Collection Framework

- collectn is a framework, which can be used to represent a group of object
- Before collectn framework, to represent a group of objects array concept can be used
- in java

Disadvantages of array

- in java size of array is fixed, we can not change further
- to work with array we have to write algorithm manually
- to overcome above problem, collectn framework introduced.

Advantages of collectn

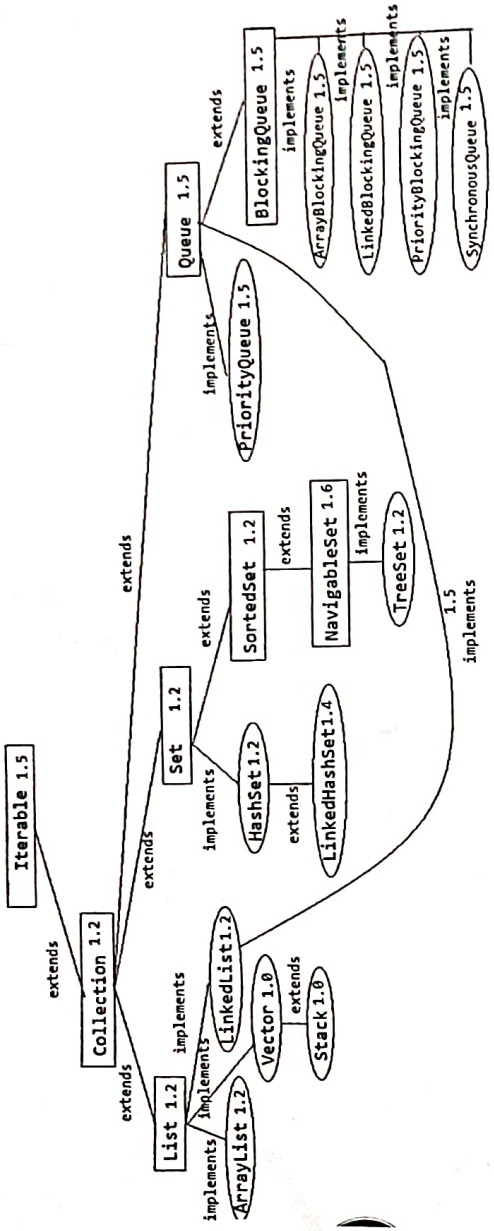
- collectn is growable, we don't need to think about size at all
- collectn framework provides readymade algorithm to work with it.
- to represent a group of objects collectn framework is best choice.

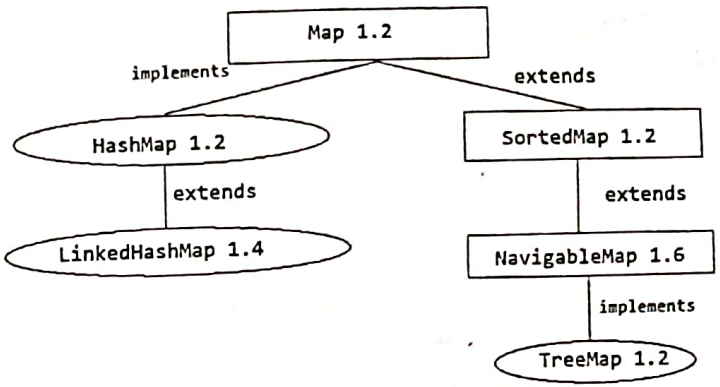
NOTE:- if we assured about size of an ~~array~~ object, array is best choice.

* How to represent collectn is object

e.g.

```
ArrayList<String> list = new ArrayList();
list.add("ABC");
list.add("DEF");
list.add("GHI");
list.add("JKL");
list.add("MNO");
```





★ Iterable

- Iterable is an interface, which is located in java.lang pkg.
- Iterable is a parent interface of collectn interface
- Iterable interface considered as a super interface in collectn framework
- Iterable interface contains iterator(), forEach and spliterator() method.
- Iterable interface can be used to represent collectn object easily
- Iterable introduced in java 1.5 version.

