# Collection Framework

# AGENDA

TRAINING CENTER

1 What is collections

2 Hierarchy of collections

3 Interfaces and Classes

4 List, Set and Map

# General information about the collections

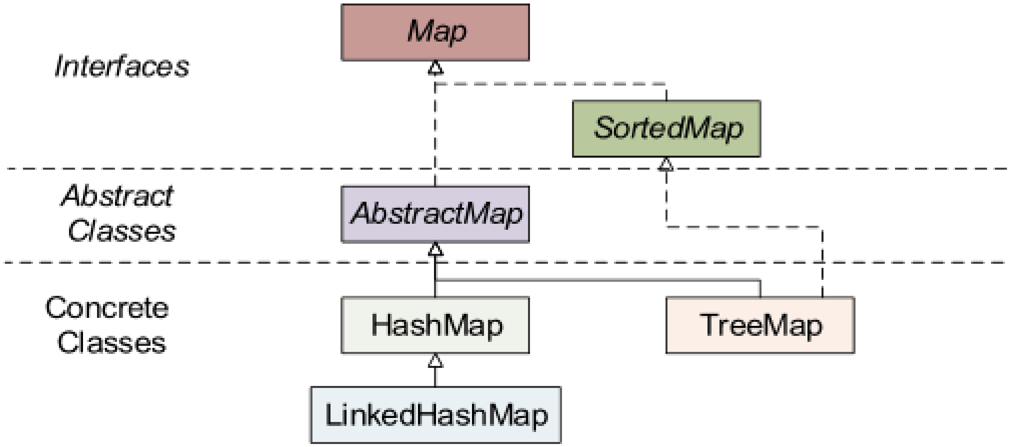
TRAINING CENTER

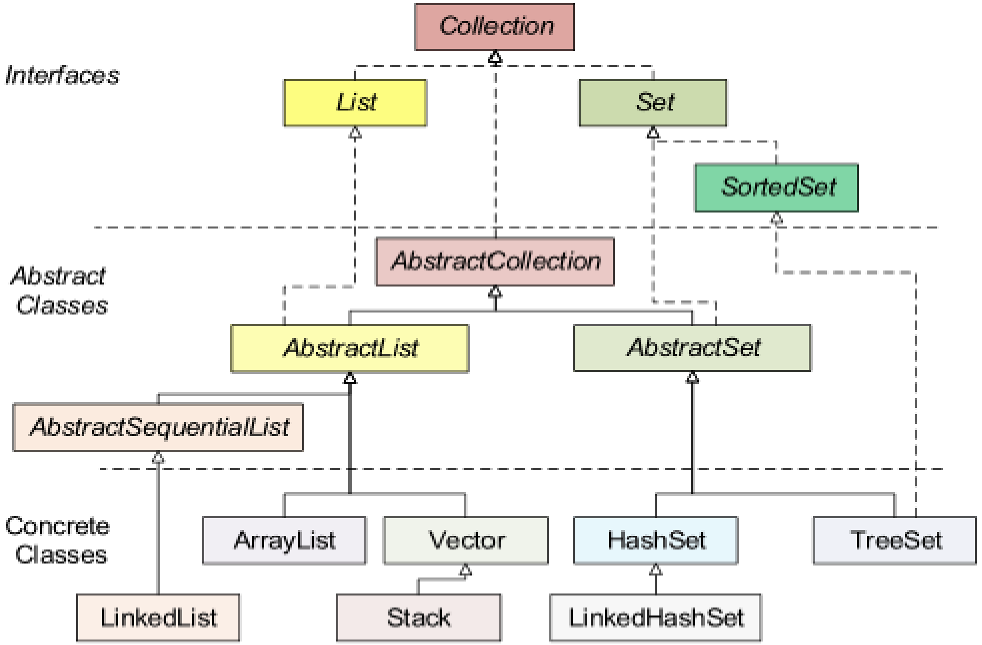
- **Collections** - storage, supporting a variety of ways of accumulation and ordering of objects in order to allow efficient access to them.

- Main operations are supported:
  - ✓ Adding a new item to the collection;
  - ✓ Removing an item from the collection;
  - ✓ Change an item in a collection

- Collections include dynamic arrays, linked lists, trees, sets, hash tables, stacks, queues, etc..

