==> Need of Collections?

-- Arrays have some limitations like they are fixed in nature, array can hole homogenous elements only

in array underlying datastructure is not there. to overcome these problems we should use Collections concept.

- Collection are growable in nature.
- Collection can hold Homegeneous and hetrogeneous elements.
-Every collection class is implemented based on some standard datastucture for every requirment readymade method

support is available.

Array	Collections
- Fixed in size	- Growable
- Memory wise Not recommended	- Recommended if Memory is critical
- Performance wise it is Recommended	- Not Recommended
- Only Homogeneous elements	- Hetrogeneous Elements
- No Undelying DataStructure Support	- UnderLying Data Structure Support
- can hold objects and premitives	- can hole only Objects

==> What is Collection?

-- Collection is group of individual Elements as a single entity.

-- Collection is like container which can hold multiple items of different types.

==> What is Collection Framework?

-- Collection framework contains several classes and interfaces which can be used to represent

group of individual objects as a single entity.

9 Key Interfaces of Collection framework :

---

1. Collection(I): -if we want to represent group of individual objects as single entity.
   -it defines most common methods which are applicable for every collection object.
   -Collection interface is considered as root interface of Collections framework.
   - There is no concrete class which implements Collection Interface directly.

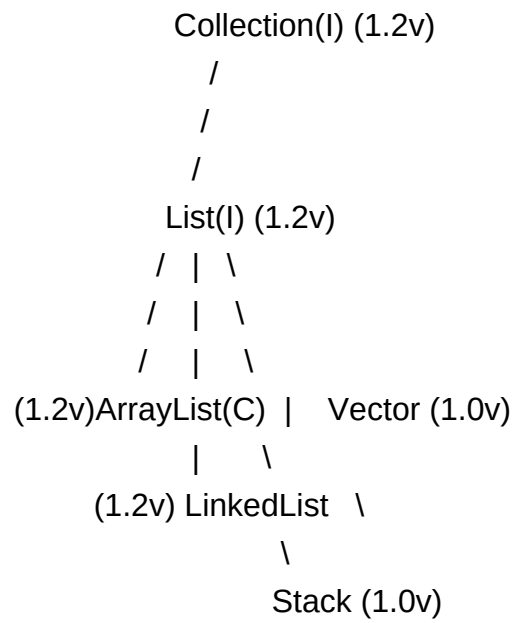
==> Collection vs Collections?

-- Collection is Interface if we want to represent group of individual objects as single entity.

-- Collections is utility class present in "java.util" package to define several utility methods for collection objects(i.e. sorting, serching etc.)

2. List(I) -it is the child interface of Collection.

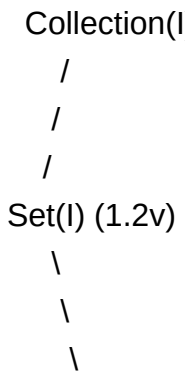
-if we want to group of objects as single entity where duplicates are allowed and insertion order must be preserved. then we should use List.

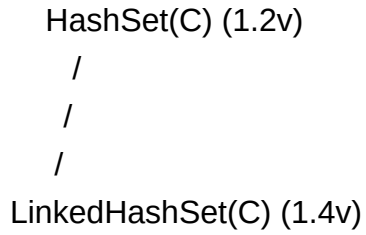


Note: In 1.2v of java "Vector" and "Stack" classes are Re-engineered to implement List interface.

3. Set(I): -it is child interface of collection.

-if we want to group of objects as single entity where duplicates are not allowed and insertion order not preserved. then we should use Set Interface.



