**Collection Framework:**

It defines several classes and interfaces used to represent a group of individual objects as a single entity.

**Collection (I):**

It is an interface used to represent a group of individual objects as a single entity.

**Collections (C):**

It is a utility class present in java.util package which defines several utility methods (Sorting, Searching...) for Collection objects.

**9 Key interfaces of Collection Framework:**

**1. Collection (I):**

It is an interface used to represent a group of individual objects as a single entity.

It defines most common methods which are applicable for any collection object.

In general Collection interface is considered as root interface of collection framework.

There is no concrete class which implements Collection interface directly.

**Collection (I) interface methods:**

**add(Object o)**

Add an object to the Collection.

**addAll(Collection c)**

Add a Collection to the existing Collection object.

**remove(Object o)**

Remove an object from the Collection.

**removeAll(Collection c)**

Remove a group of objects(Collection) from the Collection object.

**clear()**

Remove all objects from the Collection.

**retainAll(Collection c)**

Remove all the objects except the Collection c.

**isEmpty()**

To check if the Collection is empty or not.

**size()**

To get the size of the Collection object.

**contains(Object o)**

Check if an object is present in the Collections

**containsAll(Collection c)**

Check if the group of objects(Collection) are present in the Collection or not.

**toArray()**

Convert a Collection c to an Array. Return type of this method is Object.

Object[] toArray()

Object[] obj = c.toArray();

**Iterator iterator()**

To get the objects one by one. Return type of this method is iterator.

Note: Collection interface doesn’t contain any method to retrieve objects.

**2. List (I):**

It is the child interface of Collection interface.

To represent a group of individual objects where duplicates are allowed and Insertion order is preserved.

We can differentiate duplicates by using index.

We can preserve insertion order by using index. Index plays an important role in List interface.

**List interface methods:**

**void add(int index, Object o)**

Add the object at a specified index.

**boolean addAll(int index, Collection c)**

Add a group of objects(Collection) at a specified index

**Object remove(int index)**

Remove object at a specified index

**int indexOf(Object o)**

It return index of first occurrence of object o in the Collection.

**int lastIndexOf(Object o)**

It returns index of last occurrence of object o in the Collection.

**Object get(int index)**

It returns the object at the specified index.

**Object set(int index, Object o)**

Replace the old object at the specified index with new object. It returns old object.

**ListIterator listIterator()**

Returns the objects one by one from a List.

**Implementation classes of List (I):**

1. ArrayList

2. LinkedList

3. Vector

4. Stack

**1. ArrayList (C):**

Underlying data structure for ArrayList is resizeable array or growable array.

Duplicates are allowed.

Insertion order is preserved.

Heterogeneous objects are allowed.

Null insertion is possible.

Implements Serializable and Cloneable interfaces.

If our frequent operation is retrieval operation then ArrayList is the best choice because it implements RandomAccess interface.

ArrayList is the worst choice if our frequent operation is insertion or deletion in the middle, because several shift operations are required.

**Constructors in ArrayList class:**

**1. ArrayList l = new ArrayList();**

Creates an empty array list object with default initial capacity 10. Once array list reaches its maximum capacity a new array list will be created with new capacity = (currentCapacity\*3/2)+1.

All the contents of the old ArrayList are copied to the new ArrayList. Old ArrayList is eligible for garbage collection. Performance will be impacted due to this.

**2. ArrayList l = new ArrayList(int initialCapacity);**

Creates an empty array list with specified capacity, if we already know the size in advance we can create an array list using this constructor. Performance of the system will be improved by this.

**3. ArrayList l = new ArrayList(Collection c);**

Create an equivalent ArrayList object for any Collection object (LinkedList, Vector, TreeSet.....).

Note: We can use collections to hold and transfer objects from one place to another, to provide support for this requirement every Collection already implements Serializable interface and after it has reached its destination the user will clone the Collection using Cloneable interface.

**RandomAccess (I) interface:**

Only ArrayList and Vector classes implement RandomAccess (I) interface. Any random element can be accessed with the same speed.

It is present in java.util package. It has no methods. It is a marker interface.

If our frequent operation is retrieval operation then ArrayList or Vector are the best choice because they implement RandomAccess interface.

