Marwadi education foundation

Unit – 11

Sets and Maps

Prepared By

Prof. Ravikumar Natarajan

Assistant Professor, CE Dept.

KNOWLEDGE IS THE CURRENCY
FOR THE 21st CENTURY

Contents

Introduction

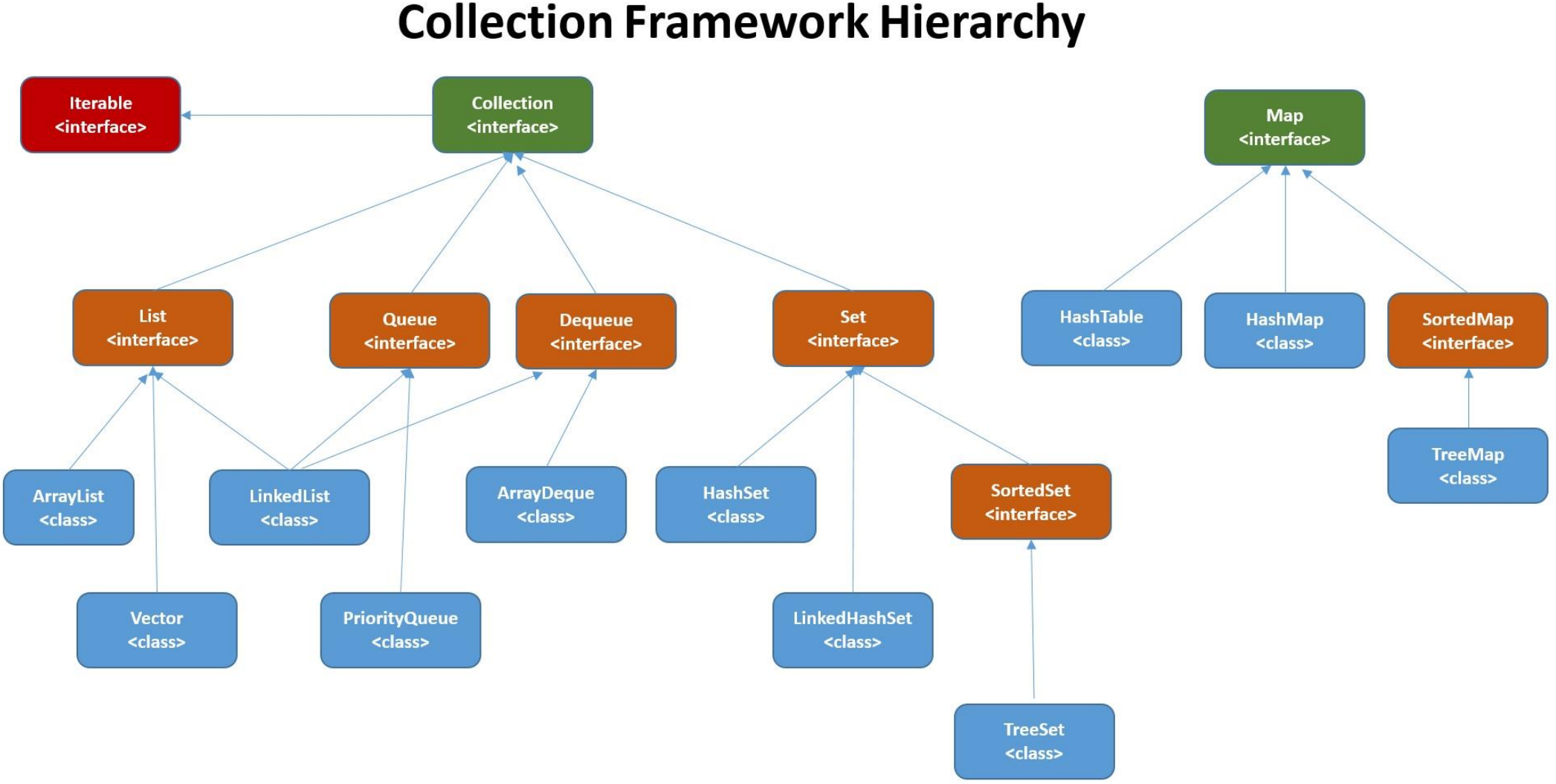
- A set is an efficient data structure for storing and processing nonduplicate elements.
- A map is like a dictionary that provides a quick lookup to retrieve a value using a key.
- For Ex. The “No Fly” list is a list where, we need to write a program that checks whether a person is on the No Fly list or not.

