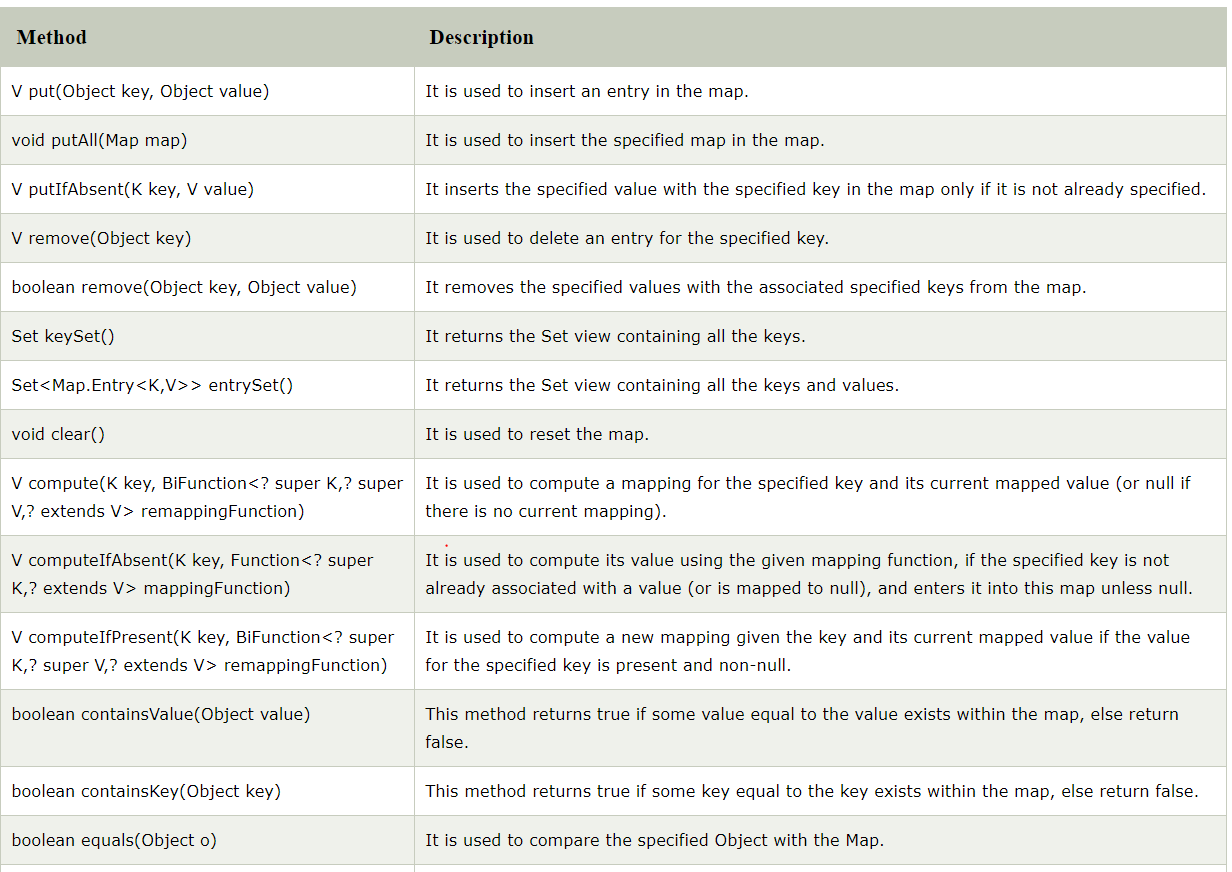
# **Map Interface in Java**

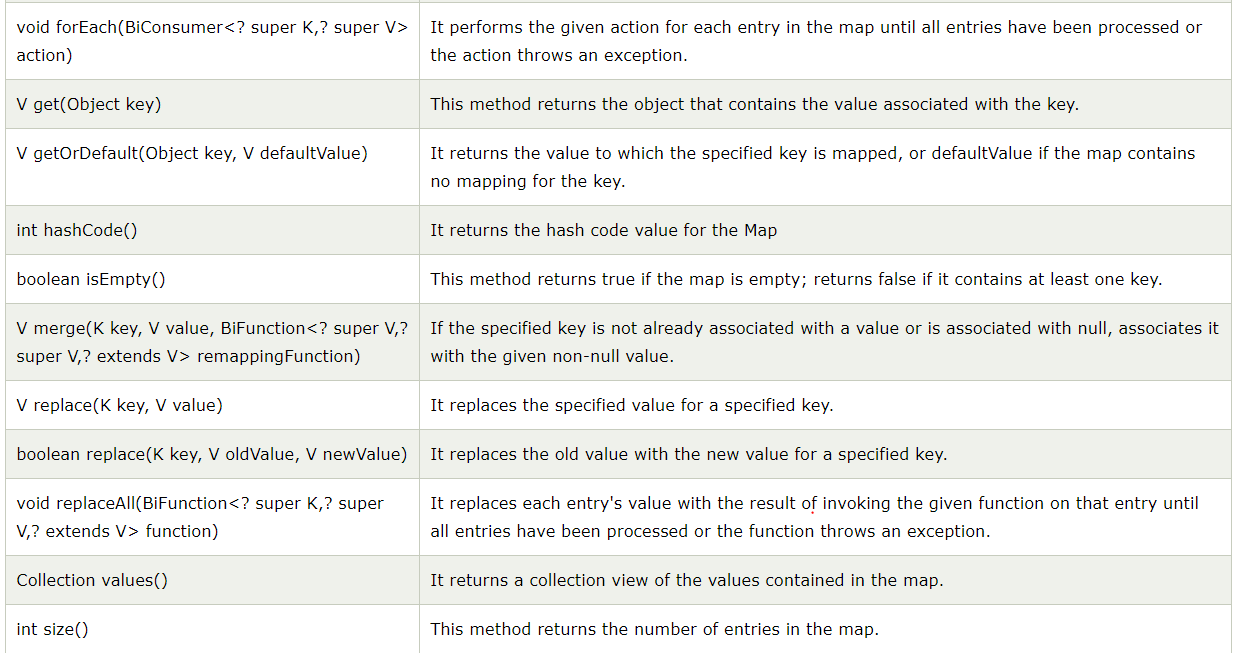
The Map interface present in [java.util](https://www.geeksforgeeks.org/java-util-package-java/) package represents a mapping between a key and a value. The Map interface is not a subtype of the [Collection interface](https://www.geeksforgeeks.org/collections-in-java-2/). Therefore it behaves a bit differently from the rest of the collection types

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

There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap





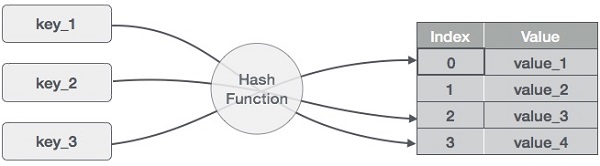
**Hashing**

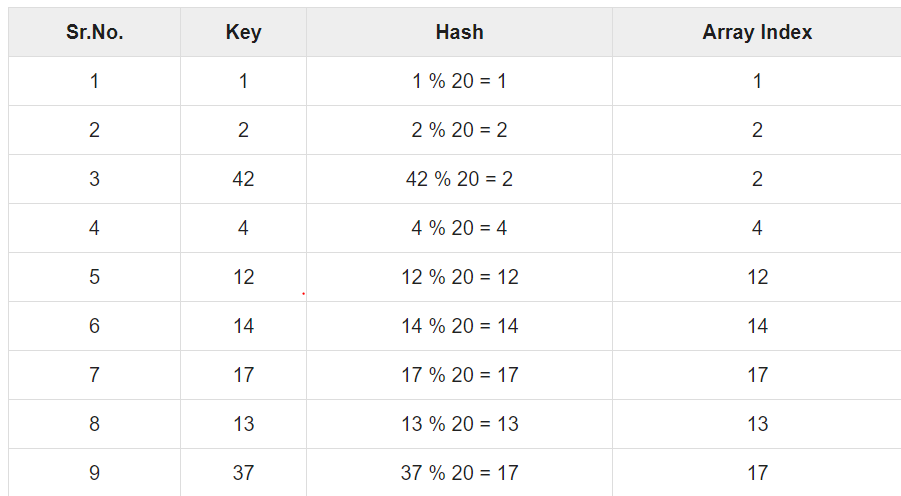
Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

## **Hashing**

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.



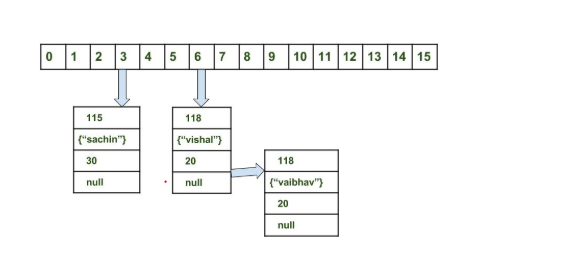


# **HashMap**

Java **HashMap** class implements the Map interface which allows us *to store key and value pair*, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.

HashMap in Java is like the legacy Hashtable class, but it is not synchronized. It allows us to store the null elements as well, but there should be only one null key. Since Java 5, it is denoted as HashMap<K,V>, where K stands for key and V for value. It inherits the AbstractMap class and implements the Map interface.

* 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.

**HashMap Changes in Java 8**

As we know now that in case of hash collision entry objects are stored as a node in a[linked-list](https://www.geeksforgeeks.org/linked-list-set-1-introduction/) and equals() method is used to compare keys. That comparison to find the correct key with in a linked-list is a linear operation so in a worst case scenario the complexity becomes O(n).  
To address this issue, Java 8 hash elements use balanced trees instead of linked lists after a certain threshold is reached. Which means HashMap starts with storing Entry objects in linked list but after the number of items in a hash becomes larger than a certain threshold, the hash will change from using a linked list to a balanced tree, which will improve the worst case performance from O(n) to O(log n).

# **LinkedHashMap class**

LinkedHashMap is just like [HashMap](http://www.geeksforgeeks.org/java-util-hashmap-in-java/) with an additional feature of maintaining an order of elements inserted into it. HashMap provided the advantage of quick insertion, search, and deletion but it never maintained the track and order of insertion which the LinkedHashMap provides where the elements can be accessed in their insertion order.

