15/6/25

Map (I) :-

→ If we want to represent a group of objects as key-value pairs Then we should go for Map. Both key & value are objects.

→ Both key & values are objects.

→ Duplicate keys are not allowed, But values can be duplicated.

→ Each key-value pair is called Entry.

Ex:-

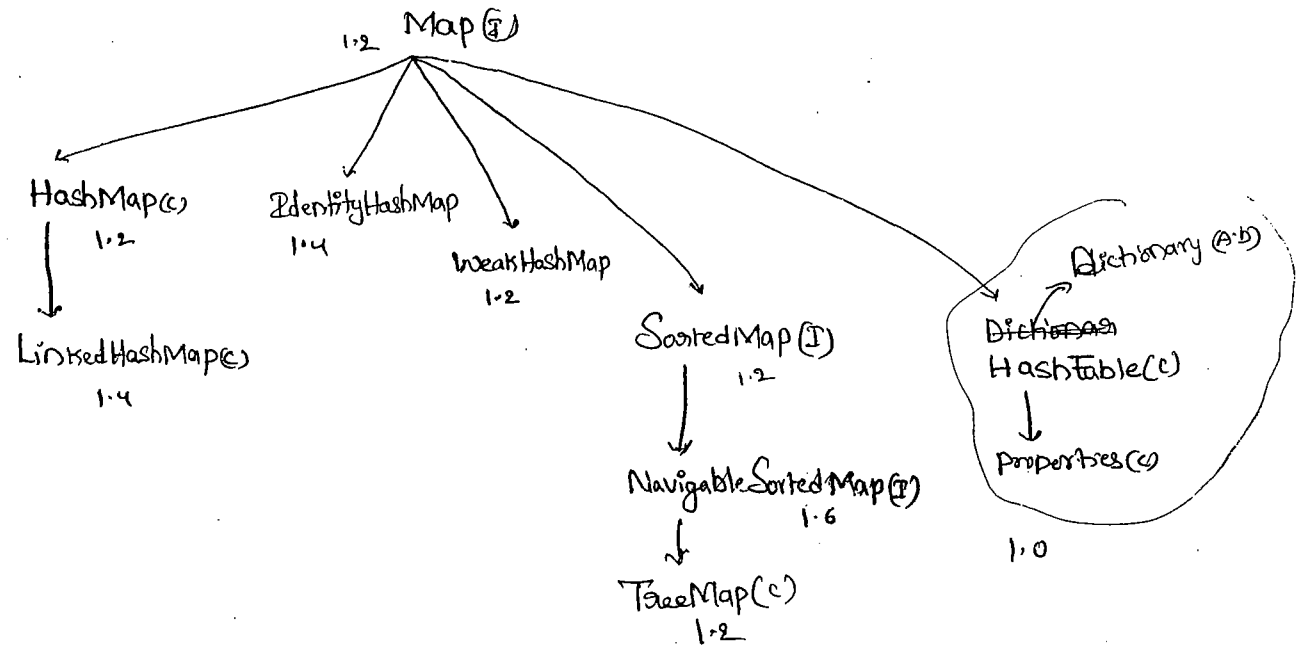
Rollno	Name
101	durga
102	Srinu
103	Ravi
104	Sambu
105	Sunder

key, value, entry

→ There is no relationship b/w Collection & Map.

→ *Collection meant for a group of individual objects where as *Map meant for a group of key-value pairs.

→ Map is not child interface of Collection.



Methods of Map Interface :-

*① Object put(Object key, Object value);

→ To add key-value pair to the map

→ If the specified key is already available then old value will be replaced with new value & old value will be returned.

② Void putAll(Map m)

→ To add a group of key-value pairs.

③ Object get(Object key)

→ Returns the value associated with specified key

→ If the key is not available then we will get Null

④ Object remove(Object key);

⑤ boolean containsKey(Object key)

⑥ boolean containsValue(Object value)

⑦ int size();

⑧ boolean isEmpty();

⑨ Void clear()

① Set keySet();
② Collection values();
③ Set entrySet();

} Collection views of the Map.

Entry (Interface) :

→ Each key-value pair is called One Entry

→ Without existing Map Object there is no chance of Entry object Hence, Interface Entry is define inside Map Interfaces.

