Agenda

- Collection
- List
- Vector
- Stack
- Cursors
- Set
- Map

Limitations of Object Arrays

Array are fixed in size . I.e Once we create an array there is no chance of increasing or decreasing size based on our requirements . Hence to use array Concept Compulsory we should know the size in advance which may not possible always .

Array can hold only Homogeneous data element i.e same time.

Array Concept not build based on Some data structure. Hence pre-defined method support is not available for every requirements . Compulsory programmer is responsible to write the logic

Advantage of Collections over Arrays

Collections are growable in nature . Hence based on our requirements we can increase or decrease the size.

Collection can hold both Homogeneous and Heterogeneous object

Every Collection class is implemented based on some data Stature . Hence readymade method support is available for most of the requirements

Collection [I]

A group of individual objects as a single entity is called “Collection”

It defines severals class and interface which can be used to represent a group of object as a single entity .

Collection interface is considered as root interface of collections framework

Collection interface defined the most common methods which can be applicable for any collection object .

Collection [I] Vs Collections [C]

Collection is an interface , Which is used to represent a group of individual objects as a single entity is called “Collection”

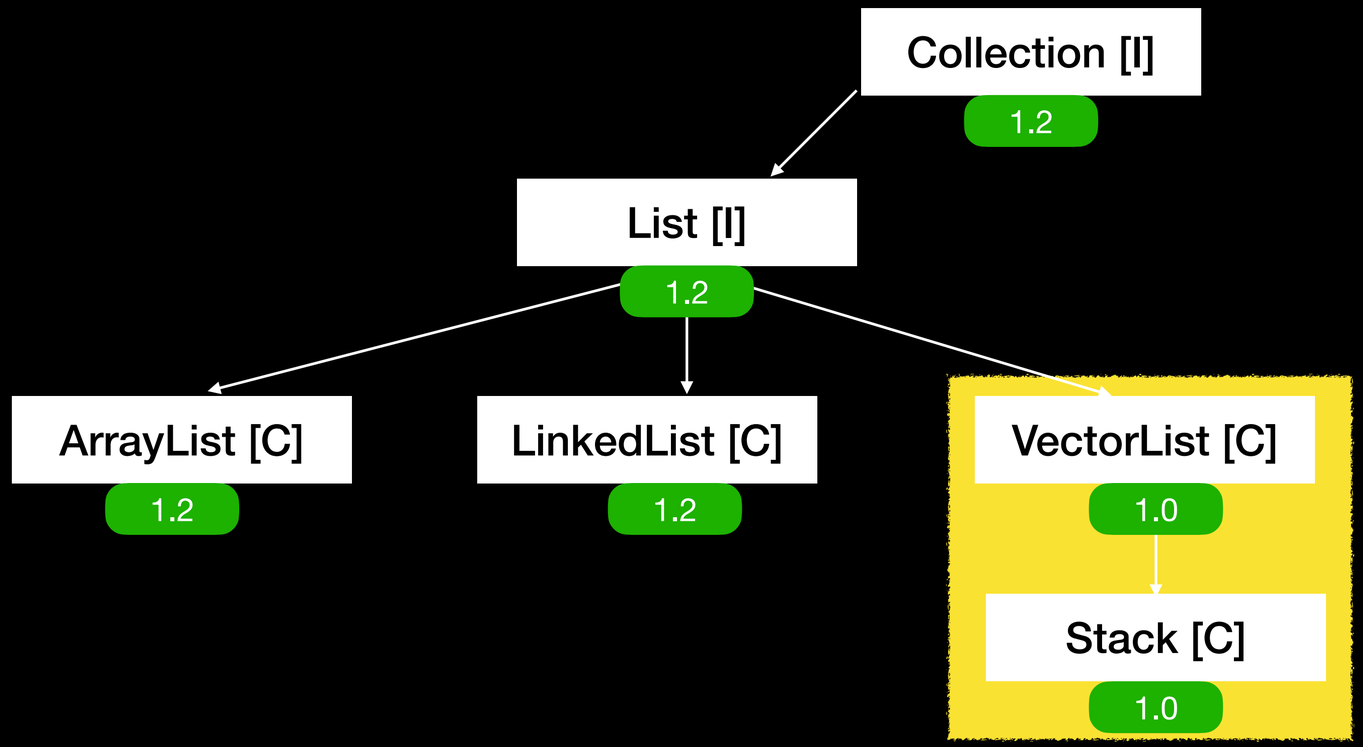
Collections is an utill class present in java.utill package to defines several utility methods for collections .