**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.

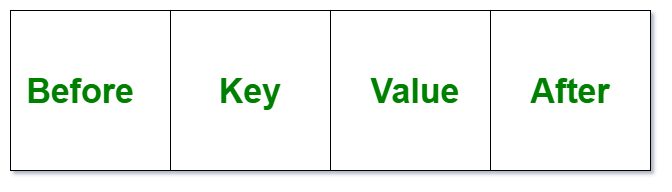


**How LinkedHashMap work internally?**

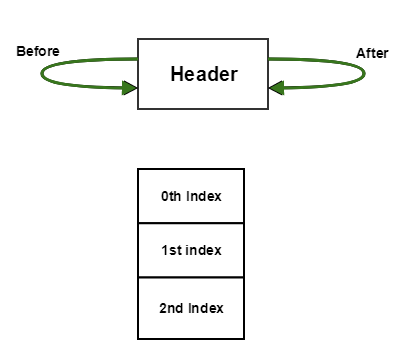
A LinkedHashMap is an extension of the HashMap class and it implements the Map interface. Therefore, the class is declared as:

*public class LinkedHashMap extends HashMap implements Map*

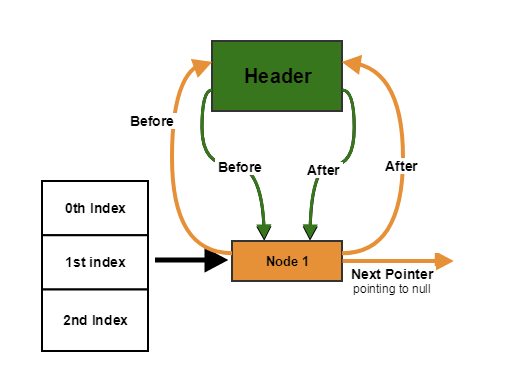
In this class, the data is stored in the form of nodes. The implementation of the LinkedHashMap is very similar to a [doubly-linked list](https://www.geeksforgeeks.org/doubly-linked-list/). Therefore, each node of the LinkedHashMap is represented as:



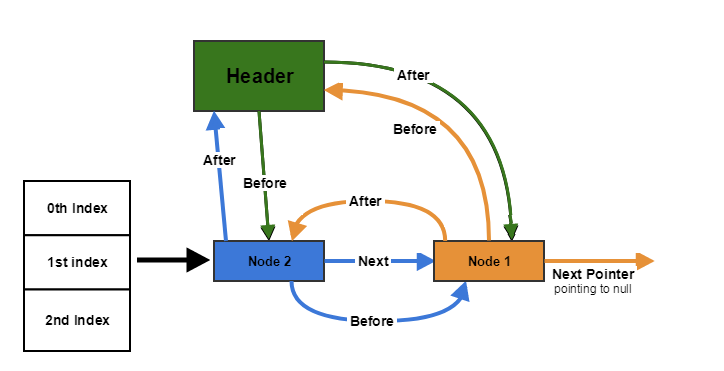
Initially, the LinkedHashMap looks like:



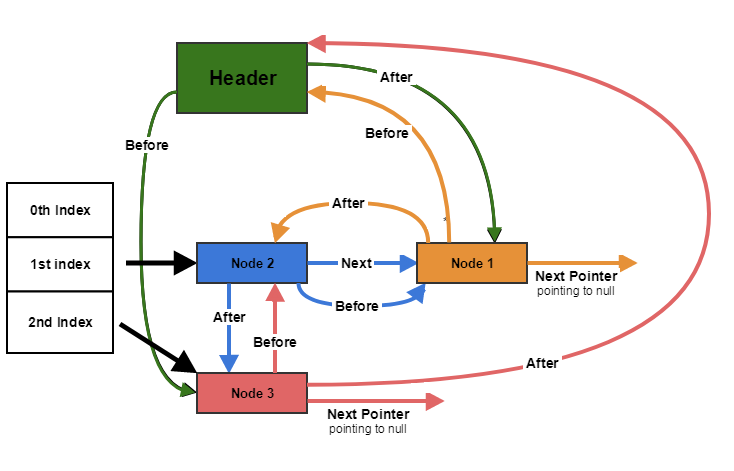
now Put 1 into it.  
The LinkedHashMap now becomes:



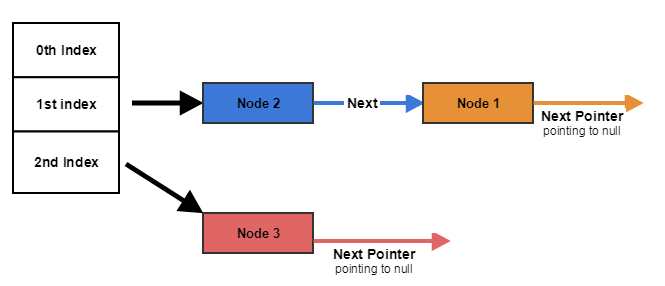
Now put another element say Node 2 and the state will then become:



Lets put Node3 which is placed at index 2 say.

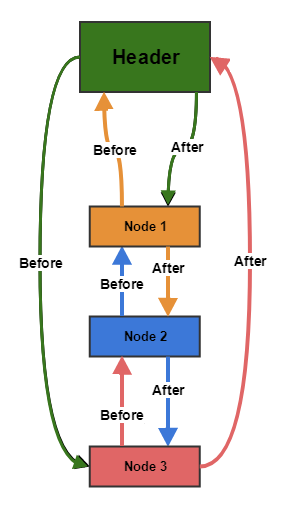


So as seen we have 2 different types of linked lists:



The one that is created with hashMaps.

And the other LinkedList created for insertion maintenance.

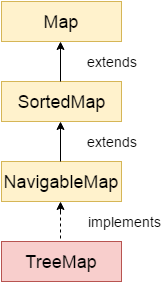


# **TreeMap**

Java TreeMap class is a red-black tree based implementation. It provides an efficient means of storing key-value pairs in sorted order.

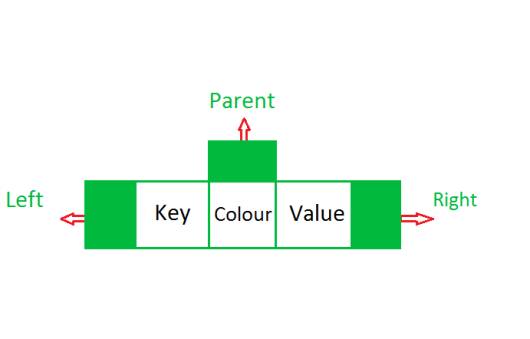
The important points about Java TreeMap class are:

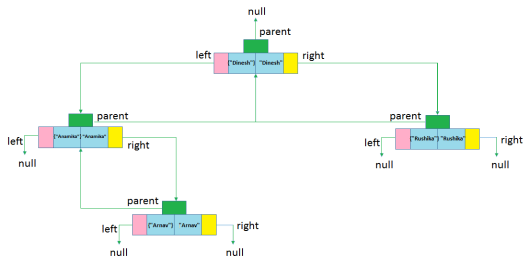
* Java TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
* Java TreeMap contains only unique elements.
* Java TreeMap cannot have a null key but can have multiple null values.
* Java TreeMap is non synchronized.
* Java TreeMap maintains ascending order



## **Internal Structure of TreeMap**

TreeMap is based on tree data structure as its name suggested. As we know that, in a tree, each node has three references its parent, right and left element. Let’s see the following diagram:



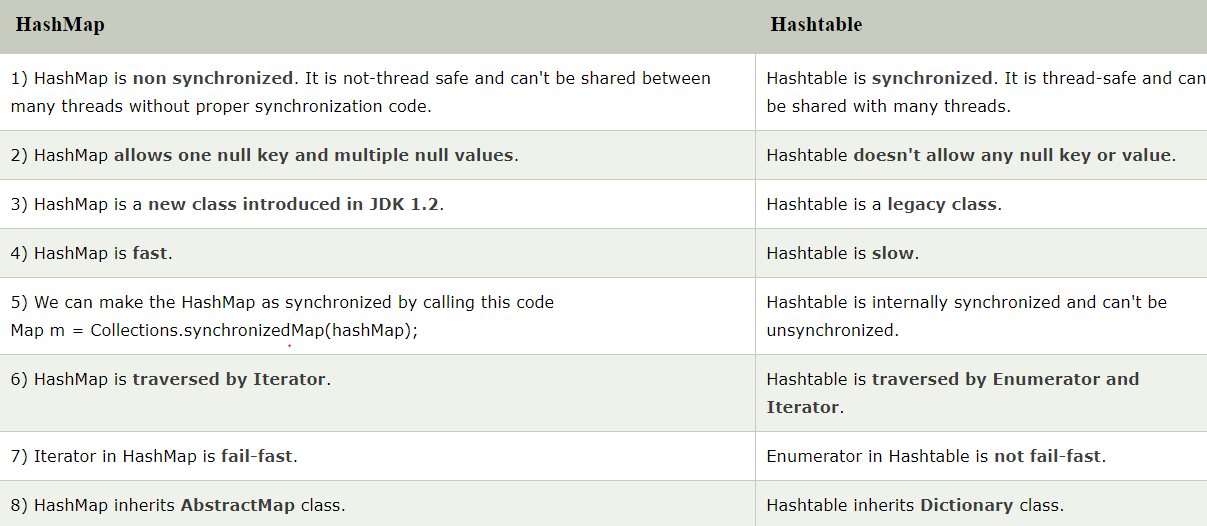


# **Java Hashtable class**

Java Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.

### **Points to remember**

* A Hashtable is an array of a list. Each list is known as a bucket. The position of the bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.
* Java Hashtable class contains unique elements.
* Java Hashtable class doesn't allow null key or value.
* Java Hashtable class is synchronized.
* The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.



## **Java EnumMap Example**

ava EnumMap class is the specialized Map implementation for enum keys. It inherits Enum and AbstractMap classes.

**import** java.util.\*;

**public** **class** EnumMapExample {

   // create an enum

**public** **enum** Days {

   Monday, Tuesday, Wednesday, Thursday

   };

**public** **static** **void** main(String[] args) {

   //create and populate enum map

   EnumMap<Days, String> map = **new** EnumMap<Days, String>(Days.**class**);

   map.put(Days.Monday, "1");

   map.put(Days.Tuesday, "2");

   map.put(Days.Wednesday, "3");

   map.put(Days.Thursday, "4");

   // print the map

**for**(Map.Entry m:map.entrySet()){

       System.out.println(m.getKey()+" "+m.getValue());

      }

   }

}

# **ConcurrentHashMap in java**

**Prerequisites:**Need of ConcurrentMap  
**ConcurrentHashMap** ConcurrentHashMap class is introduced in JDK 1.5, which implements ConcurrentMap as well as Serializable interface also. ConcureentHashMap is enhancement of HashMap as we know that while dealing with Threads in our application HashMap is not a good choice because performance wise HashMap is not upto the mark.

**Key points of ConcurrentHashMap:**

* The underlined data structure for ConcurrentHashMap is Hashtable.
* ConcurrentHashMap class is thread-safe i.e. multiple thread can operate on a single object without any complications.
* At a time any number of threads are applicable for read operation without locking the ConcurrentHashMap object which is not there in HashMap.
* In ConcurrentHashMap, the Object is divided into number of segments according to the concurrency level.
* Default concurrency-level of ConcurrentHashMap is 16.
* In ConcurrentHashMap, at a time any number of threads can perform retrieval operation but for updation in object, thread must lock the particular segment in which thread want to operate.This type of locking mechanism is known as **Segment locking or bucket locking**.Hence at a time 16 updation operations can be performed by threads.
* null insertion is not possible in ConcurrentHashMap as key or value.

### **2.1. Strong References**

The strong reference is the most common type of Reference that we use in our day to day programming:

Integer prime = 1;

The variable prime has a *strong reference*to an Integer object with value 1. Any object which has a strong reference pointing to it is not eligible for GC.

### **2.2. Soft References**

Simply put, an object that has a [*SoftReference*](https://docs.oracle.com/javase/8/docs/api/java/lang/ref/SoftReference.html) pointing to it won't be garbage collected until the JVM absolutely needs memory.

Let's see how we can create a SoftReference in Java:

|  |  |
| --- | --- |
|  | Integer prime = 1; |
|  | SoftReference<Integer> soft = **new** SoftReference<Integer>(prime); |
|  | prime = **null**; |

The prime object has a strong reference pointing to it.

Next, we are wrapping prime strong reference into a soft reference. After making that strong reference null, a prime object is eligible for GC but will be collected only when JVM absolutely needs memory.

### **2.3. Weak References**

The objects that are referenced only by weak references are garbage collected eagerly; the GC won't wait until it needs memory in that case.

We can create a *WeakReference* in Java in the following way:

|  |  |
| --- | --- |
|  | Integer prime = 1; |
|  | WeakReference<Integer> soft = **new** WeakReference<Integer>(prime); |
|  | prime = **null**; |

When we made a *prime* reference *null*, the *prime* object will be garbage collected in the next GC cycle, as there is no other strong reference pointing to it.