**Differences between ArrayList and Vector:**

|  |  |
| --- | --- |
| ArrayList | Vector |
| 1. Every method in ArrayList is Non-Synchronized  2. As multiple threads are allowed to access on ArrayList, so it is not thread safe  3. Performance of system is relatively high  4. It is non legacy (1.2 version) | 1. Most of the methods in Vector are Synchronized  2. Only one thread is allowed to access at a time, so it is thread safe  3. Performance of system is relatively low  4. It is a legacy class (1.0 version) |

**How to get synchronized version of ArrayList:**

public static List synchronizedList(List l)

ArrayList l1 = new ArrayList(); // Non-Synchronized

List l = Collections. synchronizedList(l1) // Synchronized List

Note: We can also get synchronized version of Set and Map interfaces as well,

public static Set synchronizedList(Set s)

public static Map synchronizedList(Map m)

**2. LinkedList (C):**

Underlying data structure is double linked list.

Duplicates are allowed.

Insertion order is preserved.

Heterogeneous objects are allowed.

Null insertion is possible.

Implements Serializable and Cloneable interfaces.

If our frequent operation is insertion and deletion operation then LinkedList is best choice.

LinkedList is the worst choice if our frequent operation is retrieval operation.

There is no reserving capacity or initial capacity for creating a linked list. Create a node and attach.

**Constructors in LinkedList class:**

**1. LinkedList l = new LinkedList();**

Creates an empty linked list object.

**2. LinkedList l = new LinkedList(Collection c);**

Creates an equivalent LinkedList object for the given Collection.

We can use LinkedList to implement stacks and queues. For this requirement LinkedList class defines some methods,

**void addFirst(Object o)**

Adds the object at the beginning of the LinkedList

**void addLast(Object o)**

Adds the object at the last position in the LinkedList

**Object getFirst()**

Returns the object at the beginning of the LinkedList

**Object getLast()**

Returns the object present at the last position of the LinkedList

**Object removeFirst()**

Removes the object at the first position of the LinkedList

**Object removeLast()**

Removes the object present at the last position of the LinkedList

**3. Vector (C):**

Underlying data structure resizeable or growable array.

Duplicates are allowed.

Insertion order is preserved.

Heterogeneous objects are allowed.

Null insertion is possible.

Implements Serializable and Cloneable interfaces.

If our frequent operation is retrieval operation then Vector is the best choice because it implements RandomAccess interface.

Vector is Synchronized, hence it is thread safe

**Vector class methods:**

add(Object o)

add(int index, Object o)

addElement(Object o)

Adds the object to Vector object

removeElement(Object o)

removeElementAt(int index)

removeAllElements()

Object get(int index)

Object elementAt(int index)

Object firstElement()

Object lastElement()

**int size()**

Objects present in the Vector

**int capacity()**

Total number of objects a vector can accommodate

**Enumeration elements()**

Get elements from vector one by one

Constructors in Vector class:

**1. Vector v = new Vector();**

Creates an empty Vector object with default capacity 10. Once Vector reaches its maximum capacity a new Vector object will be created with new capacity = 2\*currentCapacity.

**2. Vector v new Vector(int initialCapacity);**

Creates an empty Vector object with specified initial capacity

**3. Vector v = new Vector(int initialCapacity, int incrementalCapacity);**

Creates an empty Vector object with specified initial capacity and once Vector reaches its maximum capacity a new Vector object will be created with specified incremental capacity.

**4. Vector v = new Vector(Collection c);**

Creates an equivalent Vector object for the given Collection.

**4. Stack (C):**

Stack is the child class of Vector class

It is a specially designed class for LIFO order

**Constructors in Stack class:**

**1. Stack s = new Stack();**

Creates an empty Stack object.

**Stack class methods:**

**push(Object o)**

Adds an Object into the stack

**pop()**

Remove and return top of the stack

**peek()**

Return top of the stack.

**Empty()**

Check if the stack is empty or not

**offset search(Object o)**

To search if the element is present in the stack or not. It returns offset. If the element is present it will return offset (position of element from top). If it is not present -1 is returned.

**Three cursors in Java:**

1. Enumeration

2. Iterator

3. ListIterator

If we want to get the objects one by one from Collection, then we should go for cursors.

**1. Enumeration:**

Introduced in 1.0v (for Legacy).

We can use Enumeration to get the objects one by one from Collection Objects (Legacy Collections)

We can create Enumeration Object by using elements() method of Vector class

**public Enumeration elements();**

**Enumeration e = v.elements(); // v is a vector**

Creates an Enumeration object.