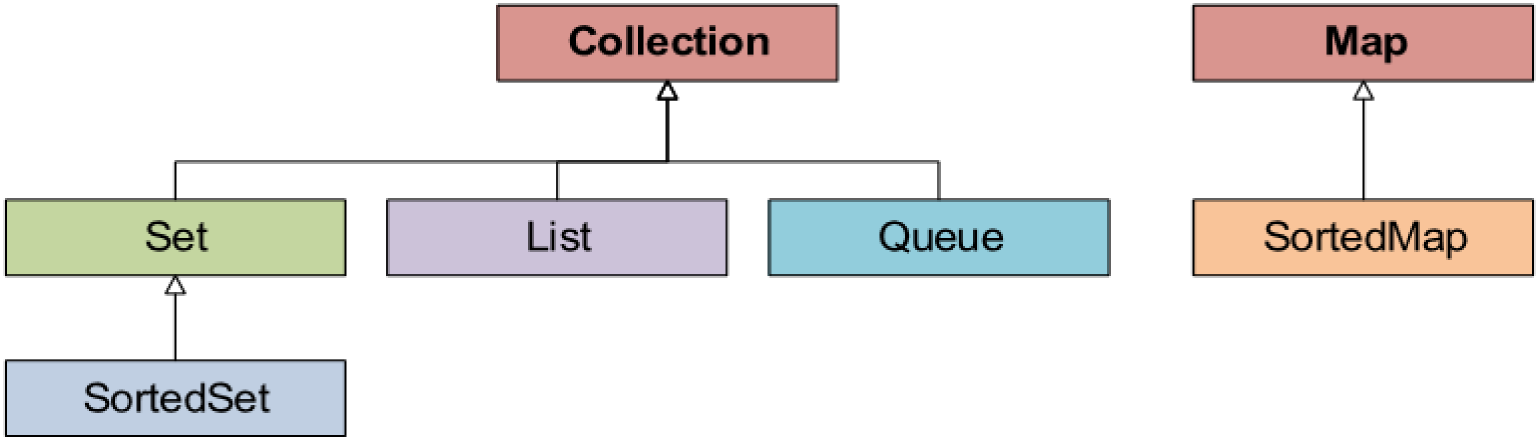
# Hierarchy of collections





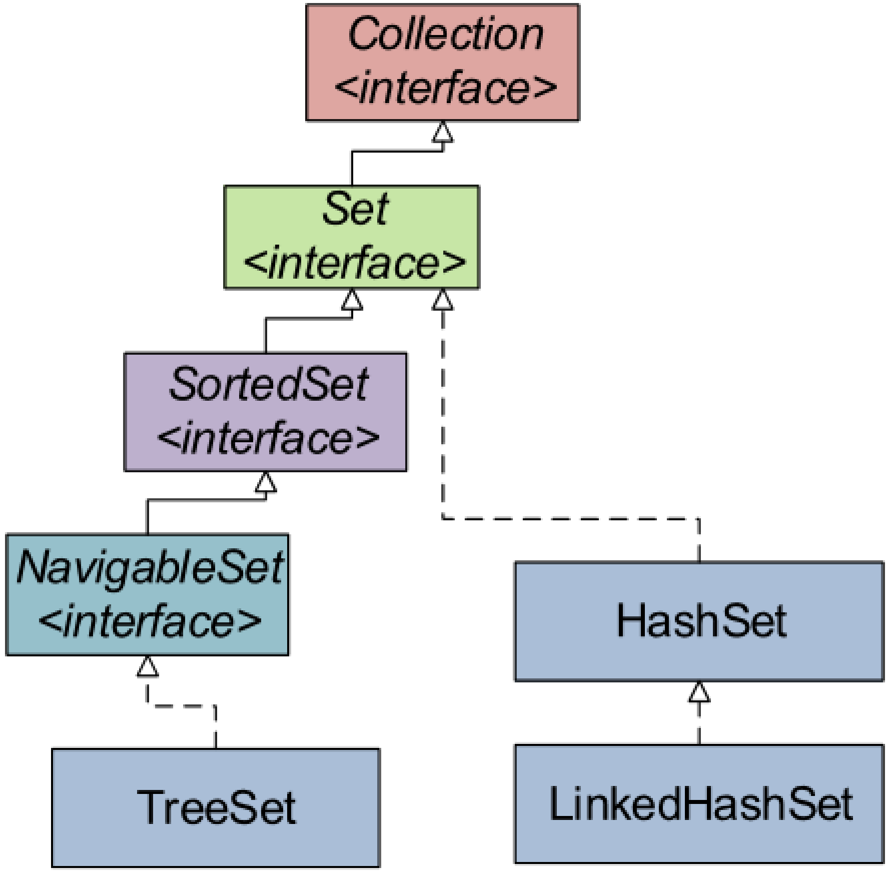
# Hierarchy of collections

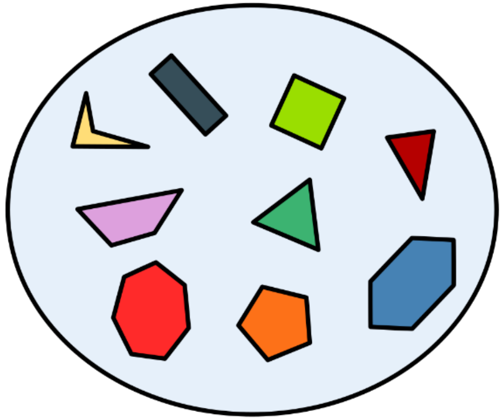
JDK does not provide direct implementations of the Collection and Map interfaces, but there are many implementations of more specific sub-interfaces :



# Sets

**Set** (set) — collection without duplicate elements.



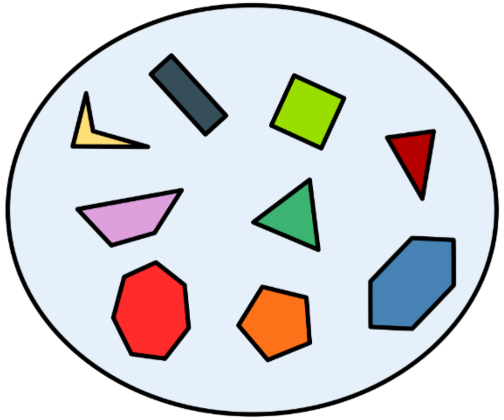


