# Collection Framework

## Why Collections Framework?

**int x = 10;**

**int y = 20;**

**int z = 30;**

Representing limited values by limited variables is not a problem.

But if we have a large number of values, say 10,000 values, then declaring 10,000 variables is not a practical approach. It decreases the **accessibility**and **readability** of the program.

### Arrays

To overcome this limitation, **Arrays** are used.

Ex. **Student[] students = new Students[10000];**

We can now access each element in the array i.e. each student by an index number, i.e. **students[0], students[88] etc…**

#### Advantages of Array

We can represent a huge number of values by using a single variable and improve the **accessibility** and **readability** of code.

#### Disadvantages of Array

1. Arrays have a fixed size –To use an array, we must know the size in advance. We can’t increase or decrease the size of array after we have specified its size.
2. Arrays can only store homogeneous data elements.

For example:

Student[] students = new Student[5];

students[0] = new Student(); // Correct

students[1] = new Faculty();// C.T.E.

**Compile Time Error:**

**Found: Faculty.**

**Required: Student**

### Object Array

If we use an Object[] instead of a normal class array, the limitation of the Array to store only homogeneous data is overcome. Consider the example:

Object[] students = new Object[5];

students[0] = new Student();

students[1] = new Faculty();

1. No API support for arrays–Arrays concept is not implemented based on some standard Data Structure, hence **ready-made method support** is not available.For every requirement such as element **insertion in middle, sorting, searching or deletion**we need to write the logic of the code. It increases the complexity and time of development.

## The Solution – Collection Framework

We can solve the above disadvantages of Arrays by using Collection Framework.

### Arrays v/sCollection Framework

|  |  |  |
| --- | --- | --- |
| **S. No** | **Arrays** | **Collection Framework** |
| 1 | Fixed in Size | Growable in size |
| 2 | Not recommended to use with respect to memory optimization, ie. if we allocate a size of 100 and use only 10 of them, the rest 90 locations are wasted. | Recommended to use with respect to memory optimization, i.e. if we allocate a size of 100 and use only 10 of them, the rest 90 locations can be released. |
| 3 | Recommended to use with respect to performance (faster execution) | Not recommended to use with respect to performance (slower execution) |
| 4 | Can store homogeneous data only | It can store both homogeneous and heterogeneous data |
| 5 | An array is not implemented using a standard Data Structure | Every collection class is implemented based on some standard Data Structure |
| 6 | Ready-made method support is not available | Ready-made method support is available |
| 7 | Arrays can be used to hold both Primitive and Object data. For Example:  int[] a = new int[6];  Student[] students = new Students[5]; | Collections can hold only object data and not Primitive Data.  Hence, Primitive data types must be first converted to Wrapper Classesand then stored. |

### Questions:

Question 1:What are the differences between an Array and

1. Collection?
2. ArrayList?
3. LinkedList?
4. Any Collection Class…

Answer:All the above differences can be listed when difference between an Array and Collection is asked as ArrayList, Vector, LinkedList etc are different types of Collections.

## Collection and Collection Framework

Collection and Collection Framework are one of the most early concepts in programming. They are renamed in Java.

* In C++, we had Container and **Standard Template Library (STL).**
* In Java, se haveCollection and Collection Framework.

### What is a Collection?

A Collection is a **group of individual objects**represented as a **single entity**.

### What is Collection Framework?

A Collection Framework defines **several interfaces and classes** that can be used to represent a group of individual objects as a single entity. All the interfaces and classes are present in **java.util package**.

### What is a Framework

A framework includes predefined interfaces, classes and functions that can be used to process input, store and process data,perform network operations, manage hardware devices etc etc etc….

### Key Interfaces in the Collection Framework

|  |  |  |  |
| --- | --- | --- | --- |
| S. No. | Interface Name | S. No. | Inteface Name |
| 1. | **Collection** | 7. | **Map** |
| 2. | **List** | 8. | SortedMap |
| 3. | **Set** | 9. | NavigableMap |
| 4. | SortedSet Queue |  |  |
| 5. | NavigableSet |  |  |
| 6. | **Queue** |  |  |

### java-collection-hierarchy.png

## Collection (I)

* Collection interface is **considered** as the **root interface** of collection framework.
* It represents a group of objects as a single entity.
* Collection interface declares the **most common methods** that are applicable toall Collection Objects.
* All collection classes implementadd(E element), addAll(), clear(), contains(), containsAll(), isEmpty(), remove(), removeAll(),retainAll(), size(), iterator() methods

