Collections Framework

Topics

1. Introduction to Collections Framework

> A collections framework is a unified architecture for representing & manipulating collections, enabling collections to be manipulated independently of implementation details. All collections frameworks contain the following:

a) Interfaces: These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In Object oriented language, interfaces generally form a hierarchy.
b) Implementations: These are the concrete implementations of the collections interfaces. In essence, they're reusable data structures.
c) Algorithms: These are the methods that perform useful computations, such as searching & sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic i.e. same method can be used on many different implementations of the appropriate collection interface.

Apart from the Java collections framework, the best – known examples of collections frameworks are the C++ Standard Template library (STL).

> Benefits of the Java Collections Framework
1. Reduces programming effort: By providing useful data structures & algorithms so you don't have to write them yourself.
2. Increases program speed & quality: This collections framework provides high – performance, high – quality implementations of useful data structures & algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. Because you're freed from the drudgery of writing our own data structures, you'll have more time to devote to improving program's quality & performance.
3. Allows interoperability between unrelated: The collection interfaces are the vernacular by which APIs pass collections back & forth. If my network administration API furnished a collection of node names & if your GUI toolkit expects a collection of column headings, our APIs will interoperated seamlessly, even though they were written independently.
4. Reduces the effort required to learn & to use new APIs
5. Reduces the effort required to design & implement: Designers & Implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.
6. Fosters software reuse: Fosters software reuse by providing a standard interface for collections & algorithms with which to manipulate them.

2. Array

> An array is an indexed collection of fixed no. of homogenous data elements.

> The main advantage of Array is we can represent multiple values by using single variable so that readability of the code will be improved.

> Limitations of Array are

a) Fixed in size i.e. once we create an array, there is no chance of increasing/decreasing the size based on our requirement. Due to this, to use Array's concept, compulsory we should know the size in advance which may not possible always.
b) Array can hold only homogenous datatype elements. (Though we can use **Object** type array)
c) Array concept is not implemented based on some standard data Structure & hence readymade method support is not available i.e. for every requirement, we have to write the code explicitly which increases complexity of programming.

> To overcome above limitations of Array, we should go for Collection concept.

> Advantages of Collection are

a) Collections are growable in nature i.e. based on our requirement we can increase/decrease the size.
b) Collections can hold both homogenous & heterogeneous elements.
c) Every collection class is implemented based on some standard data Structure hence for every requirement, readymade method support is available.

> Difference b/w Array & Collection

SNo	**Array**	**Collection**
1.	Fixed in size.	Growable in nature.
2.	With respect to memory, Arrays are not recommended to use.	With respect to memory, Collections are recommended to use.
3.	With respect to performance, Arrays are recommended to use.	With respect to performance, Collections are not recommended to use.
4.	Arrays can hold only homogenous datatype elements.	Collections can hold both homo & heterogeneous datatype elements.
5.	No underlying data structure hence no readymade methods are available.	Every collection class has underlying data structure hence readymade methods are available.
6.	Arrays can hold both primitive & objects.	Collections can hold only object types but not primitive.

3. Collection

> A collection – sometimes called a container – is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate & communicate aggregate data.
> Typically, they represent data items that form a natural group, such as poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers).

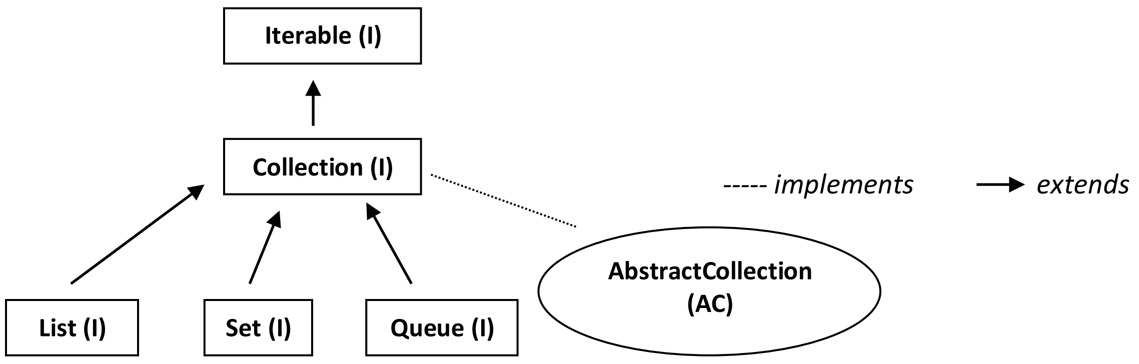
> The collection interfaces are divided into 2 groups i.e. `java.util.Collection` & `java.util.Map`