Code :

```
interface Map
{
  interface Entry
  {
    ①, Object getKey();
    ②, Object getValue();
    ③, Object setValue();
  }
}
```

(1) HashMap : (C)

→ The underlying datastructure is HashTable

→ Heterogeneous objects are allowed for both keys & values.

→ duplicate keys are not allowed but the values can be duplicated.

→ Insertion order is not preserved because it is based on HashCode of keys.

→ null key is allowed (only once)

→ null values are allowed (any number of times).

* differences b/w HashMap & HashTable :-

HashMap	HashTable
① No method is Synchronized	① Every method is Synchronized
② multiple Threads Can Operates Simultaneously & Hence HashMap Object is not Thread Safe	② At a time only one Thread is allowed to Operate on HashTable Objects. Hence it is Thread Safe.
③ Threads are not required to wait & hence relatively performance is High.	③ It increases waiting time of the Thread & Hence performance is low.
④ null is allowed for both Key & value	④ null is not allowed for Both Key & values. otherwise we will get NPE
⑤ Introduced in 1.2 version & it is non-Legacy	⑤ Introduced, in 1.0 version & it is Legaly

Q) How to get Synchronized version of HashMap?

A) → Bydefault HashMap object is not Synchronized, but we can get Synchronized version by using SynchronizedMap() of Collections class.

```
Map M = Collections.SynchronizedMap(HashMap hm);
```

Constructors :-

(i) HashMap m = new HashMap( );

→ Creates an EmptyHashMap object with default initial capacity level is 16 & default fillRatio 0.75 (75%).

(ii) HashMap m = new HashMap(int initialCapacity)

(iii) HashMap m = new HashMap(int initialCapacity, float fillRatio)

(iv) HashMap m = new HashMap(Map m)

Ex :-
```
import java.util.*;
class HashMapDemo
{
 P.S.V.m(String[] args)
 {
   HashMap m = new HashMap();
    m.put("chiranjeevi", 700);
    m.put("balaiah", 800);
    m.put("venkatesh", 1000);
    m.put("nagarjuna", 500);
    S.o.pln(m); {venkatesh=1000, balaiah=800, chiranjeevi=700, nagarjuna=500}
    S.o.pln(m.put("chiranjeevi", 1000)); 700
    Set s = m.keySet();
    S.o.pln(s); [venkatesh, balaiah, chiranjeevi, nagarjuna]
    Collection c = m.values();
    S.o.pln(c); [1000, 800, 1000, 500].
     Set S1 = m.entrySet();
     Iterator its = S1.iterator();
```

```
While (its.hasNext())
{
  Map.Entry m1 = (Map.Entry) its.next();
  S.o.pln(m1.getKey() + "----" + m1.getValues());
  if(m1.getKey().equals("nagarjuna"))
    m1.setValue(10000);
}
S.o.pln(m);
}
}
```

nagarjuna 500
venkatesh 1000
balaiah 800
chiranjeevi 1000

→ {nagarjuna=10000, venkatesh=1000, balaiah=800, chiranjeevi=1000}

ii) LinkedHashMap :-

→ It is the child class of HashMap.

→ It is Exactly same as HashMap except the following differences

HashMap	LinkedHashMap
① The underlying D.S is HashTable	① The underlying D.S is HashTable + LinkedList
② Insertion Order is not preserved	② Insertion Order is preserved
③ Introduced in 1.2 version	③ Introduced in 1.4 Version

→ In the above program if we are replacing HashMap with LinkedHashMap, the following is the O/P.

{chiranjeevi=700, balaiah=800, venkatesh=1000, nagarjuna=500}

i.e insertion order is preserved

Note:-

→ The main application area of LinkedHashSet & LinkedHashMap is are cache applications implementation where duplication is not allowed & insertion order must be preserved.

(iii) IdentityHashMap :-

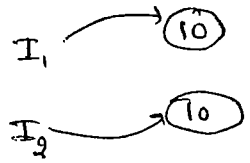
→ It is exactly same HashMap except the following difference.

→ In the case of HashMap to identify duplicate keys JVM always uses .equals(), which is mostly ment for content comparision.

→ If we want to use == operator instead of .equals() to identify duplicate keys we have to use IdentityHashMap. (== operator always ment for reference comparision).

