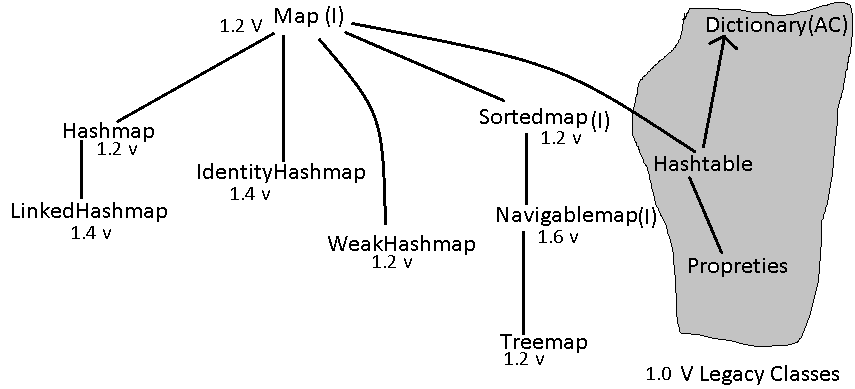
**Map(I)**



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.

|  |  |
| --- | --- |
| **Key** | **Value** |
| 101 | Durga |
| 102 | Sonu |
| 103 | Chiku |

Both keys and values are objects only. Duplicate keys are not allowed but values can be duplicated. Each key-value pair is called Entry, hence Map is considered as a collection of entry objects.

**Map interface Methods:**

1. Object put(Object key, Object value)

To add one key-value pair to the Map.

If the key is already present then old value will be replaced with new value and returns old value.

Ex:

m.put(101,”Durga”); // returns null

m.put(102,”Shiva”); // returns null

m.put(101,”Ravi”); //returns Durga

1. Void putAll(Map m)
2. Object get(Object key)

Returns the value associated with specified key.

1. Object remove(Object key)

Removes the entry associated with specified key.

1. boolean containsKey(Object key)
2. boolean containsValue(Object value)
3. boolean isEmpty()
4. int size()
5. void clear()

Other Methods

1. Set keySet()
2. Collection Values()
3. Set entrySet()

These 3 methods are called as collection views of Map.

**Entry (I):**

A map is a group of key value pairs and each key value pair is called as entry. Hence Map is considered as a collection of Entry objects.

Without existing Map object there is no chance of existing entry object hence entry interface is defined inside Map interface.

interface Map{

.

.

.

interface Entry{

Object getKey() //These 3 methods are called

Object getValue() //Entry specific method and we

Object setValue() //can apply only on Entry objects

}

}

**HashMap** :

* The underlaying data structure is Hashtable.
* Insertion order not preserved and it is based on hash code of keys.
* Duplicate keys are not allowed but values can be duplicated.
* Heterogenous objects are allowed for both keys and values.
* Null is allowed for key(only once), null is allowed for value(any number of times).
* HashMap implements serializable and cloneable interface but not RandomAccess.
* HashMap is the best choice if our frequent operation is searching.

Constructors:

1. HashMap m = new HashMap();

Creates an empty HashMap object with default initial capacity 16 an default fill ratio 0.75.

1. HashMap m = new HashMap(int initialcapacity);

Creates an empty HashMap object with specified initial capacity and default fill ratio 0.75.

1. HashMap m = new HashMap(int initialCapacity, float fillratio);

Creates an empty HashMap object with specified initial capacity and specified fill ratio.