4. 10 key interfaces of Java Collections Framework

1. Collection Interface
2. List Interface
3. Set Interface
4. SortedSet Interface
5. NavigableSet Interface
6. Queue Interface
7. Deque Interface
8. Map Interface
9. SortedMap Interface
10. NavigableMap Interface

1. Collection Interface

> If we want to represent a group of individual objects as a single entity then we should go for Collection.
> The core collection interfaces encapsulate different types of collections like List interface, Set Interface, Queue Interface, Deque Interface etc.
> These interfaces allow collections to be manipulated independently of the details of their representation.
> Collection interface defines the most common methods which are applicable for any collection object.
> In general, Collection interface is considered as root interface of Collections framework.
> There is no concrete class which implements Collection interface directly



> The Collection interface is used to pass around collections of objects where maximum generality is desired. For e.g. by convention all general – purpose collection implementations have a constructor that takes a Collection argument. This constructor, known as a conversion constructor, initializes the new collection to contain all the elements in the specified collection, whatever the given collection's sub interface or implementation type.

> Methods in Collection Interface

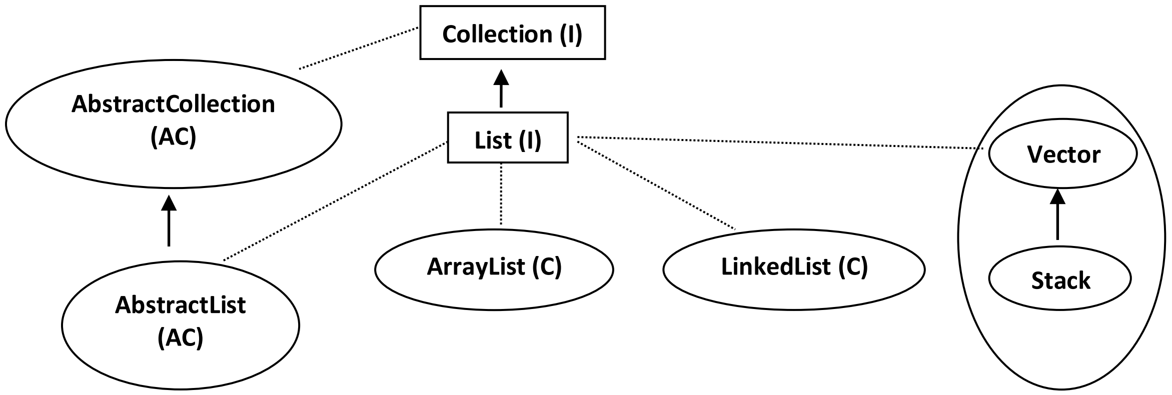
E : the type of elements in the collection
1. `boolean add(E e)` – to add an object to a collection.
2. `boolean addAll(Collection<? extends E>)` – to add a group of objects in the collection.
3. `void clear()` – to clear all objects from collection.
4. `boolean contains(Object)` – check a particular object is available or not.
5. `boolean containsAll(Collection<?>)` – check a group of objects available or not.
6. `boolean isEmpty()` – whether the collection is empty or not.
7. `boolean remove(Object)` – to remove a particular object from a collection.
8. `boolean removeAll(Collection<?>)` – to remove a group of objects from the collection.
9. `boolean removeIf(Predicate<? super E>)` – to remove all of the elements of the collection that satisfy the given predicate.
10. `boolean retainAll(Collection<?>)` – retains only the elements in the collection that are container in the specified collection i.e. removes from the collection all of its elements that are not contained in the specified collection.
11. `int size()` – how many objects are there in a collection.
12. `Iterator<E> iterator()` – to get object one by one from the collection.
13. `Spliterator<E> spliterator()` – creates a spliterator from the collection.
14. `Stream<E> stream()` – creates a sequential stream from the collection's spliterator.
15. `Stream<E> parallelStream()` – creates a sequential stream from the collection's spliterator.
16. `Object[] toArray()` – converts the collection to Array.
17. `<T> T[] toArray(T[])`

Note:
> There is also a **Collections class** in **java.util** package having utility methods for Collection objects (like sorting, searching etc.)

