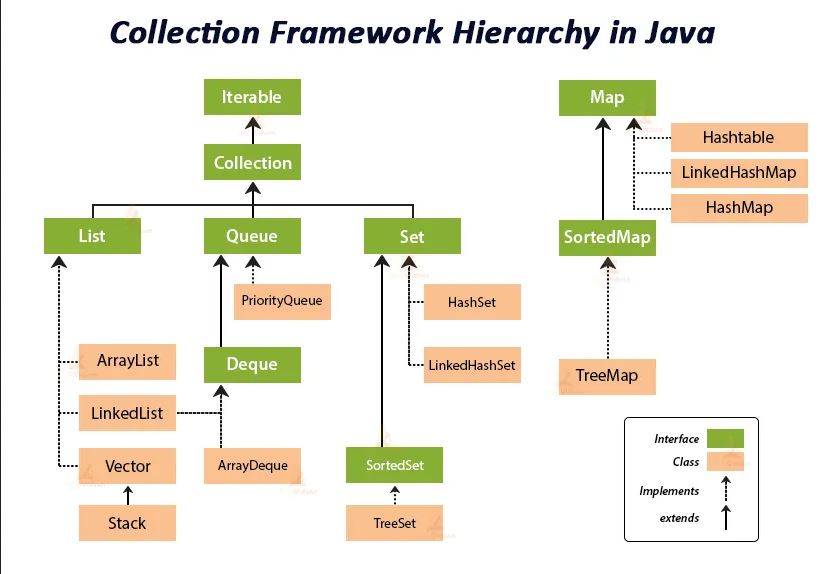
**Collection interface:**

* To represent a **group of element/objects** as a **single unit/entity**, then programmer should consider Collection
* Collection interface is considered as the **root-interface** of collection framework hierarchy
* There is **no concrete class** which implements collection interface
* Collection interface defines various **common methods** for
  1. Adding object to collection
  2. Removing an object from collection
  3. Checking size of collection
  4. Provide Iterator to iterate over collection
  5. Doing bulk operation
* These methods can be used **to operate** on any collection objects
* Present in **java.util** package and extends **java.lang.Iterable** interface



## **Collection interface in java**

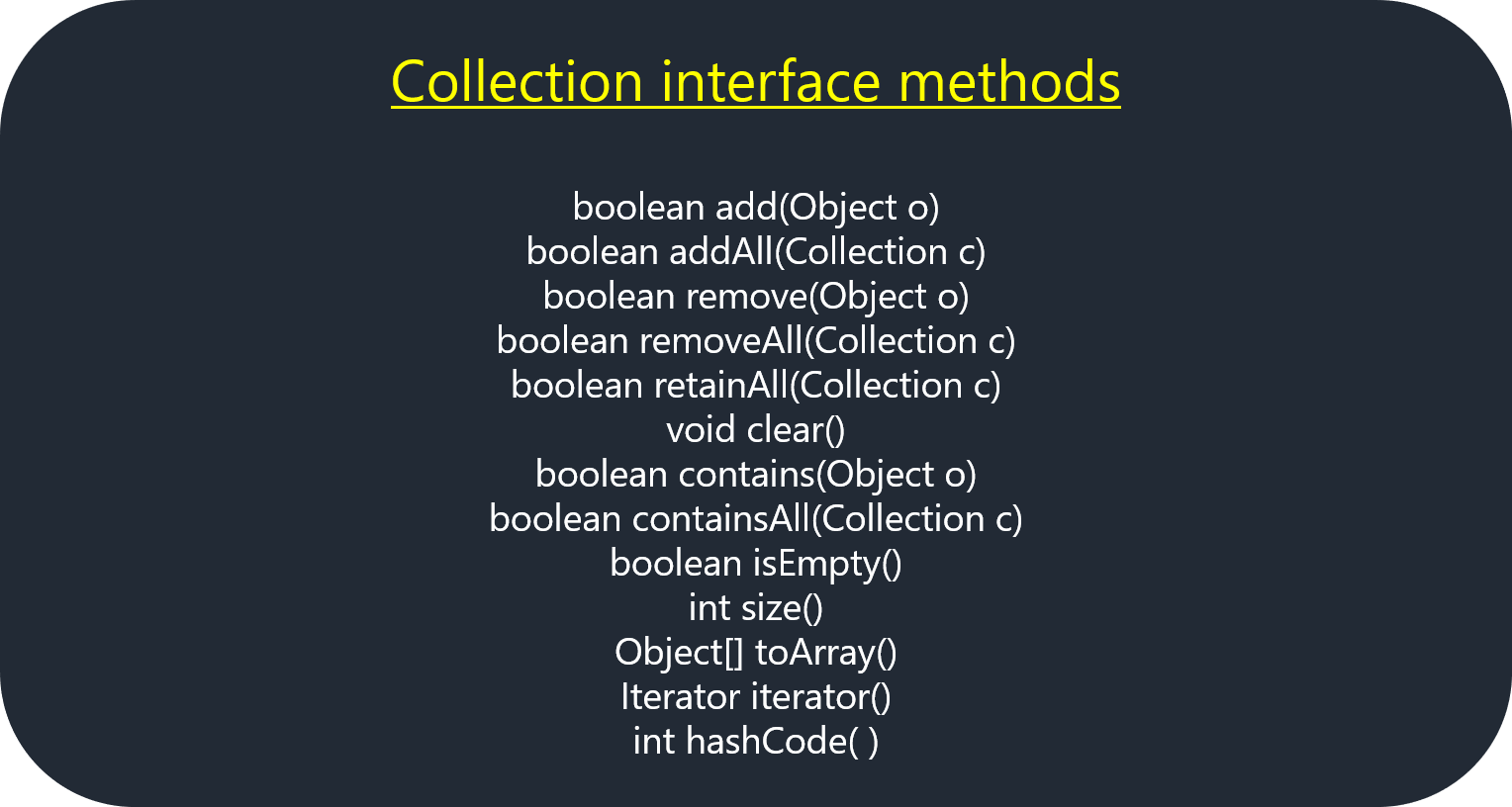
## Collection is the child interface of the Iterable interface. If we want to represent group of individual objects as a single entity then we should go for collection. Collection interface has most methods which are applicable for entire collection object. None of the concreate class implements Collection interface directly.