List [I]

It the Child Interface of Collection .

If we want to represent a group of individual objects where duplicate are allowed and insertion order is preserved . Then we should go for list.

It is Indexed based Collection .



ArrayList [c]

- **The Underlying Data Structure for ArrayList is Resizable Array ORGrowable Array.**
- **Duplicate Objects are allowed**
- **Insertion Order is Preserved.**
- **Heterogeneous Objects are allowed (Except *TreeSet* and *TreeMap* Everywhere Heterogeneous Objects are allowed).**
- **null Insertion is Possible.**

ArrayList Object Creation

- **ArrayList l = new ArrayList();**
- **List l = new ArrayList();**
- **ArrayList l = new ArrayList(int initialCapacity);**
- **Creates an Empty ArrayList Object with Default Initial Capacity 10.**
- **If ArrayList Reaches its Max Capacity then a New ArrayList Object will be Created with**

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

ArrayList characteristic

- ***ArrayList*** **and** ***Vector*** **Classes Implements** ***RandomAccess*** **Interface. So that we can Access any Random Element with the Same Speed.**

- ***RandomAccess*** **Interface Present in** ***java.util*** **Package and it doesn't contain any Methods. Hence it is a** ***Marker*** **Interface.**

- **ArrayList is Best Suitable if Our Frequent Operation is Retrieval Operation**

Every Method Present Inside ArrayListis Non – Synchronized.

At a Time Multiple Threads are allow to Operate on ArrayList Simultaneously and Hence ArrayList Object is Not Thread Safe.

By Default ArrayList Object is Non- Synchronized but we can get Synchronized Version ArrayList Object by using the following Method of Collections Class.

```
ArrayListal = new ArrayList ();
List l = Collections.synchronizedList(al);
```

- **ArrayList is the Best Choice if we want to Perform Retrieval Operation Frequently**

- **But ArrayList is Worst Choice if Our Frequent Operation is Insertion OR Deletion in the Middle. Because it required Several Shift Operations Internally.**

LinkedList [C]

- **The Underlying Data Structure is Double LinkedList.**
- **Insertion Order is Preserved**
- **Duplicate Objects are allowed.**
- **Heterogeneous Objects are allowed.**
- **null Insertion is Possible.**
- **Best Choice if Our Frequent Operation is *Insertion*OR *Deletion* in the Middle.**
- **Worst Choice if Our Frequent Operation is Retrieval.**

LinkedList Object Creation

- **LinkedList l = new LinkedList();**

- **List l = new LinkedList();**

Vector

- **The Underlying Data Structure is Resizable Array OR Growable Array.**
- **Duplicate Objects are allowed**
- **Insertion Order is Preserved.**
- **Heterogeneous Objects are allowed**
- **null Insertion is Possible.**
- **Implements *Serializable*, *Cloneable* and *RandomAccess* interfaces**
- **Every Method Present Inside Vector is Synchronized and Hence Vector Object is Thread Safe.**
- **Vector is the Best Choice if Our Frequent Operation is Retrieval.**
- **Worst Choice if Our Frequent Operation is *Insertion* OR *Deletion* in the Middle.**

Vector Object Creation

Vector v = new Vector();

- **Creates an Empty Vector Object with Default Initial Capacity 10.**
- **Once Vector Reaches its Max Capacity then a New Vector Object will be Created with**

New Capacity = (Current Capacity * 2)

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

Stack [C]

- It is the Child Class of Vector.

- It is a Specially Designed Class for Last In First Out (LIFO) Order.

Stack Object Creation

- Stack s = new Stack();

Cursors

- If we want to get Objects One by One from the Collection then we should go for Cursors.
- There are 3 Types of Cursors Available in Java.
  - Enumeration [Version - 1.0]
  - Iterator [Version - 1.2]
  - ListIterator [Version - 1.2]

Enumeration [Version - 1.0]

It is a Cursor to retrieve Object one by one from the collection .

It is applicable for legacy class.

- We can Create Enumeration Object by using elements().

public Enumeration elements();

Enumeration e = v.elements(); //Where v is Vector Object.

