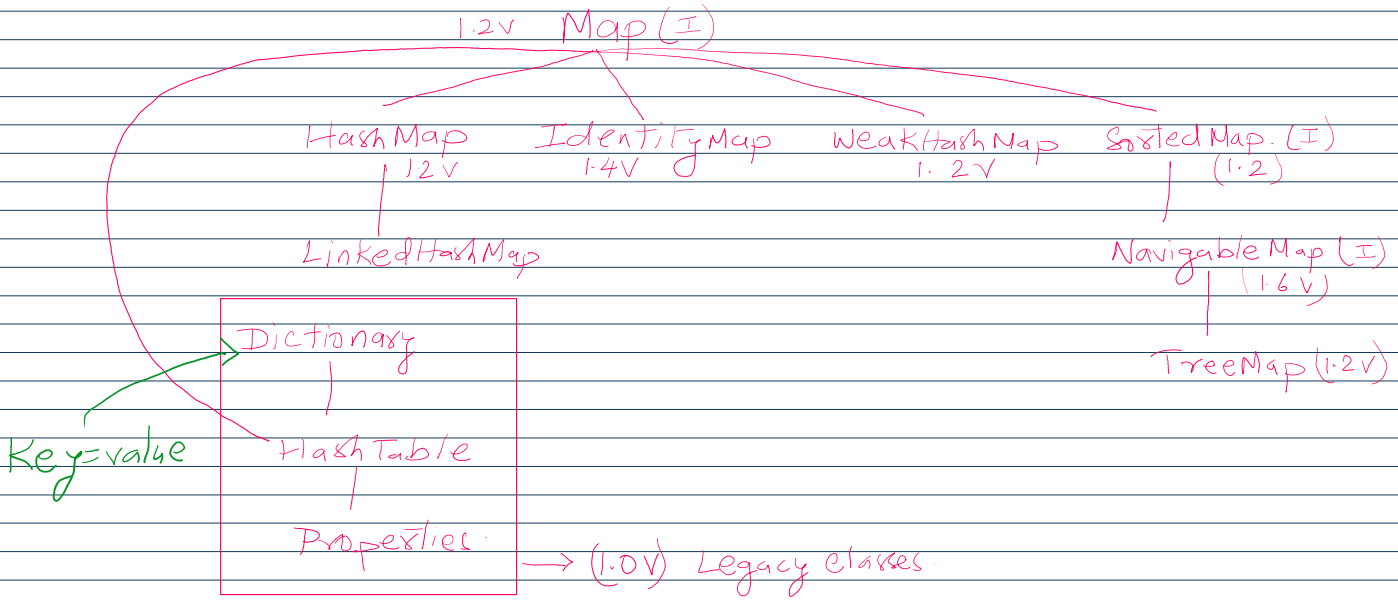
Map interface

21 May 2025 16:57



Map is not child interface of Collection.

If we want to represent a group of objects as key-value pair. Then we should go for map.

Both key 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.

Immutable Object

Key	Value
101	Akash
102	Naman
103	Aarav
104	Jiya
105	Aadya
106	Pawan

Entry

Map interface methods.

Object put (Object key, Object value)

old value

```
null ← m.put(101, "Arjun");
null ← m.put(102, "Aman");
null ← m.put(103, "Arjun");
null ← m.put(103, "Keshav");
Arjun
```

K	V
101	Arjun
102	Aman
103	~~Arjun~~ Keshav

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 the old value.

Object putAll(Map m)

Object get(Object key) → returns the value associated with specified key.

Object remove(Object key) → returns the entry associated with specified key.

boolean containsKey(Object key)

boolean containsValue(Object value)

boolean isEmpty()

int size()

void clear()

Set keySet()
Collection values()
Set entrySet()
} Collection views of Map.

Entry (I) // inner interface

```
interface Map {
    interface Entry {
        Object get();
        Object getValue();
        Object setValue(Object newObject)
    }
}
```

} Entry specific method. We can apply only on Entry objects.

A map is group of key-value pair and each key-value pair is called an Entry interface. Hence Map is considerd as a collection of Entry objects. Without existing Map object there is no chance of existing Entry objects.

public interface Map<K,V>

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

This interface takes the place of the Dictionary class, which was a totally abstract class rather than an interface.