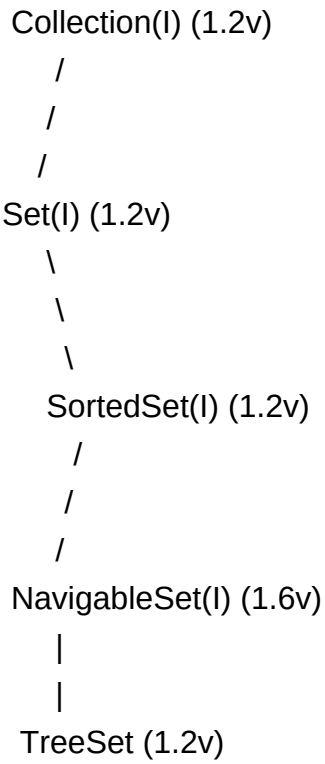


4. SortedSet(I): -it is child interface of Set.

-if we want to group of objects as single entity where duplicates are not allowed and all objects should be insertedaccording to some sorting order then we should use SortedSet Interface.

5. NavigableSet(I): -it is child interface of SortedSet.

-it contains several methods for navigation purposes.

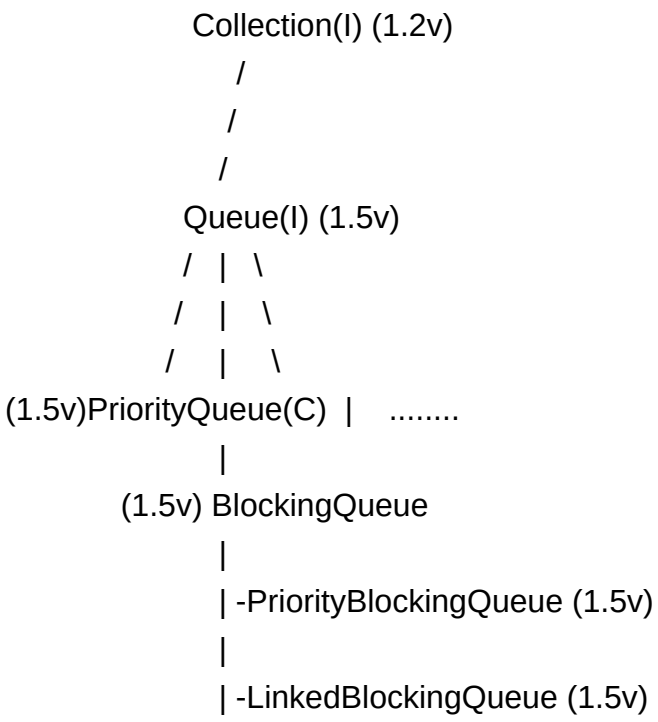


==> List vs Set

List	Set
- Duplicates are allowed	- Not allowed
- insertion order preserved	- insertion order not preserved

6. Queue(I): -it is child interface of Collection.

-if we want to group of objects "prior to processing" then we should go for Queue. usually Queue follows FIFO order. but based on our requirment we can implement our own priority Queue.

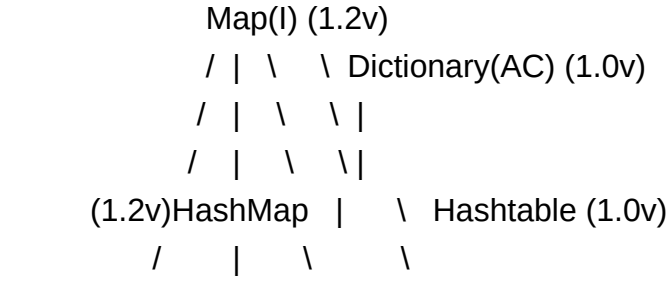


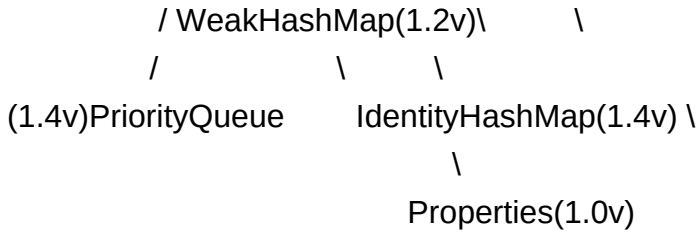
Note: all the above interfaces(Collection, List, Set, SortedSet, NavigableSet, Queue) meant for represent group of individual objects. if we want to represent group of objects as key-value pair then we should go for 'Map'.