### Collection interface methods

### **Below listed interfaces extends Collection interface**

* *List*
* *Set*
* *Queue*

|  |  |  |
| --- | --- | --- |
| **java.util.List** | **java.util.Set** | **java.util.Queue** |



**Note :** All above listed **collection methods** will be inherited to subsequent interfaces and classes

|  |  |
| --- | --- |
| **boolean retainAll(Collection c);** | **deletes** all element/objects of invoking collection **except** specified collection |
| **void clear();** | deletes all elements of invoking collection |
| **boolean contains(Object obj);** | used to search specified object from invoking collection returns true, if present; otherwise return false |
| **boolean containsAll(Collection c);** | used to search specified collection from invoking collection returns true, if present; otherwise return false |
| **Object toArray();** | converts invoked collection to array returns object array |
| **Iterator iterator();** | returns **iterator** for the invoking collection; to iterate over the collection elements in **FORWARD** direction only |
| **boolean equals(Object element);** | compares specified object with invoking collection for equality |
| **int hashCode ();** | returns **hash code** for invoking collection |

## **List interface in java**

1. List interface is the child interface of Collection interface
2. If we want to represent group of individual object as single entity where duplicates are allowed and insertion order is preserved then we should go for list
3. insertion order is also preserved by index
4. List is index based
5. we can differentiate duplicates by using index

Implementation classes for List interface are ArrayList, LinkedList, Vector, and Stack.

### List interface methods

## Implementation of List interface in java

Following classes provide the implementation of List interface.

1. ArrayList class
2. LinkedList class
3. Vector class

## **ArrayList Class**

* ArrayList is the implementation of the List interface
* If we want to represent group of individual objects as a single entity where duplicates are allowed and insertion order is preserved then we should go for ArrayList
* Underline data structure is growable or resizable. Internally ArrayList uses Object array
* Since List is index-based insertion order is preserved
* In ArrayList we can store Homogeneous as well as Heterogenous objects.

The **ArrayList** implements the **List** interface. It’s having the following features:

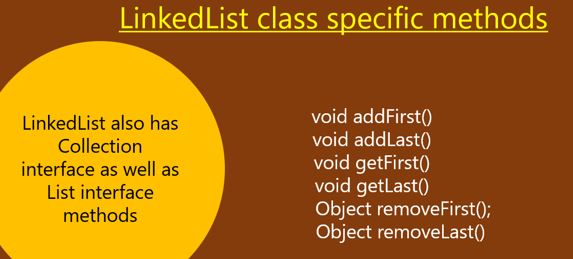
* **ArrayList** uses a dynamic array data structure to store objects and elements.
* **ArrayList** allows duplicate objects and elements.
* **ArrayList** maintains the insertion order.
* **ArrayList** is non-synchronized.
* **ArrayList** elements/objects can be accessed randomly.

