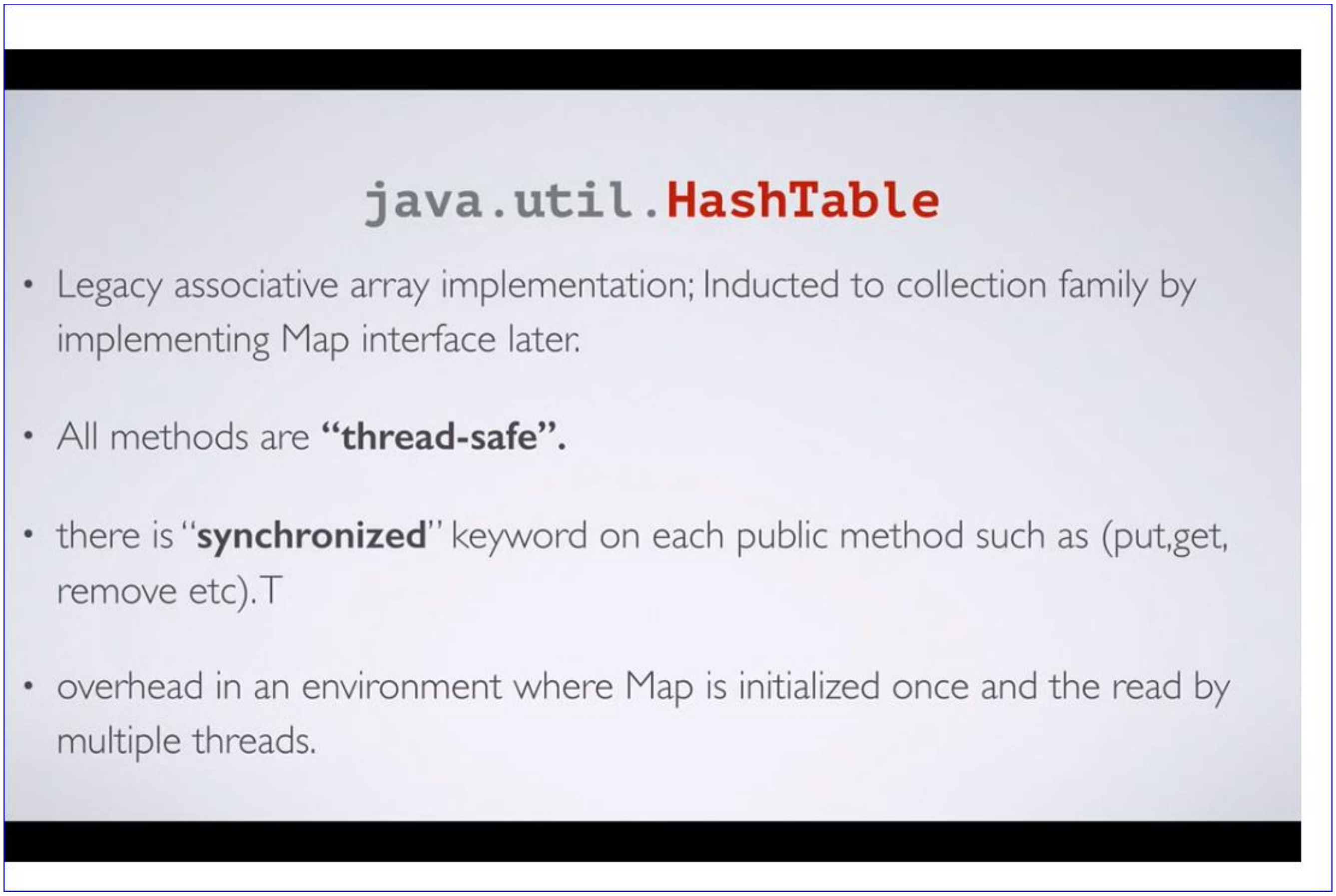
HashTable :

- thread safe, entire map is locked.
- Any non-null object can be used as a key or as a value
- Every read/write lock the whole map.
- fail-fast- ConcurrentModificationException
- Legacy class from java 1.2
- Null key and null values are not allowed.



java.util.HashTable

- Legacy associative array implementation; Inducted to collection family by implementing Map interface later.
- All methods are **"thread-safe".**
- there is "**synchronized**" keyword on each public method such as (put,get, remove etc).T
- overhead in an environment where Map is initialized once and the read by multiple threads.