A set of polygons

# Basic Sets

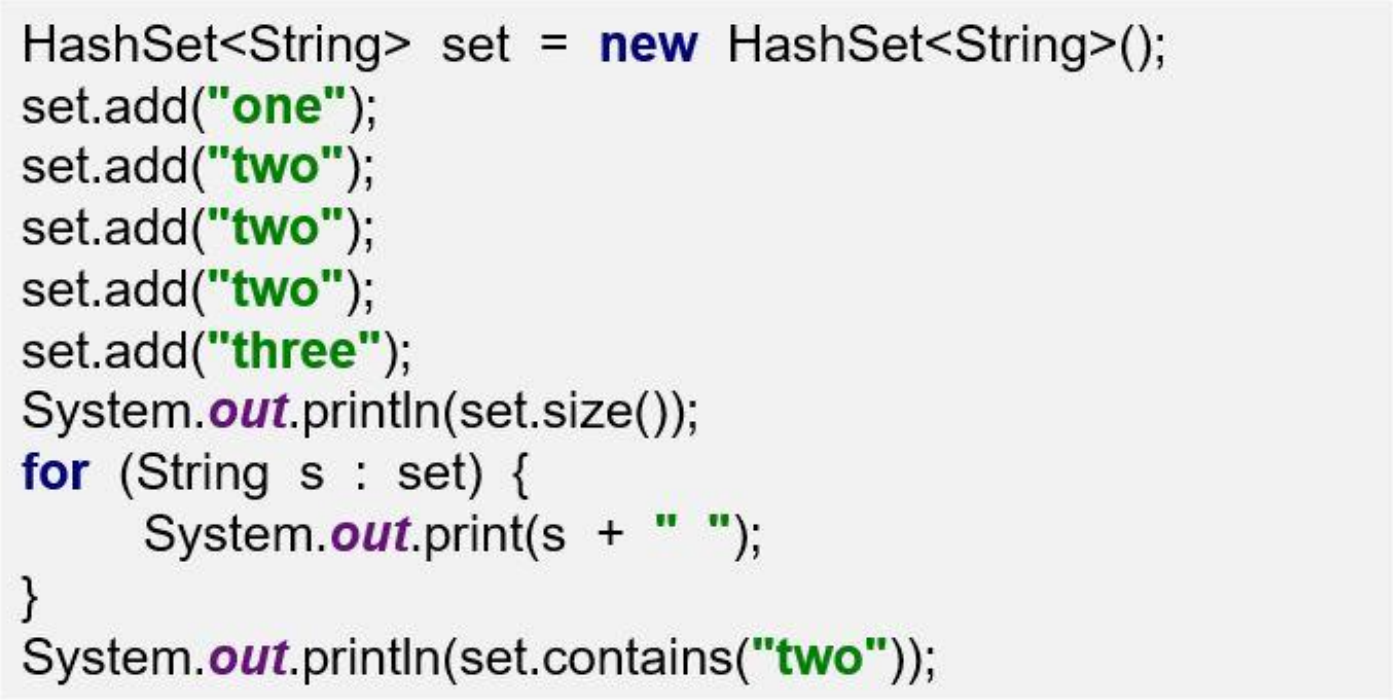
- **HashSet** - set of unordered elements (the sequence of extraction of elements may not coincide with the sequence of their addition).
- **LinkedHashSet —** with preservation of order.
- **CopyOnWriteArraySet** - implementation of the Set interface, which uses the CopyOnWriteArrayList. Thread-safe version of Set.