- You can use a list to store names in the No Fly list.
- **However, a more efficient data structure for this application is a set.**

Introduction

- Suppose your program also needs to store detailed information about person in the No Fly list.
- The detailed information such as gender, height, weight, and nationality can be retrieved using the **name** as the key.
- A **map** is an efficient data structure for such a task.
- You can create a set using one of its three concrete classes: HashSet, LinkedHashSet, or TreeSet
- **These 3 classes that implement Set must ensure that no duplicate elements can be added to the set.**

Introduction



HashSet

- **The HashSet class is a concrete class that implements Set.** You can create an empty hash set using its no-arg constructor or create a hash set from an existing collection.
- By default, the initial capacity is 16 and the load factor is 0.75. The load factor is a value between 0.0 and 1.0.
- **A HashSet can be used to store duplicate-free elements.**

HashSet

```
import java.util.*;
 public class Main {
 public static void main(String[] args) {
Set<String> set = new HashSet<>();
set.add("London");
set.add("Paris");
set.add("New York"); set.add("London");
System.out.println(set);
//for (String s: set) {
 //System.out.print(s.toUpperCase() + " ");
 } } }
```

Output:
[New York, London, Paris]

HashSet


```
import java.util.*;
 public class Main {
 public static void main(String[] args) {
Set<String> set = new HashSet<>();
set.add("London");
set.add("Paris");
set.add("New York");
System.out.println(set);
System.out.println("Is Rajkot in set?:" + set.contains("Rajkot"));
System.out.println("RemoveAll:" + set.removeAll(set));
System.out.println(set);
} }
```

Output:
[New York, London, Paris]
Is Rajkot in set?:false
RemoveAll:true
[]

LinkedHashSet

- LinkedHashSet extends HashSet with a linked-list implementation that **supports an ordering of the elements in the set.**
- The elements in a HashSet are not ordered, but the elements in a LinkedHashSet can be retrieved in the order in which they were inserted into the set.
- Since LinkedHashSet is a set, it does not store duplicate elements.
- The LinkedHashSet maintains the order in which the elements are inserted. To impose a different order (e.g., increasing or decreasing order), you can use the TreeSet class

LinkedHashSet


```
import java.util.*;

 public class Main {
 public static void main(String[] args) {
Set<String> set = new LinkedHashSet<>();

set.add("London");
 set.add("Paris");
 set.add("New York");
 set.add("Beijing");
 set.add("New York");

 System.out.println(set);
 }}
```

Output:
[London, Paris, New York, Beijing]

TreeSet

- SortedSet is a subinterface of Set, which guarantees that the elements in the set are sorted.
- Additionally, it provides the methods first() and last() for returning the first and last elements in the set.

- headSet(toElement) and tailSet(fromElement) for returning a portion of the set whose elements are less than toElement and greater than or equal to fromElement, respectively.

- The pollFirst() and pollLast() methods remove and return the first and last element in the tree set, respectively.
- Non duplicate elements will be stored.

TreeSet


```
import java.util.*;
public class Main {
public static void main(String[] args) {
TreeSet<String> set = new TreeSet<>();

set.add("London");
set.add("Paris");
set.add("New York");
set.add("Beijing");
set.add("New York");

System.out.println(set);
System.out.println("first(): " + set.first());
System.out.println("last(): " + set.last());

 System.out.println("pollFirst(): " + set.pollFirst());
 System.out.println("pollLast(): " + set.pollLast());
 System.out.println(set);   }}
```

Output:
[Beijing, London, New York, Paris]
first(): Beijing
last(): Paris
pollFirst(): Beijing
pollLast(): Paris
[London, New York]

Comparing the Performance of Sets and Lists

- Sets are more efficient than lists for storing nonduplicate elements. **Lists are useful for accessing elements through the index.**

- The elements in a list can be accessed through the index. However, sets do not support indexing, because the elements in a set are unordered.