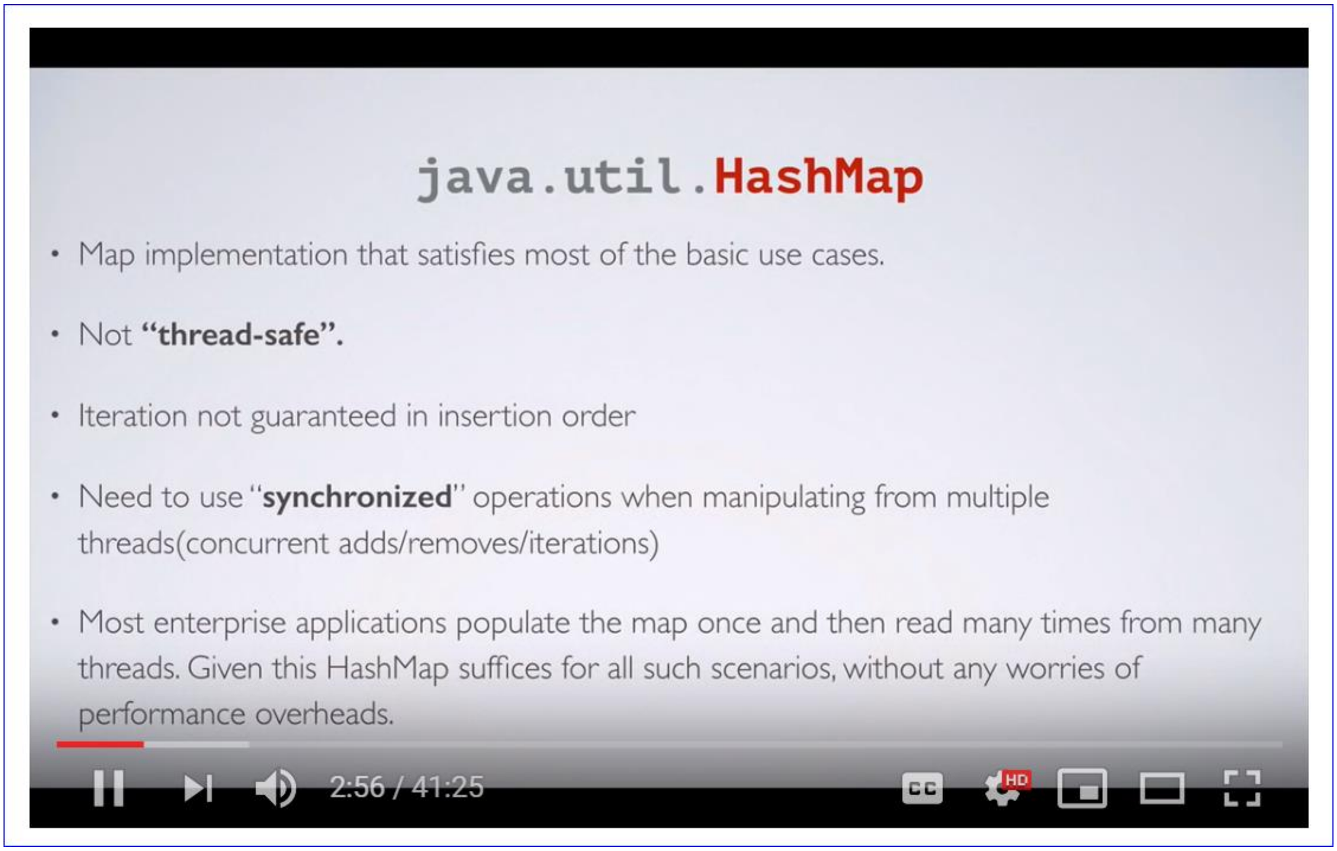
HashMap :

- permits null values and the multiple null key
- Not thread safe, calls does not block each other.
- This class makes no guarantees as to the order of the map,
- 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),