Two sets are considered equal if they contain the same elements.

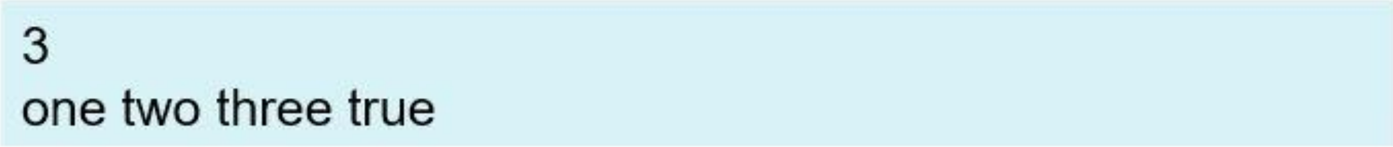


A set of polygons

# Sets: HashSet

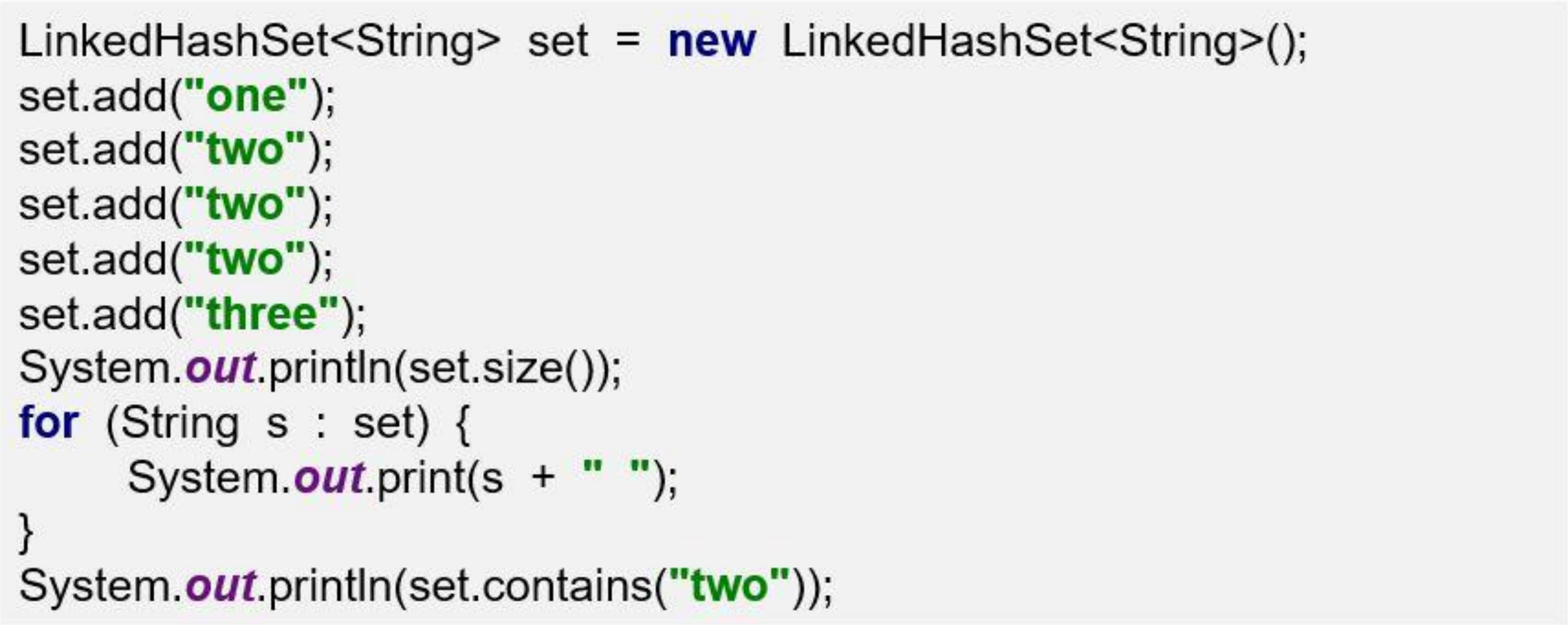


```
HashSet<String>  set  =  new  HashSet<String>();
set.add("one");
set.add("two");
set.add("two");
set.add("two");
set.add("three");
System.out.println(set.size());
for  (String  s  :  set)  {
      System.out.print(s  +  "  ");
}
System.out.println(set.contains("two"));
```