WeakHashMap is the Hash table based implementation of the Map interface, with weak keys. An entry in a WeakHashMap will automatically be removed when its key is no longer in ordinary use. More precisely, the presence of a mapping for a given key will not prevent the key from being discarded by the garbage collector, that is, made finalizable, finalized, and then reclaimed. When a key has been discarded its entry is effectively removed from the map, so this class behaves somewhat differently from other Map implementations. Few important features of a weak hashmap are:

* Both null values and null keys are supported in WeakHashMap.
* It is not synchronised.
* This class is intended primarily for use with key objects whose equals methods test for object identity using the == operator.

# Identity hash map

This class implements AbstractMap. It is similar to HashMap except that it uses reference equality when comparing the elements.

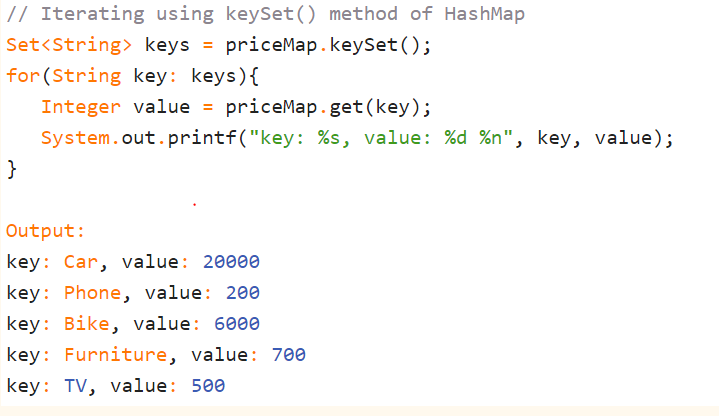
This class is not a general-purpose Map implementation. While this class implements the Map interface, it intentionally violates Map's general contract, which mandates the use of the equals method when comparing objects.

* IdentityHashMap uses equality operator “==” for comparing keys and values while HashMap uses equals method for comparing keys and values inside Map.
* Since IdentityHashMap doesn’t use equals() its comparatively faster than HashMap for object with expensive equals().
* IdentityHashMap doesn’t require keys to be immutable as it is not relied on equals().

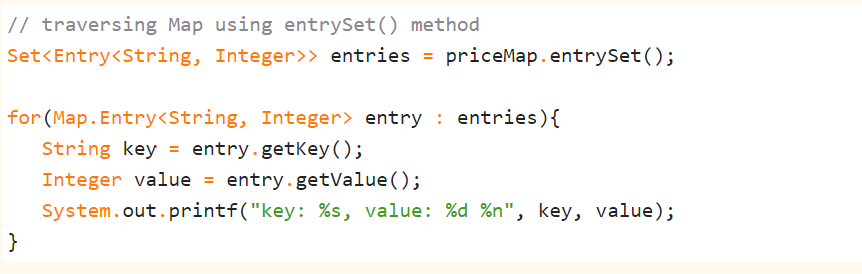
The java.util.Map interface provides three methods keySet(), entrySet() and values() to retrieve all keys, entries (a key-value pair), and values

## The keySet() method

This method returns a Set view of all the keys in the map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa.

If the map is modified while an [iteration over the Set](http://java67.blogspot.com/2012/10/how-to-iterate-over-hashset-in-java.html) is in progress (except through the iterator's own remove operation), the results of the iteration are undefined.  
  
  
The set supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove(), Set.remove(), removeAll(), retainAll(), and clear() operations, but It does not support the add() or addAll() operations.  
  


## The entrySet() method

The entrySet() method of Map interface returns a Set view of the mappings contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa.  
  
If the map is modified while an iteration over the set is in progress (except through the iterator's own remove operation, or through the setValue() operation on a map entry returned by the iterator) the results of the iteration are undefined.  
  
The Set also supports element removals, which removes the corresponding mapping from the map, via the Iterator.remove(), Set.remove(), removeAll(), retainAll(), and clear() operations. It does not support the add() or addAll() operations.  
  


## The values() methods

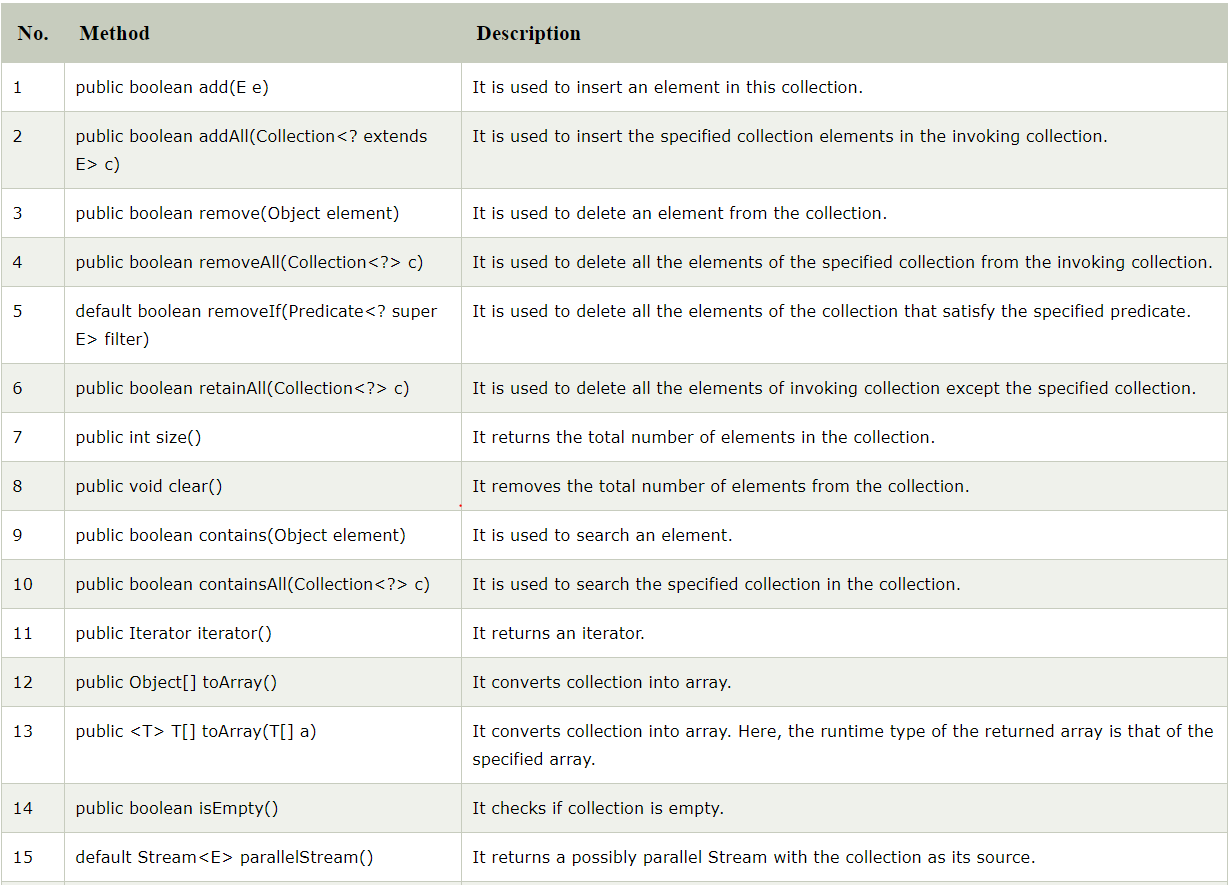
The values() methods of Map interface returns a Collection view of the values contained in this map. The collection is backed by the map, so changes to the map are reflected in the collection, and vice-versa.  
  
If the map is modified while an iteration over the collection is in progress (except through the iterator's own remove operation), the results of the iteration are undefined. The collection supports element removal, which removes the corresponding mapping from the map, via the [Iterator.remove()](http://java67.blogspot.com/2014/03/2-ways-to-remove-elementsobjects-from-ArrayList-java.html), Collection.remove(), [removeAll()](http://javarevisited.blogspot.com/2015/09/how-to-reset-arraylist-in-java-clear-vs-removeAll-example.html), retainAll() and clear() operations.  
  
Similar to keySet() and entrySet(), It does not support the add() or addAll() operations.  
  
  
Collection<Integer> values = priceMap.values();

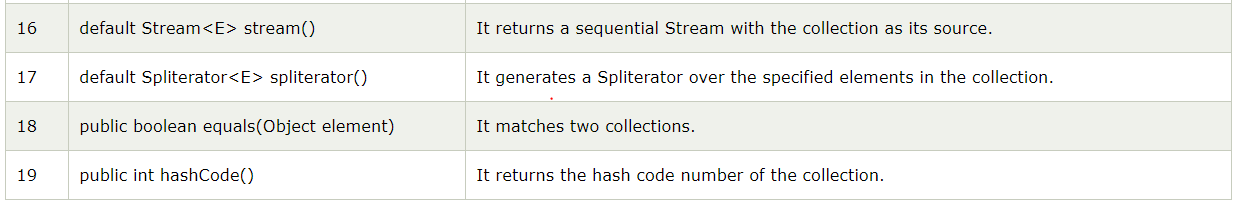
# Collection Framework

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes ([ArrayList](https://www.javatpoint.com/java-arraylist), Vector, [LinkedList](https://www.javatpoint.com/java-linkedlist), [PriorityQueue](https://www.javatpoint.com/java-priorityqueue), HashSet, LinkedHashSet, TreeSet).