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

- iterators are fail-fast

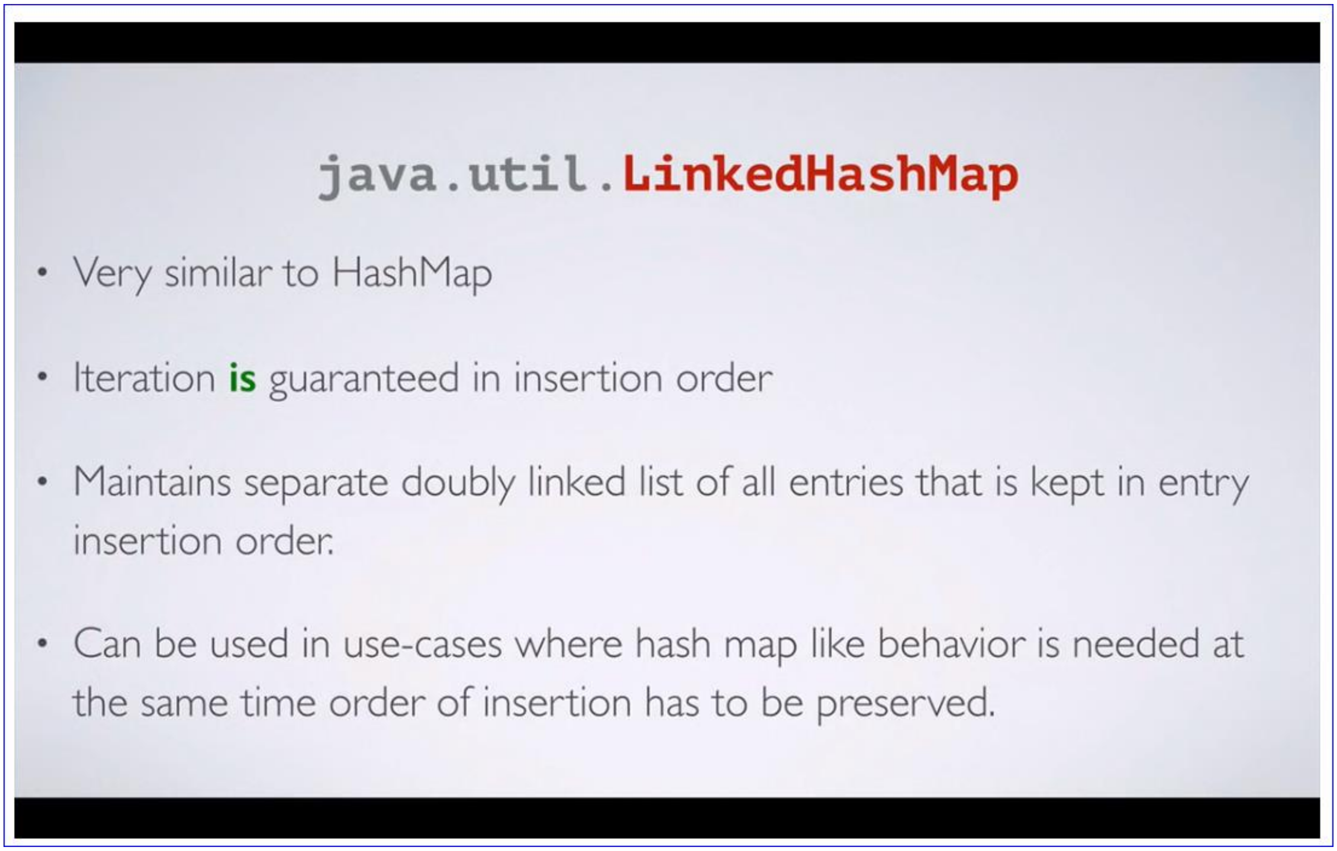


java.util.HashMap

- Map implementation that satisfies most of the basic use cases.
- Not **"thread-safe".**
- Iteration not guaranteed in insertion order
- Need to use "**synchronized**" operations when manipulating from multiple threads(concurrent adds/removes/iterations)
- Most enterprise applications populate the map once and then read many times from many threads. Given this HashMap suffices for all such scenarios, without any worries of performance overheads.

Linked Hash Map :

- with predictable iteration order.
- This implementation differs from HashMap in that it maintains a doubly-linked list running through all of its entries.
- Maintains doubly linked list internally
- Note that this implementation is not synchronized.
- Map m = Collections.synchronizedMap(new LinkedHashMap(...));
- fail-fast ConcurrentModificationException