**Enumeration methods:**

**public boolean hasMoreElements();**

Check if there are more elements present.

**public Object nextElement();**

If there are more elements are there provides next element.

**Limitations of Enumeration:**

Enumeration concept is only applicable for legacy classes, hence it is not a universal cursor.

Using this cursor we can only get read access, we cant perform remove operation.

**2. Iterator (I):**

Introduced in 1.2v

We can use Iterator on any Collection. It is a universal cursor.

We can perform read and remove operation using Iterator.

To get Iterator object we can use Iterator method present in Collection interface.

Public Iterator iterator();

Iterator itr = c.iterator();

**Iterator methods:**

**pubic boolean hasNext()**

Check if there are more elements present.

**public Object next()**

If there are more elements are there provides next element.

**pubic void remove()**

Remove the object from Collection.

**Limitations:**

1. It is only forward direction cursor.

2. We cannot perform replace and addition of new Objects operations using Iterator.

**3. ListIterator (I):**

Introduced in 1.2v

We can move either forward direction or backward direction, hence ListIterator is bidirectional.

By using ListIterator we can perform replace operation and addition of new objects.

ListIterator is the child interface of Iterator.

We can create ListIterator object using listIterator method present in List interface.

**Public ListIterator listIterator()**

**ListIterator ltr = l.listIterator(); //l is any list object**

**ListIterator methods:**

public boolean hasNext()

public Object next()

public int nextIndex()

public boolean hasPrevious()

public Object previous()

public int previousIndex()

public void remove()

public void set(Object new)

public void add(Object new)

Note: We are not creating interface objects, internally there are some implementation classes that return an object

Ex: Enumeration e = v.elements();

inside vector class there is an element method in which there is an implementation class that return the object

System.out.println(e.getClass().getName()) // this will print the corresponding class name of Vector class

Vector$1

Set (I):

It is introduced in 1.2v

It is the child interface of Collection (I) interface.

Duplicates are not allowed

No insertion order

Set interface does not have any new methods

**Implementation classes of Set(I):**

**1. HashSet (C):**

Underlying data structure is Hash table.

If we try to add a duplicate into the HashSet, we do not get any error. The add method will return boolean value – false.

No insertion order is preserved.

All objects will inserted based on hashCode of objects.

Heterogeneous objects are allowed.

Null insertion is possible

Implements Serializable and Cloneable interfaces.

If our frequent operation is Search operation, then HashSet is the best choice.

Constructors in HashSet class:

1. HashSet h = new HashSet();

Creates a new HashSet object.

Default initial capacity is 16

Default Fill Ratio: 0.75 (Load Factor)

After filling 75% ratio a new HashSet Object is created.

2. HashSet h = new HashSet(int initialCapacity);

Creates a new HashSet with specified initial capacity and with a Load Factor = 0.75

3. HashSet h = new HashSet(int initialCapacity, float loadFactor);

Creates a new HashSet with specified initial capacity and Load Factor

4. HashSet h = new HashSet(Collection c);

Create an equivalent HashSet object for the given Collection.

2. LinkedHashSet (C):

It is introduced in 1.4v

It is the child class of HashSet.

Duplicates are not allowed.

Insertion order is preserved.

Underlying data structure is Hashtable + Linked list

LinkedHashSet is the best choice to develop Cache based applications.

3. SortedSet (I):

It is the child interface of Set (I).

If we want to represent a group of individual objects, where duplicates are allowed and according to some sorting order then we need to use SortedSet (I).

No Heterogenous objects.

SortedSet specific methods:

first()

Returns the first element in the SortedSet.

last()

Returns the last element in the SortedSet.

headSet(Object obj)

Returns the elements less than the specified element.

tallSet(Object obj)

Returns the elements that are greater than the obj

SortedSet subSet(Object obj1, Object obj2)

Returns the SortedSet whose elements are >=obj1 and<obj2

Comparator comparator()

Returns the comparator object that describes the underlying sorting technique.

If we are using default natural sorting order then we will get null

Default sorting order for numbers is ascending order and for String it is alphabetical order

Note: We can apply above methods only on SortedSet implemented class objects, that is on TreeSet object.

1. TreeSet (C):

Underlying data structure is balanced tree.

Duplicates not allowed

Insertion order is not preserved

Objects are entered according to some sorting order.