The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The order of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the TreeMap class, make specific guarantees as to their order; others, like the HashMap class, do not.

Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is a key in the map. A special case of this prohibition is that it is not permissible for a map to contain itself as a key. While it is permissible for a map to contain itself as a value, extreme caution is advised: the equals and hashCode methods are no longer well defined on such a map.

All general-purpose map implementation classes should provide two "standard" constructors: a void (no arguments) constructor which creates an empty map, and a constructor with a single argument of type Map, which creates a new map with the same key-value mappings as its argument. In effect, the latter constructor allows the user to copy any map, producing an equivalent map of the desired class. There is no way to enforce this recommendation (as interfaces cannot contain constructors) but all of the general-purpose map implementations in the JDK comply.

The "destructive" methods contained in this interface, that is, the methods that modify the map on which they operate, are specified to throw UnsupportedOperationException if this map does not support the operation. If this is the case, these methods may, but are not required to, throw an UnsupportedOperationException if the invocation would have no effect on the map. For example, invoking the putAll(Map) method on an unmodifiable map may, but is not required to, throw the exception if the map whose mappings are to be "superimposed" is empty.

Some map implementations have restrictions on the keys and values they may contain. For example, some implementations prohibit null keys and values, and some have restrictions on the types of their keys. Attempting to insert an ineligible key or value throws an unchecked exception, typically NullPointerException or ClassCastException. Attempting to query the presence of an ineligible key or value may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible key or value whose completion would not result in the insertion of an ineligible element into the map may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

Many methods in Collections Framework interfaces are defined in terms of the equals method. For example, the specification for the containsKey(Object key) method says: "returns true if and only if this map contains a mapping for a key k such that (key==null ? k==null : key.equals(k))." This specification should not be construed to imply that invoking Map.containsKey with a non-null argument key will cause key.equals(k) to be invoked for any key k. Implementations are free to implement optimizations whereby the equals invocation is avoided, for example, by first comparing the hash codes of the two keys. (The Object.hashCode() specification guarantees that two objects with unequal hash codes cannot be equal.) More generally, implementations of the various Collections Framework interfaces are free to take advantage of the specified behavior of underlying Object methods wherever the implementor deems it appropriate.

Some map operations which perform recursive traversal of the map may fail with an exception for self-referential instances where the map directly or indirectly contains itself. This includes the clone(), equals(), hashCode() and toString() methods. Implementations may optionally handle the self-referential scenario, however most current implementations do not do so.

This interface is a member of the Java Collections Framework.

Since:
1.2

public static interface Map.Entry<K,V>

A map entry (key-value pair). The Map.entrySet method returns a collection-view of the map, whose elements are of this class. The only way to obtain a reference to a map entry is from the iterator of this collection-view. These Map.Entry objects are valid only for the duration of the iteration; more formally, the behavior of a map entry is undefined if the backing map has been modified after the entry was returned by the iterator, except through the setValue operation on the map entry.

Since:
1.2

https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/util/Map.html

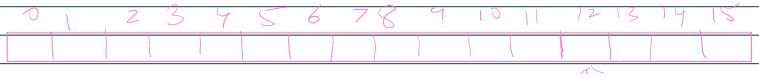
Link to know more about Map interface.

HASHMAP

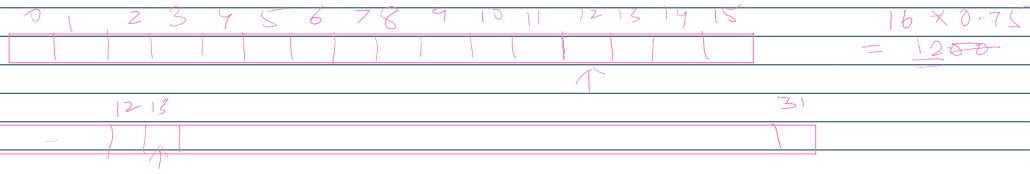
The underlying data structure is hash table.
Insertion order is not preserved and it is based on hash code of key.
Duplicate keys are not allowed, but values can be duplicated.
Heterogeneous objects are allowed for both key and value.
Null is allowed for key (only once), 'null' is allowed for values(any number of times)
HashMap implements Map<k><v>, Serilizable, Cloneable interface but not RandomAcess interface.
HashMap is the best choice if our frequent operation is search operation.