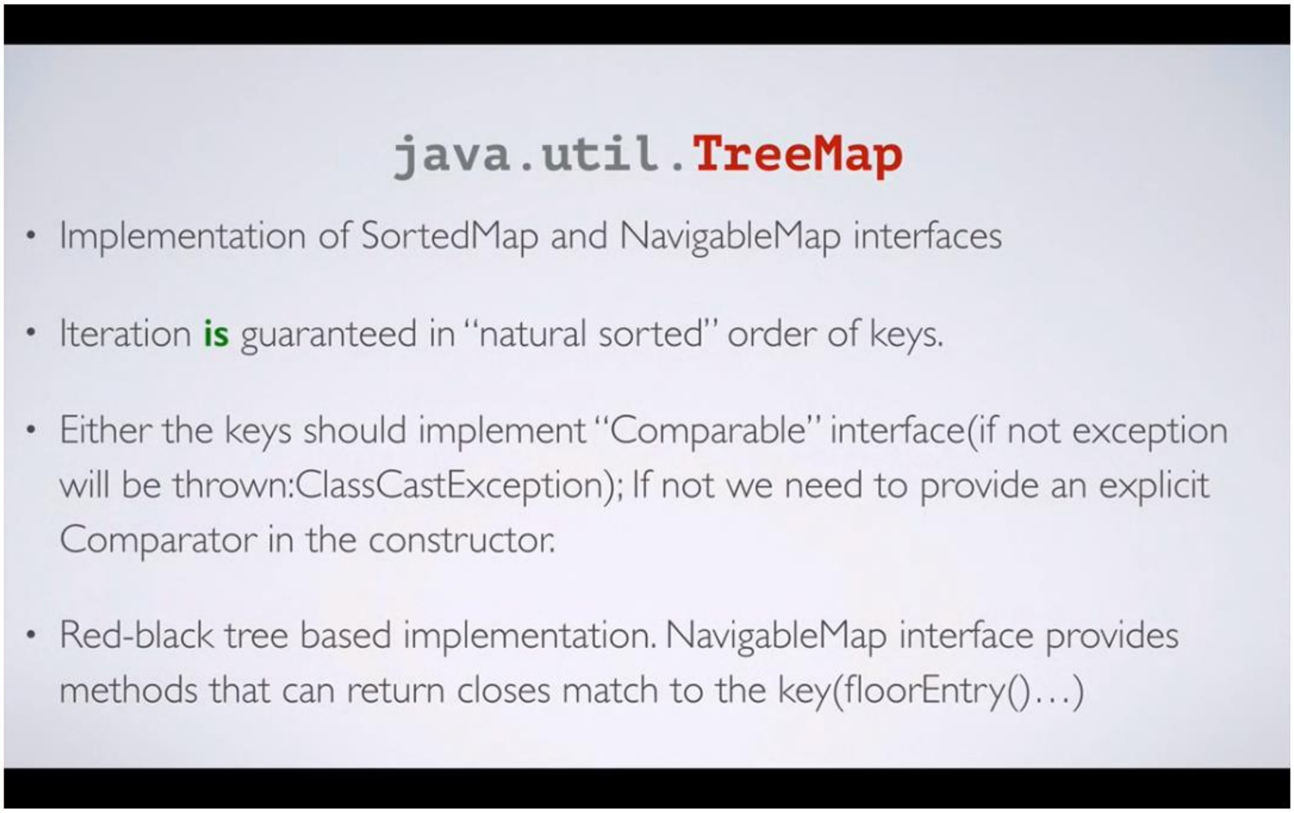


java.util.LinkedHashMap

- Very similar to HashMap
- Iteration **is** guaranteed in insertion order
- Maintains separate doubly linked list of all entries that is kept in entry insertion order.
- Can be used in use-cases where hash map like behavior is needed at the same time order of insertion has to be preserved.

Tree Map:

- key are sorted in natural sorted order in the map.
- Key must be comparator / comparable.
- The map is sorted according to the Comparator provided at map creation time, depending on which constructor is used
- This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations
- SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));
- iterator are fail-fast

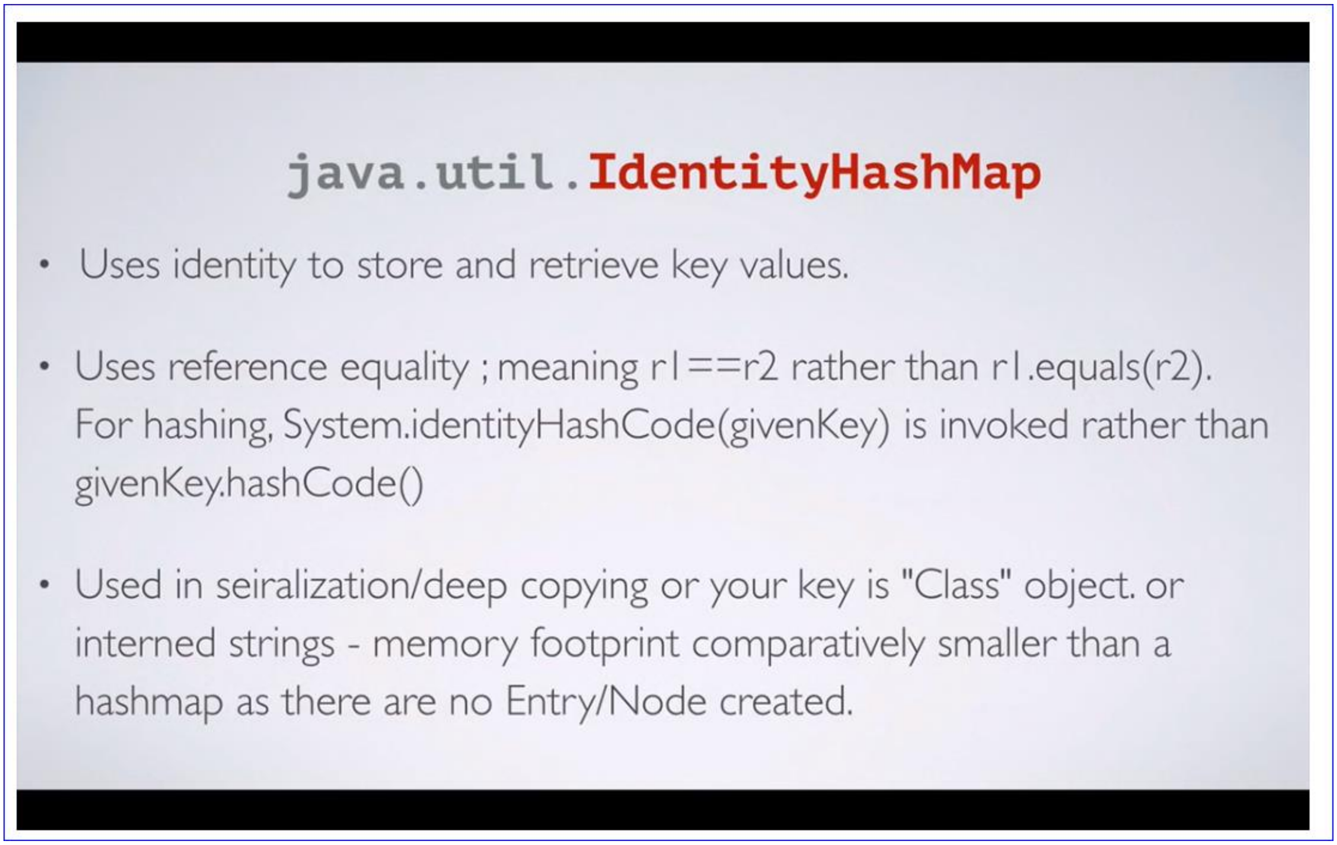


java.util.TreeMap

- Implementation of SortedMap and NavigableMap interfaces
- Iteration **is** guaranteed in "natural sorted" order of keys.
- Either the keys should implement "Comparable" interface(if not exception will be thrown:ClassCastException); If not we need to provide an explicit Comparator in the constructor.
- Red-black tree based implementation. NavigableMap interface provides methods that can return closes match to the key(floorEntry()…)