1. HashMap m = new HashMap(Map m);

Ex:

package com.company;  
import java.util.\*;  
public class MapDemo {  
 public static void main(String[] args) {  
 HashMap m = new HashMap();  
 m.put("chiranjeevi",700);  
 m.put("balaiah",800);  
 m.put("Venkatesh",200);  
 m.put("nagarjuna",500);  
 System.*out*.println(m); //{K=V, K=V,...}  
 System.*out*.println(m.put("chiranjeevi",1000));  
 Set s = m.keySet();  
 System.*out*.println(s); //[K, K, K,....]  
 Collection c = m.values();  
 System.*out*.println(c);  
 Set s1 = m.entrySet();  
 System.*out*.println(s1); //[K=V, K=V,...]  
 Iterator itr = s1.iterator();  
 while(itr.hasNext()){  
 Map.Entry m1 = (Map.Entry)itr.next();  
 System.*out*.println(m1.getKey() +" "+m1.getValue());  
 if(m1.getKey().equals("nagarjuna"))  
 m1.setValue(10000);  
 }  
 System.*out*.println(m);  
 }  
}

O/P:

{balaiah=800, chiranjeevi=700, nagarjuna=500, Venkatesh=200}

700

[balaiah, chiranjeevi, nagarjuna, Venkatesh]

[800, 1000, 500, 200]

[balaiah=800, chiranjeevi=1000, nagarjuna=500, Venkatesh=200]

balaiah 800

chiranjeevi 1000

nagarjuna 500

Venkatesh 200

{balaiah=800, chiranjeevi=1000, nagarjuna=10000, Venkatesh=200}

**Difference between HashMap and Hashtable.**

|  |  |
| --- | --- |
| HashMap | Hashtable |
| 1. Every method present in HashMap is not synchronized. 2. At a time multiple threads are allowed to operate on HashMap object and hence it is not thread safe. 3. Relatively performance is high because threads are not required to wait to operate on HashMap objects. 4. null is allowed for both keys and values. 5. Introduced in 1.2 v and it is not legacy. | 1. Every method present in Hashtable is synchronized. 2. At a time only one thread is allowed to operate on Hashtable and hence it is thread safe. 3. Relatively performance is low because threads are required to wait to operate on Hashtable object. 4. null is not allowed for keys and values otherwise we will get NullPointerException. 5. Introduced in 1.0 v and it is legacy. |

Q. How to get synchronized version of HashMap objects?

By default HashMap is non-synchronized but we can get synchronized version of HashMap by using synchronizedMap() of collections class.

HashMap m = new HashMap();

Map m1 = Collections.synchronizedMap(m);

// m = non-synchronized

//m1 = synchronized

**LinkedHashMap:**

It is the child class of HashMap.

It is exactly same as HashMap (including methods and constructors) except the following difference.

|  |  |
| --- | --- |
| HashMap | LinkedHashMap |
| 1. The underlaying data structure is Hashtable. 2. Insertion order is not preserved and it is based on keys. 3. Introduced in 1.2 version. | 1. Underlaying data structure is a combination of LinkedList and Hashtable(Hybrid data structure). 2. Insertion order is preserved. 3. Introduced in 1.4 version |

In the above HashMap program if we replace HashMap with LinkedHashMap then the output will come in inserted order.

**Difference between == operator and equals() :**

In general == operator meant for reference comparison (address comparison) where as equals() meant for content comparison.

Ex:

Integer I1 = new Integer(10);  
Integer I2 = new Integer(10);  
System.*out*.println(I1 == I2);  
System.*out*.println(I1.equals(I2));

O/P:-

false

true

**IdentityHashMap :**

It is exactly same as including methods and constructors except the following difference.

In the case of normal HashMap JVM will use .equals() method to identify duplicate keys, which is meant for content comparison.

But in the case of IdentityHashMap JVM will use == operator to identify duplicate keys. Which is meant for reference comparison(address comparison).

Ex:

HashMap m = new HashMap();  
 Integer I1 = new Integer(10);  
 Integer I2 = new Integer(10);  
 m.put(I1,"Pawan");  
 m.put(I2,"Kalyan");  
 System.*out*.println(m);

O/P:

{10=Kalyan}

Explanation :

I1 and I2 are duplicate keys because I1.equals(I2) returns true.

If we replace HashMap with IdentityHashMap then I1 and I2 are not duplicate keys because I1==I2 returns false.