★ collectn

- collectn is an interface which is located in java.util pkg
- collectn is a child interface of Iterable interface
- collectn is a parent interface of list, set & queue interface.
- collectn can be used to represent a group of object

- collectn introduced method like add(), addAll(), remove(), removeAll(), contains(), containsAll(), isEmpty(), size(), clear(), toArray() etc.
- collectn introduced in Java 1.2 version.

* List
- List is an interface which is located in java.util.pkg
- List is a child interface of collectn interface.
- List Allows null elements
- List implemented by ArrayList, LinkedList, vector, and Stack class.
- List allows duplicates elements
- List allows heterogenous elements
- List maintains order
- List introduced in Java 1.2 version.

* ArrayList
- ArrayList is a class which is located in java.util.pkg

Syntax

```
public class ArrayList extends AbstractList implements
List, Cloneable, RandomAccess, Serializable
{
    // Logic
}
```

- ArrayList allows null elements
- ArrayList allows duplicates elements
- ArrayList allows heterogenous elements
- ArrayList maintains order
- ArrayList internally uses array concept to represent a group of objects
- default initial capacity of ArrayList is 10 elements
- ArrayList increase its capacity by following formula

old formula:

newSize = ((oldSize * 3) / 2) + 1;

= 10 → 16 = 160%.

= 100 → 151 = 151%.

new formula: onwards Java 1.7

newSize = oldSize + (oldSize >> 1);

= 10 → 15 = 150%

= 100 → 150 = 150%

- ArrayList is best choice, when our frequent operatn is to display because it implements RandomAccess interface
- ArrayList is worst choice, when our frequent operation is to insertion or deletn, because it requires shifting operatn
- ArrayList is non-synchronized or thread-unsafe implementatn
- To achieve synchronized or thread-safe ArrayList, we can use synchronizedList() static method of collectn class.

e.g.

```
List newObj = Collectns.synchronizedList(oldObj);
```

- ArrayList introduced in Java 1.2 version

* Non-Generic collectn vs Generic collectn

* Non-Generic collectn

eg.

```
ArrayList list = new ArrayList();
list.add(10);
list.add("ABC");
list.add('A');
list.add(true);
```

list.add(9.4);
S.o.p(list);

- Non-Generic collection allows heterogeneous elements
- Non-Generic collection doesn't give compile-time error during heterogeneous elements
- Non-Generic collection introduced in java 1.2 version

* Generic collection

e.g.

```
ArrayList<String> list = new ArrayList();
list.add("ABC");
list.add("DEF");
list.add("GHI");
list.add("JKL");
list.add("MNO");
S.o.p(list);
```

- Generic collection allows homogeneous elements
- Generic collection gives compile-time error during heterogeneous elements
- Generic collection introduced in java 1.5 version.

* How to represent collection's object.

e.g.

```
ArrayList<String> list = new ArrayList();
list.add("ABC");
list.add("DEF");
list.add("GHI");
list.add("MNO");
list.add("JKL");
```

① by using sop()

e.g.

```
S.o.p(list);
```

② by using get() method

e.g.

```
S.o.p(list.get(i));
```

③ By using For loop

e.g.

```
for(int i=0; i<list.size(); i++)
    S.o.p(list.get(i));
```

④ by using advance For loop

e.g.

```
for(String s: list)
    S.o.p(s);
```

⑤ by using iterator() method

e.g.

```
Iterator<String> itr = list.iterator();
while(itr.hasNext())
    S.o.p(itr.next());
```

⑥ By using listIterator() method

e.g.

```
ListIterator<String> litr = list.listIterator(list.size());
while(litr.hasPrevious())
    S.o.p(litr.previous());
```

④ by using forEach() method

e.g.

list.forEach(o -> System.out.println(o));

⑤ by using spliterator() method

e.g.