IdentityHashMap :

- used == to equality check.
- two keys k1 and k2 are considered equal if and only if (k1==k2)
- Map m = Collections.synchronizedMap(new IdentityHashMap(...));
- fail-fast -ConcurrentModificationException



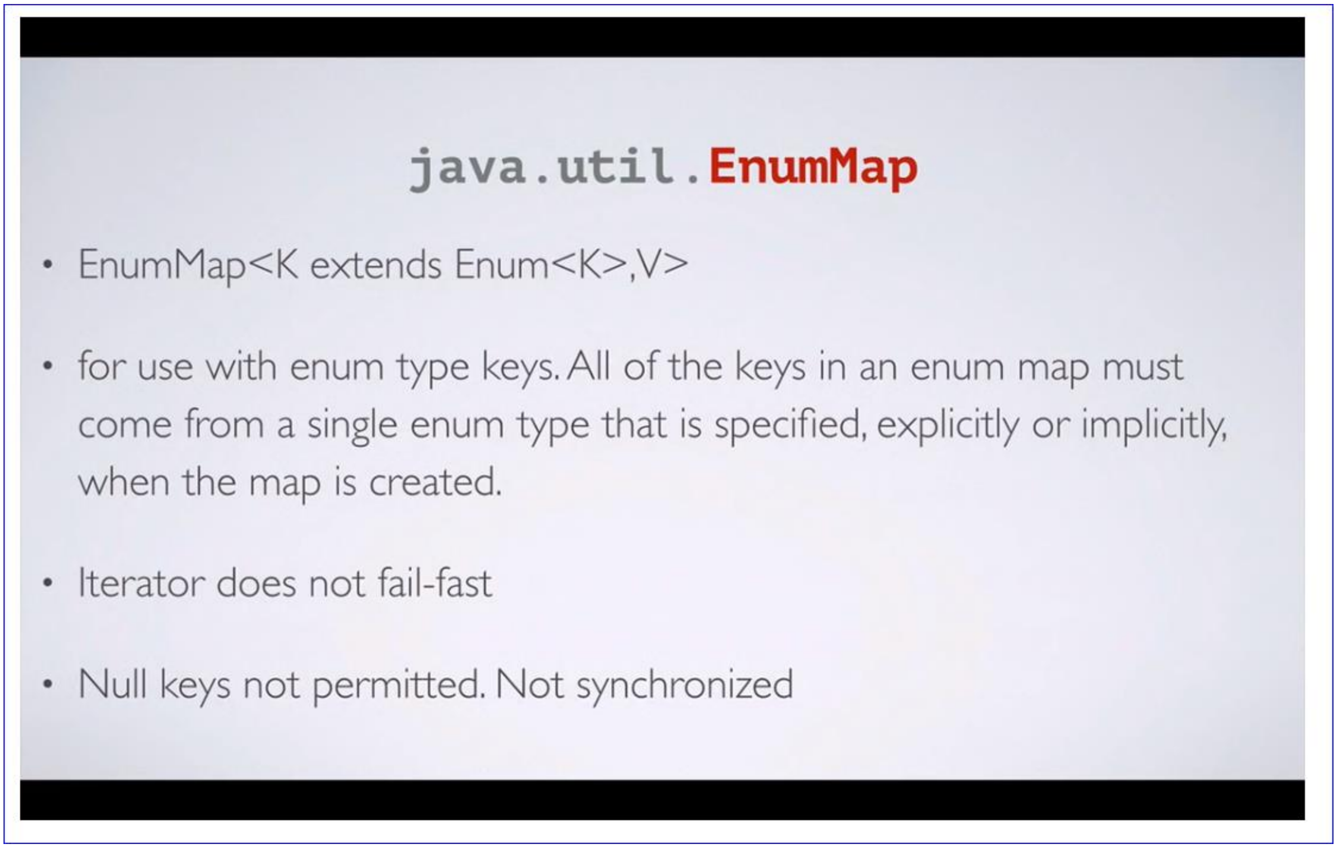
java.util.IdentityHashMap

- Uses identity to store and retrieve key values.
- Uses reference equality ; meaning r1==r2 rather than r1.equals(r2). For hashing, System.identityHashCode(givenKey) is invoked rather than givenKey.hashCode()
- Used in seiralization/deep copying or your key is "Class" object. or interned strings - memory footprint comparatively smaller than a hashmap as there are no Entry/Node created.

ENumMap :

- Enum type key,
- Null keys are not permitted. Attempts to insert a null key will throw NullPointerException.
- A specialized Map implementation for use with enum type keys
- All of the keys in an enum map must come from a single enum type that is specified
- Enum maps are represented internally as arrays. This representation is extremely compact and efficient.