Constructors

HashMap m = new HashMap(); => Creates an empty hash map object with default initial capacity 16 and default. fill ratio 0.75.



fill ratio 0.75.



HashMap m = new HashMap(int initialCapacity); => Creates empty hash map object with specified initial capacity and default filled ratio is 0.75

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

HashMap m = new HashMap(Map m);

Official Documentation:

public class HashMap<K,V>
extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable
Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

This implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

An instance of HashMap has two parameters that affect its performance: initial capacity and load factor. The capacity is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the HashMap class, including get and put). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.

If many mappings are to be stored in a HashMap instance, creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table. Note that using many keys with the same hashCode() is a sure way to slow down performance of any hash table. To ameliorate impact, when keys are Comparable, this class may use comparison order among keys to help break ties.

Note that this implementation is not synchronized. If multiple threads access a hash map concurrently, and at least one of the threads modifies the map structurally, it must be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the Collections.synchronizedMap method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

Map m = Collections.synchronizedMap(new HashMap(...));

The iterators returned by all of this class's "collection view methods" are fail-fast: if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.

This class is a member of the Java Collections Framework.

```
Since:
    1.2


/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package com.mycompany.collectionsconceptwithkeshav.MyListObjects;

import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

/**
 *
 * @author bluej
 */
public class HashMapDemo {
    public static void main(String[] args) {
        HashMap m = new HashMap();
        m.put(100, "Harsh");
        m.put(80, "Sanjay");
        m.put(120, "Ajay");
        m.put(200, "Jiya");
        m.put(130, "Aadya");
        System.out.println(m);
        System.out.println(m.put(200, "Tanisha"));
        System.out.println(m);
        Set s = m.keySet();
        System.out.println(s);
        Collection c = m.values();
        System.out.println(c);
        Set es = m.entrySet();
        System.out.println(es);

        Iterator itr = es.iterator();
        while(itr.hasNext()){
            Map.Entry m1 = (Map.Entry)itr.next();
            System.out.println(m1.getKey()+"==>"+m1.getValue());
            Integer t = (Integer)m1.getKey();
            if(t.equals(200)){
                m1.setValue("Gargi");
            }
        }
        System.out.println(m);
    }
}
```

Output:

HashMap

```
--- exec:3.1.0:exec (default-cli) @ CollectionsConceptWIthKeshav ---
{80=Sanjay, 130=Aadya, 100=Harsh, 120=Ajay, 200=Jiya}
Jiya
{80=Sanjay, 130=Aadya, 100=Harsh, 120=Ajay, 200=Tanisha}
[80, 130, 100, 120, 200]
[Sanjay, Aadya, Harsh, Ajay, Tanisha]
[80=Sanjay, 130=Aadya, 100=Harsh, 120=Ajay, 200=Tanisha]
80==>Sanjay
130==>Aadya
100==>Harsh
120==>Ajay
200==>Tanisha
{80=Sanjay, 130=Aadya, 100=Harsh, 120=Ajay, 200=Gargi}
------------------------------------------------------------------------
```

Insertion order is not preserved

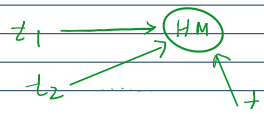
LinkedHashMap

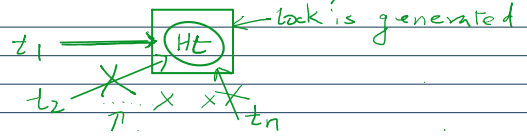
```
--- exec:3.1.0:exec (default-cli) @ CollectionsConceptWIthKeshav ---
{100=Harsh, 80=Sanjay, 120=Ajay, 200=Jiya, 130=Aadya}
Jiya
{100=Harsh, 80=Sanjay, 120=Ajay, 200=Tanisha, 130=Aadya}
[100, 80, 120, 200, 130]
[Harsh, Sanjay, Ajay, Tanisha, Aadya]
[100=Harsh, 80=Sanjay, 120=Ajay, 200=Tanisha, 130=Aadya]
100==>Harsh
80==>Sanjay
120==>Ajay
200==>Tanisha
130==>Aadya
{100=Harsh, 80=Sanjay, 120=Ajay, 200=Gargi, 130=Aadya}
------------------------------------------------------------------------
```

Insertion order is preserved

Difference between HashMap & Hashtable

HashMap	Hashtable
1. Not synchronized: Every method in HashMap is not synchronized.	1. Synchronized: Every method present in Hashtable is synchronized.