list.spliterator().forEachRemaining(o -> System.out.println(o));

✱ LinkedList

- LinkedList is a class, which is located in java.lang.pkg

Syntax

public class LinkedList extends AbstractSequentialList implements List, Deque, Cloneable, Serializable

{

// logic

}

-

- LinkedList allows null elements
- " duplicate elements
- " heterogenous elements
- LinkedList maintains order
- LinkedList internally used doubly linkedlist to represent a group of objects
- default initial capacity of linkedlist is zero
- LinkedList is best choice, when our frequent operation is to insertion or deletion because it doesn't require shifting operation
- LinkedList is worst choice when our frequent operation is to display, because it doesn't implements random access
- linkedList is non-synchronized or thread-unsafe implementation
- to achieve synchronized or thread-safe LinkedList we can use synchronizedList() static method of collection class

e.g.

List newObj = collections.synchronizedList(oldObj);

- LinkedList introduced in java 1.2 version
- Note: onwards java 1.5 version, LinkedList can be used as queue implementation

✱ vector

- vector is a class, which is a ~~class~~ located in java.long pkg
- Syntax

public class vector extends AbstractList implements List, RandomAccess, Cloneable, Serializable

{

// logic

}

- vector allows null elements
- vector allows duplicate elements
- vector allows heterogenous elements
- vector maintains order
- vector internally uses array concept, to represent a group of objects
- default initial capacity of vector is 10
- vector increase it's capacity by 2x
- vector is a synchronized or thread-safe implements
- vector is best choice, when our frequent operation is to display, because it's implements RandomAccess

interface.
- vector is worst choice, when our frequent operation is to insertion or deletion, because it requires shifting elements
- vector introduced in java 1.0 version, hence it is also known as a legacy class
- NOTE : ArrayList consumed less memory than vector and ArrayList can be used as a vector by making synchronized, hence developers ~~recommend~~ recommend to use ArrayList over vector.

* Stack
- Stack is a class located in java.util.pkg.
- Syntax

```
public class Stack, extends vector
{
    // logic
}
```

- Stack allows null elements
- Stack allows duplicate elements
- Stack allows heterogenous elements
- Stack maintains order
- default initial capacity of Stack is zero
- Stack can be used to represent group of object in last in first order (LIFO)
- Stack provides methods like push(), pop(), peek(), empty() etc.
- Stack is a synchronized or thread-safe implementation
- Stack introduced in java 1.0 version, hence it is considered as a legacy class

* Set
- Set is interface, which is located in java.util.pkg.
- Set is a child interface of collection interface
- Set is a parent interface of sorted interface
- Set is implemented by HashSet & LinkedHashSet class
- Set doesn't allow duplicate elements
- Set introduced in java 1.2 version

* HashSet
- HashSet is a class which is located in java.util.pkg.
- Syntax

```
public class HashSet extends AbstractSet implements Set
cloneable serializable
{
    // logic
}
```

- HashSet allows null elements
- HashSet doesn't allow duplicate elements
- HashSet allows heterogenous elements
- HashSet doesn't maintains order
- HashSet internally used hashtable & array concept to represent a group of objects
- default initial capacity of HashSet is 16 & load factor 0.75
- HashSet increase their capacity by 2x
- HashSet is best choice, when our frequent operation is to search & doesn't want to maintain an order
- HashSet is non-synchronized or thread-unsafe implementation

- to achieve synchronized or thread-safe HashSet, we can use synchronizedSet() static method of collection class
- e.g.
  Set newObj = collection.synchronizedSet(old obj);
- Hash introduced in java 1.2 version