## List (I)

* List is the child interface of Collection (I), i.e. List extends Collection.
* List Interface is implemented by the classes of **ArrayList, LinkedList, Vector and Stack.**
* Collection, List, ArrayList, LinkedList were included in **JDK version 1.2**
* Vector and Stack were present in **JDK version 1.0**. They were **re-engineered** and included in Collection Framework. Hence they are known as **Legacy Classes (Old Generation).**

#### Features of all Lists

* In a List, the data is stored in a **sequence**
* List indexes **start from 0**, just like arrays.
* List preserves the **insertion order**. Hence, it allows **positional access** and insertion of elements.
* In a List:
  + Duplicate values can be stored
  + null values can be stored

### *Operations on a List*

* List Interface extends Collection, hence it supports **all the operations of Collection Interface.**
* Since List preserves the **insertion order**, it allows **positional access** and insertion of elements.
* add(int **index**, E element), get(int **index**), remove(int **index**), set(int **index**, E element), last**Index**Of(E element), listIterator(int **index**)

### ArrayList (C)

An ArrayList is a class that implements List interface.

### Key Points

* An ArrayList is a **re-sizable array**, also called a **dynamic array**.
* Internally an ArrayList uses an **Object array** to store the elements.
* Java ArrayList allows **random access** because array works at the index basis.
* We**cannot** create an ArrayList of primitive types like int, char etc.
* We need to use boxed types like Integer, Character, Boolean etc
* ArrayList is **not synchronized**. To learn more, see [here](https://www.geeksforgeeks.org/synchronization-arraylist-java/)

**Memory Mapping**

|  |  |
| --- | --- |
| ArrayList_Dianamic-1.jpg | ArrayList_Dianamic-1.jpg |

**Working**

ArrayList has got 3 constructors:

* ArrayList(): Creates an ArrayList with 0 size
* ArrayList(int size): Creates an ArrayList with specified size
* ArrayList(Collection c): Creates an ArrayList with an existing Collection with its size

## How Memory Works

1. ArrayList aL = new ArrayList();

When the default constructor is called, an ArrayList with capacity 10 is created.

1. al.add(1)

As we add an element, an ArrayList with size 10 is created and the element is stored at index 0

1. As 10 elements are added, the size still remains 10.
2. As the 11th element is added,
   1. A new ArrayList with new capacity (Old Size + Old Size/2) is created, 15 in this case.
   2. All the elements are copied from the old ArrayList to the newly created ArrayList.
   3. The new element is added at position 11, index 10.
3. And the process continuous.

**Operation Efficiency**

In an ArrayList,

* Insertion/Delete operations are slow if addition/deletion occurs in the middle
* Insertion/Delete operations are fast if addition/deletion occurs in the last
* Retrieval is fast

### LinkedList (C)

An LinkedList is a class that implements List interface and Deque interface.

**Note: Deque Interface**: A linear collection that supports **element insertion and removal at both ends.** The name *deque* is short for "**double ended queue**"

### Key Points

* Linked List is a linear/sequential data structure.
* The elements in a LinkedList are not stored in contiguous locations.

