**Collection Framework – Part\_12**

* **LinkedHashMap:**

1. It is the child class of HashMap.
2. It is exactly same as HashMap including methods and constructors except the following differences.

|  |  |  |
| --- | --- | --- |
| S.No | HashMap | LinkedHashMap |
| 1 | The underlying data structure is Hashtable. | Underlying data structure is a combination of LinkedList and HashTable. (Hybrid data structure). |
| 2 | Insertion order is not preserved and it is based on hashCode() of keys. | Insertion order is preserved. |
| 3 | Introduced in 1.2 version. | Introduced in 1.4 version. |

In the above HashMap program if we replace HashMap with LinkedHashMap then output is

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

That is insertion order is preserved.

LinkedHashSet and LinkedHashMap are commonly used for developing cache based applications.

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

In general, == operator meant for reference comparison (address comparison) whereas .equals() method meant for content comparison.

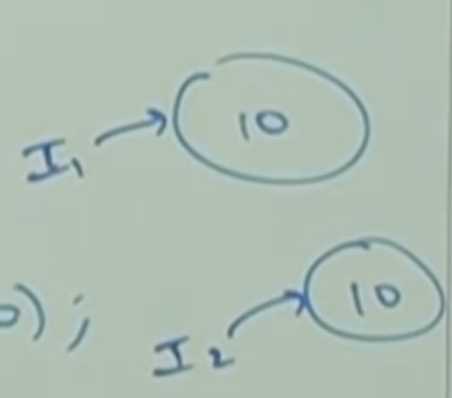
Example:

Integer i1 = new Integer(10);

Integer i2 = new Integer(10);

System.out.println(i1 == i2); //false

System.out.println(i1.equals(i2)); //true



* **IdentityHashMap:**

1. It is exactly same as HashMap 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).

Example:

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); {10=kalyan}

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 the following difference.

In the case of HashMap, even though object doesn’t have any reference it is not eligible for GC, if it is associated with HashMap. That is, HashMap dominates GC.

But in the case of WeakHashMap, if object doesn’t contain any references it is eligible for GC even though object associated with WeakHashMap. That is, garbage collector dominates WeakHashMap

* **Example:**

import java.util.\*;

class WeakHashMapDemo{

public static void main(String[] args) throws Exception{

HashMap m = new HashMap();

Temp t = new Temp();

m.put(t, “durga”);

t=null;

System.gc();

Thread.sleep(5000);

System.out.println(m);

}

}

class Temp{

public String toString(){

return “temp”;

}

public void finalize(){

System.out.println(“Finalized method called”);

}

}

In the above example temp object not eligible for GC because it is associated with HashMap. In this case output is

{temp=durga}

{temp=durga}

In the above program if we replace HashMap with WeakHashMap then temp object eligible for GC. In this case output is:

{tem=durga}

Finalize method called

{}