7. Map(I): -Map is not child interface of Collection.

-if we want to represent group of objects as key-value pair then we should go for 'Map'.

-both key and value are Objects only duplicates keys are not allowed but values can be duplicated.





8. SortedMap(I): -it is child interface of Map.

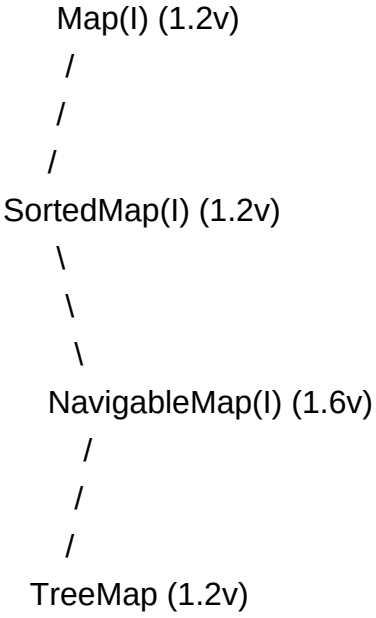
-if we want to represent group of objects as key-value pairs according to some "sorting order of keys"

then we should go for SortedMap.

-in SortedMap sorting should be based on key but not based on value.

9. NavigableMap(I): -it is child interface of SortedMap.

-it defined several methods for navigation purposes.



Note: the following are legecy characters are present in Collection:

-Enumeration(I)

-Dictionary(AC)

-Vector(C)

-Stack(C)

-HashMap(C)

-Properties(C)

Sorting | Cursors | Utility Classes |

----------|
-Comparable(I) | -Enumeration(I) | -Collections(C) |
-Comparator(I) | -Iterator(I) | -Arrays(C) |
| -ListIterator(I) | |
----------|

Collection Interface Methods:

- boolean add(Object obj)
- boolean addAll(Collection c)
- boolean remove(Object o)
- boolean removeAll(Collection c)
- boolean retainAll(Collection c) --> removes all objects execept those present in c
- void clear()
- boolean contains(Object o)
- boolean containsAll(Collection c)
- boolean isEmpty()
- int size()
- Object[] toArray()
- Iterator iterator()

List Interface methods:

- void add(int index, Object o)
- boolean addAll(int index, Collection c)
- Object remove(int index) -> returns removed object's index
- Object get(int index)
- Object set(int index, Object o) -> returns old object
- int indexOf(Object o)
- int lastIndexOF(Object o)
- ListIterator listIterator()

Implementation classes for List Interface:
---

1. ArrayList:
--> Constructors:

- ArrayList l = new ArrayList() -> creates empty ArrayList object with default initial cacapcity 10.

Once ArrayList reaches max capacity then new ArrayList Object will be created with ""new capacity = (current_capacity * (3/2)) + 1""

- ArrayList l = new ArrayList(int initial_capacity) -> creates empty ArrayList object with specified

initial cacapcity.

- ArrayList l = new ArrayList(Collection c) -> creates equivalent List object for given collection

Note: -usually we use Collections to hold and transfer objects from one location to another location(container) to provide support for this requirement every collection class by default implements "Serializable" and "Cloneable" interface.

-ArrayList and Vector classes implements "RandomAccess" interface so that any random element we can access with the same speed.

-RandomAccess: -this interface present in "java.util" package.

-doesn't contain any method. it is a marker interface. where required ability will be provided by JVM.

- ArrayList is best choice if our frequent operation is retrival object(RandomAccess). ArrayList is worst choice if our frequent operation is insertion and deletion in middle.

==> ArrayList vs Vector

ArrayList	Vector
-non synchronized mostly methods	-mostly methods synchronized
-not thread-safe	-thread-safe
-relatively-performance is high	-relatively-performance is low
-1.2v(Non legecy)	-1.0v()

==> How to get synchronized version of ArrayList Object?