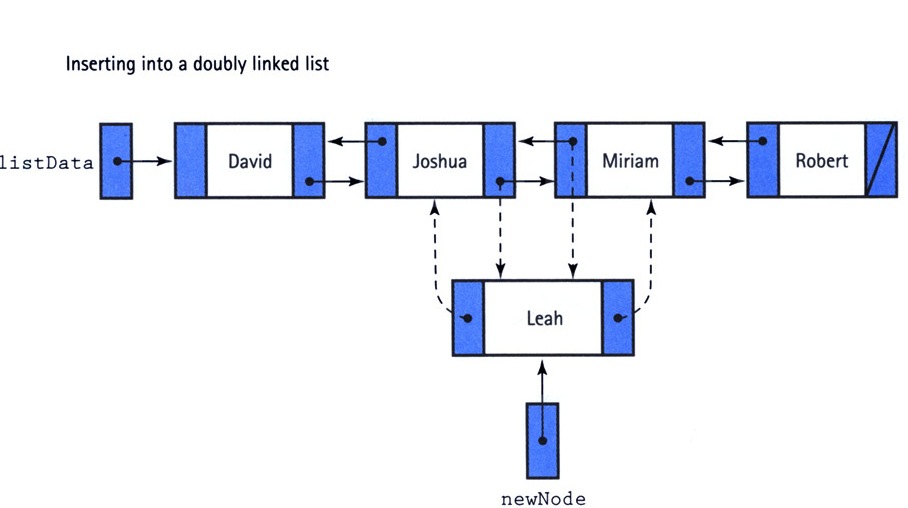
**Memory Mapping**

* A LinkedList is implemented in Java using **doubly linked list** data structure.
* A **doubly**-**linked list** is a **linked** data structure that consists of a set of sequentially **linked** records called elements or in other word **nodes**.
* The elements are linked using **pointers** and **addresses.**
* Every nodecontains three fields, **1 data field** and **2 link fields**, that are references to the previous and next node in the sequence of nodes
* The two node links allow **traversal** of the list in **either direction**.

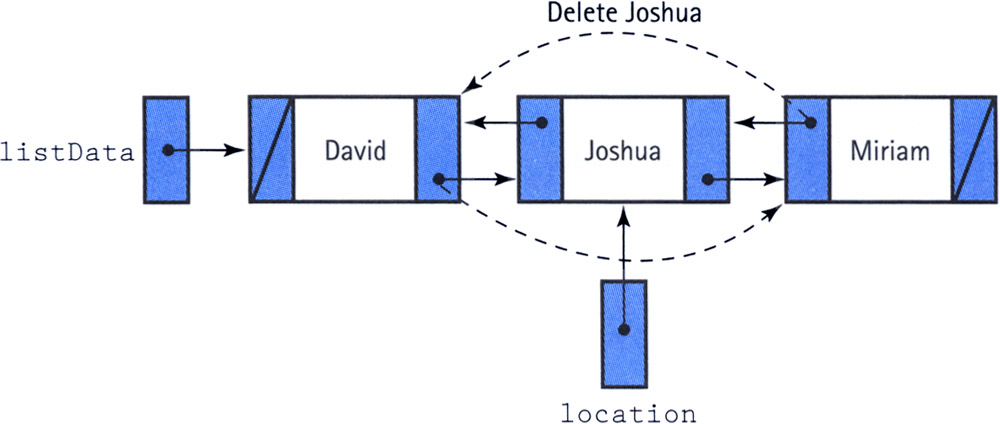
doublell.jpg

### Working

**Inserting an element**

****

**Deleting an Element**

****

**Operation Efficiency**

In a LinkedList,

* In Java LinkedList class, manipulation is fast because no shifting needs to occur.
* Due to the ease of insertions and deletions, they are preferred over ArrayList and Arrays
* Use linked list, when we require frequent adding or removing operationsin a collection.

**ArrayList v/s LinkedList**

|  |  |
| --- | --- |
| **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**. | Manipulation with LinkedList is **faster** than ArrayList |
| 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.** |

### Vector (C)

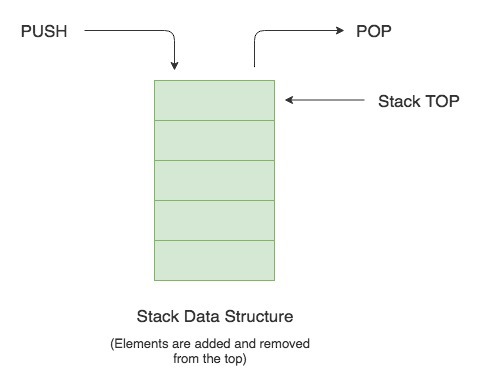
Vector is same as ArrayList except that all the Vector class methods are synchronized. Hence vector is thread-safe.

### ArrayList v/s Vector

|  |  |  |
| --- | --- | --- |
| **S. No.** | **ArrayList** | **Vector** |
| 1 | There are no synchronized methods. | All methods are synchronized. |
| 2 | No Thread safe: Multiple threads can access the array list at the same time. | Thread safe: Only one thread is allowed to operate on vector object at a time. |
| 3 | Threads are not required to wait and hence performance is high | It increases the waiting time of threads (since all the methods are synchronized) and hence performance is low |
| 4 | There are no synchronized methods. | All methods are synchronized. |
| 5 | Recommended to use in performance point of view. | Not recommended to use in performance point of view. |
| 6 | ArrayList is introuducd in Java 1.2 | Vector is introduced in Java 1.0. Hence Vector class is legacy. |