- To traverse all elements in a set, use a foreach loop.

Comparing the Performance of Sets and Lists

FOR REFERENCE

```
import java.util.*;

public class Main {
static final int N = 50000;

public static void main(String[] args) {
// Add numbers 0, 1, 2, ..., N - 1 to the array list
List<Integer> list = new ArrayList<>();
for (int i = 0; i < N; i++)
list.add(i);
Collections.shuffle(list); // Shuffle the array list

// Create a hash set, and test its performance
Collection<Integer> set1 = new HashSet<>(list);
System.out.println("Member test time for hash set is " +
getTestTime(set1) + " milliseconds");
System.out.println("Remove element time for hash set is " +
getRemoveTime(set1) + " milliseconds");

// Create a linked hash set, and test its performance
Collection<Integer> set2 = new LinkedHashSet<>(list);
System.out.println("Member test time for linked hash set is " +
getTestTime(set2) + " milliseconds");
System.out.println("Remove element time for linked hash set is "
+ getRemoveTime(set2) + " milliseconds");

// Create a tree set, and test its performance
Collection<Integer> set3 = new TreeSet<>(list);
System.out.println("Member test time for tree set is " +
getTestTime(set3) + " milliseconds");
System.out.println("Remove element time for tree set is " +
getRemoveTime(set3) + " milliseconds");

// Create an array list, and test its performance
Collection<Integer> list1 = new ArrayList<>(list);
System.out.println("Member test time for array list is " +
getTestTime(list1) + " milliseconds");
System.out.println("Remove element time for array list is " +
getRemoveTime(list1) + " milliseconds");

// Create a linked list, and test its performance
Collection<Integer> list2 = new LinkedList<>(list);
System.out.println("Member test time for linked list is " +
getTestTime(list2) + " milliseconds");
System.out.println("Remove element time for linked list is " +
getRemoveTime(list2) + " milliseconds");
}

public static long getTestTime(Collection<Integer> c) {
long startTime = System.currentTimeMillis();

// Test if a number is in the collection
for (int i = 0; i < N; i++)
c.contains((int)(Math.random() * 2 * N));

return System.currentTimeMillis() - startTime;
}

public static long getRemoveTime(Collection<Integer> c) {
long startTime = System.currentTimeMillis();

for (int i = 0; i < N; i++)
c.remove(i);

return System.currentTimeMillis() - startTime;
}
}
```

