COLLECTION FRAMEWORK

COLLECTION IN JAVA

is a framework that provides an architecture to store and manipulate a group of objects.

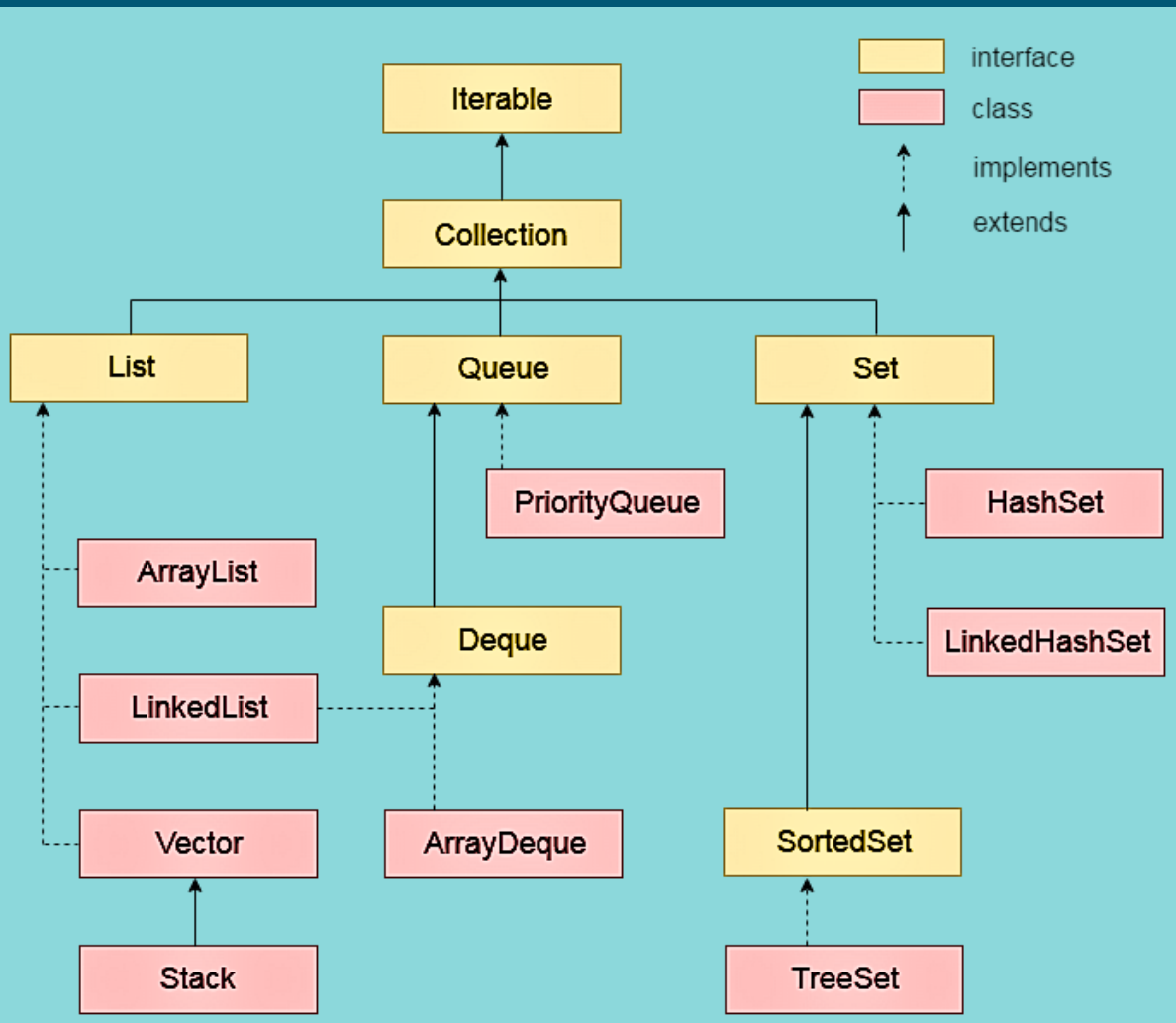
We can perform operations on data such as searching, sorting, insertion, manipulation and deletion.

INTERFACES

- Set
- List
- Queue
- Deque

CLASSES

- ArrayList
- LinkedList
- PriorityQueue
- HashSet
- LinkedHashSet
- TreeSet
- Vector
- Stack



COLLECTION INTERFACE

- `boolean add(E e)` - Insert an element in a collection.
- `boolean addAll(Collection<? extends E> c)` - Insert the specified collection elements in invoking collection.
- `boolean remove(Object element)` - To delete an element from the collection.
- `boolean removeAll(Collection<?> c)` - To delete all elements of specified collection from invoking collection.
- `boolean removeIf(Predicate<? super E> filter)` - To delete all elements of the collection that satisfy the specified predicate.
- `boolean retainAll(Collection<?> c)` - To delete all the elements of invoking collection except the specified collection.
- `int size()` - Returns a total number of elements in the collection.
- `void clear()` - Removes a total number of elements from the collection.
- `boolean contains(Object element)` - To search an element.
- `boolean containsAll(Collection<?> c)` - To search the specified collection in a collection.
- `Iterator iterator()` - It returns an iterator.
- `Object[] toArray()` - It converts a collection into an array.
- `<T> T[] toArray(T[] a)` - It converts a collection into an array with the runtime type of the specified array.
- `boolean isEmpty()` - Checks if a collection is empty.
- `Stream<E> parallelStream()` - Returns a possibly parallel Stream with the collection as its source.
- `Stream<E> stream()` - Returns a sequential Stream with the collection as its source.
- `Spliterator<E> spliterator()` - Generates a Spliterator over the specified elements in a collection.
- `boolean equals(Object element)` - Matches two collections.
- `int hashCode()` - Returns the hash code number of a collection.