Enumeration Methods

- public boolean hasMoreElements();
- public Object nextElement();

Limitations of Enumeration

- Enumeration Concept is Applicable Only for Legacy Classes and it is Not a Universal Cursor
- By using Enumeration we can Perform *Read* Operation and we can't Perform *Remove* Operation.

Iterator [Version - 1.0]

It is a Cursor to retrieve Object one by one from the collection .

- We can Apply Iterator Concept for any Collection Object. Hence it is Universal Cursor
- By using Iterator we can Able to Perform Both *Read* and *Remove* Operations.
- We can Create Iterator Object by using iterator() of Collection Interface.
  - *public Iterator iterator();*
  - Iterator itr = c.iterator(); //Where c is any Collection Object.

Iterator Methods

- public boolean hasNext()
- public Object next()
- public void remove()

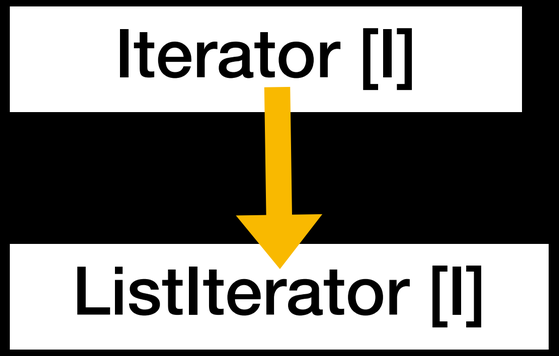
Limitations of Iterator

- **By using *Enumeration* and *Iterator* we can Move Only towards Forward Direction and we can’t Move Backward Direction. That is these are Single Direction Cursors but NotBi- Direction.**

- **By using Iterator we can Perform Only *Read* and *Remove* Operations and we can't Perform Addition of New Objects and Replacing Existing Objects.**

ListIterator [Version - 1.0]

- **ListIterator is the Child Interface of Iterator**
- **By using ListIterator we can Move Either to the Forward Direction OR to the Backward Direction. That is it is a Bi-Directional Cursor.**
- **By using ListIterator we can Able to Perform Addition of New Objects andReplacing existing Objects. In Addition to Read and Remove Operations.**
- **We can Create ListIterator Object by using listIterator().**
  - ***public List IteratorlistIterator();***
  - **ListIterator ltr = l.listIterator() // Where l is any List Object.**



ListIterator Methods

- **public boolean hasNext()**
- **public Object next()**
- **public int nextIndex()**

Forward Direction

- **public boolean hasPrevious()**
- **public Object previous()**
- **public int previousIndex()**

Backward Direction

- **public void remove()**
- **public void set(Object new)**
- **public void add(Object new)**