* LinkedHashSet
- LinkedHashSet is a class, which is located in java.util.pkg.
- Syntax
  public class LinkedHashSet extends HashSet implements Set, Cloneable, Serializable
  {
    // logic
  }
- LinkedHashSet allows null elements
- LinkedHashSet doesn't allow duplicate elements
- LinkedHash allow heterogenous elements
- LinkedHashSet maintains order
- LinkedHashSet internally use hashtable & linked list to represent a group of objects
- default initial capacity of linkedHashSet is 16 & load factor is 0.75
- LinkedHashSet increase it's capacity by 2x
- LinkedHashSet is best choice when our frequent operation is to search & wants to maintain an order
- LinkedHashSet is non-synchronized or thread-unsafe implementation
- to achieve synchronized or thread-safe LinkedHashSet we can use synchronizedSet() static method of collection class
e.g.
  Set newObj = collections.synchronizedSet(old obj);
- LinkedHashSet introduced in java 1.2 version.

* Sorted set
- Sorted set is an interface which is located in java.util pkg.
- SortedSet is a child interface of set interface
- SortedSet is a parent interface of NavigableSet interface.
- SortedSet doesn't allow duplicate elements
- SortedSet doesn't allow heterogeneous elements
- SortedSet doesn't maintain order
- SortedSet introduced method like headSet(), tailSet(), subSet, first(), last(), etc.
- Sorted set introduced in java 1.2 version

* Navigable Set
- NavigableSet is an interface, which is located in java.util pkg.
- NavigableSet is a child interface of SortedSet interface
- NavigableSet doesn't allow null elements
- NavigableSet doesn't allow duplicate elements
- NavigableSet doesn't allow heterogenous elements
- NavigableSet doesn't maintain order
- NavigableSet introduced methods like higher(), lower(), ceiling(), floor(), pollFirst(), pollLast() etc.
- Navigable introduced in java 1.6 version.

→ Treeset

- Treeset is a class, which is located in java.util.pkg

· Syntax

public class Treeset extends Abstractset implements Navigable, cloneable, serializable

{

// logic

}

- Treeset doesn't allow null elements
- Treeset doesn't allow duplicate elements
- Treeset doesn't allow heterogenous elements
- Treeset doesn't maintain order
- default initial capacity of treeset is zero
- Treeset sort primitive datatype by using CNSO (default natural sorting order)
- Treeset sort non-primitive datatype using CNSO (customized natural sorting order)
- Treeset sort allows non-primitive datatype only they are either comparable or comparator
- Treeset internally uses redblack balance search tree algorithms to sort objects.

* Comparable

- comparable is an interface which is located in java.lang.pkg.
- comparable interface contains compareTo() method
- comparable interface can be used to compare 2 objects on the basis of single property at time.
- comparable introduced in java 1.2 version

* comparator

- comparator is an interface, which is located in java.util.pkg.
- comparator interface contains compare() method.
- comparator interface can be used to compare 2 objects on the basis of multiple property at a time
- comparator interface introduced in java 1.2 version.

- Treeset is non-synchronized or thread-unsafe implementation
- to achieve synchronized or Thread-safe Treeset, we can use synchronizedSortedSet() static method of collection class

e.g.

SortedSet newobj = collections.synchronizedSortedSet(oldobj);

- Treeset is best choice, when our frequent operation is to search & want to sort each objects
- Treeset introduced in java 1.2 version.

e.g.

package com.model;
public class Student {
private int id;
private String name;
private String city;
private double percentage;
// setter & getter
// toString()
// constructor

pkg.com.model
import java.util.comparator
public class StudentByIdAsc implements comparator<student> {

```
public int compare (Student o1, Student o2) {
    return (o1.getId() > o2.getId()) ? : (o1.getId() < o2.
        getId()) ? -1 : 0;
    }
}

public class StudentByNameAsc implements Comparator
                                    <Student> {
    public int compare(Student o1, Student o2) {
        return o1.getName().compareTo(o2.getName())
    }
}

public static void main (String[] args)
{
    Treeset<Students> t = null
    switch int choice = 7;
    switch (choice) {
    case 0:
        t = new Treeset (new StudentById ASC());
        breake;
    Case 1:
    t = new Treeset (new StudentById DESC());
    breake;

        // All cases as like same

}
```

```
t.add (new Student (10, "justin", "salem", 89.44));
    —
    —
    —

for (Student s : t)
    System.out.println (s);
}
}
```