Maps

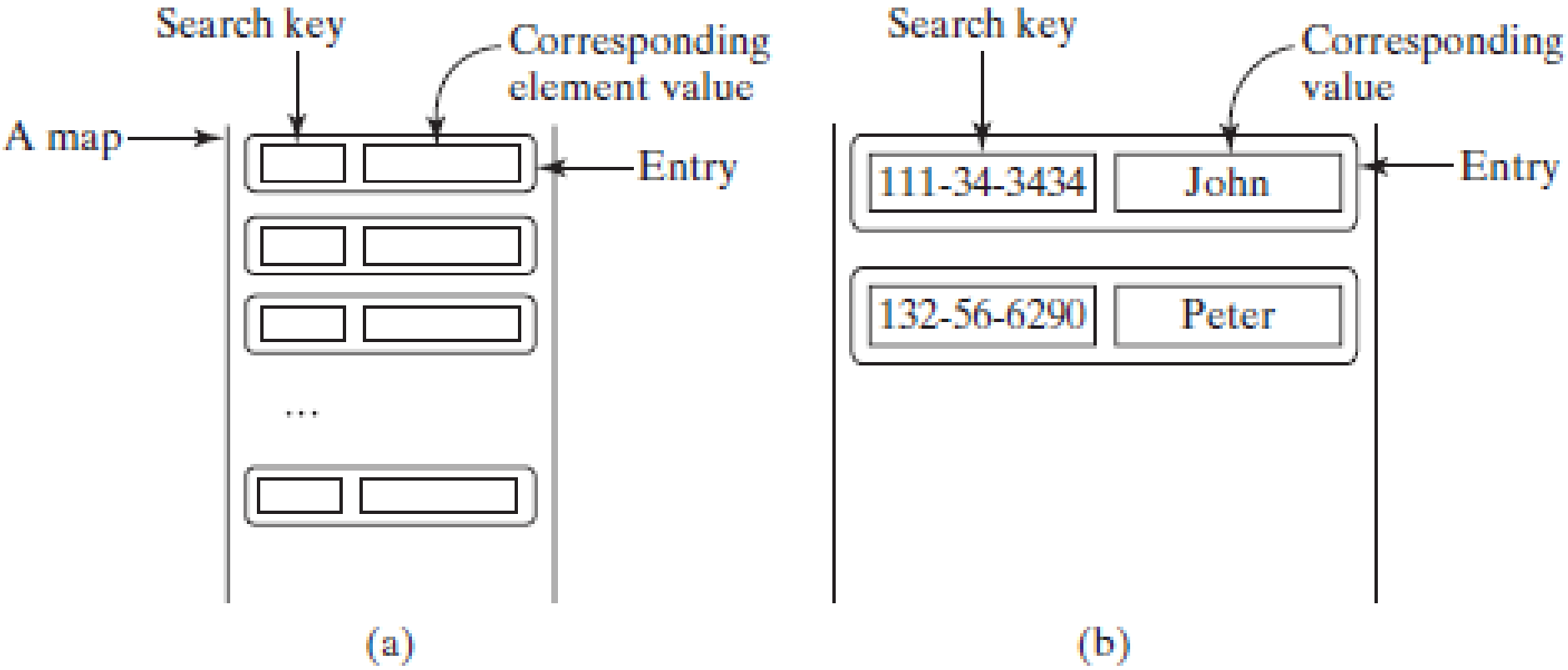
- You can create a map using one of its three concrete classes: HashMap, LinkedHashMap, or TreeMap.

- A map is a container object that stores a collection of key/value pairs. It enables fast retrieval, deletion, and updating of the pair through the key.
- A map stores the values along with the keys. The keys are like indexes.
- **In List, the indexes are integers. In Map, the keys can be any objects.**

- **A map cannot contain duplicate keys.**

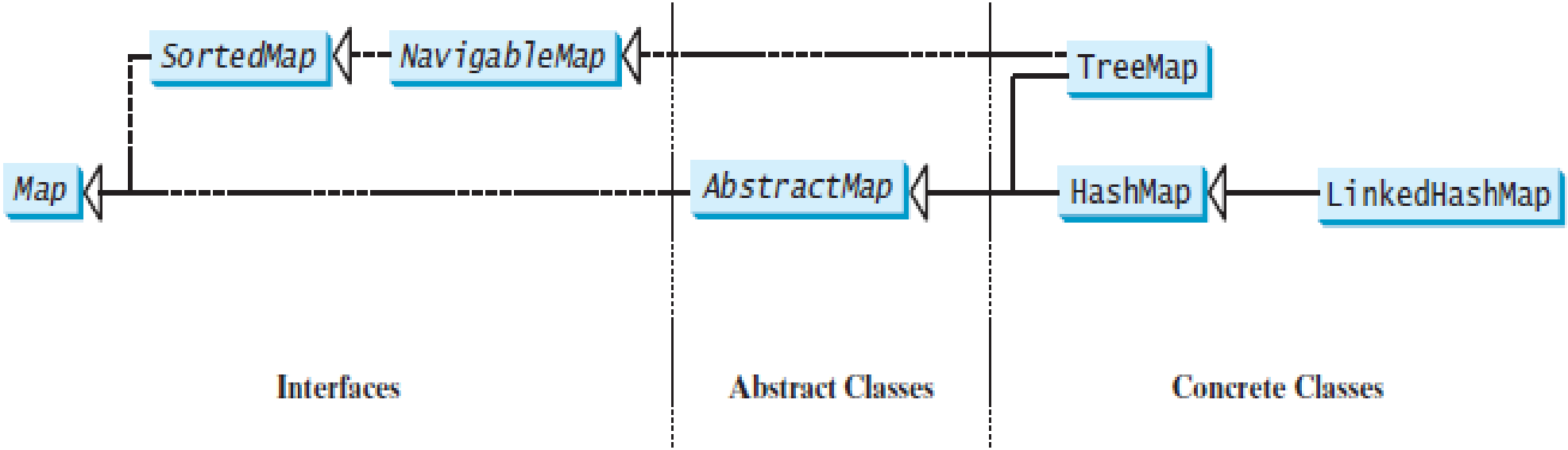
- Each key maps to one value. A key and its corresponding value form an entry stored in a map.