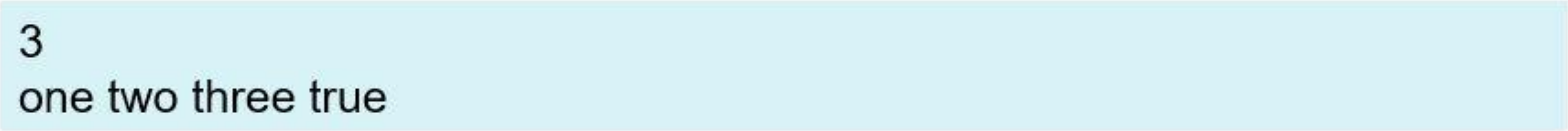


```
3
one two three true
```

# Sets: LinkedHashSet

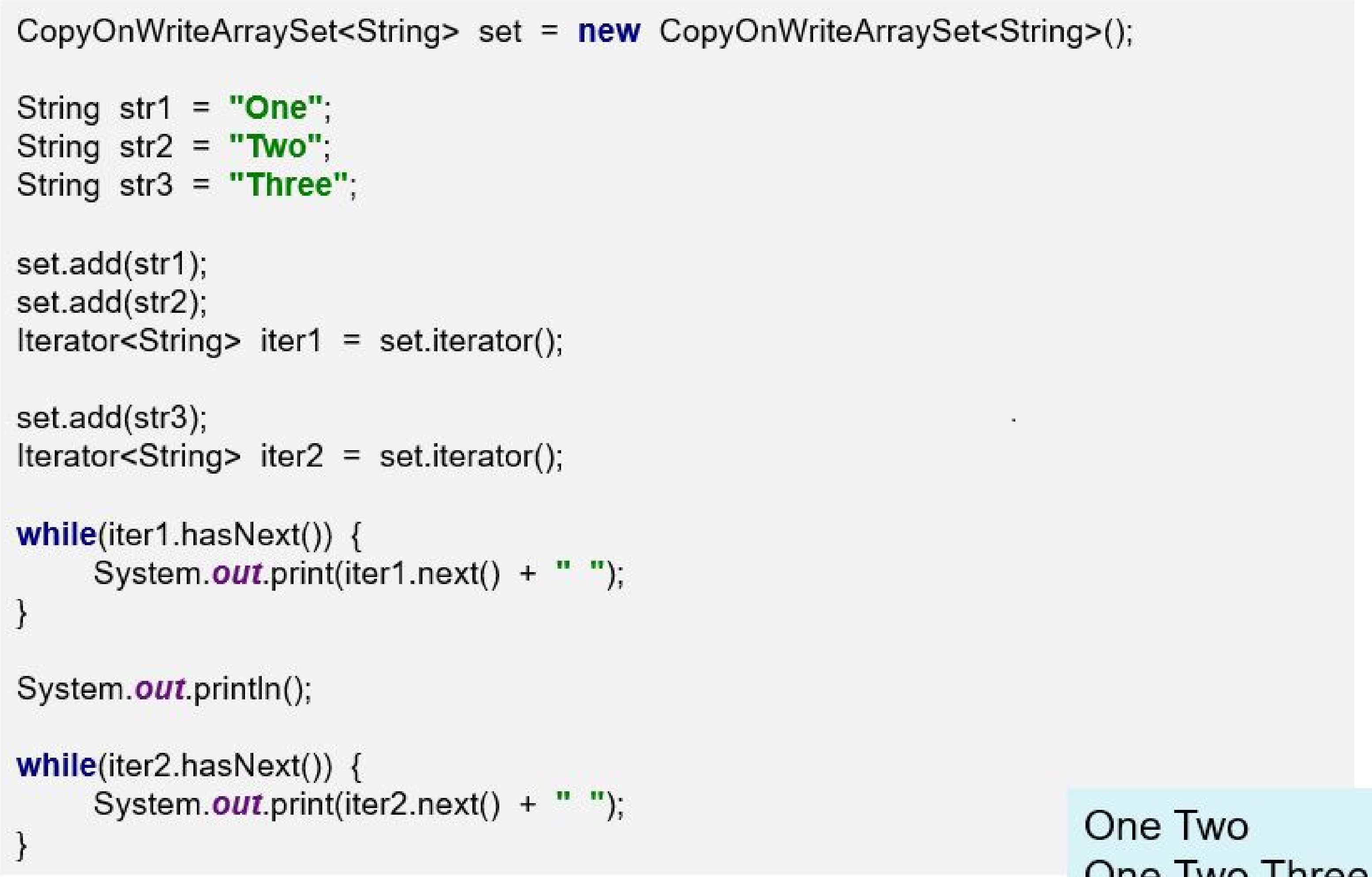


```java
LinkedHashSet<String> set = new LinkedHashSet<String>();
set.add("one");
set.add("two");
set.add("two");
set.add("two");
set.add("three");
System.out.println(set.size());
for (String s : set) {
      System.out.print(s + " ");
}
System.out.println(set.contains("two"));
```



```
3
one two three true
```