In this case output is : {10=Pawan, 10=Kalyan}

**WeakHashMap :**

It is exactly same as HashMap except following difference.

In case of HashMap even though object doesn’t have any reference it is not eligible for GC if it is associated with HashMap i.e. HashMap dominates Garbage Collector.

But in case of Weak HashMap if object doesn’t contain any reference which is eligible for GC even though object associated with WeakHashMap i.e. Garbage Collector dominates WeakHashMap.

Ex:

package com.company;  
import java.util.\*;  
public class Main {  
 public static void main(String[] args) throws InterruptedException {  
 HashMap m = new HashMap();  
 Temp t = new Temp();  
 m.put(t,"durga");  
 System.*out*.println(m);  
 t = null;  
 System.*gc*();  
 Thread.*sleep*(5000);  
 System.*out*.println(m);  
 }  
}  
class Temp{  
 public String toString(){  
 return "temp";  
 }  
 public void finalize(){  
 System.*out*.println("Finalize Method Called");  
 }  
}

In the above example temp object not applicable for GC because it is associated with HashMap.

In this case output: {temp=durga}

{temp=durga}

In the above program if we replace HashMap with WeakHashMap then temp object eligible for GC. In this output is {temp=durga}

Finalize Method Called

{}

**SortedMap :**

It is the child interface of Map. If we want to represent a group of key value pairs according to some sorting order of keys then we should go for SortedMap.

Sorting is based on keys but not based on value.

SortedMap defines the following specific methods.

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

Ex:

SortedMap -> {101=’A’, 103=’B’, 104=’C’, 107=’D’, 125=’E’, 136=’F’}

firstKey() // 101

lastKey() //136

headMap(107) //{101=’A’, 103=’B’, 104=’C’}

tailMap(107) // {107=’D’, 125=’E’, 136=’F’}

submap(103,125) //{103=’B’, 104=’C’, 107=’D’}

comparator() //null

**TreeMap :**

The underlaying data structure is Red Black tree.

Insertion order is not preserved and it is based on some sorting order of keys.

Duplicate keys are not allowed but values can be duplicated.

If we are depending on default natural sorting order then keys should be homogenous and comparable otherwise we will get runtime exception saying ClassCasteException.

If we are defining our own sorting by comparator then keys need not be homogenous and comparable we can take heterogenous noncomparable objects also.

Whether we are depending on default natural sorting order or customized sorting order there are no restrictions for value. We can take heterogenous and non comparable objects also.

**null acceptance:**

* For non empty TreeMap if we are trying to insert an entry with null key then we will get runtime exception saying NullPointerException.
* For 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 then we will get runtime exception saying NullPointerException.

Note: The above null acceptance rule applicable until 1.6 version only. From 1.7 version onwards null is not allowed for key.

But for values we can use null any number of times there is no restrictions whether it is 1.6 or 1.7 versions.

**Constructors:**

1. TreeMap t = new TreeMap();

For default natural sorting order.

1. TreeMap t = new TreeMap(Comparator c);

For customized sorting order.

1. TreeMap t = new TreeMap(Map m);
2. TreeMap t = new TreeMap(SortedMap m);

Ex: Demo program for default natural sorting order.

package com.company;  
import java.util.\*;  
public class Main {  
 public static void main(String[] args) {  
 TreeMap m = new TreeMap();  
 m.put(100,"xxx");  
 m.put(103,"zzz");  
 m.put(101,"yyy");  
 m.put(104,106);  
 System.*out*.println(m);  
 }  
}

O/P:

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

Ex 2 : Demo program for customized sorting order.

package com.company;  
import java.util.\*;  
public class Main {  
 public static void main(String[] args) {  
 TreeMap m = new TreeMap(new MyComparator());  
 m.put("XXX",10);  
 m.put("AAA",20);  
 m.put("ZZZ",30);  
 m.put("LLL",40);  
 System.*out*.println(m);  
 }  
}  
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}