* Queue
- Queue is an interface which is located in java.util pkg
- Queue is child interface of collection interface
- Queue is parent interface of BlockingQueue interface
- Queue implemented by priorityQueue
- Queue can be used to represent a group of objects in FIFO order.
- Queue introduced methods like add(), offer(), remove(), poll(), element(), peek() etc.
- Queue introduced in java 1.5 version

methods of queue
1) add()
- add() method can be used to add element into Queue
- add() method returns true or exception

2) offer()
- offer() method can be used to add elements into queue
- offer() method returns true or false

3) remove
- remove() method can be used to remove elements in from queue
- remove() method returns object or exception

4) poll()
- poll() method can be used to remove top element from queue
- poll() method returns object or null

5) element()
- element() method can be use to represent top element from queue
- elements() method returns object or exception

6) peek()
- peek() method can be used to represent top element from queue
- peek() method returns object or null

* priorityQueue
- priorityQueue is a class which is located in java.util pkg.
- Syntax
```
public class priorityQueue extends AbstractQueue
implements Queue, Serializable
{
    // logic
}
```
- priorityQueue doesn't allow null elements
- priorityQueue allows duplicate elements
- priorityQueue doesn't allows heterogeneous elements
- priorityQueue doesn't maintain order
- default initial capacity of priorityQueue is 11 elements
- priorityQueue increase capacity by 2x
- priorityQueue considered as an unbounded queue
- priorityQueue sorts primitive datatype by using default
- priority queue sorts non-primitive datatype by using CNSO
- priorityQueue allows non-primitive datatype only if they are either comparable or comparator
- priorityQueue is non-synchronized or thread-unsafe implementation
- priorityQueue can be used represent a group of objects in FIFO order on the basis of priority
- priority queue introduced in java 1.5 version

* LinkedList
- LinkedList is a class which is located in java.util.pkg
Syntax
```
public class LinkedList extends AbstractSequentialList
implements List, Deque, Serializable, Cloneable
{
    // logic
}
```
- LinkedList allow null element
- " duplicate element
- " heterogeneous elements
- LinkedList maintains order
- default initial capacity of LinkedList is zero
- onwards Java 1.5 version, LinkedList can be used a queue implementation

- LinkedList as a queue can be used to represent a group of objects in FIFO order without any restrictions
- LinkedList is non-synchronized or thread-unsafe
- LinkedList is an unbounded queue implementation
- LinkedList introduced in java 1.2 version.

* BlockingQueue
- BlockingQueue is an interface, which is located in java.util.concurrent pkg
- BlockingQueue is a child interface of queue interface
- BlockingQueue interface implements by ArrayBlocking queue, PriorityBlockingQueue, LinkedBlockingQueue & SynchronousQueue class
- BlockingQueue can be used to represent a group of objects in FIFO order & wants to solve produced & consumer problem
- BlockingQueue introduced in java 1.5 version

* ArrayBlockingQueue
- ArrayBlockingQueue is a class, which is located in java.util.concurrent pkg
- Syntax

```
public class ArrayBlockingQueue extends AbstractQueue
implements BlockingQueue Serializable
{
    //logic
}
```

- ArrayBlockingQueue doesn't allow null elements
- ArrayBlockingQueue allows duplicate elements
- ArrayBlockingQueue allow heterogeneous elements
- ArrayBlockingQueue maintains order
- ArrayBlockingQueue is a bounded queue
- ArrayBlockingQueue is a synchronized or thread-safe implementation
- ArrayBlockingQueue can be used in appn like limited login device like wifi connect
- ArrayBlockingQueue introduced in java 1.5 version.