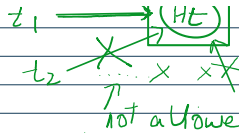


At a time multiple threads are allowed to operate on HashMap object and hence it is not Thread safe.

2. Relatively performance is high because threads are not required to wait to operate on HashMap objects.

3. 'null' is allowed for Key (only once). 'null' is allowed for the value (multiple times).

4. It is not legacy, it is introduced in 1.2V



not allowed to access, because thread 't1' has acquired the lock, the other threads has to wait, till t1 thread releases the lock.

At a time only one thread is allowed to operate on Hashtable object and hence it is Thread safe.

2. Relatively performance is low because threads are required to wait to operate on Hashtable object.

3. 'null' such type of concept is not available. if given, then NullPointerException will occur.

4. It is a legacy, introduced in 1.0v.

LinkedHashMap

HashMap → LinkedHashMap [Hashtable + LinkedList]

HashSet → LinkedHashSet [LL + HT]

- LinkedHashMap is a child class of HashMap.
- It is exactly same as HashMap (including methods and constructor except the following differences.)

HashMap	LinkedHashMap
The underlying data structure is Hashtable.	Underlying data structure is a combination of LinkedList and Hashtable. (hybrid data structure)
Insertion order is not preserved and it is based on hash code of keys.	Insertion order is preserved.
Introduced in 1.2 version.	Introduced in 1.4 version.

Official Documentation:

```
public class LinkedHashMap<K,V>
extends HashMap<K,V>
implements Map<K,V>
```

Hash table and linked list implementation of the Map interface, with predictable iteration order. This implementation differs from HashMap in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion-order). Note that insertion order is not affected if a key is re-inserted into the map. (A key k is reinserted into a map m if m.put(k, v) is invoked when m.containsKey(k) would return true immediately prior to the invocation.)

This implementation spares its clients from the unspecified, generally chaotic ordering provided by HashMap (and Hashtable), without incurring the increased cost associated with TreeMap. It can be used to produce a copy of a map that has the same order as the original, regardless of the original map's implementation:

```
void foo(Map m) {
    Map copy = new LinkedHashMap(m);
    ...
}
```

This technique is particularly useful if a module takes a map on input, copies it, and later returns results whose order is determined by that of the copy. (Clients generally appreciate having things returned in the same order they were presented.)

A special constructor is provided to create a linked hash map whose order of iteration is the order in which its entries were last accessed, from least-recently accessed to most-recently (access-order). This kind of map is well-suited to building LRU caches. Invoking the put, putIfAbsent, get, getOrDefault, compute, computeIfAbsent, computeIfPresent, or merge methods results in an access to the corresponding entry (assuming it exists after the invocation completes). The replace methods only result in an access of the entry if the value is replaced. The putAll method generates one entry access for each mapping in the specified map, in the order that key-value mappings are provided by the specified map's entry set iterator. No other methods generate entry accesses. In particular, operations on collection-views do not affect the order of iteration of the backing map.

The removeEldestEntry(Map.Entry) method may be overridden to impose a policy for removing stale mappings automatically when new mappings are added to the map.

This class provides all of the optional Map operations, and permits null elements. Like HashMap, it provides constant-time performance for the basic operations (add, contains and remove), assuming the hash function disperses elements properly among the buckets. Performance is likely to be just slightly below that of HashMap, due to the added expense of maintaining the linked list, with one exception: Iteration over the collection-views of a LinkedHashMap requires time proportional to the size of the map, regardless of its capacity. Iteration over a HashMap is likely to be more expensive, requiring time proportional to its capacity.

A linked hash map has two parameters that affect its performance: initial capacity and load factor. They are defined precisely as for HashMap. Note, however, that the penalty for choosing an excessively high value for initial capacity is less severe for this class than for HashMap, as iteration times for this class are unaffected by capacity.