**Stack (C)**

* The Stack class represents a last-in-first-out (LIFO) stack of objects.
* It extends class Vector with five operations that allow a vector to be treated as a stack.
* It contains 5 methods: push(), pop(), peek(), isEmpty(), search()
* Learn about Stack implementation in Java [here](https://www.tutorialspoint.com/javaexamples/data_stack.htm)



**Note:** Vector and Stack are legacy classes and now they are not much used.

## Set (I)

* Set is the child interface of Collection (I)
* Set is an ordered collection of objects where
  + Duplicates are not allowed
  + Insertion order is not preserved
  + null is not allowed

Set (I)

LinkedHashSet (C)

HashSet (C)

SortedSet (I)

NavigableSet (I)

TreeSet (C)

### SortedSet(I)

* It is the child interface of Set (I)
* We use SortedSet when we want all objects to be inserted according to some sorting order
* Itrepresents a set that is sorted according to the **natural order** (i.e. in ascending order for numerals and alphabetically for strings)
* Sorting invloves comparison between collection elements.
* Hence, sortingcan take place only between objects of same type.
* null objects can’t be sorted.

### NavigableSet (I)

* It is the child interface of SortedSet (I)
* It inherits the methodsof a SortedSet, i.e. the elements arrange according to their natural order.
* It declares several methods for navigation purposes.

### TreeSet (C)

* TreeSet implements the **NavigableSet**interface
* Objects in a TreeSet are stored in a **sorted and ascending order**.
* TreeSet does not preserve the insertion order of elements but elements are **sorted by keys**.
* TreeSet **does not** allow inserting**Heterogeneous objects** as it implements SortedSet (I).
* It will throw **ClassCastException** at Runtime if trying to add hetrogeneous objects.
* TreeSet **cannot** contain null values as it implements SortedSet(I).
* It has faster access and retrieval time
* TreeSet internally uses a **TreeMap** to store elements.
* TreeSet class is **not thread-safe**(not synchronized)

### Important

Two things must be kept in mind while creating and adding elements into a TreeSet:

1. Firstly, **insertion of null** into a TreeSet throws **NullPointerException** because while insertion of null, it gets compared to the existing elements and null cannot be compared to any value.
2. Secondly, if we are depending on default natural sorting order, compulsory the object should be **homogeneous** and **comparable** otherwise we will get RuntimeException:***ClassCastException***

### HashSet(C)

* Java HashSet class **implements Set interface**
* It **extends** the **AbstractSet class**
* Underlying data structure for HashSet is **hash table**.
* HashSet **does not allow duplicates**
* HashSet **allows null value.**
* Hashset **allows heterogeneous objects**
* HashSet class is **non synchronized.**
* HashSet is the best approach for search operations.
* We **can’t sort** HashSet elements
* We can however sort it by converting into to anyList type or TreeSet

### List v/s Set

|  |  |  |
| --- | --- | --- |
| S. No. | List | Set |
| 1. | Duplicates are allowed | Duplicates are not allowed |
| 2. | Insertion order is preserved | Insertion order is not preserved |

### Queue (I)

* It is the child interface of Collection (I)
* It follows the FIFO or the First-In-First-Out principle.
* It is an ordered list of objects
* It is used to insert elements at the end of the list and delete elements from the start of list
* The child interface of Queue is DeQueue
* PriorityQueue is the direct implemenation of Queue interface

**DeQueue(I)**

* It is the child interface of Queue.
* It is a linear collection that supports **element insertion and removal at both ends.**
* The name *deque* is short for "**double ended queue**"
* LinkedList implements DeQueue

Collection (I)

Queue (I)

PriorityQueue (C)

PriorityBlockingQueue (C)

BlockingQueue (I)

LinkedBlockingQue (C) (C)