* LinkedBlockingQueue
- LinkedBlockingQueue is class which is located in java.util.concurrent pkg
- Syntax

```
public class LinkedBlockingQueue extends AbstractQueue
implements BlockingQueue Serializable
{
    //logic
}
```

- LinkedBlockingQueue doesn't allow null elements
- LinkedBlockingQueue allows duplicate elements
- ↳ allows heterogeneous elements
- ↳ maintains order
- initial default capacity LinkedBlockingQueue is (2^31)-1.
- LinkedBlockingQueue considered as an unbounded queue if size is not mentioned
- LinkedBlockingQueue considered as bounded queue if size is mentioned
- LinkedBlockingQueue can be used to represent a group of objects in FIFO order without any restrictions & wants to solve produced & consumer problem
- LinkedBlockingQueue can be used in applicatn like zoom, tv Broadcasting, email forwarding etc.
- LinkedBlockingQueue introduced in java 1.5 version

* priorityBlockingQueue
- priorityBlockingQueue is a class which is located in java.util.concurrent pkg.
- Syntax
public class priorityBlockingQueue extends AbstractQueue implements BlockingQueue, Serializable
{
// logic
}
- priorityBlockingQueue doesn't allow null elements
- -"- allows duplicate elements
- -"- doesn't allow heterogeneous elements
- -"- doesn't maintain order
- default initial capacity of priorityBlockingQueue is 11 elements
- priorityBlockingQueue increase its capacity by 2x
- -"- considered as an unbounded queue
- -"- sort primitive datatype by using CNSO
- -"- sort non-primitive datatype by using CNSO
- -"- allows non-primitive datatype only if they either comparable or comparator
- -"- is synchronized or thread-safe implementation
- -"- can be used to represent a group of objects in FIFO order on the basis of priority & counts to solve producer & consumer problem
- -"- can be used in appln like amazon prime, mobile tower appn
- -"- introduced in java 1.5 version.

* SynchronousQueue
- SynchronousQueue is a class, which is located in java.util.concurrent pkg.
- Syntax
public class SynchronousQueue extends AbstractQueue implements BlockingQueue, Serializable
{
// logic
}
- SynchronousQueue doesn't allow null elements
- -"- allow single element only to be added
- -"- considered as a bounded queue.
- -"- is a synchronized or thread-safe implementation
- -"- can be used in appln like google pay like single login, bluetooth connectivity
- -"- introduced in java 1.5 version.

* Map
- Map is an interface which is located in java.util pkg.
- Map is a parent interface of SortedMap interface
- Map implemented by HashMap & LinkedHashMap class
- Map doesn't allow duplicate key
- Map can be used to group of objects in key & value pair
- Map introduced methods likes size(), isEmpty(), containsKey(), containsValue(), get(), put(), putAll(), remove(), clear(), KeySet(), entrySet() etc
- Map introduced in java 1.2 version

* HashMap
- HashMap is a class which is located in java.util pkg.
- syntax
public class HashMap extends AbstractMap implements Map, Cloneable, Serializable
{
// logic
}
- HashMap doesn't allow duplicate key
- HashMap allow null key
- HashMap allow heterogeneous key
- HashMap doesn't maintain order key
- HashMap internally, hashtable & array to represent a group of object
- default initial capacity of HashMap is 16 & load factor is 0.75
- HashMap increase it's capacity by 2x
- HashMap is a non-synchronized or thread-unsafe implementation
- to achieve synchronized or thread-safe HashMap we can used synchronizedMap() static method of collection class

```
Map<K,V> newObj = Collections.synchronizedMap(oldObj);
```

- HashMap is best choice when our frequent operation is to search & doesn't want to maintain an order
- HashMap introduced in java 1.2 version.