Maps



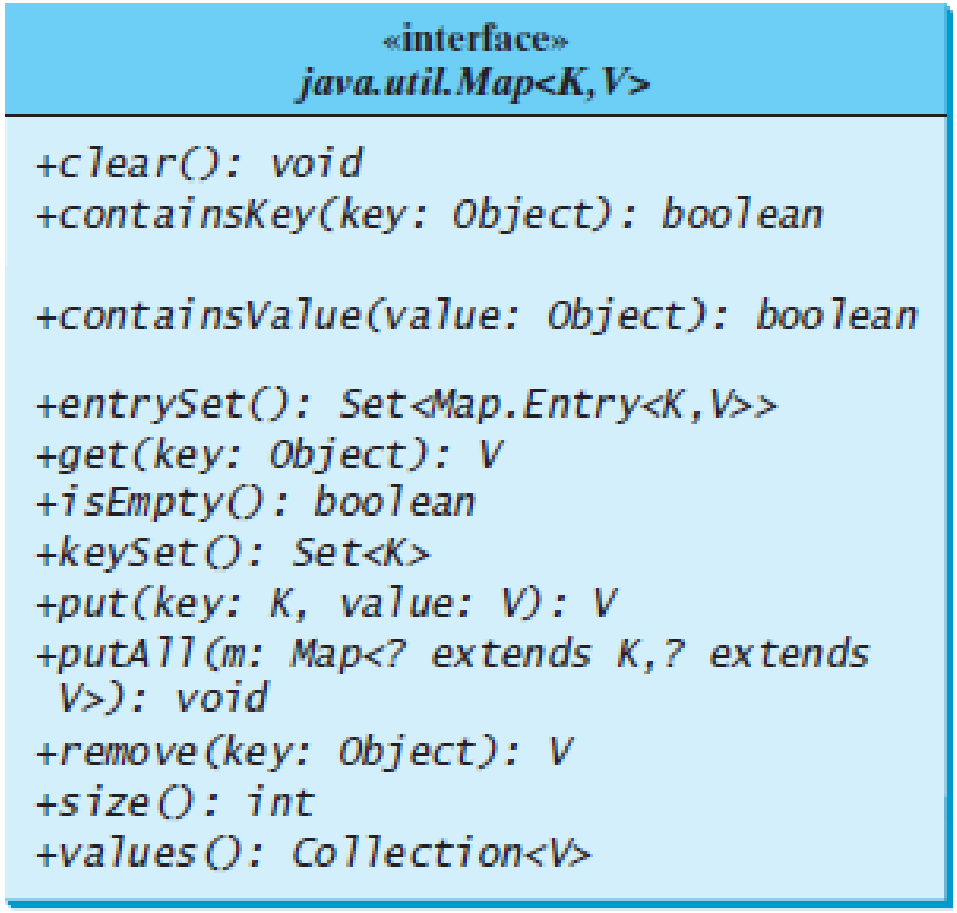
The entries consisting of key/value pairs are stored in a map.

Maps



A map stores key/value pairs.

Maps



Removes all entries from this map.

Returns true if this map contains an entry for the specified key.

Returns true if this map maps one or more keys to the specified value.

Returns a set consisting of the entries in this map.

Returns the value for the specified key in this map.

Returns true if this map contains no entries.

Returns a set consisting of the keys in this map.

Puts an entry into this map.

Adds all the entries from `m` to this map.

Removes the entries for the specified key.

Returns the number of entries in this map.

Returns a collection consisting of the values in this map.