2. List (Interface) [Order Collection or Sequence]

> It is child interface of Collection interface.

> If we want to represent a group of individual objects as a single entity where **duplicates are allowed** & **insertion order must be preserved** then we should go for list.

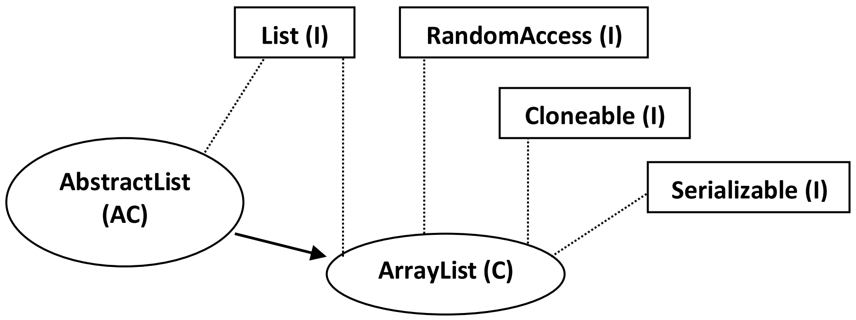


> Important methods of List interface

1. `void add (int index, E element)` – inserts the specified element at the specified position in the list.
2. `boolean addAll (int index, Collection<? Extends E> c)` – inserts all of the elements in the specified collection into the list at the specified position onwards.
3. `E remove (int index)` – removes the element at the specified position in the list.
4. `E get (int index)` – returns the element at the specified position in the list.
5. `E set (int index, E element)` – replaces the element at the specified position in this list with the specified element.
6. `int indexOf(Object obj)` – returns the index of the first occurrence of the specified element in the list or -1 if the list does not contain the element.
7. `int lastIndexOf(Object obj)` – returns the index of the last occurrence of the specified element in the list or -1 if the last does not contain the element.
8. `ListIterator<E> listIterator ()` – returns a list iterator over the elements in the list.
9. `Spliterator<E> spliterator()` – creates a spliterator over the elements in the list.
10. `List<E> subList(int fromIndex, int toIndex)` – returns a view of the portion of the list between the specified fromIndex inclusive and toIndex exclusive.
11. `void sort (Comparator<? super E> c)` – sorts this list according to the order induced by the specified Comparator. Internally the implementation is a stable, adaptive, iterative mergesort.

List interface implementation classes

a) ArrayList class



> The underlying data structure is Resizable array or Growable array.
> Duplicates are allowed.
> Insertion order is preserved.
> Heterogeneous objects are allowed (except TreeSet & TreeMap, everywhere Heterogeneous objects are allowed)
> null insertion is possible.

> Constructors

a) `ArrayList l = new ArrayList ();`
> creates an empty ArrayList object with default initial capacity of 10.
> Once ArrayList reaches its max capacity then a new ArrayList object will be created with

New Capacity = (Current Capacity * 3/2) + 1

b) `ArrayList l = new ArrayList (int initialCapacity);`
> creates an empty ArrayList Object with specified initial capacity.

c) `ArrayList l = new ArrayList (Collection c);`
> creates an equivalent ArrayList object for the given collection.
> ***This constructor is meant for inter conversion between Collection objects.***

Note:
> Usually, we can use Collections to hold & transfer objects from one location to another (container). To provide support for this requirement, every Collection class by default implements Serializable & Cloneable interface.
> Only ArrayList & Vector classes implements RandomAccess interface so that any random element we can access with the same speed.

> ArrayList is the best choice if our frequent operation is retrieval operation (as it implements RandomAccess interface).
> ArrayList is the worst choice if our frequent operation is insertion or deletion in the middle (because of shift operation).

RandomAccess interface
> It is present in java.util package & it does not contain any methods.
> It is a marker interface, where required ability internally automatically provided by JVM.

Q. How to get synchronized version of ArrayList object?
Ans: - By default ArrayList is non – synchronized but we can get synchronized version of ArrayList object by using synchronizedList () method of Collections class.

e.g.