- Map<EnumKey, V> m
  = Collections.synchronizedMap(new EnumMap<EnumKey, V>(...));



java.util.EnumMap

- EnumMap<K extends Enum<K>,V>
- for use with enum type keys. All of the keys in an enum map must come from a single enum type that is specified, explicitly or implicitly, when the map is created.
- Iterator does not fail-fast
- Null keys not permitted. Not synchronized

When reading , updating not allowed is called failed fast.

package com.pune.it.a04;
 	
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
 	
public class TestHashMap {
 	
    public static void main(String[] args) {
 	
        Map<String, Integer> map=new HashMap<>();
        map.put("User1", 1);
        map.put("User2", 1);
        map.put("User3", 1);
        map.put("User4", 1);
        map.put("User5", 1);
        map.put("User6", 1);
        map.put("User7", 1);
        map.put("User8", 1);
	Exception in thread "main" java.util.ConcurrentModificationException
	    at java.util.HashMap$HashIterator.nextNode(HashMap.java:1445)
	    at java.util.HashMap$KeyIterator.next(HashMap.java:1469)
	    at main(TestHashMap.java:28)

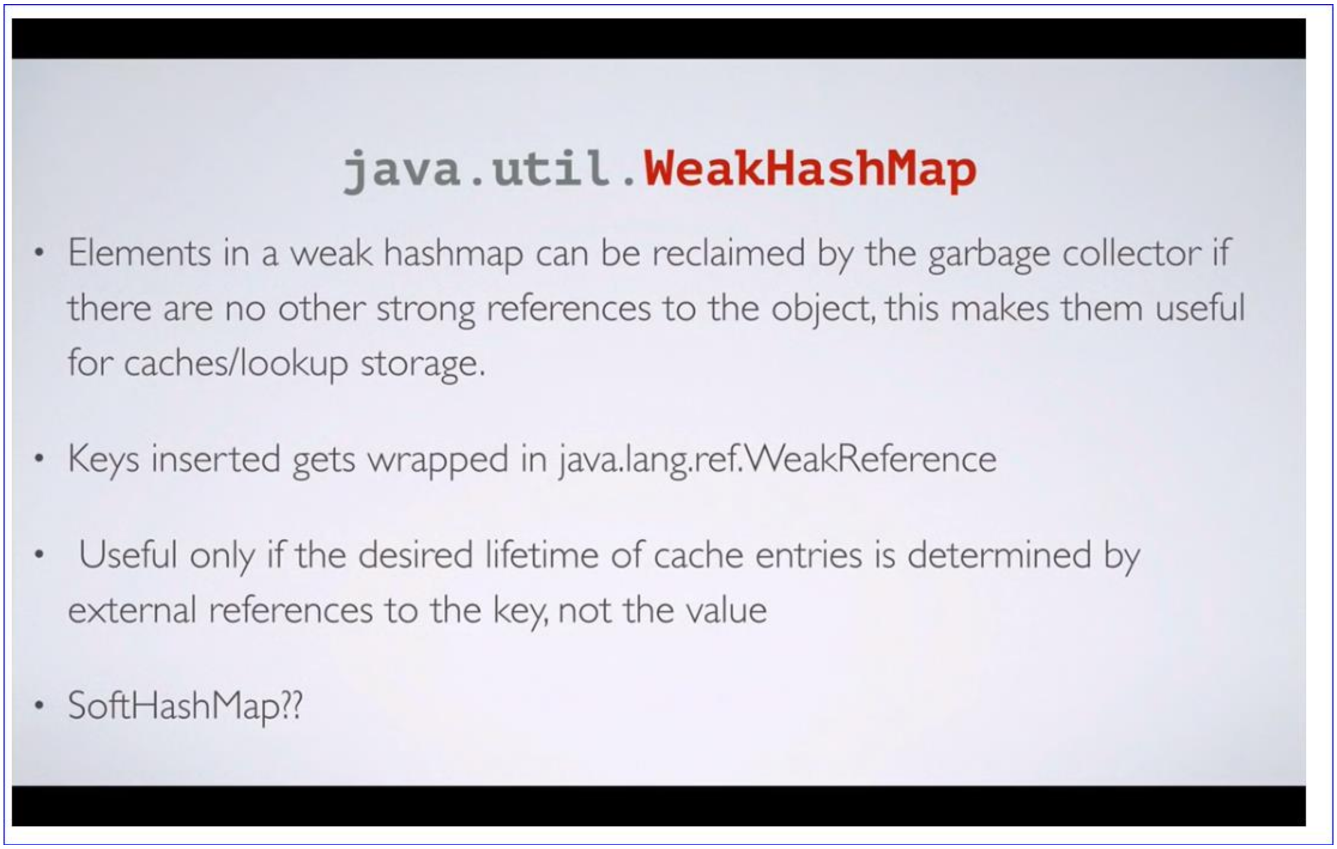
```
            map.put("User9", 1);


            Iterator<String> iterator = map.keySet().iterator();

            while(iterator.hasNext()) {
                  System.out.println(map.get(iterator.next()));

                  map.put("User10", 10);
            }

      }
}
```

WeakHashMap : used to maintain cache.

- Key stored with week reference will be GC /reclaimed.
- When there is not strong reference, that will be garbage collected.

