Collections Framework in Java

Abhinav Dogra
Department of Computer science

RNS Institute of Technology
Benagluru, India
Email-: abhinav26dogra@gmail.com

Abstract— This paper aims at providing introduction to Java Collections Framework (JCF). A lot of classes and methods have been developed in framework to enhance the speed of programs in terms of computational complexity. There are several methods that must be taken in consideration; time complexity, code reusability, interoperability.

Keywords— Interface, API, Instantiation.

INTRODUCTION

The Java Collections Framework standardizes the way in which groups of objects are handled by your programs. Collections were not part of the original Java release, but were added by J2SE 1.2. Prior to the Collections Framework, Java provided ad hoc classes such as Dictionary, Vector, Stack, and Properties to store and manipulate groups of objects. Although these classes were quite useful, they lacked a central, unifying theme. The way that you used Vector was different from the way that you used Properties, for example. Also, this early, ad hoc approach was not designed to be easily extended or adapted. Collections are an answer to these problems. The Collections Framework was designed to meet several goals. First, the framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) are highly efficient. Second, the framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability. Third, extending and/or adapting a collection had to be easy. Toward this end, the entire Collections Framework is built upon a set of standard interfaces. Several standard implementations (such as LinkedList, HashSet, and TreeSet) of these interfaces are provided that you may use as-is. You may also implement your own collection, if you choose.[4]

EASE OF USE

REDUCES PROGRAMMING EFFORT

By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work.

INCREASES PROGRAM SPEED AND QUALITY

This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementation.

ALLOWS INTEROPERABILITY AMONG UNRELATED APIS

The collection interfaces are the vernacular by which APIs pass collections back and forth. If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings, our APIs will interoperate seamlessly, even though they were written independently.

FOSTERS SOFTWARE REUSE

New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

Abbreviations and Acronyms

J2SE- Java platform, Standard Edition

GUI- Graphical user interface

LIFO- Last-in-first-out

FIFO- First-in -first-out

Hierarchy of the Collections Framework

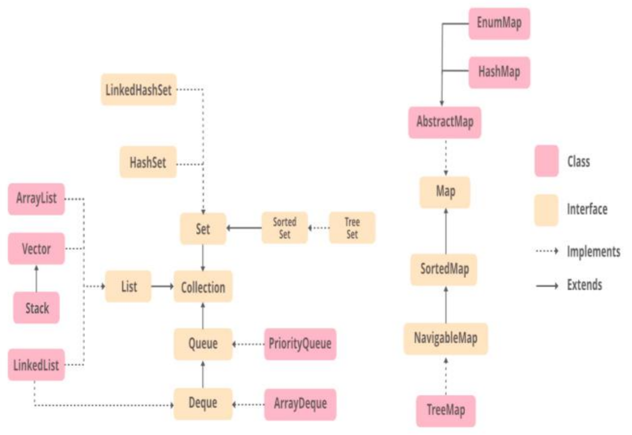


Fig1: Hierarchy of the collections framework[3]

- Class: A class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type.

- Interface: An interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body). Interfaces specify what a class must do and not how. It is the blueprint of the class.

Interfaces which extend the Collections Interface

- Iterable Interface : This is the root interface for the entire collection framework. The collection interface extends the iterable interface. Therefore, inherently, all the interfaces and classes implement this interface. The main functionality of this interface is to provide an iterator for the collections. Therefore, this interface contains only one abstract method which is the iterator.[2]
- Collection Interface : This interface extends the iterable interface and is implemented by all the classes in the collection framework. This interface contains all the basic methods which every collection has like adding the data into the collection, removing the data, clearing the data, etc. All these methods are implemented in this interface because these methods are implemented by all the classes irrespective of their style of implementation.
- List Interface : This is a child interface of the collection interface. This interface is dedicated to the data of the list type in which we can store all the ordered collection of the objects. This also allows duplicate data to be present in it. This list interface is implemented by various classes like ArrayList, Vector, Stack etc.

```
List <T> al = new ArrayList<> ();
List <T> ll = new LinkedList<> ();
List <T> v = new Vector<> ();
Where T is the type of the object
```

Instantiation a list object with different classes.[2]

THE CLASSES WHICH IMPLEMENT THE LIST INTERFACE:-

- ArrayList : ArrayList provides us with dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.[1] The size of an ArrayList is increased automatically if the collection grows or shrinks if the objects are removed from the collection. Java ArrayList allows us to randomly access the list. ArrayList can not be used for primitive types, like int, char, etc. We will need a wrapper class for such cases.