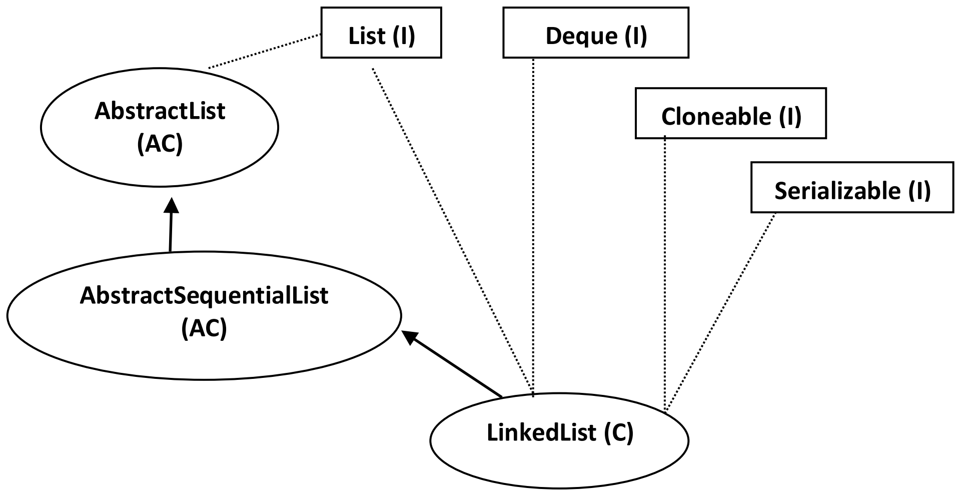
```
public static List synchronizedList (List l)
```

```
ArrayList unsyncList = new ArrayList();
List syncList = Collections.synchronizedList(unsyncList);
```

> Similarly we can get synchronized version of Set & Map objects by using the following methods of Collections class:

```
public static Set synchronizedSet (Set s)
public static Map synchronizedMap (Map m)
```

b) LinkedList class



> The underlying data structure is Doubly Linked list.
> Insertion order is preserved.
> Duplicate Objects are allowed.
> Heterogenous objects are allowed.
> null Insertion is possible.
> LinkedList does not implement RandomAccess interface.
> LinkedList is the best choice if our frequent operation is insertion or deletion in the middle.
> LinkedList is the worst choice if our frequent operation is retrieval operation.

> Constructors

a) `LinkedList l = new LinkedList ();`
> creates an empty LinkedList object.

b) `LinkedList l = new LinkedList (Collection c);`
> creates an equivalent LinkedList object for the given collection.
> ***This constructor is meant for inter conversion between Collection objects.***

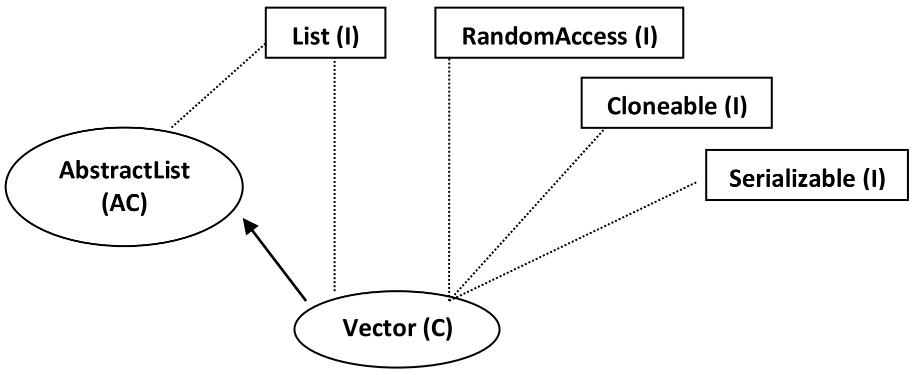
> Methods

1. `void addFirst (E e)` – Inserts the specified element at the beginning of the list.
2. `void addLast (E e)` – Inserts the specified element to the end of the list.
3. `E getFirst ()` – returns the first element in the list.
4. `E getLast ()` – returns the last element in the list.
5. `E removeFirst ()` – removes & returns the first element from the list.
6. `E removeLast ()` – removes & returns the last element from the list.