Note that this implementation is not synchronized. If multiple threads access a linked hash map concurrently, and at least one of the threads modifies the map structurally, it must be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the Collections.synchronizedMap method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

```
   Map m = Collections.synchronizedMap(new LinkedHashMap(...));
```

A structural modification is any operation that adds or deletes one or more mappings or, in the case of access-ordered linked hash maps, affects iteration order. In insertion-ordered linked hash maps, merely changing the value associated with a key that is already contained in the map is not a structural modification. In access-ordered linked hash maps, merely querying the map with get is a structural modification.)

The iterators returned by the iterator method of the collections returned by all of this class's collection view methods are fail-fast: if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.

The spliterators returned by the spliterator method of the collections returned by all of this class's collection view methods are late-binding, fail-fast, and additionally report Spliterator.ORDERED.

This class is a member of the Java Collections Framework.

Implementation Note:
    The spliterators returned by the spliterator method of the collections returned by all of this class's collection view methods are created from the iterators of the corresponding collections.
Since:
    1.4

Constructors

Constructor	Description
`LinkedHashMap()`	Constructs an empty insertion-ordered LinkedHashMap instance with the default initial capacity (16) and load factor (0.75).
`LinkedHashMap(int initialCapacity)`	Constructs an empty insertion-ordered LinkedHashMap instance with the specified initial capacity and a default load factor (0.75).
`LinkedHashMap(int initialCapacity, float loadFactor)`	Constructs an empty insertion-ordered LinkedHashMap instance with the specified initial capacity and load factor.
`LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)`	Constructs an empty LinkedHashMap instance with the specified initial capacity, load factor and ordering mode.
`LinkedHashMap(Map<? extends K,? extends V> m)`	Constructs an insertion-ordered LinkedHashMap instance with the same mappings as the specified map.

https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html
To get the details of the method visit the Link.

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package com.mycompany.collectionsconceptwithkeshav.MyListObjects;

import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Set;

/**
 *
 * @author bluej
 */
public class LinkedHashMapDemo {
    public static void main(String[] args) {
        LinkedHashMap m = new LinkedHashMap();
        m.put(100, "Harsh");
        m.put(80, "Sanjay");
        m.put(120, "Ajay");
        m.put(200, "Jiya");
        m.put(130, "Aadya");
        System.out.println(m);
        System.out.println(m.put(200, "Tanisha"));
        System.out.println(m);
        Set s = m.keySet();
        System.out.println(s);
        Collection c = m.values();
        System.out.println(c);
        Set es = m.entrySet();
        System.out.println(es);

        Iterator itr = es.iterator();
        while(itr.hasNext()){
            Map.Entry m1 = (Map.Entry)itr.next();
            System.out.println(m1.getKey()+"==>"+m1.getValue());
            Integer t = (Integer)m1.getKey();
            if(t.equals(200)){
                m1.setValue("Gargi");
            }
        }
        System.out.println(m);
    }
}
```

Output:

```
--- exec:3.1.0:exec (default-cli) @ CollectionsConceptWithKeshav ---
{100=Harsh, 80=Sanjay, 120=Ajay, 200=Jiya, 130=Aadya}
Jiya
{100=Harsh, 80=Sanjay, 120=Ajay, 200=Tanisha, 130=Aadya}
[100, 80, 120, 200, 130]
[Harsh, Sanjay, Ajay, Tanisha, Aadya]
[100=Harsh, 80=Sanjay, 120=Ajay, 200=Tanisha, 130=Aadya]
100==>Harsh
80==>Sanjay
120==>Ajay
200==>Tanisha
130==>Aadya
{100=Harsh, 80=Sanjay, 120=Ajay, 200=Gargi, 130=Aadya}
------------------------------------------------------------------------
```

LinkedHashSet and LinkedHashMap are commonly used for developing Cache based application.

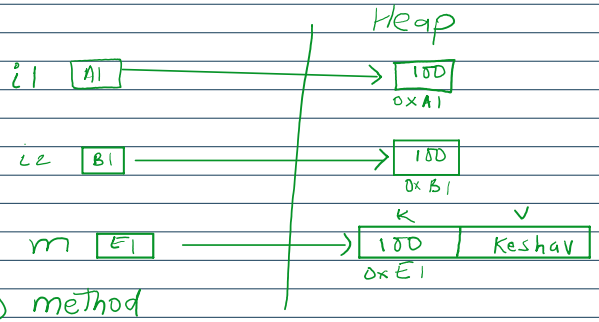
IdentityHashMap

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package com.mycompany.collectionsconceptwithkeshav.MyListObjects;

import java.util.HashMap;

/**
 *
 * @author bluej
 */
public class IdentityHashMapDemo {
    public static void main(String[] args) {
        HashMap m = new HashMap();
        Integer i1 = new Integer(100);
        Integer i2 = new Integer(100);

        m.put(i1, "Keshav");
        m.put(i2, "Aadya");
        System.out.println("" + m);
    }
}
```