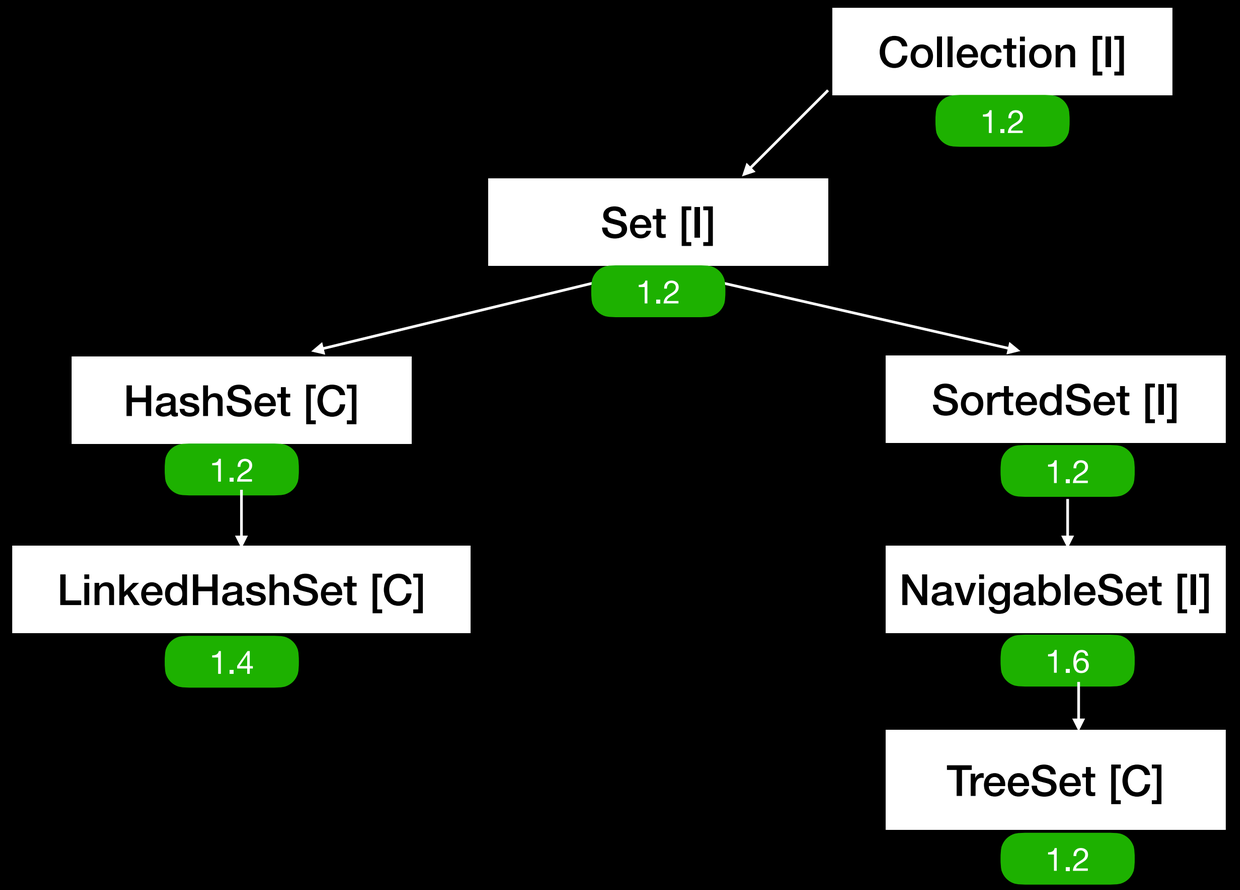
Comparison [Cursor]

Property	Enumeration	Iterator (1.2 v)	ListIterator(1.2)
It is Legacy	YES(1.0 Version)	NO(1.2 Version)	NO(1.2 Version)
It is applicable for	Only for Legacy	For any collection object	Only for List Object
Accessibility	Only Read	Read & remove	Read/remove/replace/add
Movement	**Single Direction (Only Forward)**	**Single Direction (Only Forward)**	**Bi-Direction**

Set [I]

It the Child Interface of Collection .

- If we want to Represent a Group of Individual Objects as a Single Entity where Duplicates are Not allowed and Insertion Order is Not Preserved then we should go for Set.



HashSet [C]

It the Child Class of Set .

- If we want to Represent a Group of Individual Objects as a Single Entity where Duplicates are Not allowed go for **HashSet.**
- The Underlying Data Structure for HashSet is **HashTable**.
- Insertion Order is **Not** Preserved.
- All Object are sorted according to the **HashCode** of the Object .
- Heterogeneous Object are allowed
- NULL Insertion is possible , but only once .
- It Implements **serializable** and **cloneable** Interfaces

HashSet [C] Object Creation

HashSet hashset = new HashSet();

- Creates an Empty HashSet Object with Default Initial Capacity 16 and Default Fill Ratio : 0.75.

HashSet hashed = new HashSet(int initialCapacity);

- Creates an Empty HashSet Object with specified Initial Capacity and Default Fill Ratio or Load Factor : 0.75 OR 75 %
- HashSet h = new HashSet(intinitialCapacity, float fillRatio);
- HashSet h = new HashSet(Collection c);

Set hashset = new HashSet();

LinkedHashSet [C]

- **It the Child Class of Set .**

- If we want to Represent a Group of Individual Objects as a Single Entity where Duplicates are Not allowed go for **LinkedHashSet.**

The Underlying Data Structure is a Combination of **LinkedList and Hashtable**

- Insertion Order is Preserved.

- It Implements **serializable** and **cloneable** Interfaces

LinkedHashSet [C] Object Creation

- This constructor is used to create a default HashSet

  - Set<E> linkedHashSet = new LinkedHashSet<E> ();

- Used in initializing the HashSet with the elements of the collection C

```
LinkedHashSet<E> linkedHashSet = new LinkedHashSet<E>(Collection c);
```

- Used to initialize the size of the LinkedHashSet mentioned in the parameter

```
LinkedHashSet<E> hs = new LinkedHashSet<E>(int size);
```

LinkedHashSet [C] Load Factor

- The default initial capacity is 16

- The default load factor is 0.75

LinkedHashSet Vs HashSet

HashSet	LinkedHashSet
The Underlying Data Structure is Hashtable.	The Underlying Data Structure is a Combination of LinkedList and Hashtable
Insertion Order is Not Preserved.	Insertion Order will be Preserved.
Introduced in 1.2 Version.	Introduced in 1.4 Version.