> Difference between ArrayList & LinkedList

No.	ArrayList	LinkedList
1.	ArrayList is the best choice if our frequent operation is retrieval operation because it implements RandomAccess interface.	LinkedList is the best choice if our frequent operation is insertion or deletion in the middle.
2.	It is worst choice if our frequent operation is insertion/ deletion in the middle due to shift operation.	It is the worst choice if our frequent operation is retrieval operation.
3.	In ArrayList, the elements will be stored in consecutive memory locations & hence retrieval operation will become easy.	In LinkedList, the elements won't be stored in consecutive memory locations & hence retrieval operation will become difficult.

c) Vector class



> The underlying data structure is Resizable array/Growable array.
> Insertion order is preserved.
> Duplicates are allowed.
> Heterogenous objects are allowed.
> null insertion is possible.
> It implements RandomAccess interface.

> ***Every method present in the vector is synchronized & hence vector object is thread – safe.***

> Constructors

a) `Vector v = new Vector ();`
> creates an empty vector object with initial capacity of 10.

a) `Vector v = new Vector (int initial_Capacity);`

> creates an empty vector object with specified initial capacity.

a) `Vector v = new Vector (int initial_Capacity, int incremental_Capacity);`
> creates an empty vector object with specified initial capacity & how much increment in size of vector is specified by incremental_Capacity.

d) `Vector v = new Vector (Collection c);`
> creates an equivalent vector object for the given collection.
> ***This constructor is meant for inter conversion between Collection objects.***

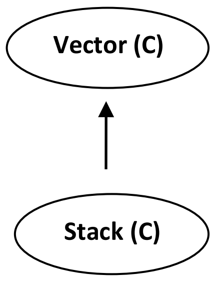
> Methods

1. `public synchronized void addElement (E e)` – adds the specified element at the end of the vector.
2. `public synchronized void removeElement (Object obj)` – removes the first occurrence of the argument from the vector.
3. `public synchronized void removeElementAt (int index)` – deletes the component at the specified index & each component in the vector with an index greater or equal to the specified index is shifted downward to have an index one smaller than the value it had previously.
3. `public synchronized void removeAllElements ()` – removes all components from the vector & sets its size to zero.
3. `public synchronized E elementAt (int index)` – returns the component at the specified index.
3. `public synchronized E firstElement ()` – returns the first element in the vector.
3. `public synchronized E lastElement ()` – returns the last element in the vector.
3. `public synchronized int size ()` – returns the number of components in the vector.
3. `public synchronized int capacity ()` – returns the current capacity of the vector.
3. `public Enumeration<E> elements ()` – return an enumeration of the components of the vector.

> Difference between ArrayList & Vector

No.	ArrayList	Vector
1.	Every method present in the ArrayList is non synchronized.	Every method present in the vector is synchronized.
2.	At a time, multiple threads are allowed to operate on ArrayList object & hence it is not thread – safe.	At a time, only on thread is allowed to operate on vector object & hence it is thread – safe.
3.	Relatively performance is high because threads are not required to wait to operate on ArrayList object.	Relatively performance is low because threads are required to wait to operate on Vector object.
4.	Introduced in 1.2 V & it is non – legacy.	It is introduced in 1.0 V & it is legacy.

d) Stack class



> It is the child class of Vector.
> It is a specially designed class for Last In First Out (LIFO) order.

> Constructors

a) `Stack s = new Stack ();`
> creates an empty stack object.

> Methods

1. `public E push (E item)` – pushes an item onto the top of the stack.
2. `public synchronized E pop ()` – removes the object at the top of the stack & returns that object as the value.
3. `public synchronized E peak ()` – looks at the object at the top of this stack without removing it from the stack.
4. `public boolean empty ()` – tests if the stack is empty.
5. `public synchronized int search ()`

3 Cursors of Java Collections Framework

> If we want to get Objects one by one from a collection, then we should go for cursor.
> There are 3 types of cursors available in Java

1. Enumeration Interface
2. Iterator Interface
3. ListIterator

1. Enumeration Interface
   > We can use Enumeration to get objects one by one from legacy collection object.
   > We can create Enumeration object by using element () method of Vector class.

   > Methods
   1. `boolean hasMoreElements ()` – tests if the enumeration contains more elements.
   2. `E nextElement ()` – returns the next element of the enumeration.

   > Limitations of Enumeration
   1. We can apply enumeration concept only for legacy classes & it is not universal cursor.
   2. By using Enumeration, we can get only read access & we can't perform remove operation.

   > To overcome these limitations, we should go for Iterator.