null ← m.put(i1, "Keshav");
Keshav ← m.put(i2, "Aadya"); ⟶ .equals() method is called to identify the Keys i1.equals(i2),
if it returns true, then value is replaced.

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package com.mycompany.collectionsconceptwithkeshav.MyListObjects;

import java.util.HashMap;

/**
 *
 * @author bluej
 */
public class IdentityHashMapDemo {
    public static void main(String[] args) {
        HashMap m = new HashMap();
        Integer i1 = new Integer(100);
        Integer i2 = new Integer(100);

        System.out.println(m.put(i1, "Keshav"));
        System.out.println(m.put(i2, "Aadya"));
        System.out.println("" + m);
    }
}
```

Output:

```
null
Keshav
- {100=Aadya}
--------------
```

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package com.mycompany.collectionsconceptwithkeshav.MyListObjects;

import java.util.HashMap;
import java.util.IdentityHashMap;

/**
 *
 * @author bluej
 */
public class IdentityHashMapDemo {
    public static void main(String[] args) {
        IdentityHashMap m = new IdentityHashMap();
        Integer i1 = new Integer(100);
        Integer i2 = new Integer(100);

        System.out.println(m.put(i1, "Keshav"));
        System.out.println(m.put(i2, "Aadya"));
        System.out.println("" + m);
    }
}
```

→ If we replace HashMap with IdentityHashMap, then I1 and I2 are not duplicate keys.

Because, IdentityHashMap uses '==' operator instead of .equals() method. The '==' operator is used to compare identity of the objects or the reference of the object, which is always unique.

Therefore in case of IdentityHashMap JVM will use '==' operator to identify duplicate keys which is ment for reference comparison (address comparison).

Output:

```
null
null
{100=Aadya, 100=Keshav}
```

IdentityHashMap is exactly same as HashMap (including methods and constructors).

Official Documentation:

public class IdentityHashMap<K,V>
extends AbstractMap<K,V>
implements Map<K,V>, Serializable, Cloneable

This class implements the Map interface with a hash table, using reference-equality in place of object-equality when comparing keys (and values). In other words, in an IdentityHashMap, two keys k1 and k2 are considered equal if and only if (k1==k2). (In normal Map implementations (like HashMap) two keys k1 and k2 are considered equal if and only if (k1==null ? k2==null : k1.equals(k2)).)

This class is not a general-purpose Map implementation! While this class implements the Map interface, it intentionally violates Map's general contract, which mandates the use of the equals method when comparing objects. This class is designed for use only in the rare cases wherein reference-equality semantics are required.

A typical use of this class is topology-preserving object graph transformations, such as serialization or deep-copying. To perform such a transformation, a program must maintain a "node table" that keeps track of all the object references that have already been processed. The node table must not equate distinct objects even if they happen to be equal. Another typical use of this class is to maintain proxy objects. For example, a debugging facility might wish to maintain a proxy object for each object in the program being debugged.

This class provides all of the optional map operations, and permits null values and the null key. This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

This class provides constant-time performance for the basic operations (get and put), assuming the system identity hash function (System.identityHashCode(Object)) disperses elements properly among the buckets.

This class has one tuning parameter (which affects performance but not semantics): expected maximum size. This parameter is the maximum number of key-value mappings that the map is expected to hold. Internally, this parameter is used to determine the number of buckets initially comprising the hash table. The precise relationship between the expected maximum size and the number of buckets is unspecified.

If the size of the map (the number of key-value mappings) sufficiently exceeds the expected maximum size, the number of buckets is increased. Increasing the number of buckets ("rehashing") may be fairly expensive, so it pays to create identity hash maps with a sufficiently large expected maximum size. On the other hand, iteration over collection views requires time proportional to the number of buckets in the hash table, so it pays not to set the expected maximum size too high if you are especially concerned with iteration performance or memory usage.