SortedSet [I]

- It is the Child Interface of Set.
- If we want to Represent a Group of Individual Objects as a Single Entity where Duplicates are Not allowed go for **SortedSet.**
- Objects will be Inserted According to Some Sorting Order
  - Sorting can be Either Default Natural Sorting OR Customized Sorting Order.
  - For String Objects Default Natural Sorting is Alphabetical Order.
  - For Numbers Default Natural Sorting is Ascending Order.

TreeSet [C]

- The Underlying Data Structure is Balanced Tree.
- Insertion Order is Not Preserved and it is Based on Some Sorting Order.
- Heterogeneous Objects are Not Allowed. If we are trying to Insert we will get Runtime Exception Saying ClassCastException.
- Duplicate Objects are Not allowed.
- Implements *Serializable* and *Cloneable* Interfaces but Not *RandomAccess* Interface.

NULL Acceptance [Before java 7]

- For Empty TreeSet as the 1st Element null Insertion is Possible. But after inserting that null if we are trying to Insert any Element we will get NullPointerException.
- For Non- Empty TreeSet if we are trying to Insert null we will get NullPointerException.

After java 7 *TreeSet* no longer supports the addition of *null value*

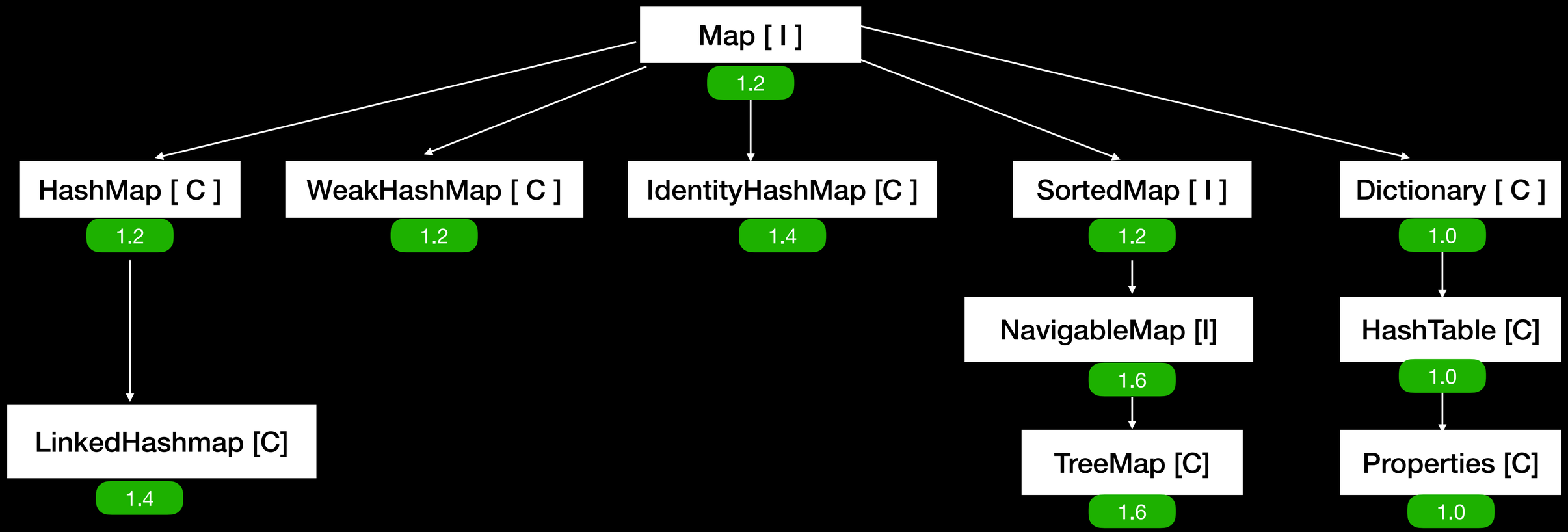
TreeSet [C] Object Creation

- Creates an Empty TreeSet Object where all Elements will be Inserted According to Default Natural Sorting Order.
  - TreeSet treeset = new TreeSet();
- Creates an Empty TreeSet Object where all Elements will be Inserted According to Customized Sorting Order which is described by Comparator Object.
  - TreeSet treeset = new TreeSet(Comparator c);
- TreeSet t = new TreeSet(Collection c);
- TreeSet t = new TreeSet(SortedSet s);