- Vector : A vector provides us with dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed. This is identical to ArrayList in terms of implementation. However, the primary difference between a vector and an ArrayList is that a Vector is synchronized and an ArrayList is non-synchronized.

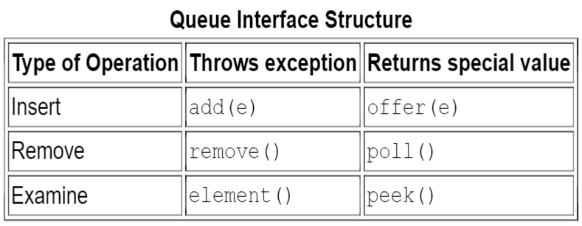
- LinkedList: LinkedList class is an implementation of the LinkedList data structure which is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node.

- Stack : Stack class models and implements the Stack data structure. The class is based on the basic principle of *last-in-first-out*. In addition to the basic push and pop operations, the class provides three more functions of empty, search and peek. The class can also be referred to as the subclass of Vector.

- **Queue Interface :** A queue interface maintains the FIFO(First In First Out) order similar to a real-world queue line. This interface is dedicated to storing all the elements where the order of the elements matter. For example, whenever we try to book a ticket, the tickets are sold at the first come first serve basis. Therefore, the person whose request arrives first into the queue gets the ticket. There are various classes like PriorityQueue, Deque, ArrayDeque, etc.

```
Queue <T> pq = new PriorityQueue<> ();
Queue <T> ad = new ArrayDeque<> ();
Where T is the type of the object.
```

Instantiation a queue object with different classes.[2]



Queue Interface Structure

Type of Operation	Throws exception	Returns special value
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

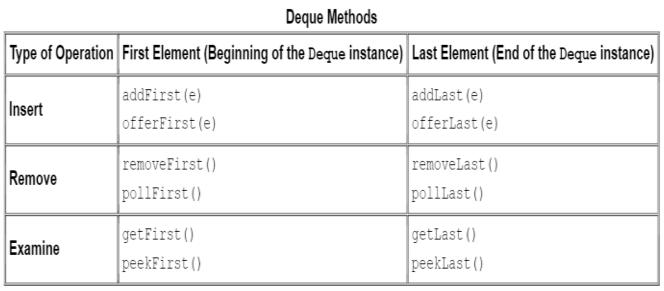
Fig2: Queue Interface

- Priority Queue : A PriorityQueue is used when the objects are supposed to be processed based on the priority. It is known that a queue follows the First-In-First-Out algorithm, but sometimes the elements of the queue are needed to be processed according to the priority and this class is used in these cases. 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.[4]

- **Deque Interface** : Usually pronounced as deck, a deque is a double-ended-queue. A double-ended-queue is a linear collection of elements that supports the insertion and removal of elements at both end points. The Deque interface is a richer abstract data type than both Stack and Queue because it implements both stacks and queues at the same time. The Deque interface, defines methods to access the elements at both ends of the Deque instance. Methods are provided to insert, remove, and examine the elements. Predefined classes like ArrayDeque and LinkedList implement the Deque interface.

```
Deque<T> ad = new
ArrayDeque<> ();
Where T is the type of the
object.
```

Instantiation a deque object with ArrayDeque class.[2]



Deque Methods

Type of Operation	First Element (Beginning of the Deque instance)	Last Element (End of the Deque instance)
Insert	`addFirst(e)` `offerFirst(e)`	`addLast(e)` `offerLast(e)`
Remove	`removeFirst()` `pollFirst()`	`removeLast()` `pollLast()`
Examine	`getFirst()` `peekFirst()`	`getLast()` `peekLast()`

Fig3: Deque Methods

- ArrayDeque : ArrayDeque class which is implemented in the collection framework provides us with a way to apply resizable-array. This is a special kind of array that grows and allows users to add or remove an element from both sides of the queue. Array deques have no capacity restrictions and they grow as necessary to support usage.[2] Null elements are prohibited in the array deques. They are faster than Stack and LinkedList. They are not thread-safe; in the absence of external synchronization.[4]