* LinkedHashMap
- LinkedHashMap is a class, which is located in java.util pkg.
- syntax
public class LinkedHashMap extends AbstractMap implements Map, Cloneable, Serializable
{
// logic
}
- LinkedHashMap allow null key
- " doesn't allow duplicate key
- " allow heterogenous key
- " maintain order
- " internally uses doubly LinkedList & Hashtable to represent a group of objects
- default initial capacity of LinkedHashMap is 16 & loadfactor is 0.75
- LinkedHashMap is non-synchronized or thread-unsafe implementation
- to achieve synchronized or thread-safe LinkedHashMap, we can used synchronizedMap() static method of collection class
- Map<K,V> newObj = collections.synchronizedMap(oldObj);

- LinkedHashMap is best choice, when our frequent operation is to search & wants to maintain order of key
- LinkedHashMap introduced in java 1.4 version

* SortedMap
- SortedMap is an interface which is located in java.util.pkg.
- SortedMap is a child interface of Map interface
- SortedMap is a parent interface of NavigableMap interface
- SortedMap doesn't allow duplicate Key
- SortedMap doesn't allow null key
- SortedMap doesn't maintain order
- SortedMap doesn't allow heterogeneous Key
- SortedMap introduced methods like headMap(), tailMap(), subMap(), firstKey(), lastKey() etc
- SortedMap introduced in java 1.2 version.

* NavigableMap
- NavigableMap is an interface which is located in java.util pkg
- NavigableMap is a child interface of SortedMap interface
- NavigableMap implemented by TreeMap class
- NavigableMap doesn't allow null key
- NavigableMap doesn't allow duplicate key
- it doesn't maintain order
- it doesn't allow heterogeneous key
- NavigableMap introduced methods like ceilingKey() ceilingEntry(), floorKey(), floorEntry(), higherKey(), higherEntry(), lowerKey(), lowerEntry(), pollFirstEntry() pollLastEntry() etc
- Navigable Map introduced in java 1.6 version

* TreeMap
- TreeMap is a class which is located in java.util pkg
- Syntax
public class TreeMap extends AbstractMap implements NavigableMap, Cloneable, Serializable
{
// logic
}
- TreeMap doesn't allow duplicate key
- TreeMap doesn't allow null key
- TreeMap doesn't allow heterogeneous key
- TreeMap doesn't maintain order of key
- default initial capacity of TreeMap is zero
- TreeMap internally uses redblack balance search tree to sorts each key
- TreeMap sorts primitive key by using DNSO
- TreeMap sorts non-primitive key by using CNSO
- TreeMap allow non-primitive datatype if they are either comparable or comparator
- TreeMap is non-synchronized or thread-unsafe implementation
- to achieve synchronized or thread-safe TreeMap we can use synchronizedMap() static method of collection class
e.g.
SortedMap<K,V> newobj = Collections.synchronizedMap(oldobj);
- TreeMap is best choice, when our frequent operation is to search & wants to sorts each key
- TreeMap introduced in java 1.2 version

Pg 1.

```
public static void main (String[] args) {
    Map<String, String> m = new LinkedHashMap();
    m.put ("e", "elephant");
    m.put ("a", "aeroplane");
    m.put ("d", "dragon");
    m.put ("b", "basket");
    m.put ("c", "champion");
    System.out.println (m);
    S.o.p (m.get ("d"));
    S.o.p (m.containsKey ("e"));
    S.o.p (m.containsValue ("champion"));
    S.o.p (m.size());
    S.o.p (m.isEmpty());
    Set <String> set = m.keySet();
    for (String s: set)
        S.o.p (s + "\t" + m.get(s));
    S.o.p ("-----");
    Set<Entry<String, String>> set1 = m.entrySet();
    for (Entry<String, String> e: set1)
    S.o.p (e.getKey() + "\t" + e.getValue());
}
}
```