- 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

- Both null values and the null key are supported.
- this class is not synchronized. A synchronized WeakHashMap may be constructed using the Collections.synchronizedMap
- for object identity using the == operator.Once such a key is discarded it can never be recreated,
- Fail-fast iterators throw ConcurrentModificationExceptionon a best-effort basis



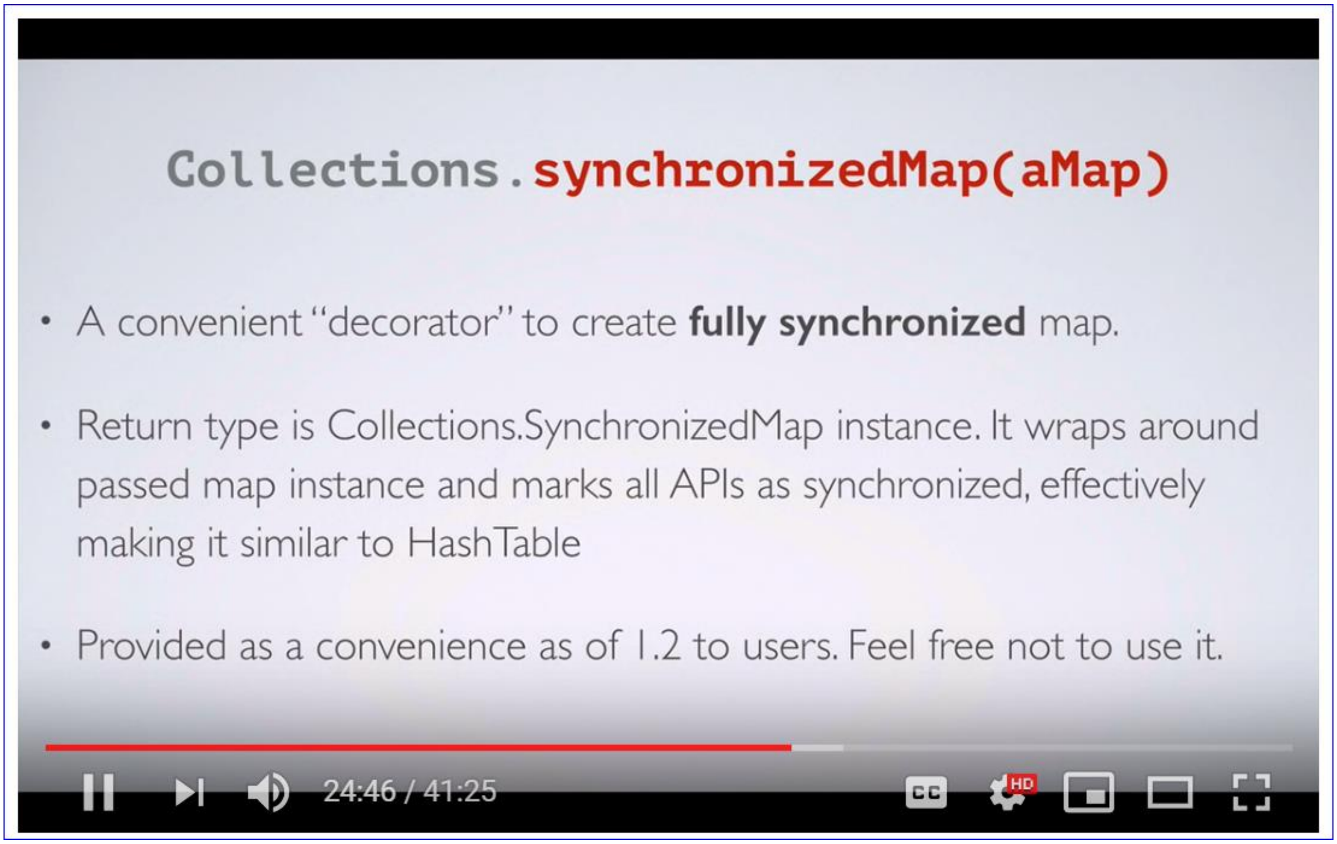
java.util.WeakHashMap

- Elements in a weak hashmap can be reclaimed by the garbage collector if there are no other strong references to the object, this makes them useful for caches/lookup storage.
- Keys inserted gets wrapped in java.lang.ref.WeakReference
- Useful only if the desired lifetime of cache entries is determined by external references to the key, not the value
- SoftHashMap??

```
package com.pune.it.a04;

import java.util.Map;
import java.util.WeakHashMap;

class Order{

                int id;
                String details;
                public Order(int id, String details) {
                        super();
                        this.id = id;
                        this.details = details;
                }



}
public class TestWeakHashMap {

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

```
3
1
```

```
                Map<Order, Integer> map=new WeakHashMap<Order, Integer>();

                map.put(new Order(1, "A"), 1);
                map.put(new Order(2, "B"), 2);

                Order o1=new Order(3, "C");
                map.put(o1, 2);
                System.out.println(map.size());
                System.gc();
                Thread.sleep(2000);