2. Iterator Interface
   > We can apply Iterator concept for any collection object; hence it is Universal Cursor.
   > By using Iterator we can perform both read & remove operations.
   > We can create Iterator object by using iterator () method present in Collection interface.

   > Methods
   1. `boolean hasNext ()` – returns true if the iteration has more elements.
   2. `E next ()` – returns the next element in the iteration.
   3. `void remove ()` – removes from the underlying collection the last element returned by the iterator.

   > Limitations of Enumeration
   1. By using Enumeration & Iterator, we can always move only towards forward direction not backward direction. So, these are single direction cursors not bidirectional cursor.
   2. By using Iterator, we can get only read & remove operation & we can't perform replacement or addition of new Objects.

   > To overcome these limitations, we should go for ListIterator.

2. ListIterator Interface
   > It is child interface of Iterator; hence all methods of Iterator by default available to the ListIterator.
   > ListIterator is a bidirectional cursor.
   > By using ListIterator, we can perform replacement & addition of new objects along with read & remove operations.
   > We can create ListIterator by using listIterator () method of List interface.

> Methods

1. `void add (E e)` – inserts the specified element into the list.
2. `boolean hasNext ()` – returns true if the list iterator has more elements when traversing the list in forward direction.
3. `boolean hasPrevious ()` – returns true if the list iterator has more elements when traversing the list in reverse direction.
4. `E next ()` – returns the next element in the list & advances the cursor position.
5. `int nextIndex ()` – returns the index of the element that would be returned by a subsequent call to next ().
6. `E previous ()` – returns the previous element in the list & moves the cursor position backwards.
7. `int previousIndex ()` – returns the index of the element that would be returned by a subsequent call to previous ().
8. `void remove ()` – removes from the list the last element returned by next () or previous ().
9. `void set (E)` – replaces the last element returned by next () or previous () with the specified element.

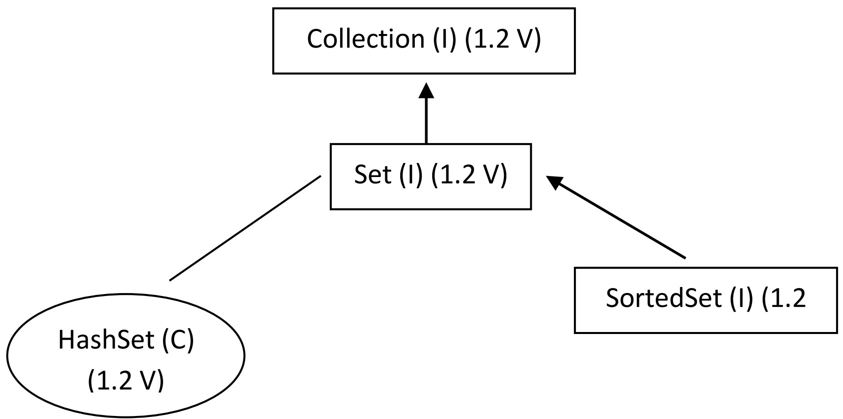
> The most powerful cursor is ListIterator but its limitation is it is applicable only for List objects.

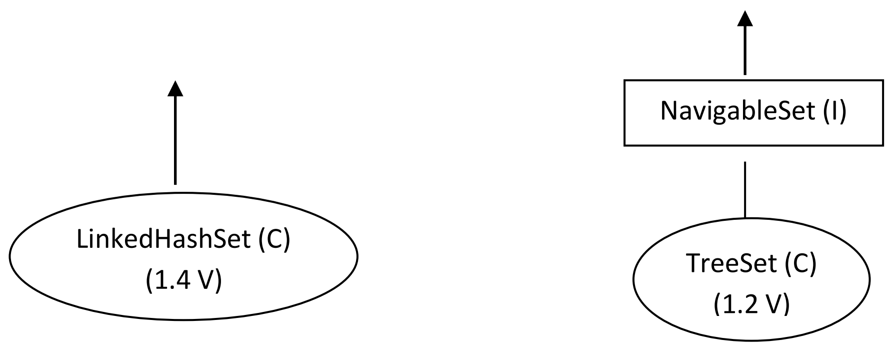
No.	Property	Enumeration	Iterator	ListIterator
1.	Where we can apply?	Only for legacy classes	For any Collection objects	Only for List objects
2.	Is it legacy?	Yes	No	No
3.	Movement	Forward direction	Forward direction	Bidirectional
4.	Allowed operations	Only read	Read, Remove	Read, Remove, Add, Replace
5.	How can we get?	By using element () method of Vector class.	By using iterator () method of Collection interface.	By using listIterator method of List interface.