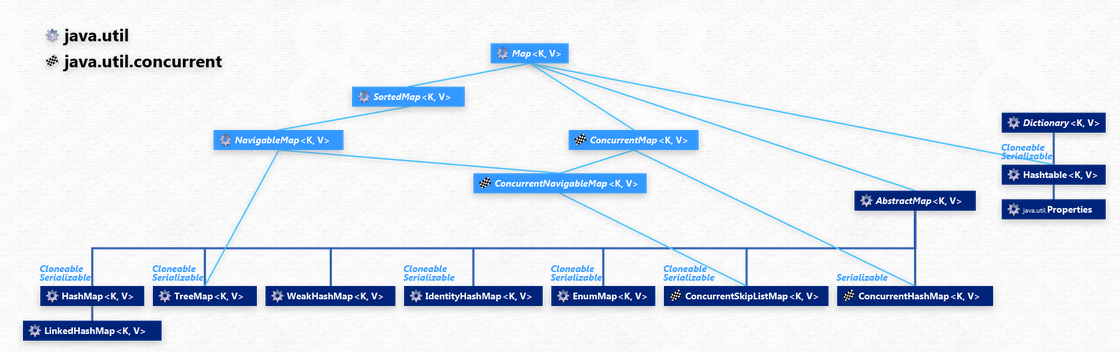
**PriorityQueue (C)**

We know that Queue follows **F**irst-**I**n-**F**irst-**O**ut model but sometimes we need to process the objects in the queue based on the priority. That is when Java PriorityQueue is used.

* A PriorityQueue is used when the objects are supposed to be processed based on the priority
* PriorityQueue doesn’t permit null.
* We can’t create PriorityQueue of Objects that are non-comparable

**Map (I)**

* A Map is an object that **maps keys to values**, or is a collection of **attribute-value pairs**.
* A Map is not considered to be a true collection, as the Map (I)does not extend Collection (I)
* Instead, it is an independent branch in the Java Collections Framework, as shown below:



Find the original diagram [here](http://www.falkhausen.de/Java-8/java.util/Map-Hierarchy.html)

### Key Points

* A Map is useful if you have to search, update or delete elements on the basis of a **key**.
* A Map doesn't allow **duplicate keys.**
* Each key can map to at most one value
* But a map can have duplicate values.
* This interface takes the place of the **Dictionary class**, which was a totally **abstract class** rather than an interface.
* A Map can't be traversed, so we need to convert it into Set using ***keySet()*** or ***entrySet()*** methods.

### HashMap(C)

* HashMap provides the **basic implementation of the Map interface** of Java
* It is present in **java.util** package
* It stores the data in **(Key, Value)** pairs.
* To access a value one must know **its key**.
* HashMap is known as HashMap because it uses a technique **called Hashing**.

Hashing

* Hashing is a technique of converting a **large String to small String** that represents the same String.
* A shorter value helps in indexing and faster searches.
* HashSet also uses HashMap internally.

Key points -

* A HashMap **cannot** contain **duplicate keys**.
* A HashMap **can** contain **duplicate values**.
* HashMap allows **null key also but only once**
* HashMap allows**multiple null values.**
* HashMap is an **unordered collection**. It does not guarantee any specific order of the elements.
* Java HashMap is not thread-safe.

**SortedMap (I)**

* SortedMap extends the Map interface.
* SortedMap orders the keys by their natural ordering, or by a specified **comparator.**
* **Since sorting has come into picture,** null key are not permitted.

**NavigableMap(I)**

* NavigableMap extends the SortedMap interface.
* It provides convenient navigation methods to the implementing map

**TreeMap(C)**

* TreeMap implements SortedMap and NavigableMap interfaces
* TreeMap is sorted according to the natural ordering of its keys.
* We can navigate through the TreeMap using methods of NavigableMap

**Dictionary (Abstract Class)**

* **Dictionary** is an abstract class
* **It came in JDK 1.0, hence it is a legacy class**
* **It**represents a **key-value**relation and works similiar to a map.
* Given a key you can store values and when needed can retrieve the value back using its key.
* Thus, it is a list of key-value pair.
* **This class is obsolete.**
* **New implementations should implement the Map interface, rather than extending this class.**

**Hashtable(C)**

* Hashtable class extends Dictionary class
* It implements Map interface
* Any non-null object can be used as a key or as a value.
* Hashtable is similar to HashMap except it is synchronized.

**Enumeration (I)**

* Enumeration is aninterface
* It is present in **java.util** pacakage
* 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.
* Methods are provided to enumerate through the elements of a vector, the keys of a hashtable, and the values in a hashtable
* The functionality of this interface is duplicated by the **Iterator** interface.
* In addition, **Iterator** adds an optional remove operation, and has shorter method names.
* New implementations should consider using **Iterator** in preference to **Enumeration**

**Hashmap v/s Hashtable**