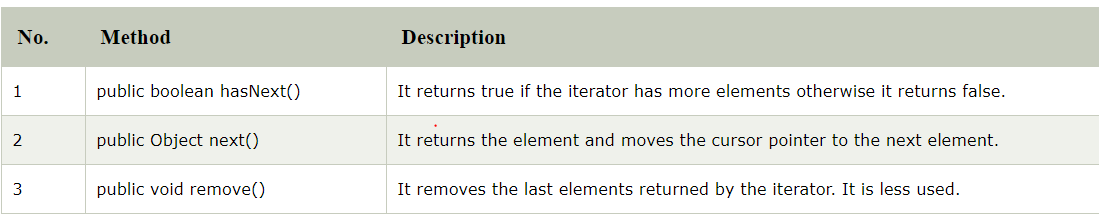




## Iterator interface

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

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



## 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.,

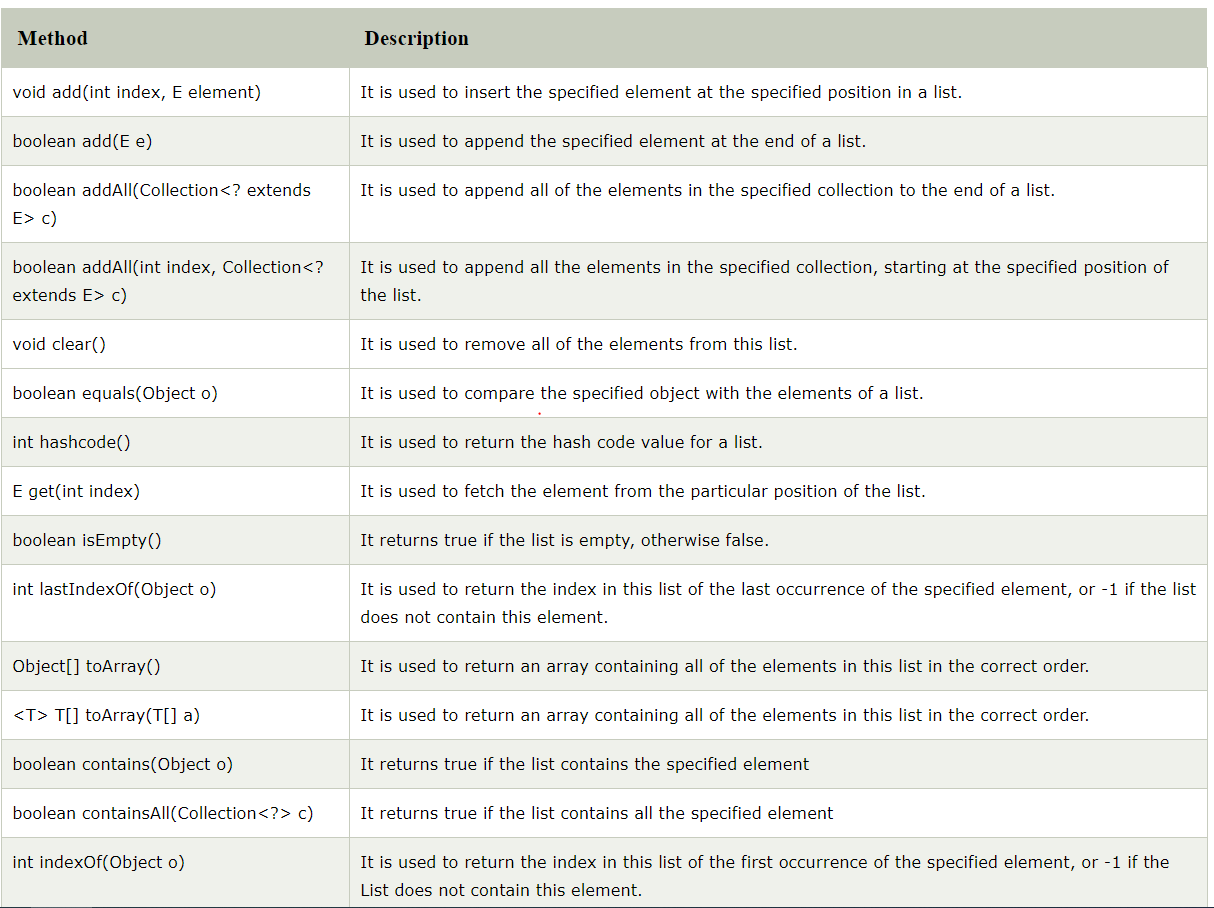
Iterator<T> iterator()

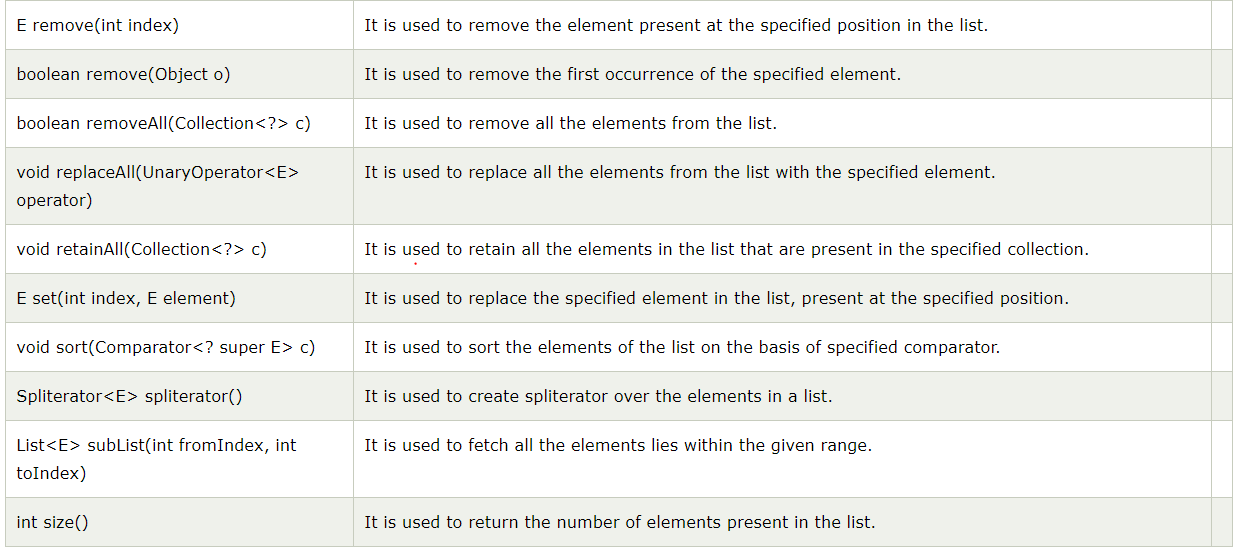
It returns the iterator over the elements of type T.

## Java List

**List** in Java provides the facility to maintain the *ordered collection*. It contains the index-based methods to insert, update, delete and search the elements. It can have the duplicate elements also. We can also store the null elements in the list.

The List interface is found in the java.util package and inherits the Collection interface. It is a factory of ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions. The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector. The ArrayList and LinkedList are widely used in Java programming. The Vector class is deprecated since Java 5.

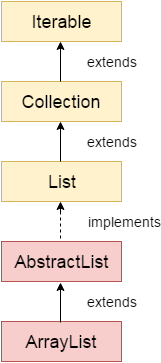




# Java ArrayList

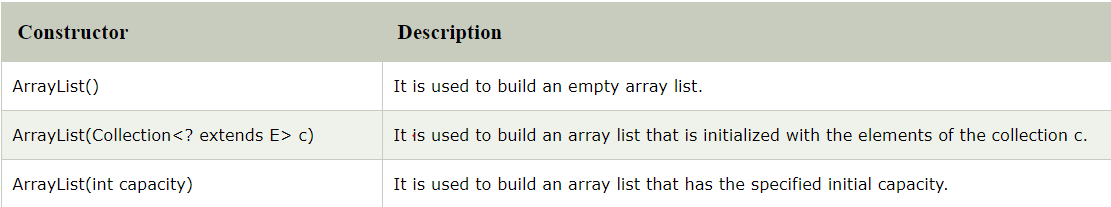
Java **ArrayList** class uses a dynamic [*array*](https://www.javatpoint.com/array-in-java) for storing the elements. It is like an array, but there is no size limit. We can add or remove elements anytime. So, it is much more flexible than the traditional array. It is found in the java.util package.

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



The important points about 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 array works at the index basis.
* In ArrayList, manipulation is 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.



## Java Non-generic Vs. Generic Collection

Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.

Java new generic collection allows you to have only one type of object in a collection. Now it is type safe so typecasting is not required at runtime.

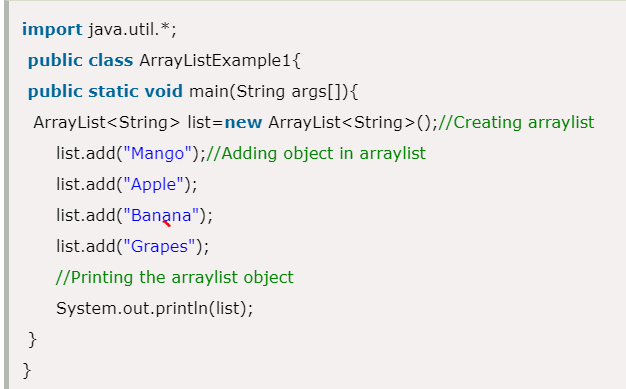
Let's see the old non-generic example of creating java collection.

1. ArrayList list=**new** ArrayList();//creating old non-generic arraylist

Let's see the new generic example of creating java collection.

1. ArrayList<String> list=**new** ArrayList<String>();//creating new generic arraylist

In a generic collection, we specify the type in angular braces. Now ArrayList is forced to have the only specified type of objects in it. If you try to add another type of object, it gives *compile time error*.



## LinkedList class

Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

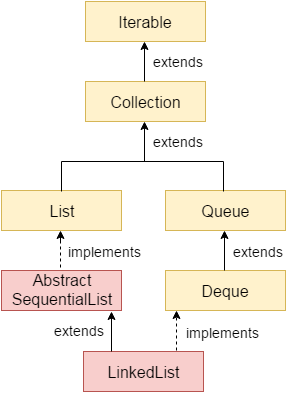
The important points about Java LinkedList are:

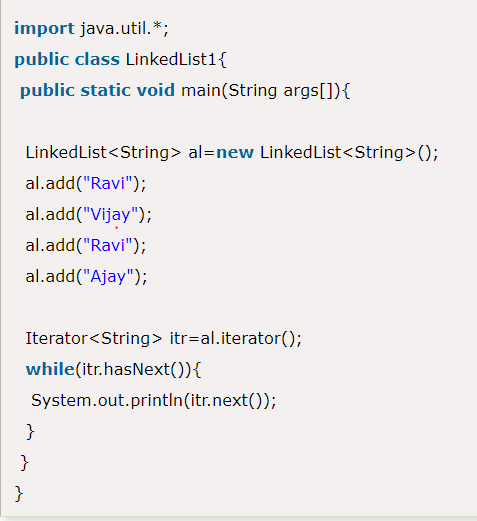
* Java LinkedList class can contain duplicate elements.
* Java LinkedList class maintains insertion order.
* Java LinkedList class is non synchronized.
* In Java LinkedList class, manipulation is fast because no shifting needs to occur.
* Java LinkedList class can be used as a list, stack or queue.

## Doubly Linked List

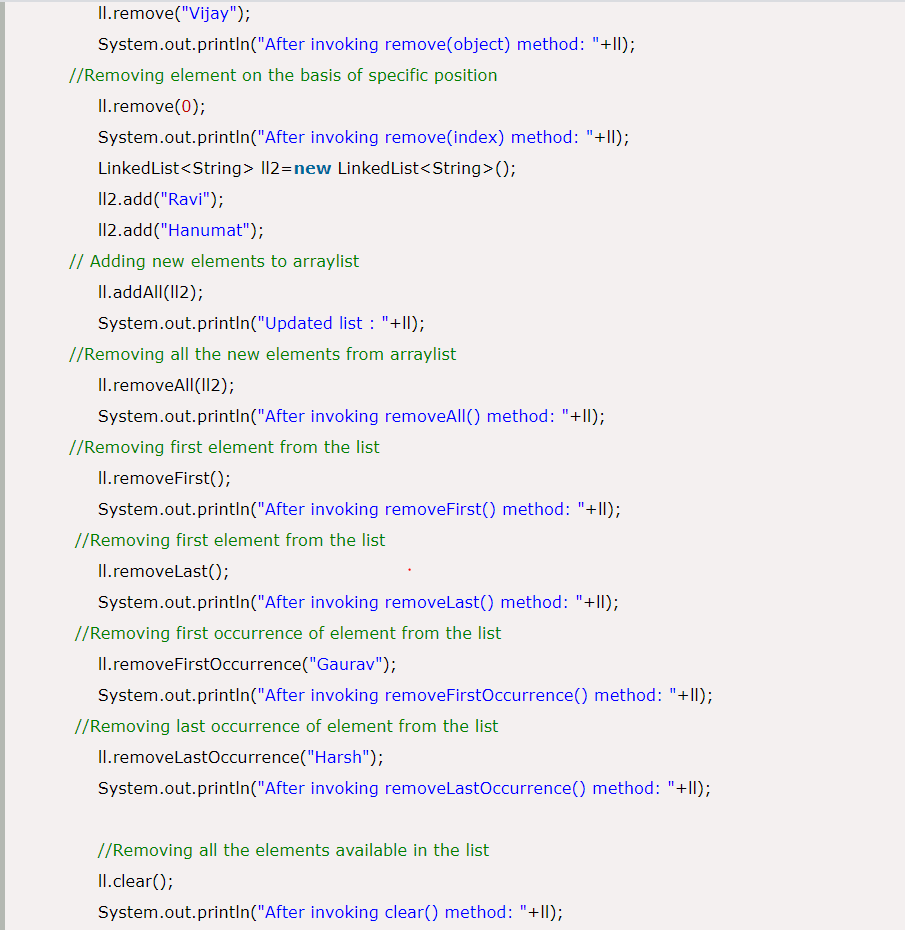
In the case of a doubly linked list, we can add or remove elements from both sides.