Pg. 2

```
public class App{
    public static void main (String[] args)
    {
    Map<String, Set<String>> m = new HashMap();
    String q1 = "what is java?";
    Set <String> ans1 = new HashSet();
    ans1.add ("java is VM");
    ans1.add ("java is programing language");
    ans1.add ("java is device");
    ans1.add ("java is OS");
    m.put(q1, ans1);

    String q2 = "what is variable in java";
    Set <String> ans2 = new HashSet();
    ans2.add ("variable is a program");
    ans2.add ("variable is operator");
    ans2.add ("variable is memory");
    ans2.add ("variable is contained");
    m.put (q2, ans2)

    String q3 = "Interface can be used";
    Set <String> ans3 = new LinkedHashSet();
    ans3.add ("as a blueprint of appln");
    ans3.add ("to remove tightly coupled code");
    ans3.add ("to achieve multiple inheritance");
    ans3.add ("all of the above");
    m.put (q3, ans3);

    String q4 = "usage of method";
    Set <String> ans4 = new LinkedHashSet();
    ans4.add ("to store multiple value");
    ans4.add ("to iterate statement");
    ans4.add ("to hide data");
    ans4.add ("none of these");
    m.put (q4, ans4);
```

```
int questionNo=0;
Set<String> set = m.keySet();
for(String s: set)
{
System.out.println(++questionNo+")"+s);
 int a=0;
 Set<String> ans = m.get(s);
 for(String s1:ans)
  S.o.p((char)(a++ +97)+")"+s1);
  S.o.p("-------------");
 }
}
```

eg.3

```
Map<Student, Map<String, List>> Student = HashMap
```

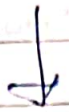
Student S1 = new Student();

```
Map<String, List> details = new HashMap();
List<String> listofImage = new ArrayList()
        listofCertificate =
        listofAchievement =
        listofReviews =
```

```
public static void main (String args[]) {
  Map<Student, Map<String, List>> Students = new
                    TreeMap(new StudentByIdAsc());
Student S1 = new Student(19,"Naitan","banglore",89.51);
Map<String, List> detailsOfS1 = new HashMap();
List<String> listofImagesOfS1 = new ArrayList();
listofImagesOfS1.add("img001.jpg");
                 ("img001.jpg");
                 ("img001.jpg");
                 ("img001.jpg");
detailsOfS1.put("listofImages", listofImagesOfS1);
List<String> listofCertificatesOfS1 = new ArrayList();
listofCertificatesOfS1.add("diploma in photography");
                       ("diploma in portrait photography");
                       ("certificate in designing & animation");
detailsOfS1.put("listofCertificates", listofCertificatesOfS1);
List<String> listofAchievementsOfS1 = new ArrayList();
listofAchievementsS1.add("youngest photographer");
                     ("1st rank in wildlife photography");
                     ("top magazine model photography");
detailsOfS1.put("listofAchievements", listofAchievementsOfS1);
List<String> listofReviewsOfS1 = new ArrayList();
listofReviewsOfS1.add("nice student");
listofReviewsOfS1.add("brilliant student");
detailsOfS1.put("listofReviews", listofReviewsOfS1);
 Students.put(S1.detailsOfS1);

Student S2 = new Student(12,"aman","delhi",79.11);
Map<String, List> detailsOfS1 = new HashMap();
```

```
detailsOfS2.put("listofReviews", listOfReviewsOfS2);
students.put(S2, detailsOfS2);
Set<Student> student = students.keySet();
for
```



```
. For (Student S: Student)
  {
   System.out.println(S);
   System.out.println("details");
   Map<String, List> details = student.get(S);
   System.out.print("images:");
   for (Object O: details.get("listofImages"))
       System.out.print(O+", ");
   System.out.println();
   System.out.print("certificates");
   for (Object O: details.get("listofcertificates"))
     System.out.print(O+", ");
   System.out.println();
   System.out.print("achievements:");
   for (Object O: details.get("listofAchievements"))
    System.out.print(O+", ");
   System.out.println();
   System.out.print("reviews:");
   for (Object O: details.get("listofReviews"))
    System.out.print(O+", ");
    System.out.println();
    System.out.println("---------");
  }
}
```