No heterogeneous objects are not allowed, runtime exception: class cast exception

Null acceptance – only once

**Constructors in TreeSet:**

1. TreeSet t = new TreeSet();

Creates an empty TreeSet object where elements will be inserted according to default natural sorting order

2. TreeSet t = new TreeSet(Comparator c);

Creates an empty TreeSet object where elements will be inserted according to customized sorting order.

3. TreeSet t = new TreeSet(Collection c);

Creates an equivalent TreeSet object for the given Collection.

4. TreeSet t = new TreeSet(SortedSet s);

Creates an equivalent TreeSet object for the given SortedSet object.

Note: Null Acceptance – For empty TreeSet null insertion is possible as a first element.

After inserting null if we try to enter any other element we will get NullPointerException.

We cannot enter null into a Non empty TreeSet, we will get NullPointerException.

If we are depending on natural sorting order the objects should be homogeneous and Comparable.

StringBuffer class does not implement Comparable interface (java.lang.comparable) hence we cannot add StringBuffer objects using the constructor TreeSet t = new TreeSet();

String and all Wrapper classes already implements comparable interface.

Comparable (I) interface:

It is present in java.lang package.

It has only one method

compareTo()

public int compareTo(Object obj)

Ex: obj1.compareTo(obj2)

returns -ve if obj1 has to come before obj2

returns +ve if obj1 has to come after obj2

returns 0 if obj1 and obj2 are equal

If we are depending on natural sorting order then internally JVM will call compareTo() method while inserting objects to the TreeSet. Hence the objects must be comparable

Ex: obj1.compareTo(obj2)

obj1 is the element which is being inserted, obj2 is the element which is already present

If we need customized sorting order then we need to use Comparator

Comparator (I) interface:

Customized sorting order

It is present java.util package

compare() & equals

public int compare(Object obj1, Object obj2)

returns -ve if obj1 has to come before obj2

returns +ve if obj1 has to come after obj2

returns 0 if obj1 and obj2 are equal

public boolean equals()

when ever we are implementing comparator interface we need to provide implementation for compare() method

Implementing equals method is optional because it already available in every java class from object class through inheritance

TreeSet t = new TreeSet(new myComaparator())

using myComparator() implementation class we can write out own customization logic and use in TreeSet

Map (I):

If we want to represent a group of objects as key-value pairs we can use Map.

Map is not child interface of Collection (I).

Both keys and values are objects.

Duplicate keys are not allowed, but values can be duplicated.

Each key-value pair is called entry, hence Map is considered as a collection of entry objects

Map interface methods:

1. put(Object key, Object value)

To add one key value pair to the Map. If a duplicate key is entered then the old value will be replaced with new value.

2. putAll(Map m):

m.get(key);

To get the corresponding value

m.remove(key);

To remove key from the Map (also removes the value)

m.containsKey(key);

To check if the key is available or not.

containsValue(value);

To check if the value is available or not.

m.isEmpty()

To chcek if the Map is empty or not

m.size()

To get the size of the Map

**Collection views of Map**

Set keySet()

To get all the keys from the Map

Collection values()

To get all the values from the Map

Set entrySet()

To get the set of key and values from the Map

Set s = m.entrySet();

Entry (I):

Without existing map object there is no chance of existing entry object, Hence entry interface is defined inside Map interface.

Interface Map{

.

.

.

interface Entry{

Object getKey()

Object getValue()

Object setValue(Object newValue)

}

}

Implementation classes of Map (I):

1. HashMap (C):

underlying data structure is Hash table

inerstion order is not preserved, inserted based on hash code of keys.

Duplicate keys not allowed, but values are allowed

heterogeneous allowed

Null is allowed only once for keys, for values any number of times.

Implements Serializable and Cloneable

Best choice if our operation is search operation

Constructors in HashMap:

1. HashMap m = new HashMap();

Creates an empty HashMap object with default size 16 and default fill ratio 0.75

2. HashMap m = new HashMap(int initialCapacity);

Creates an empty HashMap object with specified initial capacity and default fill ratio 0.75

3. HashMap m = new HashMap(int initialCapacity, float fillRatio);

Creates an empty HashMap object with specified initial capacity and specified fill ratio 0.75

4. HashMap m = new HashMap(Map m);

Creates an equivalent HashMap object for the given Map object.