3. Set (Interface)
> It is the child interface of Collection interface.
> If we want to represent a group of individual objects as a single entity where duplicates are not allowed & insertion order not required to be preserved then we should go for Set interface.

4. SortedSet (Interface)
> It is the child interface of Set interface.
> If we want to represent a group of individual objects as a single entity where duplicates are not allowed & all objects should be inserted according to some sorting order then we should go for SortedSet.

5. NavigableSet (Interface)
> It is the child interface of SortedSet interface.
> It contains several methods for Navigation purposes.





> Difference b/w List & Set

No.	List	Set
1.	Duplicates are allowed.	Duplicates are not allowed.
2.	Insertion order preserved.	Insertion order not preserved.

6. Queue (Interface)

> It is the child interface of Collection interface.
> If we want to represent a group of individual objects prior to processing then we should go for Queue.

7. Deque (Interface)

>

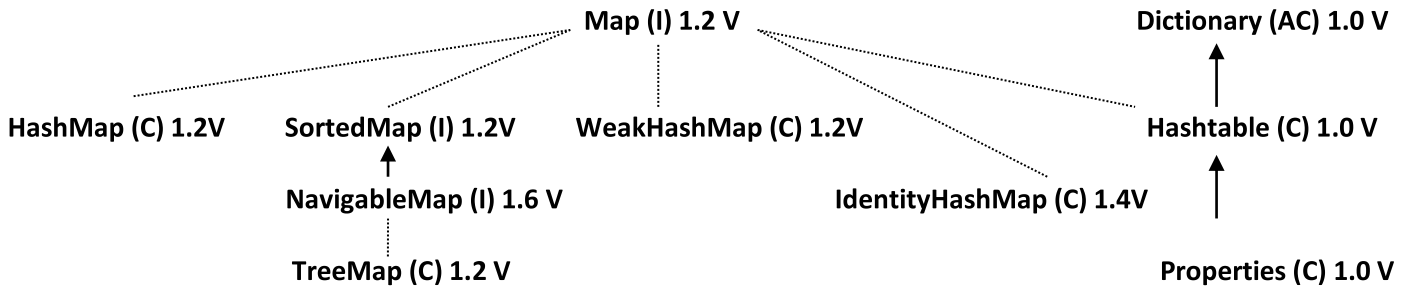
8. Map (Interface)

> Map is not child interface of Collection interface.
> If we want to represent a group of Objects as key – value pairs then we should go for Map interface.

e.g.

Key	Value
101	Sam
102	SRV
103	Code

> Both key & value are objects only & duplicate keys are not allowed but values can be duplicated.

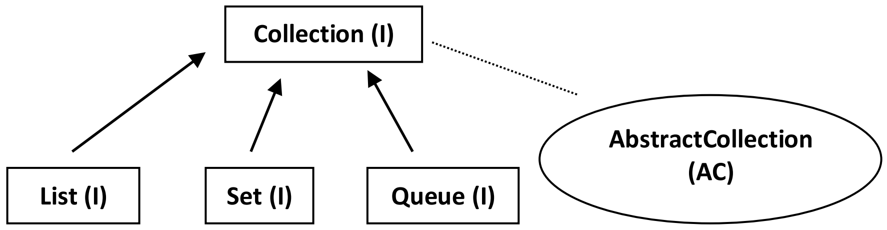


9. SortedMap (Interface)
> It is the child interface of Map interface.
> If we want to represent a group of key – value pairs according to some sorting order of keys then we should go for SortedMap.
> In SortedMap, the sorting should be based on key but not based on value.

10. NavigableMap (Interface)
> It is the child interface of Sorted Map.
> It defines several methods for Navigation purpose.

Note:
The following are the legacy classes & interfaces in Collections Framework

1. Enumeration (I)
2. Dictionary (AC)
3. Vector (C)
4. Stack (C)
5. Hashtable (C)
6. Properties (C)


Collection (I)
List (I)
Set (I)
Queue (I)
AbstractCollection (AC)