HashSet Vs LinkedHashSet Vs TreeSet

Property	HashSet	LinkedHashSet	TreeSet
Underlying Data Structure	Hashtable	Hashtable and LinkedList	Balanced Tree
Insertion Order	Not Preserved	Preserved	Not Preserved
Sorting Order	Not Applicable	Not Applicable	Applicable
Heterogeneous Object	Allowed	Allowed	Not Allowed
Duplicate Object	Not Allowed	Not Allowed	Not Allowed
NULL Acceptance	**Allowed** (Only Once)	**Allowed** (Only Once)	For Empty TreeSet as the 1st Element null Insertion is Possible. In all Other Cases we will get NullPointerException.

Map [I]


Map [I]
1.2
HashMap [C]
1.2
WeakHashMap [C]
1.2
IdentityHashMap [C]
1.4
SortedMap [I]
1.2
Dictionary [C]
1.0
LinkedHashmap [C]
1.4
NavigableMap [I]
1.6
TreeMap [C]
1.6
HashTable [C]
1.0
Properties [C]
1.0

Map [I]

- Map is Not Child Interface of Collection.
- If we want to Represent a Group of Objects as Key- Value Pairs then we should go for Map.
- Both Keys and Values are Objects Only
- Duplicate Keys are Not allowed. But Values can be Duplicated.
- Each Key- Value Pair is Called an Entry.

Map Methods

- Object put(Object key, Object value);
- void putAll(Map m)
- Object get(Object key)
- Object remove(Object key)
- boolean containsKey(Object key)
- boolean containsValue(Object value)
- booleanisEmpty()
- int size()
- void clear()
- Set keySet()
- Collection values()
- Set entrySet()

Entry [I]

- Each Key- Value Pair is Called One Entry.
- Without existing Map Object there is No Chance of existing Entry Object.
- Hence Interface Entry is Define Inside Map Interface.

```
interface Map {
    interface Entry {
            Object getKey();
            Object getValue();
            Object setValue(Object new);
        }
}
```

HashMap [C]

- The Underlying Data Structure is Hashtable.
- Duplicate Keys are Not Allowed. But Values can be Duplicated.
- Heterogeneous Objects are allowed for Both Keys and Values.
- Insertion Order is not preserved and it is based on hash code of the keys.
- null Insertion is allowed for Key (Only Once) and allowed for Values (Any Number of Times)

HashMap Object Creation

HashMap m = new HashMap();

- HashMap m = new HashMap(intinitialcapacity);
- HashMap m = new HashMap(int initialcapacity, float fillRatio);
- HashMap m = new HashMap(Map m);

HashMap [C] Load Factor

- The default initial capacity is 16
- The default load factor is 0.75

LinkedHashMap [C]

- It is the Child Class of HashMap.

- It is Exactly Same as HashMap Except the following Differeces.

The Underlying Data Structure is Combination of Hashtable and LinkedList.

- Insertion Order is preserved.

Introduced in 1.4 Version.

LinkedHashMap Object Creation

```
LinkedHashMap m = new LinkedHashMap();
```

- `LinkedHashMap m = new LinkedHashMap(int initialcapacity);`
- `LinkedHashMap m = new LinkedHashMap(int initialcapacity,  float fillRatio, boolean accessorder);`

LinkedHashMap [C] Load Factor

- The default initial capacity is 16
- The default load factor is 0.75

WeakHashMap [C]

- It is the Child Class of HashMap.
- It is Exactly Same as HashMap Except the following Differences.
  - In Case of WeakHashMap if an Object doesn't contain any References then it is Always Eligible for GC Even though it is associated with WeakHashMap
- In Java Collections, *WeakHashMap* class has been declared as follows
  - public class WeakHashMap<K, V> extends AbstractMap<K, V> implements Map<K, V>