Eg:-

```
HashMap m = new HashMap();
Integer i1 = new Integer(10);
Integer i2 = new Integer(10);
m.put(i1, "pavan");
m.put(i2, "kalyan");
S.o.pln(m); {10=kalyan}
```



.equals() → Content
== → reference

I1 == I2 → false
I1.equals(I2) → True

→ In the above code i_1 & i_2 are duplicate keys because i_1.equals(i_2) returns true.

→ If we replace HashMap with IdentityHashMap then the o/p is {10=pavan, 10=kalyan}

→ i_1 & i_2 are not duplicate keys because $i_1 == i_2$ returns false.

WeakHashMap :-

→ It is exactly same as HashMap except the following difference.

→ In the case of HashMap, Object is not eligible for g.c eventhough it doesn't have any external references if it is associated with HashMap. i.e, HashMap dominates Garbage Collector (g.c).

→ But in the case of WeakHashMap Eventhough object associated with WeakHashMap, it is eligible for g.c, if it doesnot have any external references. i.e G.c dominates WeakHashMap.

Eg:-

```
import java.util.*;
class WeakHashMapDemo
{
  P. S. V. m (String[] args) throws InterruptedException
  {
    HashMap m = new HashMap();
    Temp t = new Temp();
    m.put(t, "durga");
    S.o.pln(m);        {temp = durga}
    t = null;
    System.gc();
    Thread.sleep(5000);
    S.o.pln(m);
  }                    {temp = durga}
}
```

```
Class Temp
{
  public String toString()
  {
    return "temp";
  }
  public void finalize()
  {
    System.out.println("finalize method called");
  }
}
```

O/P: {temp = durga}
{temp = durga}

→ If we replace HashMap with WeakHashMap then the o/p is

{temp = durga}
finalize method called
{}

(ii) SortedMap (I) :-

→ If we want to represent a group of entries according to some sorting order then we should go for SortedMap. The sorting should be done based on the keys but not based on the values.

→ SortedMap interface is the child interface of Map.

→ SortedMap interface defines the following 6 specific methods

1. Object firstKey();
2. Object lastKey();
3. SortedMap headMap(Object key1);
4. SortedMap tailMap(Object key1);
5. SortedMap subMap(Object key1, Object key2)
6. Comparator comparator();

(iii) TreeMap (C) :-

→ The underly D.S is RED-BLACK Tree,

→ Insertion order is not preserved & all entries are inserted according to some sorting order of keys.

→ If we are depending on default natural sorting order then the keys should be Homogeneous & Comparable. otherwise we will get ClassCastException (CCE).

→ If we are defining our own sorting order by Comparator then

The Keys need not be Homogeneous & Comparable.

→ There are no restrictions on values, they can be Heterogeneous & non-Comparable.

→ duplicate keys are not allowed but values can be duplicated.

Null acceptance:-

→ for the Empty TreeMap as the first Entry with Null key is allowed. but after inserting that Entry if we are trying to insert any other Entry we will get NullPointerException (NPE).

→ for the NON-Empty TreeMap if we are trying to insert Entry with null key we will get NullPointerException (NPE)

→ There are no restrictions on null values. i.e, we can use null any no. of times any where for Map values.

Constructors:-

(i) TreeMap t = new TreeMap()

for default natural Sorting order.

(ii) TreeMap t = new TreeMap(Comparator c)

for Customized Sorting order.

(iii) TreeMap t = new TreeMap(Map m)

(iv) TreeMap t = new TreeMap(SortedMap m)

Eg:-

```
import java.util.*;
class TreeMapDemo3
{
  p.s.v.m(String[] args)
  {
    TreeMap m = new TreeMap();
    m.put(100, "zzz");
    m.put(103, "yyy");
    m.put(101, "xxx");
    m.put(104, 106);
    m.put(107, null);
    //m.put("FFFF", "xxx"); //CCE
    // m.put(null, "xxx"); //NPE
    S.o.p(m);
  }
}
```

{100=zzz, 101=xxx, 103=yyy, 104=106, 107=null}

O/P:-

{100=zzz, 101=xxx, 103=yyy, 104=106, 107=null}

Eg:-

```
import java.util.*;

Class TreeMapDemo
{
  P.S.V.m(String[] args)
  {
    TreeMap t = new TreeMap(new MyComparator());
    t.put("xxx", 10);
    t.put("AAA",20);
    t.put("zzz", 30);
    t.put("LLL", 40);
    S.o.pln(t);
  }
}

Class MyComparator implements Comparator
{
  public int compare(Object obj1, Object obj2)
  {
    String s1 = obj1.toString();
    String s2 = obj2.toString();
    return s2.compareTo(s1);
  }
}
```

o/p:- {zzz = 30, xxx = 10, LLL=40, AAA =20}

Hashtable(c):

→ The Underlying datastructure is HashTable.