# Sets: CopyOnWriteArraySet



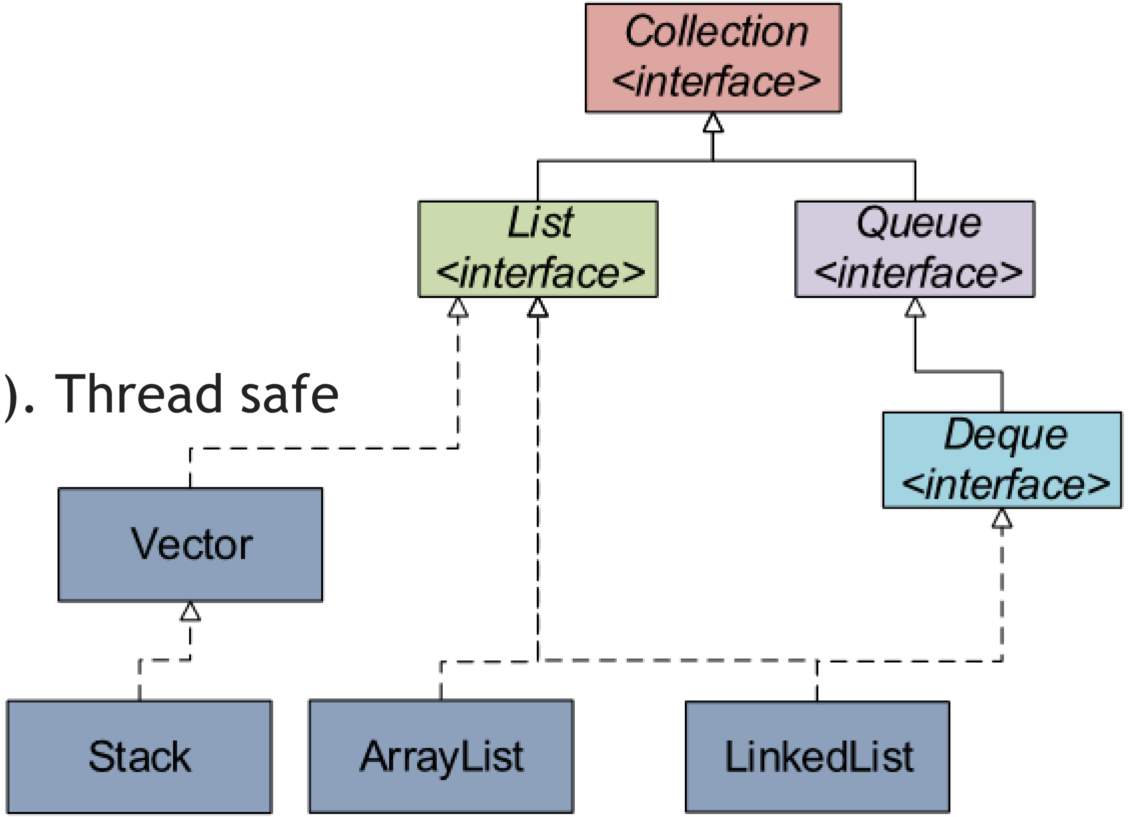
```
CopyOnWriteArraySet<String> set = new CopyOnWriteArraySet<String>();

String str1 = "One";
String str2 = "Two";
String str3 = "Three";

set.add(str1);
set.add(str2);
Iterator<String> iter1 = set.iterator();

set.add(str3);
Iterator<String> iter2 = set.iterator();

while(iter1.hasNext()) {
    System.out.print(iter1.next() + " ");
}

System.out.println();

while(iter2.hasNext()) {
    System.out.print(iter2.next() + " ");
}
```

```
One Two
One Two Three
```