WeakHashMap Object Creation

```
WeakHashMap <K, V>  whm = new WeakHashMap <K, V>( );
```

```
WeakHashMap <K, V>  whm = new WeakHashMap <K, V>( int initialCapacity );
```

```
WeakHashMap <K, V>  whm = new WeakHashMap <K, V>( int initialCapacity, float loadFactor );
```

```
WeakHashMap <K, V>  whm = new WeakHashMap <K, V>( Map m );
```

IdentityHashMap [C] Load Factor

- The default initial capacity is 16

- The default load factor is 0.75

IdentityHashMap [C]

- It is the Child Class of HashMap.
- It is Exactly Same as HashMap Except the following Differences.
  - In *HashMap* JVM will Use .equals() to Identify Duplicate Keys, which is Meant for *Content*Comparision
  - In *IdentityHashMap* JVM will Use == Operator to Identify Duplicate Keys, which is Meant for *Reference* Comparison.

IdentityHashMap Object Creation

```
IdentityHashMap <K, V>  ihm = new IdentityHashMap<K, V>( );
```

```
Map <K, V>  ihm = new IdentityHashMap<K, V>( );
```

```
Map <K, V>  ihm = new IdentityHashMap<K, V>(int ExpectedMaxSize);
```

```
IdentityHashMap <K, V>  ihm = new IdentityHashMap(Map m);
```

IdentityHashMap [C] Load Factor

- The default initial capacity is 16

- The default load factor is 0.75

SortedMap [I]

- It is the Child Interface of Map
- If we want to Represent a Group of Key - Value Pairs According Some Sorting Order of Keys then we should go for SortedMap.

Methods

- SortedMapDefines the following Specific Methods
  - Object firstKey();
  - Object lastKey();
  - SortedMapheadMap(Object key)
  - SortedMaptailMap(Object key)
  - SortedMapsubMap(Object key1, Object key2)
  - Comparator comparator()

TreeMap [C]

- The Underlying Data Structure is Red -Black Tree.
- Duplicate Keys are Not Allowed. But Values can be Duplicated.
- Insertion Order is Not Preserved and it is Based on Some Sorting Order of Keys.
- If we are depending on Default Natural Sorting Order then the Keys should be *Homogeneous* and *Comparable*. Otherwise we will get Runtime Exception Saying ClassCastException.
- If we defining Our Own Sorting by Comparator then Keys can be *Heterogeneous* and *Non- Comparable*.
- But there are No Restrictions on Values. They can be *Heterogeneous* and *Non- Comparable*

NULL Acceptance [Before JAVA 7]

- For Empty TreeMap as the 1st Entry with null Key is Allowed. But After inserting that Entry if we are trying to Insert any Other Entry we will get RE: NullPointerException
- For Non- Empty TreeMap if we are trying to Insert null Entry then we will get Runtime Exception Saying NullPointerException.
- There are No Restrictions on null Values.

After java 7 *TreeSet* no longer supports the addition of *null value*

TreeMap Object Creation

- TreeMap t = new TreeMap();
- TreeMap t = new TreeMap(Comparator c);
- TreeMap t = new TreeMap(SortedMap m);
- TreeMap t = new TreeMap(Map m);

HashTable [C]

- The Underlying Data Structure for Hashtable is Hashtable Only.
- Duplicate Keys are Not Allowed. But Values can be Duplicated
- Insertion Order is Not Preserved and it is Based on Hashcode of the Keys.
- Heterogeneous Objects are Allowed for Both Keys and Values.
- null Insertion is Not Possible for Both Key and Values. Otherwise we will get Runtime Exception Saying NullPointerException.
- Every Method Present in Hashtable is Synchronized and Hence Hashtable Object is Thread Safe.

HashTable Object Creation

- Hashtable h = new Hashtable();
- Hashtable h = new Hashtable(intinitialcapacity);
- Hashtable h = new Hashtable(intinitialcapacity, float fillRatio);
- Hashtable h = new Hashtable(Map m);

HashTable [C] Load Factor

- The default initial capacity is 11
- The default load factor is 0.75

Properties [C]

- It is the child class of HashTable.

- If we required to have DB User name , Password , URL etc then we should use the Properties .

Properties Object Creation

```
Properties p = new Properties();
```

Properties Methods

- public String getProperty(String pname);
- public String setProperty(String pname, String pvalue);
- public Enumeration propertyNames();
- public void load(InputStream is);
- public void store(OutputStreamos, String comment);

Thank You
Happy Learning
Keep Watching