--by default ArrayList is non-synchronized but we can get synchronized version of ArrayList Object by using "synchronizedList()" method of Collections class.

ArrayList l = new ArrayList();
List l2 = Collections.synchronizedList(l); --> public static List synchronizedList(List l)
^ ^
| |
synchronized Nnon-sychronized

--similarly we can get synchronized version of set and Map objects by using following methods:
- public static Map synchromizedMap(Map m)
- public static Set synchronisedSet(Set s)

2. LinkedList(C): -underlying datastructure is duble LinkedList.
-insertion order is preserved and duplicates are allowed.
-null insertion is possible.
-linked implementes implements Serializable and Cloneable interfaces but not RandomAccess interface.
-linkedlist is best choice if our frequent operation is insertion or deletion in middle. and worst choice if our frequent operation is retrieval.
--> Constructors:
-LinkedList l = new LinkedList() --> creates an empty LinkedList object.

-LinkedList l = new LinkedList(Collection c) --> creates an equivalent List object for given collection.

-> LinkedList class specific methods: usually we can use LinkedList to develop Stack and Queue to provide support
for this requirment LinkedList class defines following methods:
- void addFirst(Object O)
- void addLast(Object O)
- Object getFirst()
- Object getlast()
- Object removeFirst()
- Object removeLast()

==> LinkedList vs ArrayList:

---|
ArrayList | LinkedList |
---|
-best choice for retrieval | --best for insertion & deletion in middle |
-worst for insertion and deletion in middle | --worst for retrieval(frequent) |
-elements stored in consecutive memory locations | --elements are stored in consecutive memory locations|
---|

3. Vector(C): -underlying datastructure is resizable array or growable array.
-insertion order is preserved.
-duplicates are allowed
-hetrogeneous objects are allowed and null insertion is possible
-it implements "Serializable", "Clonable" and "RandomAccess" interfaces.
-every method present in vector is sychronized and hence Vector object is thread-safe

-->Constructors:
-Vector v = new Vector() --> create new empty vector object with initial capacity of 10. once vector
reaches its max capacity then a new Vector object will be created with new capacity is double of old capacity.

-Vector v = new Vector(int initial_cap) --> creates new empty vector object with specified initial
capacity.

-Vector v = new Vector(int initial_cap, int increamental_cap)

-Vector v = new Vector(Collection c)
--> Vector specific methods:
--to add elements:
- add(Objects o) ---> C
- add(int index, Object o) ---> L
- addElement(Object o) ----> V
--to remove Objects:
- remove(Object o) ---> C
- removeElement(Object o) ---> V
- remove(int index) ---> L

- removeElementAt(int index) ---> V
- clear() ---> C
- removeAllElements() ---> V

--to get Objects:

- Object get(int index) ---> L
- Object elementAt(int index) ---> V
- Object firstElement() ---> V
- Object lastElement() ---> V

--other methods:

- int size()
- int capacity()
- Enumeration elements()

4. Stack(C): -it is a child class of Vector. it is specially designed class for LIFO order.

--> Constructors:

- Stack s = new Stack();

--> methods:

- Object push(Object o) -->to insert object into stack
- Object pop() --> to remove and return top of the Stack.
- Object peak() --> to return top of the stack without removal
- boolean empty() --> returns true if the stack is empty.
- int search(Object o)--> returns offset if the element is available otherwise returns -1.

==> The Three Cursors of Java:

--If we want to get objects one by one from the collection then we should go for Cursor.

--there are three types of cursor in java.

I. Enumeration(I): -- we can use Enumeration to get Objects one-by-one from legecy Collection Object.

-- we can create Enumeration Object by using "elenments()" method of Vector class.

Enumeration e = v.elements();

--Methods:

-boolean hasMoreElements()

-Object nextElement()

--Limitations of Enumeration:

--We can apply enumeration concept only for legecy classes and its not a universal

cursor

--By using enumeration we can get only read access and we cant performe remove operation.