LIST INTERFACE

- Child interface of Collection Interface.
- Stores ordered a collection of objects.
- Can have duplicate values.
- Implemented by ArrayList, LinkedList, Stack and Vector classes.

ArrayList

- Uses a dynamic array to store different data type elements.
- It maintains insertion order and is non-synchronized.
- Stored elements can be randomly accessed.

LinkedList

- Uses a doubly linked list internally to store elements.
- It maintains insertion order and is non-synchronized.
- Manipulation is fast because no shifting is required.

Vector

- Uses a dynamic array to store data elements.
- Similar to ArrayList, but is Synchronized and has many methods other than the collection framework.

Stack

- Subclass of Vector, implements LIFO(Last In First Out) data structure (Stack).
- Has all methods of vector and its own methods i.e. push(), pop(), peek() methods .

QUEUE INTERFACE

- Maintains First In First Out order (FIFO).
- Objects will be added from one end (tail) and removed from another end (head).
- Can have duplicate values.
- Implemented by PriorityQueue, Deque, and ArrayDeque classes.

DEQUE INTERFACE

- Stands for a double-ended queue.
- Extends Queue interface.
- Can add & remove elements from both ends.

Priority Queue

- Holds objects which need to be processed by priority.
- It doesn't allow null values.

ArrayDeque

- Implements Deque interface.
- Faster than ArrayList & Stack.
- Has no capacity restrictions.

SET INTERFACE

- Represent an unordered set of objects.
- Duplicate elements are not allowed.
- Can have only one null value.
- Implemented by HashSet, LinkedHashSet and TreeSet classes.

HASHSET

- Uses a Hash table for storage.
- Hashing is used for storing elements.
- It contains unique items.

SORTEDSET INTERFACE

- Maintains elements in ascending (sorted) order.
- Provides additional methods for ordering objects.

LinkedHashSet

- LinkedList implementation of Set interface.
- Maintains insertion order and permits null elements.

TreeSet

- Uses a Tree for storage.
- Access & retrieval time is quite fast.
- Objects are stored in sorted order.

MAP INTERFACE

- Map interface is part of the collection framework but it doesn't inherit Collection interface.
- Stores data in Key-Value pairs, the key is associated with a value.
- Key will be always unique.
- Implemented by HashMap, LinkedHashMap and TreeMap.

SORTEDMAP INTERFACE

- Extends Map interface.
- Maintains Sorted order of elements on the basis of key of each element.
- The sorting order is determined by the natural order of keys, which must implement a Comparable interface or by Comparator passed to its constructor.

HashMap

- Same as the legacy Hashtable class.
- It's not Synchronized.
- Can store null elements, but only one null key is allowed.
- extends AbstractMap which implements Map interface

LinkedHashMap

- Internally uses Hashtable and LinkedList.
- Maintains insertion order of elements.
- extends the HashMap and is not affected a key is reinserted into it.

TreeMap

- Uses Red-Black Tree for efficient storage.
- Cannot have a null key, but can have multiple null values.
- Maintains ascending (Sorted) order of elements.

PROPERTIES

✓ : Allowed

✗ : Not Allowed

1 : Only One value

IO : Insertion Order

NO : No Order

PAC : Placed According to Supplied Comparator or Natural order

	Null	Duplicate	Synchronized	Order of Elements
ArrayList	✓	✓	✗	IO
Vector	✓	✓	✓	IO
LinkedList	✓	✓	✗	IO
Stack	✓	✓	✓	LIFO
Priority Queue	✗	✓	✗	PAC
ArrayDeque	✗	✓	✗	LIFO/FIFO
HashSet	1	✗	✗	NO
LinkedHashSet	1	✗	✗	IO
TreeSet	✗	✗	✗	PAC
HashMap	1	✗	✗	NO
LinkedHashMap	1	✗	✗	IO
TreeMap	✗	✗	✗	PAC

Note: In case of Map, the Key can't be Null or Duplicate, but Values can be.

PERFORMANCE

COMPLEXITY OF OPERATIONS

ArrayList	Vector	LinkedList	Stack	Priority Queue	ArrayDeque	HashSet	LinkedHashSet	TreeSet	HashMap	LinkedHashMap	TreeMap	
O(1)	O(1)	O(1)	O(1)	O(log(n))	O(1)	O(1)	O(1)	O(log(n))	O(1)	O(1)	O(log(n))	Insertion
O(1)	O(1)	O(1)	O(1)	O(log(n))	O(n)	O(1)	O(1)	O(log(n))	O(1)	O(1)	O(log(n))	Removal
O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)	O(log(n))	O(1)	O(1)	O(log(n))	Retrieval

Same as ArrayList, But slower due to Synchronization

Popping of a required element

WHEN TO USE?

ArrayList : Need to perform more search operations than insertion & removal.

Vector : In need of a Synchronized list.

LinkedList : Need to perform insertion & removal more.

Stack : Need to perform Synchronized insertion from one end & removal from the other end.

PriorityQueue : Need to perform tasks on the basis of priority.

ArrayDeque : Need to implement Queue data structure or Stack.

HashSet : Want unique elements without any particular order.

LinkedHashSet : Want only unique elements in insertion order.

TreeSet : Want only unique elements in a specific order.

HashMap : Need Key-Value pairs without any order.

LinkedHashMap : Need Key-Value pairs in insertion order.

TreeMap : Need Key-Value pairs sorted in some specific order.