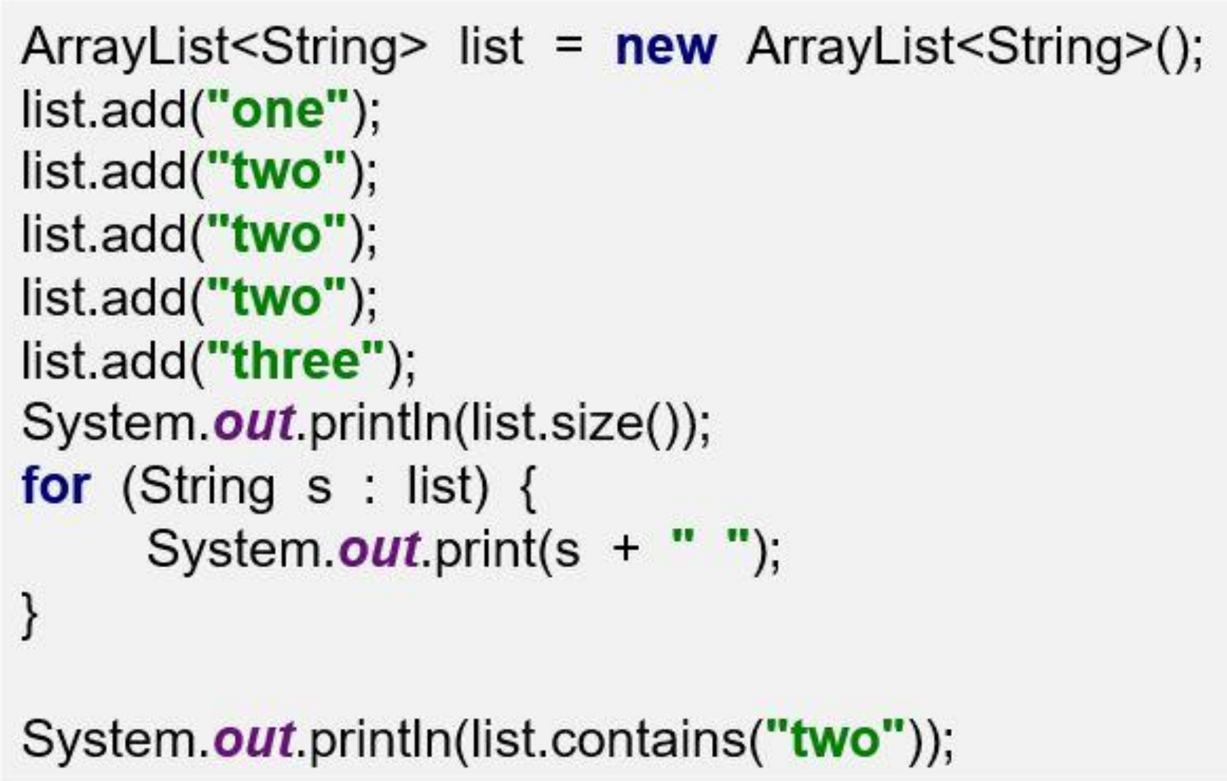
# Lists

**List** is an ordered collection (preserves the sequence of adding elements and allows access to the element by index).

- **ArrayList —** Array-based list.
- **LinkedList —** Doubly-linked list.
- **Stack** – A list of type last-in-first-out (LIFO). Thread safe