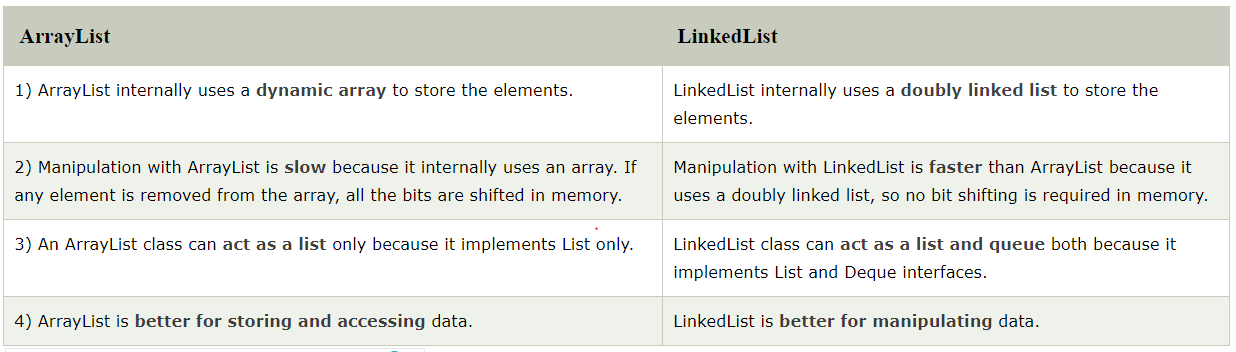




## Difference between ArrayList and LinkedList

ArrayList and LinkedList both implements List interface and maintains insertion order. Both are non synchronized classes.

However, there are many differences between ArrayList and LinkedList classes that are given below.



## Java HashSet



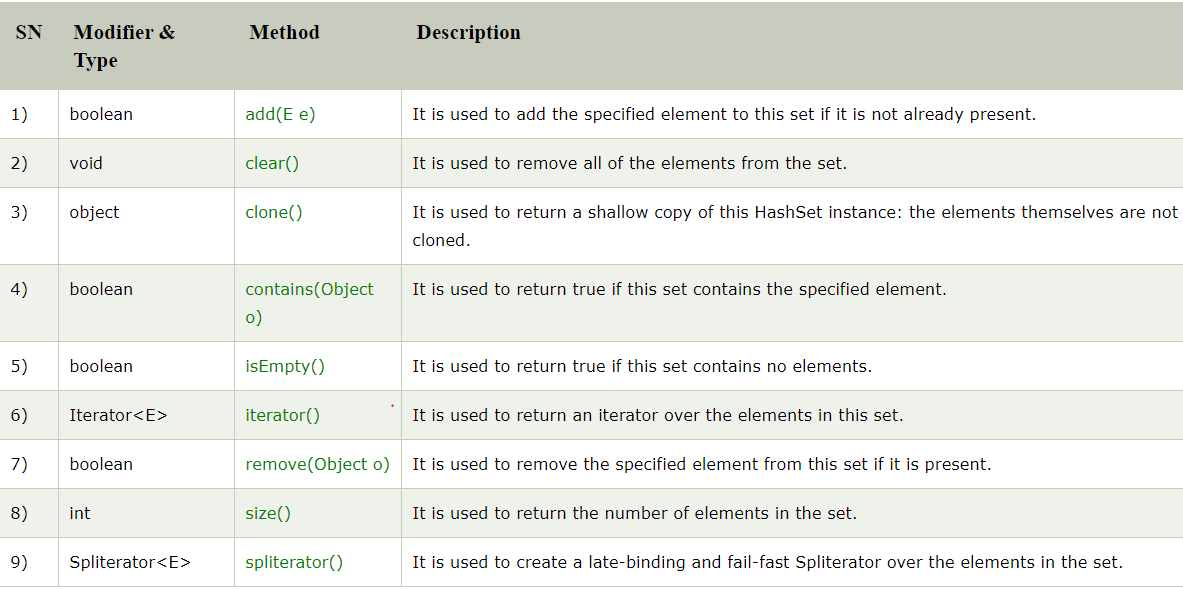
Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

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.

## Difference between List and Set

A list can contain duplicate elements whereas Set contains unique elements only.



**HashSet** uses HashMap internally to store it’s 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 HashSet are stored as **keys** of this HashMap object. The value associated with those keys will be a **constant**

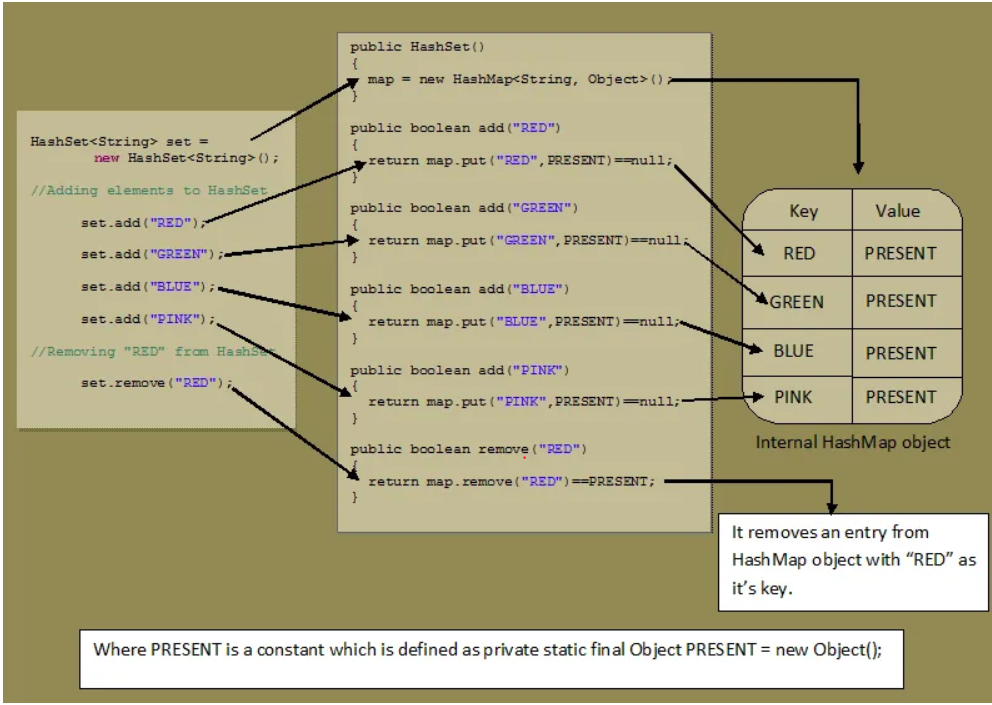


Whenever you insert an element into HashSet using **add()** method, it actually creates an entry in the internally backing HashMap object with element you have specified as it’s key and constant called “**PRESENT**” as it’s value. This “PRESENT” is defined in the HashSet class as below.

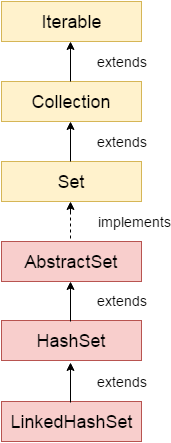
|  |  |
| --- | --- |
|  | // Dummy value to associate with an Object in the backing Map  private static final Object PRESENT = new Object(); |

Let’s have a look at add() method of HashSet class.

|  |  |
| --- | --- |
|  | public boolean add(E e)  {          return map.put(e, PRESENT)==null;  } |



# Java LinkedHashSet class

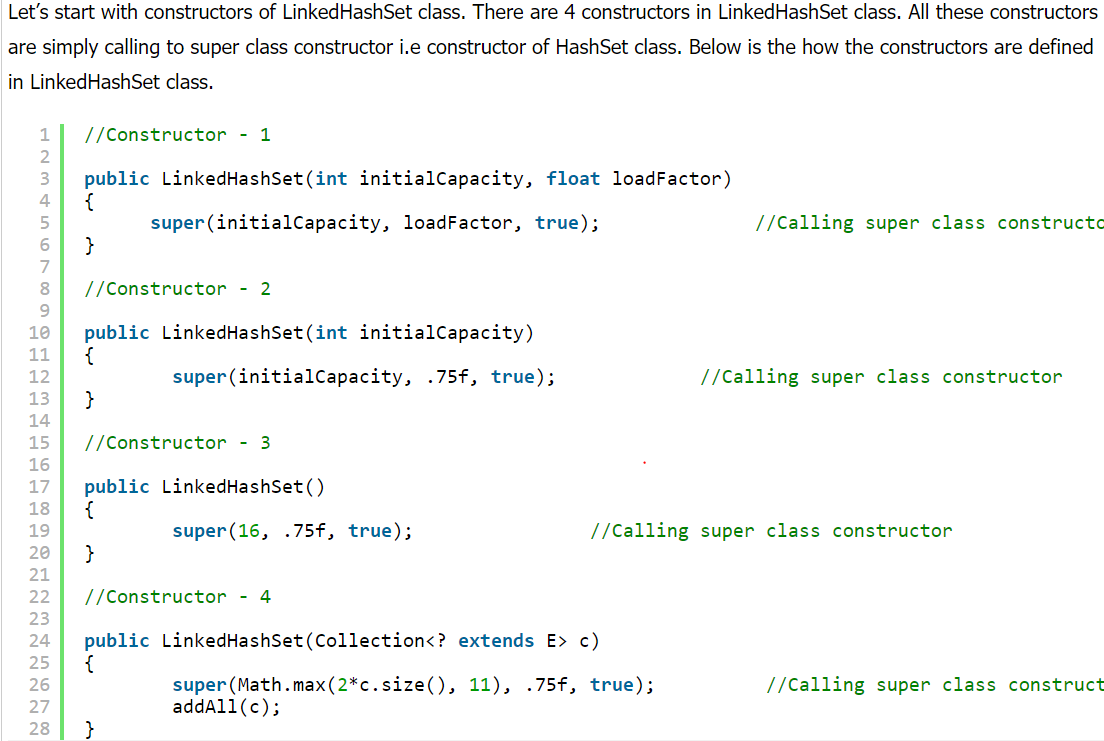


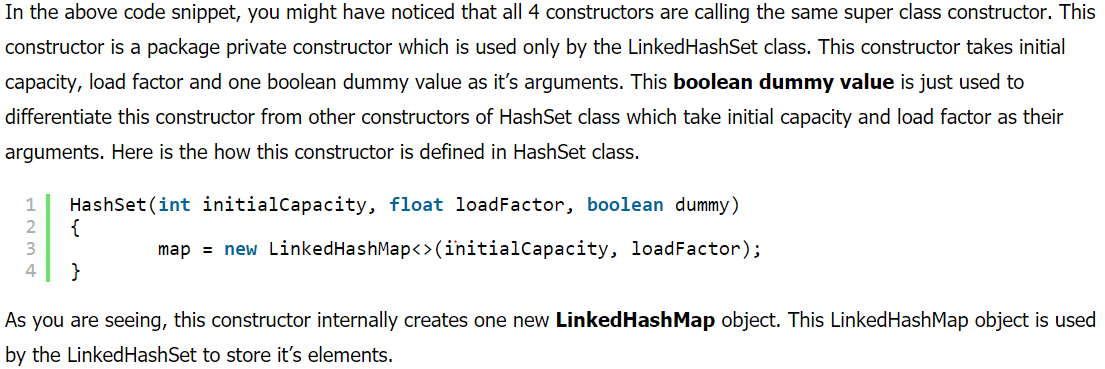
Java LinkedHashSet class is a Hashtable and Linked list implementation of the set interface. It inherits HashSet class and implements Set interface.