→ Heterogeneous objects are allowed for both keys & values

→ Insertion order is not preserved & it is based on HashCode of the keys.

→ null is not allowed for both key & values otherwise we will get NullPointerException (NPE).

→ duplicate keys are not allowed, but values can be duplicated.

→ All methods are Synchronized & Hence HashTable object is ThreadSafe.

Constructor:

(i) Hashtable h = new Hashtable()

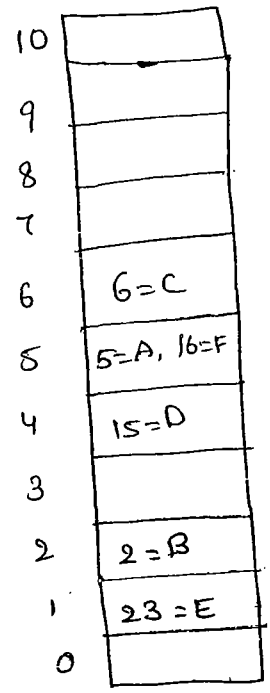
→ Creates an Empty Hashtable object with default initial capacity is "11" & default fillratio 75% (0.75).

(ii) Hashtable h = new Hashtable (int initialCapacity)

(iii) Hashtable h = new Hashtable (int initialCapacity, float fillratio)

(iv) Hashtable h = new Hashtable (Map m);

eg:-
```
import java.util.*;
Class HashtableDemo
{
 P.S.V.m(String[] args)
 {
  Hashtable h = new Hashtable();
  h.put(new Temp(5), "A");
  h.put(new Temp(2), "B");
  h.put(new Temp(6), "C");
  h.put(new Temp(15), "D");
  h.put(new Temp(23), "E");
  h.put(new Temp(16), "F");
  //h.put("durga", null); //NPE
  System.out.println(h);
 }
}
Class Temp
{
 int i;
 Temp(int i)
 {
  this.i = i;
 }
 public int hashCode()
 {
  return i;
 }
 Public String toString()
 {
   return i+" ";
 }
}
```



{6=C, 16=F, 5=A, 15=D, 2=B, 23=E}

→ from top to bottom & Right to Left

(iv) Properties (C):-

→ It is the child class of Hashtable

→ In our program if any thing which changes frequently (like database usernames, passwords, URL) never recommended to hardCode the value in the java program. Because for Every change, we have recompile, rebuild, redeploye the application & Sometime even Sever restart also required. which Creates a big business impact to the client.

→ we have to Configuare those variables (properties inside properties files & we have to read those values from javaCode.

→ the main advantage of this approach is, If any change in the properties file Just redeployement is enough which is not a business impact to the client.

Constructor:-

(i) Properties p = new Properties();

→ In the Case of Properties both key & value should be String type

Methods:-

(i) String getProperty(String PropertyName)

→ returns the value associated with Specified property.

(ii) String setProperty(String pname, String pvalue);

→ to Set a new property.

(iii) Enumeration propertyNames();

* (iv) Void load(InputStream is)

→ To load the properties from properties files into java properties object.

(v) Void store(OutputStream os, String Comment)

→ To update properties from properties object into propeties file.

Eg:-

```
import java.util.*;
import java.io.*;
class PropertiesDemo
{
 P.S.V.m(String[] args) throws IOException
 {
  Properties P = new Properties();
  FileInputStream fis = new FileInputStream("abc.properties");
  P.load(fis);
  System.out.println(P);
  String s = P.getProperty("venki");
  S.o.pln(s);
  P.setProperty("nag", "999999");
  FileOutputStream fos = new FileOutputStream("abc.properties");
  P.store(fos, "Updated by durga for SCJP Demo Class");
 }
}
```

user = scott
venki = 8888
pwd = tiger

abc.properties