- **Set Interface :** A set is an unordered collection of objects in which duplicate values cannot be stored. This collection is used when we wish to avoid the duplication of the objects and wish to store only the unique objects.[1] This set interface is implemented by various classes like HashSet, TreeSet, LinkedHashSet, etc. Since all the subclasses implement the set, we can instantiate a set object with any of these classes.[4]

```
Set<T> hs = new HashSet<> ();
Set<T> lhs = new LinkedHashSet<> ();
Set<T> ts = new TreeSet<> ();
Where T is the type of the object.
```

Instantiation of set object with different classes[2]

- HashSet : The HashSet class is an inherent implementation of the hash table data structure. The objects that we insert into the HashSet do not guarantee to be inserted in the same order. The objects are inserted based on their hashcode. This class also allows the insertion of NULL elements. HashSet was introduced in Java 2.

- LinkedHashSet : A LinkedHashSet is very similar to a HashSet. The difference is that this uses a doubly linked list to store the data and retains the ordering of the elements. HashSet is an unordered & unsorted collection of the data set, whereas the LinkedHashSet is an ordered and sorted collection of HashSet. LinkedHashSet was introduced in Java 4.[3]

- **Sorted Set Interface:** This interface is very similar to the set interface. The only difference is that this interface has extra methods that maintain the ordering of the elements. The sorted set interface extends the set interface and is used to handle the data which needs to be sorted. The class which implements this interface is TreeSet. Since this class implements the SortedSet, we can instantiate a SortedSet object with this class.[2]

```
SortedSet<T> ts = new TreeSet<>
();
Where T is the type of the object.
```

- TreeSet : The TreeSet class uses a Tree for storage. The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided. This must be consistent with equals if it is to correctly implement the Set interface. It can also be ordered by a Comparator provided at set creation time, depending on which constructor is used.[3]

- **Map Interface :** A map is a data structure which supports the key-value pair mapping for the data. This interface doesn't support duplicate keys because the same key cannot have multiple mappings. A map is useful if there is a data and we wish to perform operations on the basis of the key. This map interface is implemented by various classes like HashMap, TreeMap etc. Since all the subclasses implement the map, we can instantiate a map object with any of these classes.[2]

```
Map<T> hm = new HashMap<> ();
Map<T> tm = new TreeMap<> ();
Where T is the type of the object.
```

Instantiation of set object with different classes.[2]

- HashMap : HashMap provides the basic implementation of the Map interface of Java. It stores the data in (Key, Value) pairs. To access a value in a HashMap, we must know its key. HashMap uses a technique called Hashing. Hashing is a technique of converting a large String to small String that represents the same String so that the indexing and search operations are faster. HashSet also uses HashMap internally.

Summary of Interfaces

The Java Collections Framework hierarchy consists of two distinct interface trees :

- The first tree starts with the Collection interface, which provides for the basic functionality used by all collections, such as add and remove methods. Its subinterfaces — Set, List, and Queue — provide for more specialized collections.
- The Set interface does not allow duplicate elements. This can be useful for storing collections such as a deck of cards or student records. The Set interface has a subinterface, SortedSet, that provides for ordering of elements in the set.
- The List interface provides for an ordered collection, for situations in which you need precise control over where each element is inserted. You can retrieve elements from a List by their exact position.
- The Queue interface enables additional insertion, extraction, and inspection operations. Elements in a Queue are typically ordered in on a FIFO basis.
- The Deque interface enables insertion, deletion, and inspection operations at both the ends.
- Elements in a Deque can be used in both LIFO and FIFO.
- The second tree starts with the Map interface, which maps keys and values similar to a Hashtable.
- Map's subinterface, SortedMap, maintains its key-value pairs in ascending order or in an order specified by a Comparator.[4]

ACKNOWLEDGMENT

I would like to thank our project mentor Dr.Bhavanishankar K Assistant Prof. Dept. of CSE, RNSIT Bengaluru.

REFERENCES

[1] Herbert Schildt, The Complete Reference , JAVA 9th edition. Java.util Part 2: The Collection Framework, 498-510.

[2] GeeksForGeeks, Collection in Java https://www.geeksforgeeks.org/collections-in-java-2/

[3] Tutorials point, Java-The Collections Framework , https://www.tutorialspoint.com/java/java_collections.htm

[4] Oracle,The JAVA tutorials,Intro to Collections, https://docs.oracle.com/javase/tutorial/collections/intro/index.html.