The important points about Java LinkedHashSet class are:

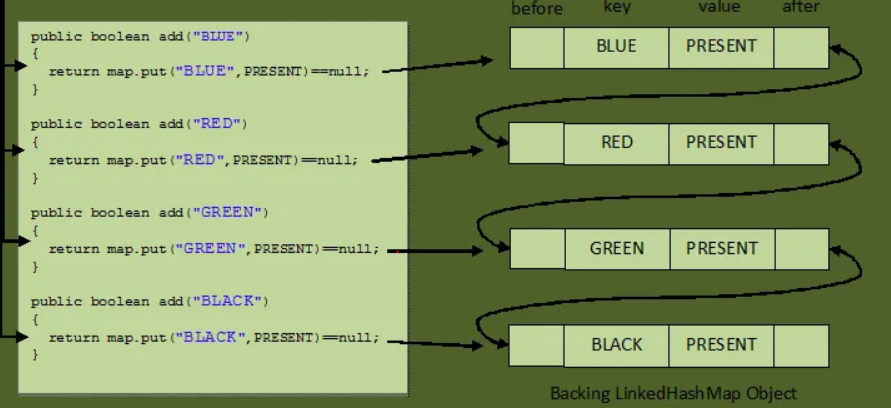
* Java LinkedHashSet class contains unique elements only like HashSet.
* Java LinkedHashSet class provides all optional set operation and permits null elements.
* Java LinkedHashSet class is non synchronized.
* Java LinkedHashSet class maintains insertion order.

LinkedHashSet is an **extended version** of HashSet. HashSet doesn’t follow any order where as LinkedHashSet maintains **insertion order**. HashSet uses **HashMap object** internally to store it’s elements where as LinkedHashSet uses **LinkedHashMap object** internally to store and process it’s elements



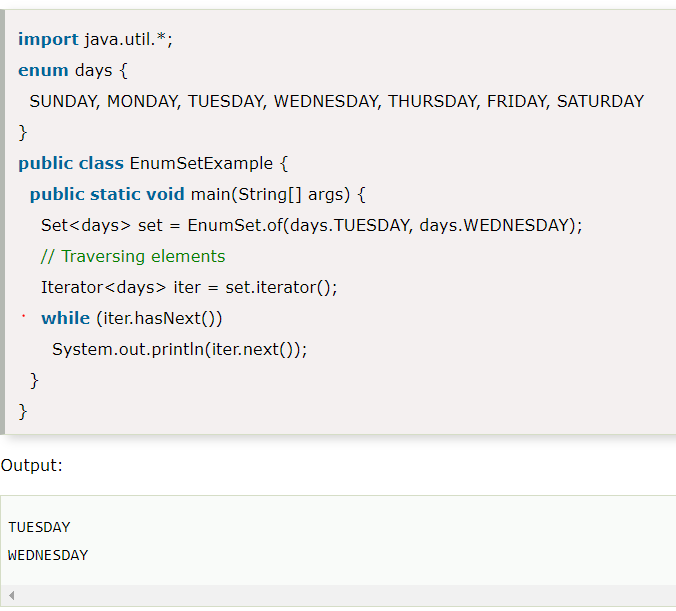


LinkedHashSet doesn’t have it’s own methods. All methods are inherited from it’s super class i.e HashSet. So. all operations on LinkedHashSet work in the same manner as that of HashSet. The only change is the internal object used to store the elements. In hashSet, elements you insert are stored as **keys of HashMap** object. Where as in LinkedHashSet, elements you insert are stored as **keys of LinkedHashMap** object. The values of these keys will be the same constant i.e “**PRESENT**“

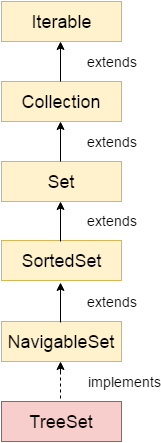


# Java EnumSet class

Java EnumSet class is the specialized Set implementation for use with enum types. It inherits AbstractSet class and implements the Set interface.



# 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.

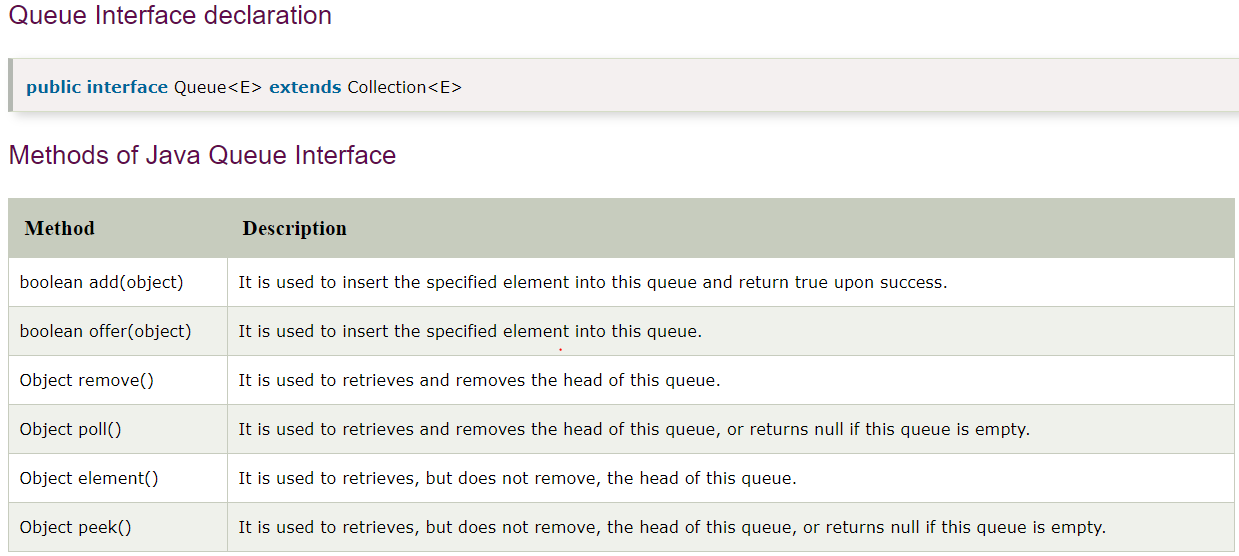
The important points about Java TreeSet class are:

* 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.

The TreeSet class internally uses a [TreeMap](https://www.callicoder.com/java-treemap/) to store elements. The elements in a TreeSet are sorted according to their natural ordering. You may also provide a custom [Comparator](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) to the TreeSet at the time of creation to let it sort the elements based on the supplied comparator.

# Java Queue Interface

Java Queue interface orders the element in FIFO(First In First Out) manner. In FIFO, first element is removed first and last element is removed at last.

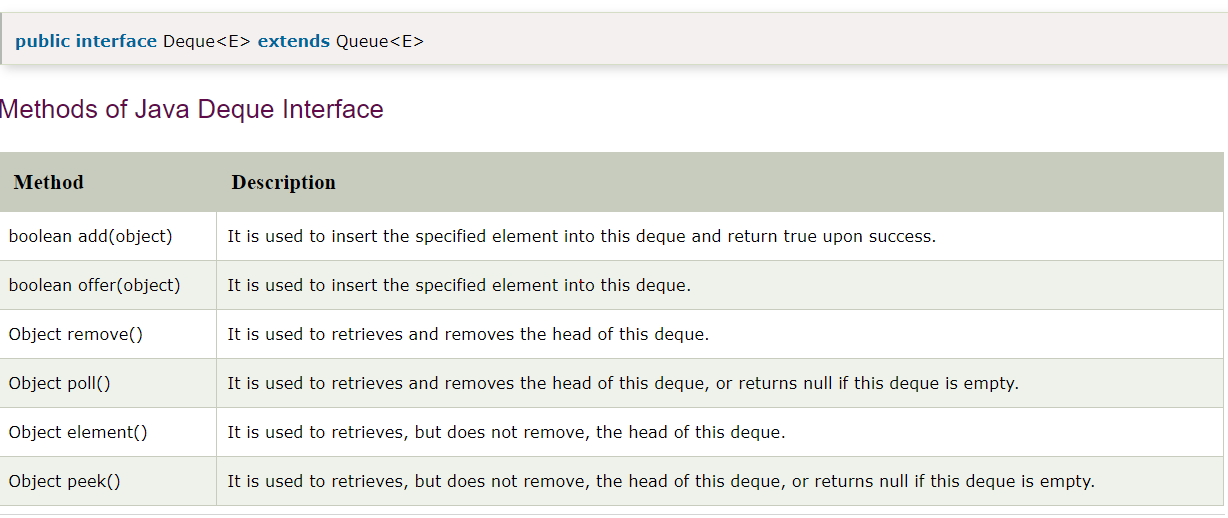


# PriorityQueue in Java

A PriorityQueue is used when the objects are supposed to be processed based on the priority. It is known that a queue follows First-In-First-Out algorithm, but sometimes the elements of the queue are needed to be processed according to the priority, that’s when the PriorityQueue comes into play. The PriorityQueue is based on the priority heap. The elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.

# 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.

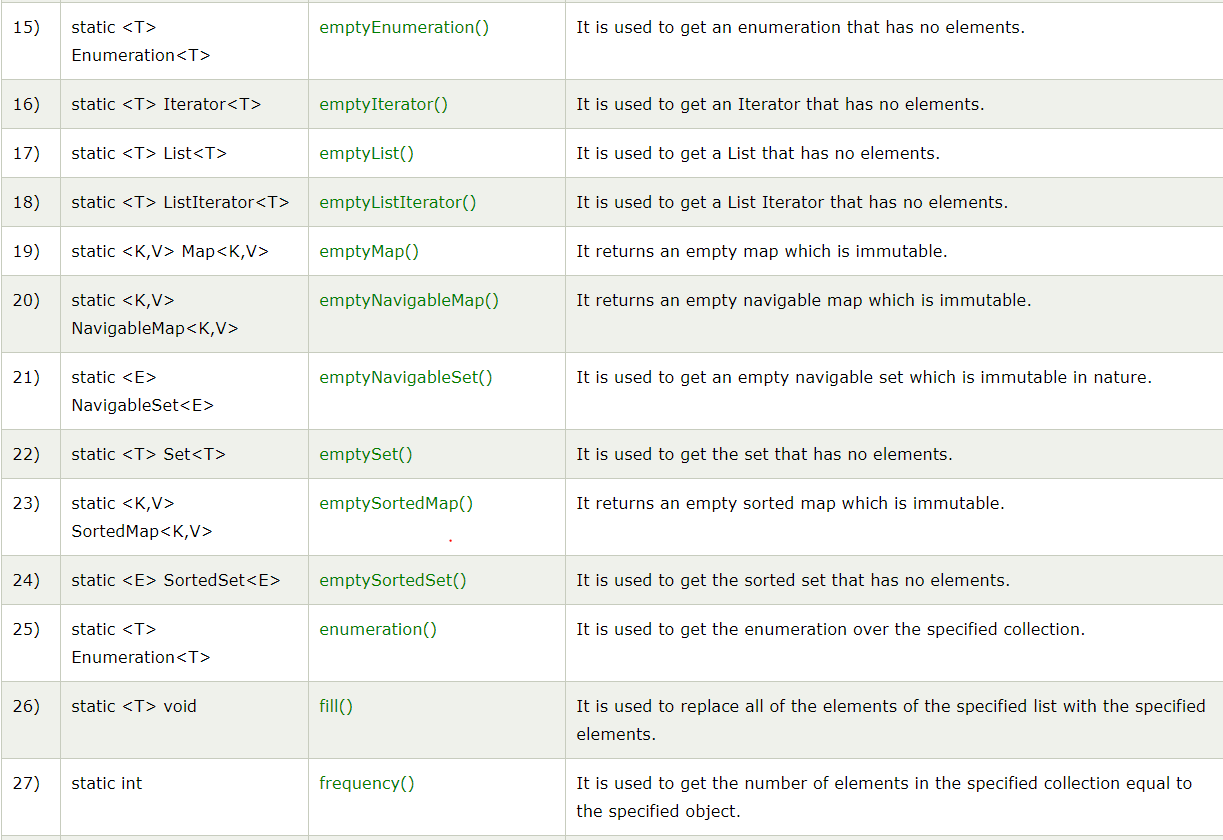
# Java Collections class

Java collection class is used exclusively with static methods that operate on or return collections. It inherits Object class.