--To overcome above limitations we should go for Iterator.

II. Iterator(I): -- we can apply Iterator concept for any Collectionobject hence it is universal cursor.

--By using iterator we can preform both read and write operation.

--we can create Iterator Object by using "iterator()" method of Collection interface.

Iterator itr = c.iterator();

--Methods:

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

--Limitations of Iterator:

--By using Enumeration and Iterator we can always move only in forward direction but we cant move

in backward direction these are single direction Cursor but not Bi-directional cursor

--By using Iterator we can perform only read and remove operations and we can't perform replace of new Objects.

--To overcome above limitations we should go for ListIterator.

III. ListIterator(I): --By using ListIterator we can move either to forward direction or backwrad direction

hence it is bi-directional cursor.

--by using LisIterator we can perform replacement of new Objects in addition to read and remove operations.

--we can create ListIterator Object by using "listIterator()" method of List interface.

ListIterator litr = l.ListIterator();

--method:

--ListIterator is child interface of Iterator hence methods present in Iterator by default

present in ListIterator.

--ListIterator has following 9 methods:

-boolean hasNext()

-Object next()

-int nextIndex()

-boolean hasPrevious()

-Object previous()
-int previousIndex()
-void remove()
-void add(Object o)
-void set(Object o)

Note: most powerful cursor is ListIterator but its limitation is it is only applicable for List Objects.

==> Comparision of three cursors:

Property	Enumeration	Iterator	ListIterator
-application	-only for legecy calsses	-any Collection object	-Only List Objects
-Legecy?	-Yes(1.0v)	-No(1.2v)	-No(1.2v)
-Movement	-Uni-directional(forwrd)	-Uni-directional	-Bi-directional
-allowed ope.	-Read	-Read, Remove	-Read, Remove, Replace, Add
-Object Creation	-using elements() method of Vector class.	-Using "iterator()" method of Collection interface.	-Using "listIterator()" method of List interface.
-#methods	-2 methods	-3 methods	-9 methods

==> Intrenal implementations of Cursors:

```
Vector v = new Vector();
Enumeration e = v.elements();
Iterator itr = v.iterator();
ListIterator litr = v.listIterator();

System.out.println(e.getClass().getName());    // java.util.Vector$1
System.out.println(itr.getClass().getName());  //java.util.Vector$Itr
System.out.println(litr.getClass().getName()); //java.util.Vector$ListItr
```

Set(I):

methods: set interface doesn't contain any new method we have to use only Collection iterface methods.

HashSet(C): --underlying datastructure is Hashtable.

--duplication is not allowed and insertion order is not preserved and it is based on hashcode of objects.

--null insertion is possible(only once).

--implements "Serializable", "Clonable" and not "RandomAccess" interface.

Note: in hashSet duplicates are not allowed. if we try to insertany duplicate we won't get any runtime or compile time

Error. but add method simply returns false.

--Constructors:

-HashSet h = new HashSet() --> creates an empty HashSet object with default initial capacity with 16 and

default fill-ratio of 0.75.

-HashSet h = new HashSet(int initial_cap) --> creates an empty HashSet object with specified initial capacity

and default fill-ratio of 0.75.

-HashSet h = new HashSet(int initial_cap, float fillratio) --> creates an empty HashSet object with specified

initial capacity and specified fill-ratio.

-HashSet h = new HashSet(Collection c)

LinkedHashSet(C): --underlying datastructure is LinkedList and Hashtable.

-- it is a child class of HashSet. it is exactly same as HashSet(including Constructor and methods)

HashSet	LinkedHashSet
-underlying datastructure is Hashtable	-combination of Hashtble and LinkedList
-insertion order is not preserved	-insertion order is preserved.
-1.2v	-1.4v

Note: in general we can used LinkedHashSet to develoved Cache based application where duplicates are not allowed and insertion order is preserved.