**Use ArrayList when there is more retrieving**

## **LinkedList Class**

* LinkedList is the implementation of the List interface
* If we want to represent group of individual objects as a single entity where duplicates are allowed and insertion order is preserved then we should go for LinkedList
* LinkedList follows doubly link list data structure internally so LinkedList is best choice if frequent operation is insertion and deletion

LinkedList implements the List interface. It’s having the following features:

* **LinkedList** uses a **doubly linked list** data structure to store elements.
* **LinkedList** allowed storing the duplicate elements.
* **LinkedList** maintains the insertion order.
* **LinkedList** is not synchronized.
* **LinkedList** manipulation is fast because no shifting is required.
* **Use linked list when there is more delete/insertion manipulation**

## Vector Class

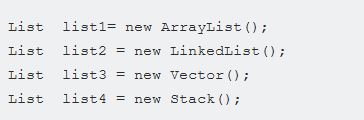
Vector Class implements List interface. It’s having the following features:

* **Vector** is similar to the ArrayList class.
* **Vector** class uses data structure as a dynamic array to store the data elements.
* **Vector** is synchronized.
* **Vector** contains many methods that are not the part of Collection Framework.

## Stack Class

The **Stack** is the subclass of the Vector class. It’s having the following features:

* **Stack** implements the Vector data structure with the (**LIFO**)last-in-first-out.
* **Stack** contains all of the methods of the Vector class.
* **Stack** also provides its methods like **boolean push()**, **boolean peek()**, **boolean push(object o)**, which defines its features.



**Set Interface**

**Set Interfac**e extends **Collection Interface** and present in **java.util** package.

* **Set** doesn’t allow duplicate elements or objects.
* **Set** store elements in an unordered way.
* **Set** allows only one null value.
* **Set** is implemented by **HashSet**, **LinkedHashSet**, and **TreeSet**.

## **HashSet**

**HashSet** class implements **Set Interface**. It’s having the following features:

* **HashSet** internally uses data structure like a **hash table** for storage.
* **HashSet** uses **hashing technique** for storage of the elements.
* **HashSet** always contains unique items.

 Duplicates are not allowed but if we are trying add any duplicate value then we do not get any compile time error instead add() method returns false

 Insertion order is not preserved because all objects are inserted based on Hashcode of Objects

 Heterogeneous objects are allowed

 null insertion is possible

 HashSet is best choice if operation is frequent search operation

## LinkedHashSet

## **LinkedHashSet** class implements Set Interface. It’s having the following features:

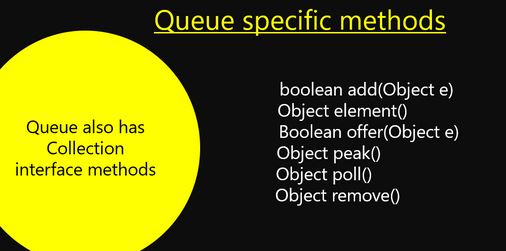
* **LinkedHashSet** store items in LinkedList.
* **LinkedHashSet** store unique elements.
* **LinkedHashSet** maintains the insertion order.
* **LinkedHashSet** allows null elements.

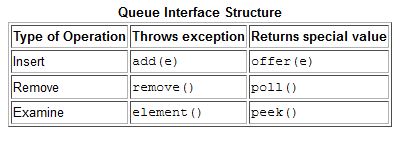
## **Queue interface in java**

## Queue is the child interface of Collection interface.

* In queue elements are stored in first come first out manner.
* If we want to represent group of individual objects as single entity for prior to processing then we should go for Queue

### Queue Specific methods

****

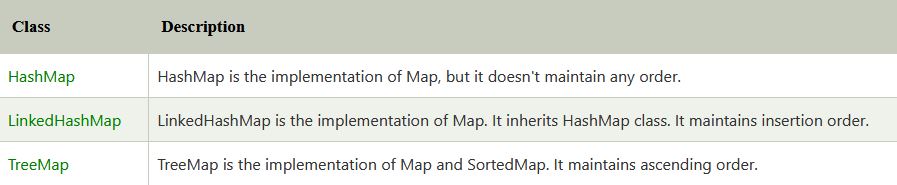


There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap. The hierarchy of Java Map is given below:



A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.

A Map can't be traversed, so you need to convert it into Set using *keySet()* or *entrySet()* method.



**Map Interface**

In the collection framework, a map contains values on the basis of key and value pair. This pair is known as an entry. A map having the following features:

* **Map** contains unique keys.
* **Map** allows duplicate values.
* **Map** is useful to search, update or delete elements on the basis of a key.
* **Map** is the root interface in Map hierarchy for Collection Framework.
* **Map** interface is extended by SortedMap and implemented by HashMap, LinkedHashMap.
* **Map** implementation classes HashMap and LinkedHashMap allow null keys and values but TreeMap doesn’t allow null key and value.
* **Map** can’t be traversed, for traversing needs to convert into the set using method keySet() or entrySet().

**Hashtable** class implements a **Map** interface and extends Dictionary class to store key and values as pairs. It’s having the following features:

* **HashTable** store values as an array of the list where each list is known as a bucket of the node(key and value pair).
* **HashTable** class is in **java.util** package.
* **Hashtable** contains unique elements.
* **Hashtable** doesn’t allow null key or value.
* **Hashtable** is synchronized.
* **Hashtable** initial default **capacity** is 11 whereas the **load factor** is 0.75.