Hashtable

Hashtable (Java Platform SE 8) (oracle.com)

https://docs.oracle.com/javase/8/docs/api/java/util/Hashtable.html

```
package com.hdfc.collections;

import java.util.Collection;
import java.util.Hashtable;
import java.util.Map;
import java.util.Set;

public class MapTest3 {
    public static void main(String[] args) {

        Map<String, String> hashtable = new Hashtable<>(12); //java 1.2
        hashtable.put(null, null); //not allowed, java.lang.NullPointerException
        hashtable.put("a","A"); // a= key  A=value
        hashtable.put("b","B");
        hashtable.put("c","C");
        hashtable.put("d","D");

        for (Map.Entry<String, String> entry : hashtable.entrySet()){
            String key = entry.getKey();
            String value = entry.getValue();
        }

        hashtable.clear();//removes all entires from map
        String value=  hashtable.get("a");//A

        boolean result = hashtable.containsKey("e");

        String value2 = hashtable.remove("a");

        Set<String> setOfKeys = hashtable.keySet(); // b c d
        Collection<String> values = hashtable.values();

    }
}
```

HashMap

HashMap (Java Platform SE 8) (oracle.com)
https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

```
package com.hdfc.collections;

import java.util.*;

public class MapTest3 {
    public static void main(String[] args) {

        Map<String, String> hashMap = new HashMap<>();
        hashMap.put(null, null); //not allowed, java.lang.NullPointerException
        hashMap.put("a","A"); // a= key  A=value
        hashMap.put("b","B");
        hashMap.put("c","C");
        hashMap.put("d","D");

        for (Map.Entry<String, String> entry : hashMap.entrySet()){
            String key = entry.getKey();
            String value = entry.getValue();
        }

        hashMap.clear();//removes all entires from map
        String value=  hashMap.get("a");//A

        boolean result = hashMap.containsKey("e");

        String value2 = hashMap.remove("a");

        Set<String> setOfKeys = hashMap.keySet(); // b c d
        Collection<String> values = hashMap.values();

    }
}
```

EnumMap

EnumMap (Java Platform SE 8) (oracle.com)
https://docs.oracle.com/javase/8/docs/api/java/util/EnumMap.html

```
package com.hdfc.collections;

import java.util.EnumMap;


public class EnumMapTest {


    enum Size {
        SMALL, MEDIUM, LARGE

    }

    public static void main(String[] args) {

        EnumMap<Size, String> map = new EnumMap<>(Size.class);
        //hashcode and equal()
        //Enum class has hashcode and equal()
        //Enum extends from Object class

        map.put(Size.MEDIUM, "MEDIUM");
        map.put(Size.SMALL, "SMALL");
        map.put(Size.LARGE, "LARGE");

        System.out.println(map.get(Size.LARGE));
    }
}
```

EnumMapTest

```
package com.hdfc.collections;

import java.util.Collections;
import java.util.EnumMap;
import java.util.Map;

//class
//interface
//enum
//annotation
public class EnumMapTest {


    enum COLOR {
        RED, YELLOW, WHITE, BLACK
    }

    enum Size { //enum introduced in java 5
        SMALL, MEDIUM, LARGE

    }

    public static void main(String[] args) {

        EnumMap map1 = new EnumMap<>(Size.class); //BEFORE JAVA 5
        EnumMap<Size, String> map = new EnumMap<>(Size.class); //After JAVA 5 with Generics
        //hashcode and equal()
        //Enum class has hashcode and equal()
        //Enum extends from Object class

        Map<Size, String> sizeStringMap = Collections.synchronizedMap(map); //thread safe


        map.put(Size.MEDIUM, "MEDIUM");
        map.put(Size.SMALL, "SMALL");
        map.put(Size.LARGE, "LARGE");

        //map.put(COLOR.RED, "MEDIUM");///.ClassCastException

        System.out.println(map.get(Size.LARGE));
    }
}
```

GcExample

```
package com.hdfc.collections;

public class GcExample {

    public static void main(String[] args) {

        GcExample gc = new GcExample();
        gc.test();

    }

    public void test(){
        test2 ();
    }


    public void test2 (){
        String hello = "hello";
    }
}
```

WeakHashMap

WeakHashMap (Java Platform SE 8) (oracle.com)
https://docs.oracle.com/javase/8/docs/api/java/util/WeakHashMap.html

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.*

```
package com.hdfc.collections;

import java.util.HashMap;
import java.util.Map;
import java.util.Objects;
import java.util.WeakHashMap;

class Order{

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

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Order order = (Order) o;

        if (id != order.id) return false;
        return Objects.equals(details, order.details);
    }

    @Override
    public int hashCode() {
        int result = id;
        result = 31 * result + (details != null ? details.hashCode() : 0);
        return result;
    }
}

//main ()
// gc thread-  <JVM>

public class TestWeakHashMap {

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


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

        Order o2 = new Order(1);
        map.put(o2, 1);
        map.put(new Order(2), 2);//gc
        Order o1=new Order(3);
        map.put(o1, 2);

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

        System.gc(); // invoke garbage collection
        //we do not write System.gc()
        Thread.sleep(2000);//sure

        System.out.println(map.size());
        System.out.println(map.get(new Order(1))); //null
        System.out.println(map.get(new Order(2)));//null
        System.out.println(map.get(o1));//2
    }
```

 Contact: 8087883669

```
}
```

Comparable

Comparable (Java Platform SE 8) (oracle.com)
https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's `compareTo` method is referred to as its *natural comparison method*.