Maps


```
import java.util.*;
public class Main {
public static void main(String[] args) {

Map<String, String> map = new LinkedHashMap<>();

map.put("123", "John Smith");
map.put("111", "George Smith");
map.put("123", "Steve Yao");
map.put("222", "Steve Yao");
System.out.println(map);

}
}
```

Output:
{123=Steve Yao, 111=George Smith, 222=Steve Yao}

Singleton and Unmodifiable Collections and Maps

- You can create singleton sets, lists, and maps and unmodifiable sets, lists, and maps using the static methods in the Collections class.

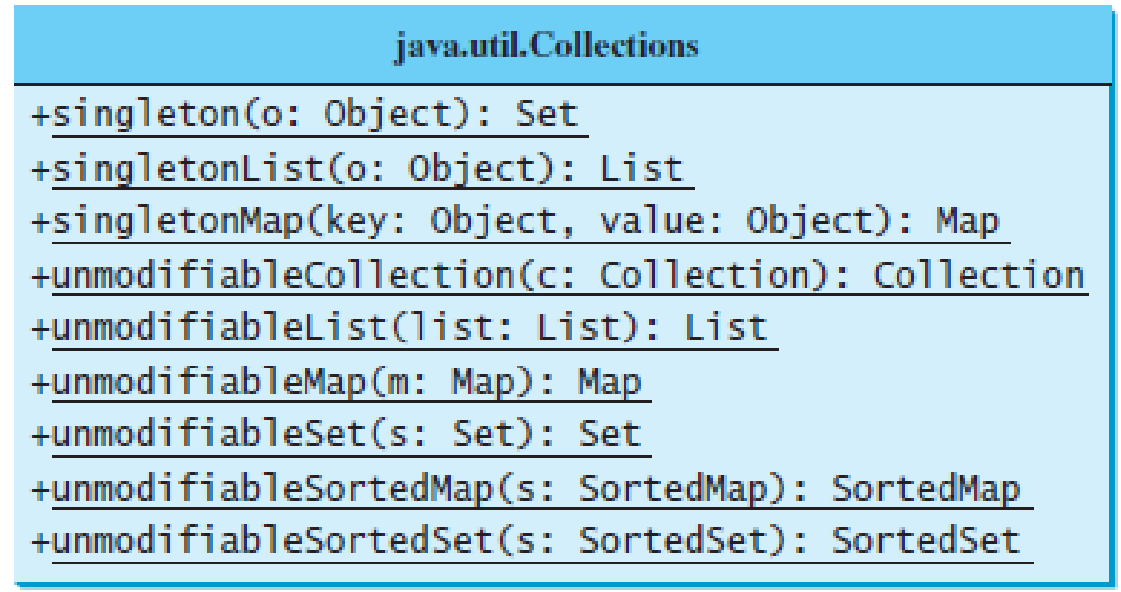
- The Collections class contains the static methods for lists and collections. It also contains the methods for creating immutable singleton sets, lists, and maps, and for creating read only sets, lists, and maps.
- The Collections class defines three constants **EMPTY_SET, EMPTY_LIST, and EMPTY_MAP** for an empty set, an empty list, and an empty map. These collections are immutable.

Singleton and Unmodifiable Collections and Maps

- The class also provides the singleton(Object o) method for creating an immutable **set** containing only a single item, the singletonList(Object o) method for creating an immutable **list** containing only a single item, and the singletonMap(Object key, Object value) method for creating an immutable **map** containing only a single entry.

- It also provides six static methods for returning **read only views** for collections, which we can not modify and if we try then will cause an **UnsupportedOperationException**.

Singleton and Unmodifiable Collections and Maps



Returns an immutable set containing the specified object.
Returns an immutable list containing the specified object.
Returns an immutable map with the key and value pair.
Returns a read-only view of the collection.
Returns a read-only view of the list.
Returns a read-only view of the map.
Returns a read-only view of the set.
Returns a read-only view of the sorted map.
Returns a read-only view of the sorted set.

The Collections class contains the static methods for creating singleton and read-only sets, lists, and maps.

Summary

- List vs Map
- Performance of list and map
- Singleton
- Unmodifiable collections

END OF UNIT - 11