SortedSet(I): --SortedSet is child interface of Set.
--arrange elements according to some sorting order without duplication.
--methods: -SortedSet interface defines the folloing specific methods:
-Object first() -> returns first element of SortedSet
-Object last() -> returns last element of SortedSet
-SortedSet headSet(Object o) -> returns SortedSet whose elements are less than obj.
-SortedSet tailSet(Object o) -> returns SortedSet whose elements are >= obj.
-SortedSet subSet(Object start, Object end) -> returns SortedSet whose elements are >= start and < end.
-Comparator comparator() -> returns Comparator object that describe underlying sorting technique. if we
are using default natural sorting order then we will get null.

-Note: the default natural sorting order for number is asscending order and for string objects alphabatical order.

TreeSet(C): --Underlying dataStructure for TreeSet is BalancedTree.
--insertion order is not preserved.
--Hetrogeneous elements are not allowed.
--null insertion is possible.(only once).
-Constuctors:
-TreeSet t = new TreeSet() --> creates an empty TreeSet object where all object will be inserted
according to default natural sortig order.
-TreeSet t = new TreeSet(Comparator c) -->creates an empty TreeSet object where all object will be
inserted according to customized sortig order.
-TreeSet t = new TreeSet(Collection c)
-TreeSet t = new TreeSet(SortedSet s)
-Null acceptance:
-for Non-empty TreeSet if we trying to insert null then we will get "NullPointerException".
-for empty TreeSet as first element null is allowed but after inserting the null if we try to insert any other element we will get "NullPointerElement"

*Note: untill 1.6v null is allowed as a first element to the empty TreeSet from 1.7v null is not allowed even as
the first element. null is not applicable for TreeSet from 1.7v.

Note: --if we are depending on default natural sorting order compulsory objects should be "Homegeneous and
Comparable" otherwise we will get RuntimeException saying "ClassCastException".
--object is said to be comparable if and if corresponding class implements Comparable interface. String and all Wrapper classes already implement Comparable interface but StringBuffer dosn't implement Comparable interface.

Comparable(I): -it is present in Java.lang" package and it contains only one method "compareTo()".

public int compareTo(Object obj)

the object which is to be inserted — object which is already inserted

```
obj1.compateTo(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.
```

Note: -if default natural sorting oreder not available or if we are not satisfied with default natural sorting
order then we can go for customized sorting by using Comparator.
-Comparable meant for default natural sorting order whereas Comparator is meant for customized sorting order.

Comparator(I): -it is present in "java.util" package. and it defines two methods "compare()" and "equal()".

```
-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(Object obj)

--whenever we are implementing Compartor interface. compulsion we should provide implementation for compare() method and we are not required to provide implementation to equals() method. it is already availble to our class from Object Class through inheritance.

==> Various possible implementation of compare() method:

```
-----------------------------------------------------
   public int compare(Object obj1, Object obj2){
Integer i1 = (Integer)obj1;
Integer i2 = (Integer)obj2;