Lists (and arrays) of objects that implement this interface can be sorted automatically by `Collections.sort` (and `Arrays.sort`). Objects that implement this interface can be used as keys in a sorted map or as elements in a sorted set, without the need to specify a comparator.

The natural ordering for a class `C` is said to be *consistent with equals* if and only if `e1.compareTo(e2) == 0` has the same boolean value as `e1.equals(e2)` for every `e1` and `e2` of class `C`. Note that `null` is not an instance of any class, and `e.compareTo(null)` should throw a `NullPointerException` even though `e.equals(null)` returns `false`.

It is strongly recommended (though not required) that natural orderings be consistent with equals. This is so because sorted sets (and sorted maps) without explicit comparators behave "strangely" when they are used with elements (or keys) whose natural ordering is inconsistent with equals. In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the `equals` method.

For example, if one adds two keys `a` and `b` such that `(!a.equals(b) && a.compareTo(b) == 0)` to a sorted set that does not use an explicit comparator, the second `add` operation returns false (and the size of the sorted set does not increase) because `a` and `b` are equivalent from the sorted set's perspective.

Virtually all Java core classes that implement `Comparable` have natural orderings that are consistent with equals. One exception is `java.math.BigDecimal`, whose natural ordering equates `BigDecimal` objects with equal values and different precisions (such as 4.0 and 4.00).

For the mathematically inclined, the *relation* that defines the natural ordering on a given class C is:

```
{(x, y) such that x.compareTo(y) <= 0}.
```

The *quotient* for this total order is:

```
{(x, y) such that x.compareTo(y) == 0}.
```

It follows immediately from the contract for `compareTo` that the quotient is an *equivalence relation* on `C`, and that the natural ordering is a *total order* on `C`. When we say that a class's natural ordering is *consistent with equals*, we mean that the quotient for the natural ordering is the equivalence relation defined by the class's `equals(Object)` method:

```
{(x, y) such that x.equals(y)}.
```

This interface is a member of the Java Collections Framework.

```
package com.hdfc.collections;

public class Player implements Comparable<Player> {
```

```
    private int rating; // 1,2,3,4,5
    private String name;
    private int age;

    public Player(int rating, String name, int age) {
        this.rating = rating;
        this.name = name;
        this.age = age;
    }

    public int getRating() {
        return rating;
    }

    public void setRating(int rating) {
        this.rating = rating;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Player{" +
                "rating=" + rating +
                ", name='" + name + '\'' +
                ", age=" + age +
                '}';
    }

    // 0 = both are same
    // -1 : less   this.rating is less than o.getRating()
    //+ 1 : more  this.rating is more than o.getRating()
    @Override
    public int compareTo(Player o) {
        //Number
         return Integer.compare(this.rating, o.getRating());  //rating
        //Double, Byte, Short

        //String
        //return this.name.compareTo(o.name);

       /* if (this.rating == o.rating) {
            return 0;
        } else if (this.rating < o.rating) {
            return -1;
        } else {
            return 1;
        }*/

        //return this.rating - o.getRating(); //avoid
    }
}
```

```
package com.hdfc.collections;

import java.util.*;

public class PlayerTest {
```

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

        Player p2 = new Player(4, "A", 20);
        Player p3 = new Player(3, "F", 20);
        Player p4 = new Player(2, "G", 40);
        Player p5 = new Player(5, "X", 50);//
        Player p6 = new Player(5, "A", 60);//
        Player p1 = new Player(1, "E", 10);



        Set<Player> players =  new TreeSet<>();
        players.add(p1);
        players.add(p2);
        players.add(p3);
        players.add(p4);
        players.add(p5);
        players.add(p6);

        System.out.println(players);


        Set<Integer> number =  new TreeSet<>(); //order natural order
        number.add(4);
        number.add(9);
        number.add(2);
        //no sorting
        //System.out.println(number);

        Set<String> name =  new TreeSet<>(); //order natural order
        name.add("Sumit");
        name.add("Amit");
        name.add("Ravi");
        name.add("Mayank");
        name.add("Sourabh");

        //no sorting
       // System.out.println(name);


    }
}
```