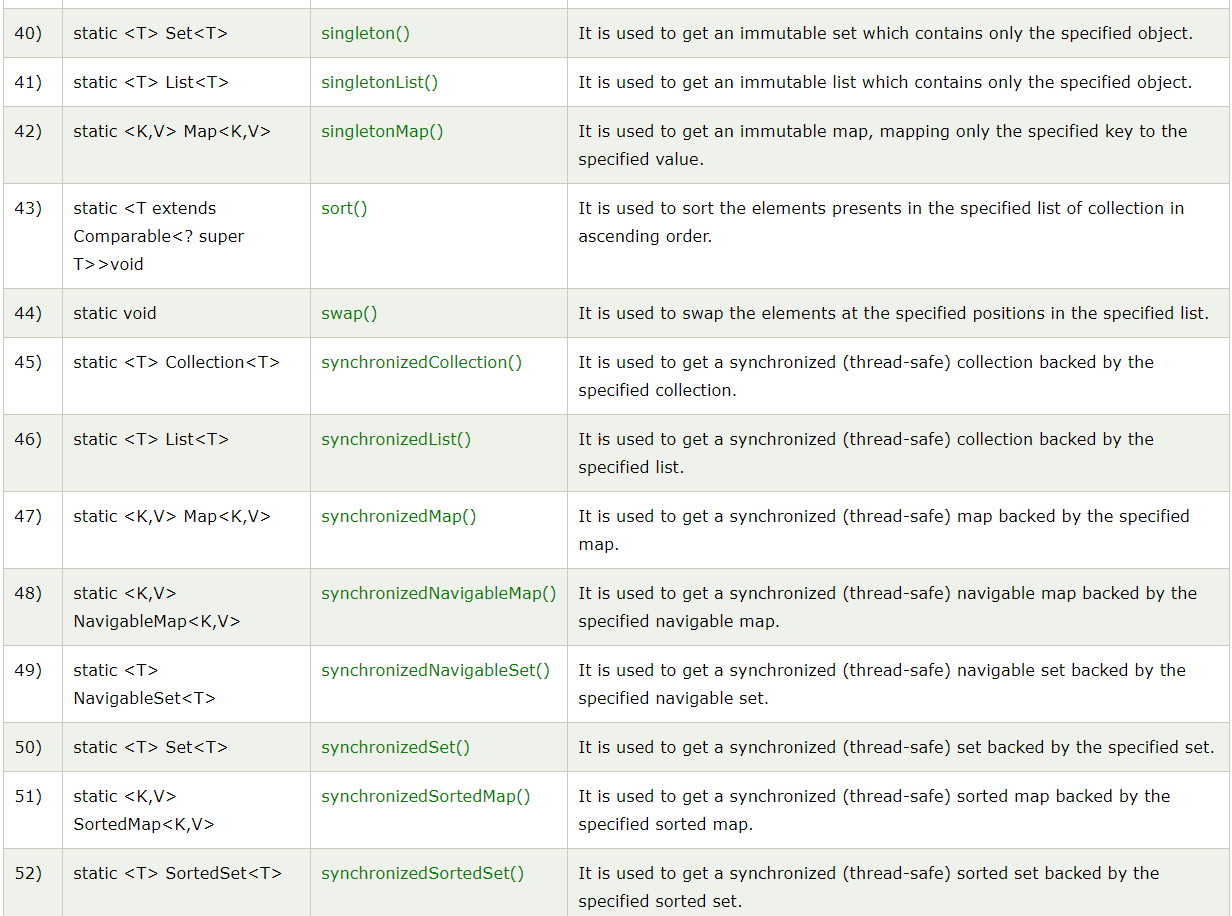
The important points about Java Collections class are:

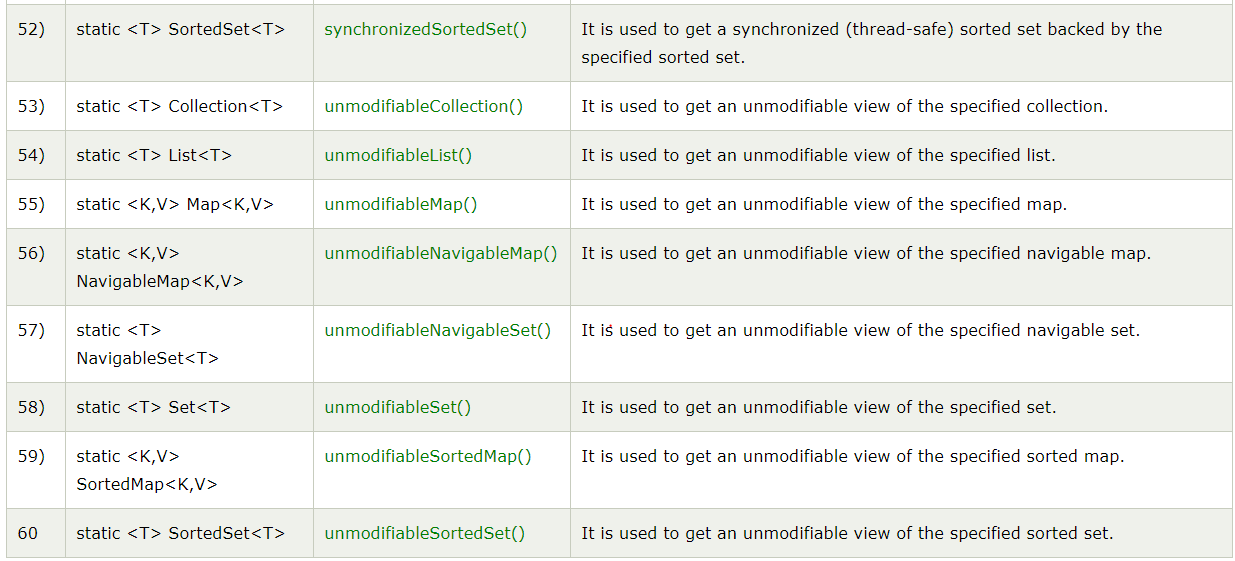
* Java Collection class supports the **polymorphic algorithms** that operate on collections.
* Java Collection class throws a **NullPointerException** if the collections or class objects provided to them are null.

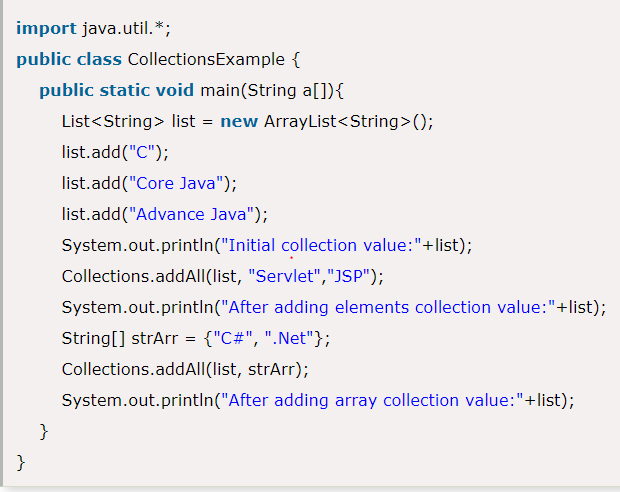












# Java Comparable interface

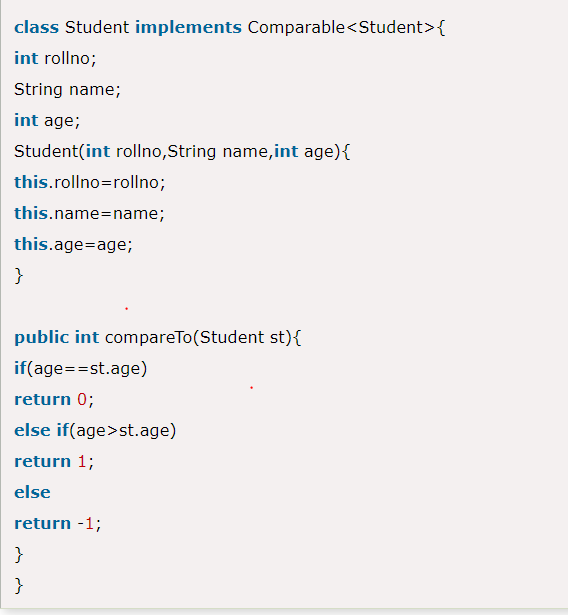
Java Comparable interface is used to order the objects of the user-defined class. This interface is found in java.lang package and contains only one method named compareTo(Object). It provides a single sorting sequence only, i.e., you can sort the elements on the basis of single data member only. For example, it may be rollno, name, age or anything else.

## compareTo(Object obj) method

**public int compareTo(Object obj):** It is used to compare the current object with the specified object. It returns

* positive integer, if the current object is greater than the specified object.
* negative integer, if the current object is less than the specified object.
* zero, if the current object is equal to the specified object.

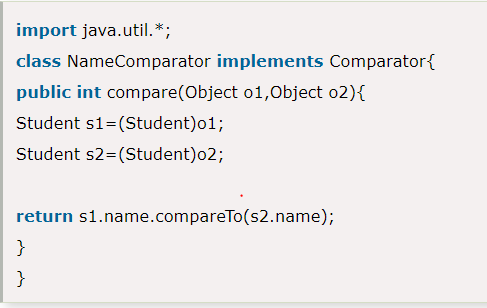
#### Note: String class and Wrapper classes implement the Comparable interface by default. So if you store the objects of string or wrapper classes in a list, set or map, it will be Comparable by default.

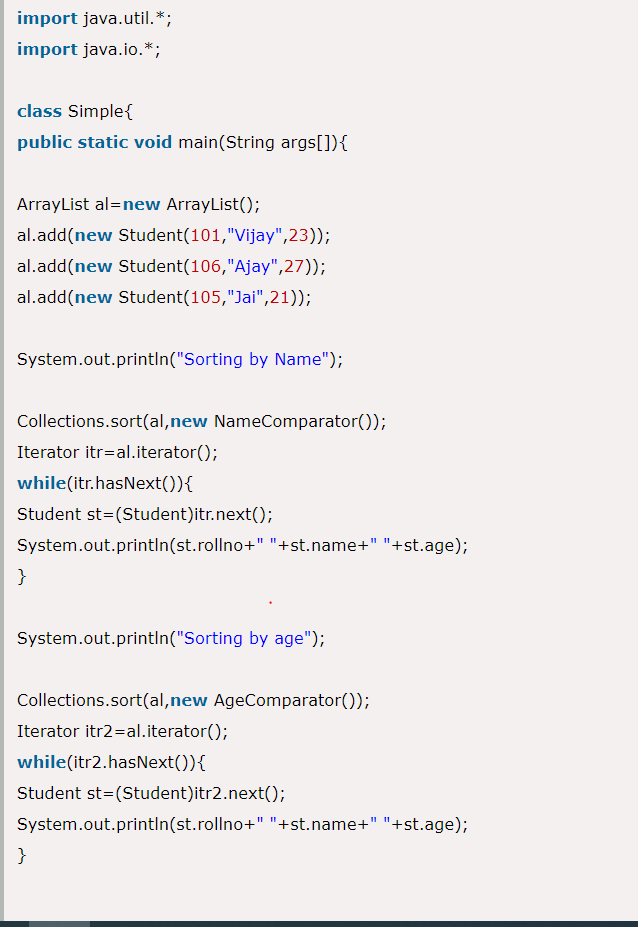


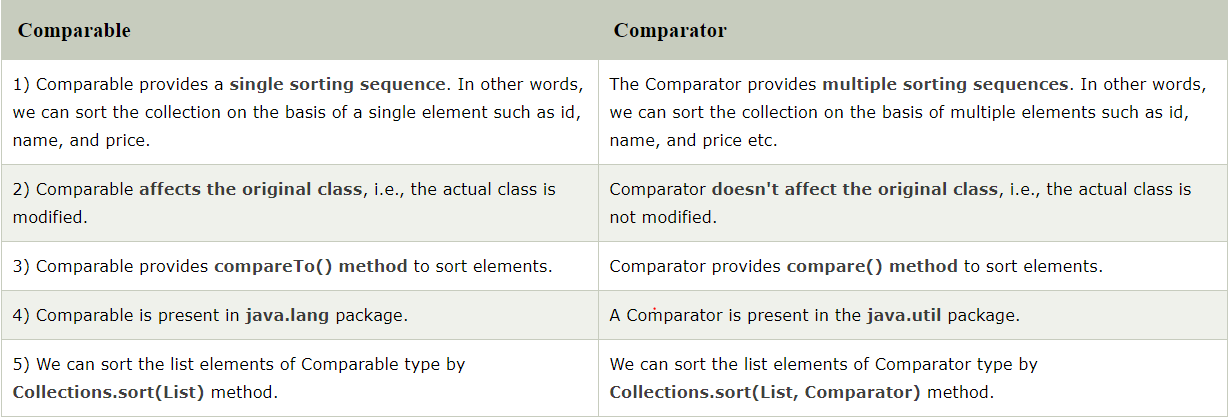
# Java Comparator interface

**Java Comparator interface** is used to order the objects of a user-defined class.

This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).