// return i2 - i1;
// return i1.compareTo(i2);  // --> Assending Order
// return -i1.compareTo(i2); // --> Decending order
// return i2.compareTo(i1);  // --> decending order
// return +1; // [insertion order]
// return -1;  // [reverse of insertion order]
// return 0; // only first element is inserted & other will be considered as duplicates
      }
```

note: if we are depending on default natural sorting order compulsory objects should be homogeneous and comparable otherwise we will get ClassCastException. if we are defining our own customized sortng by comparator then objects need not be comaparable and homogeneous hence objects can be hetrogeneous and non-comparable also.

==> comparable vs Comparator:

Comparable	Comparator
-Default natural sorting order	-Customized sorting order
-java.lang package	-java.util package
-only 1 method	-2 methods
-implemented by String and all	-implemented by Collator and

Wrapper classes | RuledBasedCollator(GUI based) |
---|

Map(I):
methods:
-Object put(objects key, Object value) --> returns old vlaue if key already present and value id replaced
with new value else returns null.
-void putAll(Map m)
-Object get(Object key)
-Object remove(Object key)
-boolean containsKey(Object key)
-boolean containsValue(Object value)
-boolean isEmpty()
-int size()
-void clear()
-Set keySet()
-Collection values()
-Set emtrySet()

Entry(I): -Map is a group of a key-value pair and each key-value pair is called an Entry. hence Map is considered as
Collection of Entry objects. without Existing Map object there is no chance of existing Entry object. hence Entry object is defined inside Map interface.
methods:
-Object getKey()
-Object getValue()
-Object setValue(Object value)

HashMap: -underlying datastructure is Hashtable
-insertion order is not preserved. and it is based on hashcode of keys.
-duplicate keys are not allowed but values can be duplicated.
-hetrogeneous Objects are allowed for both key and values.
-null is allowed for key(only once) and for values(any number of times).
-HashMap implements Serilizable and Cloneable nut not RandomAccess.
-HashMap is best choice if frequent operation is Search operation.
'

-Constructor:

-HashMap h = new HashMap() --> creates an empty HashMap object with default initial capacity of 16 and

default fill ratio of 0.75.

-HashMap h = new HashMap(int initial_cap)

-HashMap h = new HashMap(int initial_cap, float fill_ratio)

-HashMap h = new HashMap(Map m)

==> HashMap vs Hashtable:

HashMap	Hashtable
-all methods are non-synchronized	-all methods are synchronized
-not thread-safe	-thread-safe
-relativly performance is high	-relatively preformnce is low
-null for key and value is allowed	-null is not allowed for keys and values otherwise we will get NPE
-not legecy(1.2v)	-it is legecy(1.0v)

LinkedHashMap: --it is child class of HashMap. it is exactly as HashMap including constructors except following differences:

HashMap	LinkedHashMap
-underlying datastructure is Hashtable	-underlying data structure is Hashtable + LinkedList(Hybrid).
-insertion order is not preserved and it is based on hashcode of keys.	-insertion order is preserved.
-introduced in 1.2v	-introduced in 1.4v.

IdentityHashMap: it is exactly same as HashMap(including methods and constructors) except following differences:

-In HashMap JVM will use "equals()" to identify duplicates keys. which is meant for content comparison. but in case of IdentityHashMap JVM will use "=="operator to idetify duplicate keys which is meant for reference comparision.