Note that this implementation is not synchronized. If multiple threads access an identity hash map concurrently, and at least one of the threads modifies the map structurally, it must be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the Collections.synchronizedMap method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

```
Map m = Collections.synchronizedMap(new IdentityHashMap(...));
```

The iterators returned by the iterator method of the collections returned by all of this class's "collection view methods" are fail-fast: if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: fail-fast iterators should be used only to detect bugs.

Implementation note: This is a simple linear-probe hash table, as described for example in texts by Sedgewick and Knuth. The array alternates holding keys and values. (This has better locality for large tables than does using separate arrays.) For many JRE implementations and operation mixes, this class will yield better performance than HashMap (which uses chaining rather than linear-probing).

This class is a member of the Java Collections Framework.

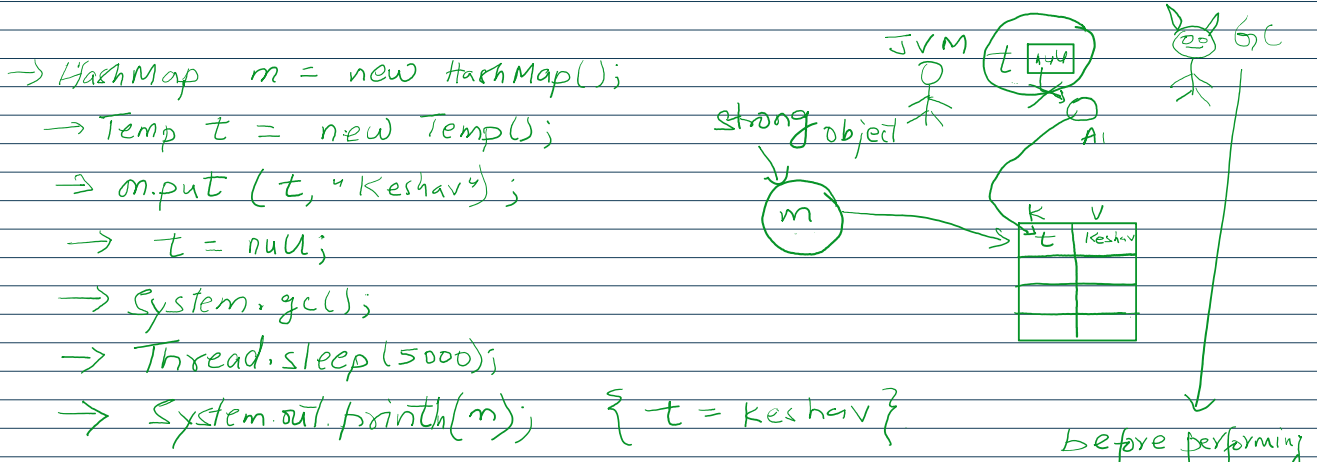
Since:
1.4

Constructors

Constructor	Description
`IdentityHashMap()`	Constructs a new, empty identity hash map with a default expected maximum size (21).
`IdentityHashMap(int expectedMaxSize)`	Constructs a new, empty map with the specified expected maximum size.
`IdentityHashMap(Map<? extends K,? extends V> m)`	Constructs a new identity hash map containing the keys-value mappings in the specified map.

https://docs.oracle.com/javase/8/docs/api/java/util/IdentityHashMap.html
To know about methods click on the link

WeakHashMap



→ System.out.println(m); {t = keshav}

before performing clean/upate it will call finalize().

finalize(){

}

```
package com.mycompany.collectionsconceptwithkeshav.MyListObjects;

import java.util.HashMap;

/**
 *
 * @author bluej
 */
public class WeakHashMapDemo {
    public static void main(String[] args) throws InterruptedException {
        HashMap m = new HashMap();
        Temp t = new Temp();
        m.put(t, "Keshav");
        t = null;
        System.gc();
        Thread.sleep(5000);
        System.out.print("\n"+m);
    }
}

public class Temp {
    @Override
    public String toString() {
        return "Temp";
    }

    @Override
    public void finalize(){
        System.out.println("Finalize method is called.");
    }
}

Output:
{Temp=Keshav}


WeakHashMap Demo Code:
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this
license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package com.mycompany.collectionsconceptwithkeshav.MyListObjects;

import java.util.HashMap;
import java.util.WeakHashMap;

/**
 *
 * @author bluej
 */
public class WeakHashMapDemo {
    public static void main(String[] args) throws InterruptedException {
        //HashMap m = new HashMap();
        WeakHashMap m = new WeakHashMap();
        Temp t = new Temp();
        m.put(t, "Keshav");
        t = null;
        System.gc();
        Thread.sleep(5000);
        System.out.println(m);
        if(m.isEmpty()){
            System.out.println("Map is empty");
        }
    }
}
```

stronger than garbage collector.