# Iterators

Iterators are used in [Collection framework](https://www.geeksforgeeks.org/collections-in-java-2/) in Java to retrieve elements one by one. There are three iterators.

[**Enumeration**](https://www.geeksforgeeks.org/iterators-in-java/#Enumeration)**:**

It is a interface used to get elements of legacy collections(Vector, Hashtable). Enumeration is the first iterator present from JDK 1.0, rests are included in JDK 1.2 with more functionality. Enumerations are also used to specify the input streams to a SequenceInputStream. We can create Enumeration object by calling elements() method of vector class on any vector object

// Here "v" is an Vector class object. e is of

// type Enumeration interface and refers to "v"

Enumeration e = v.**elements**();

There are **two** methods in Enumeration interface namely :

// Tests if this enumeration contains more elements

**public boolean hasMoreElements();**

// Returns the next element of this enumeration

// It throws **NoSuchElementException**

// if no more element present

**public Object nextElement();**

**Limitations of Enumeration :**

* Enumeration is for **legacy** classes(Vector, Hashtable) only. Hence it is not a universal iterator.
* Remove operations can’t be performed using Enumeration.
* Only forward direction iterating is possible.

[**Iterator**](https://www.geeksforgeeks.org/iterators-in-java/#Iterator)**:**

It is a **universal** iterator as we can apply it to any Collection object. By using Iterator, we can perform both read and remove operations. It is improved version of Enumeration with additional functionality of remove-ability of a element.

Iterator must be used whenever we want to enumerate elements in all Collection framework implemented interfaces like Set, List, Queue, Deque and also in all implemented classes of Map interface. Iterator is the **only** cursor available for entire collection framework.

Iterator object can be created by calling iterator() method present in Collection interface.

// Here "c" is any Collection object. itr is of

// type Iterator interface and refers to "c"

Iterator itr = c.**iterator**();

Iterator interface defines **three** methods:

// Returns true if the iteration has more elements

**public boolean hasNext();**

// Returns the next element in the iteration

// It throws **NoSuchElementException** if no more

// element present

**public Object next();**

// Remove the next element in the iteration

// This method can be called only once per call

// to next()

**public void remove();**

**Limitations of Iterator :**

* Only forward direction iterating is possible.
* Replacement and addition of new element is not supported by Iterator.

[**ListIterator**](https://www.geeksforgeeks.org/iterators-in-java/#ListIterator)**:**

It is only applicable for List collection implemented classes like arraylist, linkedlist etc. It provides bi-directional iteration.

ListIterator must be used when we want to enumerate elements of List. This cursor has more functionality(methods) than iterator.

ListIterator object can be created by calling listIterator() method present in List interface.

// Here "l" is any List object, ltr is of type

// ListIterator interface and refers to "l"

ListIterator ltr = l.**listIterator**();

ListIterator interface extends Iterator interface. So all three methods of Iterator interface are available for ListIterator. In addition there are **six** more methods.

// Forward direction

// Returns true if the iteration has more elements

**public boolean hasNext();**

// same as next() method of Iterator

**public Object next();**

// Returns the next element index

// or list size if the list iterator

// is at the end of the list

**public int nextIndex();**

// Backward direction

// Returns true if the iteration has more elements

// while traversing backward

**public boolean hasPrevious();**

// Returns the previous element in the iteration

// and can throws **NoSuchElementException**

// if no more element present

**public Object previous();**

// Returns the previous element index

// or -1 if the list iterator is at the

// beginning of the list

**public int previousIndex();**

// Other Methods

// same as remove() method of Iterator

**public void remove();**

// Replaces the last element returned by

// next() or previous() with the specified element

**public void set(Object obj);**

// Inserts the specified element into the list at

// position before the element that would be returned

// by next(),

**public void add(Object obj);**

The hasPrevious() and the previous operations are exact analogues of hasNext() and next() The previous operation moves the cursor backward, whereas next moves it forward

ListIterator has no current element; its cursor position always lies between the element that would be returned by a call to previous() and the element that would be returned by a call to next()

*set()* method can throw four exceptions

* *UnsupportedOperationException* – if the set operation is not supported by this list iterator
* *ClassCastException :*If the class of the specified element prevents it from being added to this list
* *IllegalArgumentException :*If some aspect of the specified element prevents it from being added to this list
* *IllegalStateException :*If neither next nor previous have been called, or remove or add have been called after the last call to next or previous

*add()* method can throw three exceptions

* *UnsupportedOperationException :*If the add method is not supported by this list iterator
* *ClassCastException :*If the class of the specified element prevents it from being added to this list
* *IllegalArgumentException :*If some aspect of this element prevents it from being added to this list

**Important Common Points**

**1 :** Please note that initially any iterator reference will point to the index just before the index of first element in a collection.

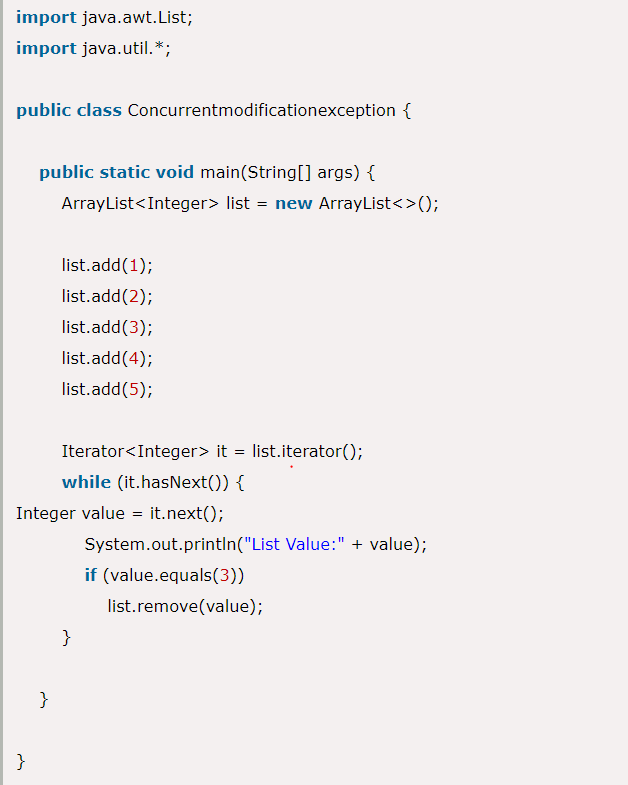
**2 :** We don’t create objects of Enumeration, Iterator, ListIterator because they are interfaces. We use methods like elements(), iterator(), listIterator() to create objects. These methods have anonymous [Inner classes](https://www.geeksforgeeks.org/inner-class-java/) that extends respective interfaces and return this class object

## ConcurrentModificationException

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**.

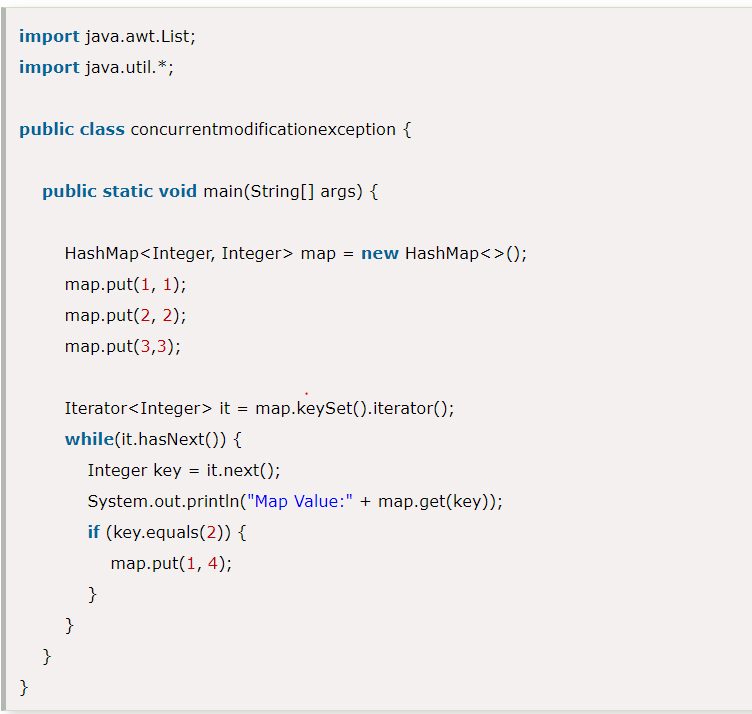
**For Example** - It is not permissible for a thread to modify a Collection when some other thread is iterating over it. This is because the result of the iteration becomes undefined with it. Some implementation of the Iterator class throws this exception, including all those general-purpose implementations of Iterator which are provided by the JRE. Iterators which do this are called **fail-fast** as they throw the exception quickly as soon as they encounter such situation rather than facing undetermined behavior of the collection any time in the future.

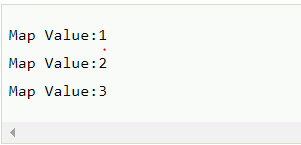
#### Note: It is not mandatory that this exception will be thrown only when some other thread tries to modify a Collection object. It can also happen if a single thread has some methods called which are trying to violate the contract of the object. This may happen when a thread is trying to modify the Collection object while it is being iterated by some fail-fast iterator, the iterator will throw the exception





This message says that the exception is thrown when the next method is called as the iterator is iterating the list and we are making modifications in it simultaneously. But if we make modifications in hashmap like given below, then it will not throw any such exception as the size of the hashmap won't change.





### To Avoid ConcurrentModificationException in multi-threaded environment

1. You can convert the list to an array and then iterate on the array. This approach works well for small or medium size list but if the list is large then it will affect the performance a lot.
2. You can lock the list while iterating by putting it in a synchronized block. This approach is not recommended because it will cease the benefits of multithreading.
3. If you are using JDK1.5 or higher then you can use **ConcurrentHashMap** and **CopyOnWriteArrayList** classes. This is the recommended approach to avoid concurrent modification exception.

convertArray to list

how to make collections thread safe

concurrent modification exception

fail-fast , fail-safe

Arrays