                System.out.println(map.size());
        }
}
```



Collections.synchronizedMap(aMap)

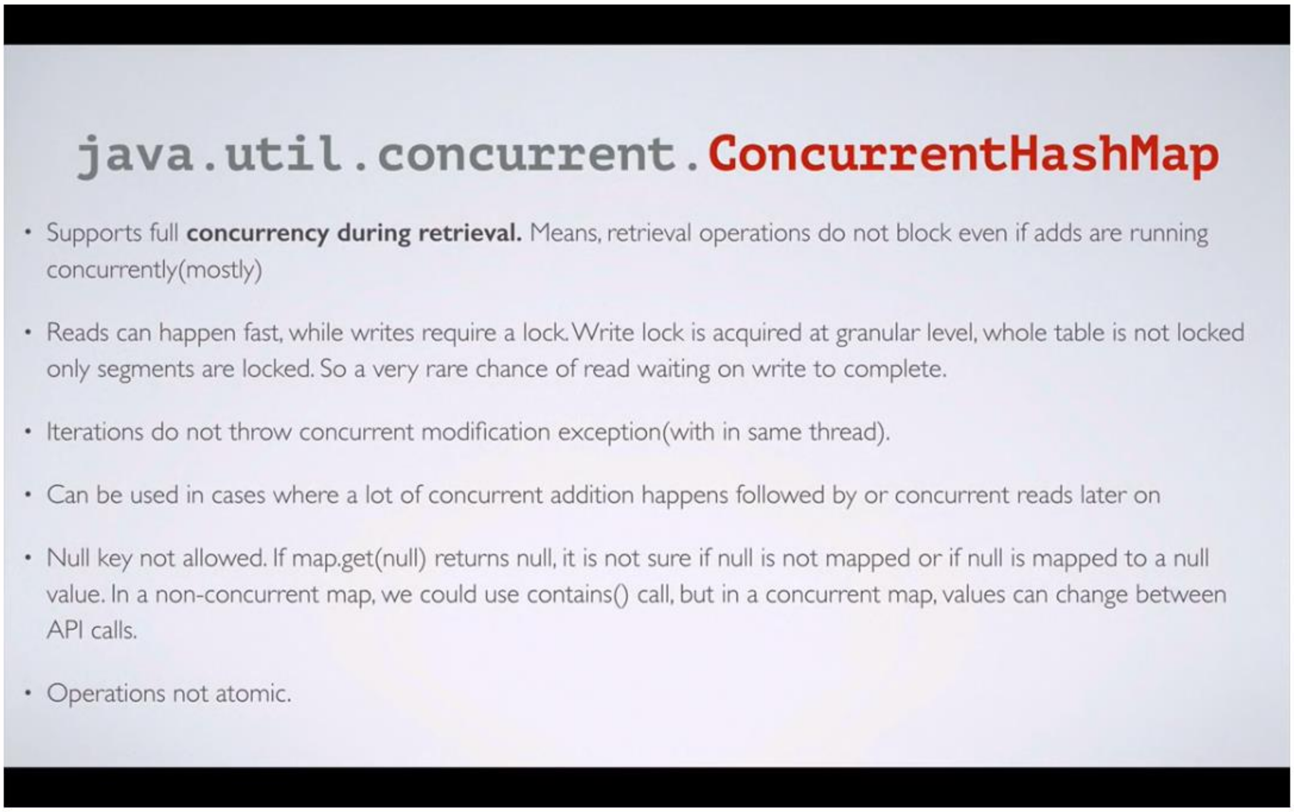
- A convenient "decorator" to create **fully synchronized** map.
- Return type is Collections.SynchronizedMap instance. It wraps around passed map instance and marks all APIs as synchronized, effectively making it similar to HashTable
- Provided as a convenience as of 1.2 to users. Feel free not to use it.

Concurrent HashMap

Lock on the segment not on the whole map, write has less amount of lock,
16 threads can write concurrently
Read is faster.

Retrieval operations (including get) generally do not block
They do ***not*** throw ConcurrentModificationException
However, iterators are designed to be used by only one thread at a time
this class does not allow null to be used as a key or value.

ConcurrentHashMaps support a set of sequential and parallel bulk operations that, unlike most Stream methods



java.util.concurrent.ConcurrentHashMap

- Supports full **concurrency during retrieval.** Means, retrieval operations do not block even if adds are running concurrently(mostly)
- Reads can happen fast, while writes require a lock. Write lock is acquired at granular level, whole table is not locked only segments are locked. So a very rare chance of read waiting on write to complete.
- Iterations do not throw concurrent modification exception(with in same thread).
- Can be used in cases where a lot of concurrent addition happens followed by or concurrent reads later on
- Null key not allowed. If map.get(null) returns null, it is not sure if null is not mapped or if null is mapped to a null value. In a non-concurrent map, we could use contains() call, but in a concurrent map, values can change between API calls.
- Operations not atomic.

ConcurrentSkipListMap (Treemap concurrent version)

A scalable concurrent ConcurrentNavigableMap implementation
The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

Iterate MAP

```java
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class MapTest2 {

    public static void main(String[] args) {


        Map<Integer, Integer> map = new HashMap<>();
        map.put(1,100);
        map.put(2,200);
        map.put(3,300);
        map.put(4,400);
        map.put(5,500);


        //entry set
        for(Map.Entry<Integer, Integer> m: map.entrySet()){
            Integer key = m.getKey();
            Integer value = m.getValue();
            System.out.println(key);
            System.out.println(value);
        }

        //iterable
        Iterator<Integer> iterator = map.values().iterator();
        while(iterator.hasNext()){
            System.out.println(iterator.next());
        }

      //iterable
        Iterator<Integer> iterator2 = map.keySet().iterator();
        while(iterator2.hasNext()){
            System.out.println(iterator2.next());
        }


    }
}
```