When 'null' is assigned 'GC' gets activated

even the referencing sleeping.

→ WeakHashMap is exactly same as HashMap except the following differences.

In case of HashMap even though Object doesn't have any reference, it is eligible for GC, if it is associated with HashMap. i.e., HashMap dominates Garbage collector.

But in case of WeakHashMap, if the Object doesn't contain any reference, it is elligible for gc(), even though object associated with WeakHashMap, i.e., gc() dominates the WeakHashMap.

```
    }
  }
}
```

contain any reference, it is elligble for gc(), even though object associated with WeakHashMap, i.e., gc() dominates the WeakHashMap.

Output:

```
Finalize method is called.
{}
Map is empty
```

← t is deleted by the gc() method because Map object is weak.

```
public class WeakHashMap<K,V>
extends AbstractMap<K,V>
implements Map<K,V>
```

Hash table based implementation of the Map interface, with weak keys. An entry in a WeakHashMap will automatically be removed when its key is no longer in ordinary use. More precisely, the presence of a mapping for a given key will not prevent the key from being discarded by the garbage collector, that is, made finalizable, finalized, and then reclaimed. When a key has been discarded its entry is effectively removed from the map, so this class behaves somewhat differently from other Map implementations.

Both null values and the null key are supported. This class has performance characteristics similar to those of the HashMap class, and has the same efficiency parameters of initial capacity and load factor.

Like most collection classes, this class is not synchronized. A synchronized WeakHashMap may be constructed using the Collections.synchronizedMap method.

This class is intended primarily for use with key objects whose equals methods test for object identity using the == operator. Once such a key is discarded it can never be recreated, so it is impossible to do a lookup of that key in a WeakHashMap at some later time and be surprised that its entry has been removed. This class will work perfectly well with key objects whose equals methods are not based upon object identity, such as String instances. With such recreatable key objects, however, the automatic removal of WeakHashMap entries whose keys have been discarded may prove to be confusing.

The behavior of the WeakHashMap class depends in part upon the actions of the garbage collector, so several familiar (though not required) Map invariants do not hold for this class. Because the garbage collector may discard keys at any time, a WeakHashMap may behave as though an unknown thread is silently removing entries. In particular, even if you synchronize on a WeakHashMap instance and invoke none of its mutator methods, it is possible for the size method to return smaller values over time, for the isEmpty method to return false and then true, for the containsKey method to return true and later false for a given key, for the get method to return a value for a given key but later return null, for the put method to return null and the remove method to return false for a key that previously appeared to be in the map, and for successive examinations of the key set, the value collection, and the entry set to yield successively smaller numbers of elements.

Each key object in a WeakHashMap is stored indirectly as the referent of a weak reference. Therefore a key will automatically be removed only after the weak references to it, both inside and outside of the map, have been cleared by the garbage collector.

Implementation note: The value objects in a WeakHashMap are held by ordinary strong references. Thus care should be taken to ensure that value objects do not strongly refer to their own keys, either directly or indirectly, since that will prevent the keys from being discarded. Note that a value object may refer indirectly to its key via the WeakHashMap itself; that is, a value object may strongly refer to some other key object whose associated value object, in turn, strongly refers to the key of the first value object. If the values in the map do not rely on the map holding strong references to them, one way to deal with this is to wrap values themselves within WeakReferences before inserting, as in: m.put(key, new WeakReference(value)), and then unwrapping upon each get.

The iterators returned by the iterator method of the collections returned by all of this class's "collection view methods" are fail-fast: if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its

```
correctness: the fail-fast behavior of iterators should be used only to detect bugs.

This class is a member of the Java Collections Framework.

Since:
    1.2
```