```
HashMap h = new HashMap();                    IdentityHashMap h = new IdentityHashMap()
Integer I1 = new Integer(10);                 Integer I1 = new Integer(10);
Integer I2 = new integer(10);                 Integer I2 = new Integer(10);
h.put(I1, "shubham");                         h.put(I1, "shubham");
h.put(I2, "asati");                           h.put(I2, "asati");
SOP(h); // { 10=asati }          -->          SOP(h); // { 10=shubham, 10=asati }
```

WeakHashMap: it is exactly same as HashMap except the following diferences:

-in the case of HashMap eventhough object doesnt have any reference it is not eligible for garbage collector hence HashMap dominates garbage collector.

-in the case of WeakHashMap, if object doesn't contain any refreences it is eligible for garbage collector even it is associated eith WeakHashMap. hence garbage collector dominates WeakHashMap.

SortedMap(I) -- it is the child interface of Map. if we want to represent group of key-value pairs according to some

sorting order of keys. them we should go for SortingMap

--sorting is based on keys but not values.

methods:

-Object firstKey()

-Object lastKey()

-SortedMap headMap(Object key)

-SortedMap tailMap(Object key)

-SortedMap subMap(Object start, Object end)

-Comaparator comparator()

TreeMap(): --underlying data structure is Rad-Black tree.

--insertion order is not preserved and it i based on hashcode of keys.

--duplicate not allowed for key values can be dulicated.

--if we are depending default sortng then keys should be homogeneous and comaparable otherwise we eill get rutime eeror saying ClassCastException if we are defining custom sorting then keys can hetrogeneous.

--wheather we are depending on default or custom sorting we an add hetrogeneous non comparable object also.

--Null Acceptance:

--first ever key can be null otherwise we will get NullPointerException until 1.6v from 1.7v null can't be inserted as key.

--Constructors:

TreeMap t = new TreeMap() ----> default sorting

TreeMap t = new TreeMap(Comaparator c) --> custom sorting

TreeMap t = new TreeMap(Collection c)

TreeMap t = new TreeMap(SortedMap sm)

Hashtable: --underlying data structure is Hashtable itself.

--insertion order is not preserved and it is basedon hashcode of keys.

--duplicate keys not allowed values can be.

--Hetrogeneous objects are allowed for both key and value

--null is not allowed for both key and value

--it implements serielizable and Cloneable interfaces but not RandomAccess.

--every method present in Hashtable is synchronised hence it is Thred-safe.

--it is best choice for search operations.

--Constructors:

-Hashtable h = new Hashtable() --> default initial capacity is 11 and fill ratio 0.75

-Hashtable h = new Hashtable(int initial_cap)

-Hashtable h = new Hashtable(int initial_cap, float fill_ratio)

-Hashtable h = new Hashtable(Map m)\

Properties: --in our program if anythin which changes frequently(username, password etc.) not recommanded to

hardcode in java programe if there is any change. to reflect that change recompilation, rebuild and redeploy of application is required.which crerate big bussiness impact to client.

--we can overcome this problem by using Properties file. such type of variable things we have to configure in propreties file. from properties file we have to read into java program and we can use those properties. advantage of this approach is. if there is change in property file to reflect that change redeploy is enough. which doesn.t create any bussiness impact.

--we can use java "Properties" object to hold properties which are comming from Properties file.

--in normal map(HashMap, Hashtable etc.) key-value can be of any type. but in Properties key and value should be of String type.

--Constructor:

Properties p = new Properties()

--method:

-String getProperty(String PName)

-String setProperty(String PName, String PValue)

-Enumeration propertyNames()

-void load(InputStream is) --> to load properties file into java properties object.

-void store(OutputStream os, String comment)--> to store Java Properties object to properties file.

1.5v inhencements Queue(I): --it is child interface of Collection

---------- --if we want to represent group of individual objects prior to processing then we should use Queue.

--usually Queue follows FIFO but we can implement our own priority order(PriorityQueue).

--1.5v onwards LinkedList class also implements Queue interface. LinkedList implementation of Queue always follows FIFO.

--methods:

-boolean offer(Object o) --> add an object to queue

-Object poll() --> remove and return head element -> returns null if queue is empty

-Object remove() --> remove and return head element -> RE: NoSuchElementException if queue is empty

-Object peak() --> returns head elemr=ent --> returns null if queue is empty

-Object element() ---> returns head element --> RE: NoSuchElementException if queue is empty

PriorityQueue: --based on some priority elements are processed

--priority can be eithe default sorting or customize sorting defined by caomaarotor

--insertion order is not prserved

--duplication not allowed

--if we are depending on default sorting then object should be Homogenous and comparable otherwise we will get RE: ClassCastException.

--if we are depending on custom sorting then object need notto be Homogenous and comparable.

--null is not allowed.

--constructors:

-PriorityQueue q = new PriorityQueue(); --> initial capacity 11 and DNSO

-PriorityQueue q = new PriorityQueue(int initial_cap);

-PriorityQueue q = new PriorityQueue(int initial_cap, Comparator c);

-PriorityQueue q = new PriorityQueue(SortedSet s);
-PriorityQueue q = new PriorityQueue(Collection c);

Note: some platform won't provide proper support for thread-priorities and PriorityQueue

NavigbleSet:
--methods:
-

NavigableMap:
--methods:

Collections(C): -Collections class defines several utitlity method for collection objects like sorting, searching, reverse etc..

--sorting elements of List:
I. public static void sort(List l)
II. public static void sort(List l, Comparator c)

--searching element:
I. public static int binarySearch(List l, Object target) --> returns insertion point if not found
II. public static int binarySearch(List l, Object target, Comparator c)