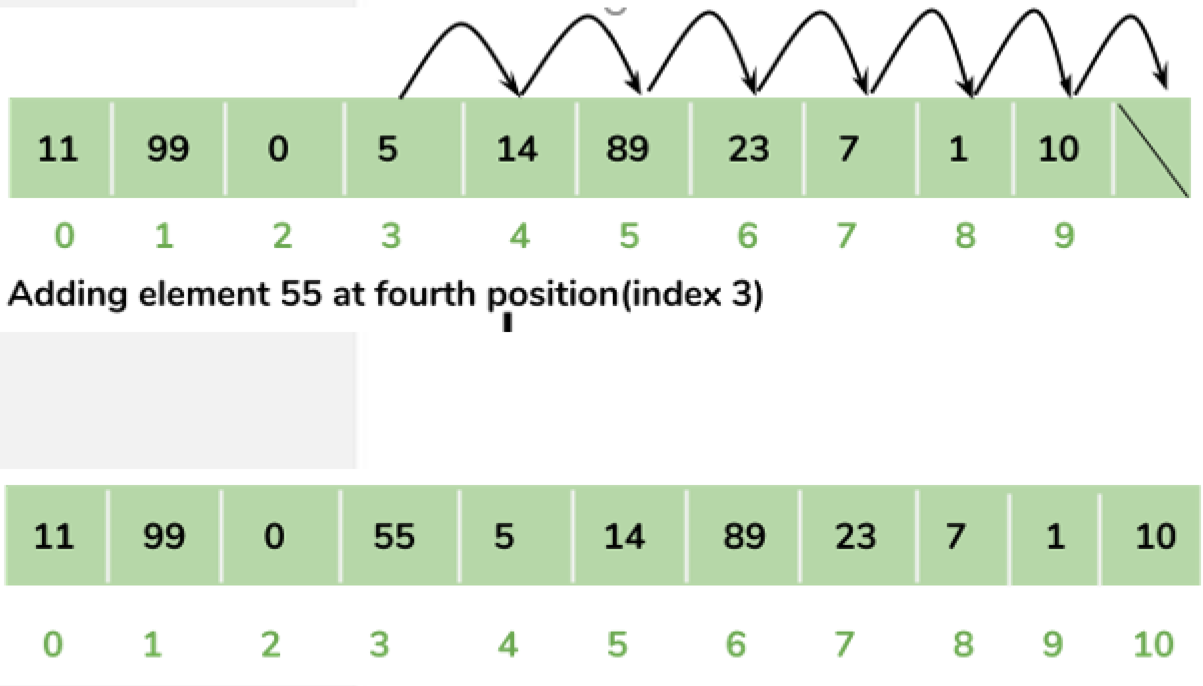


# Lists: ArrayList



```
ArrayList<String> list = new ArrayList<String>();
list.add("one");
list.add("two");
list.add("two");
list.add("two");
list.add("three");
System.out.println(list.size());
for (String s : list) {
        System.out.print(s + " ");
}

System.out.println(list.contains("two"));
```

```
5
one two two two three true
```



| 11 | 99 | 0 | 5 | 14 | 89 | 23 | 7 | 1 | 10 | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | |

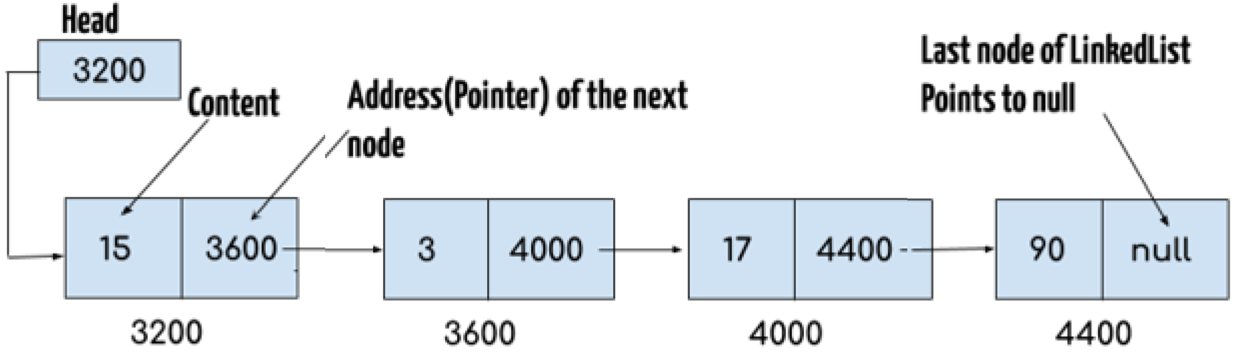
Adding element 55 at fourth position(index 3)

| 11 | 99 | 0 | 55 | 5 | 14 | 89 | 23 | 7 | 1 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

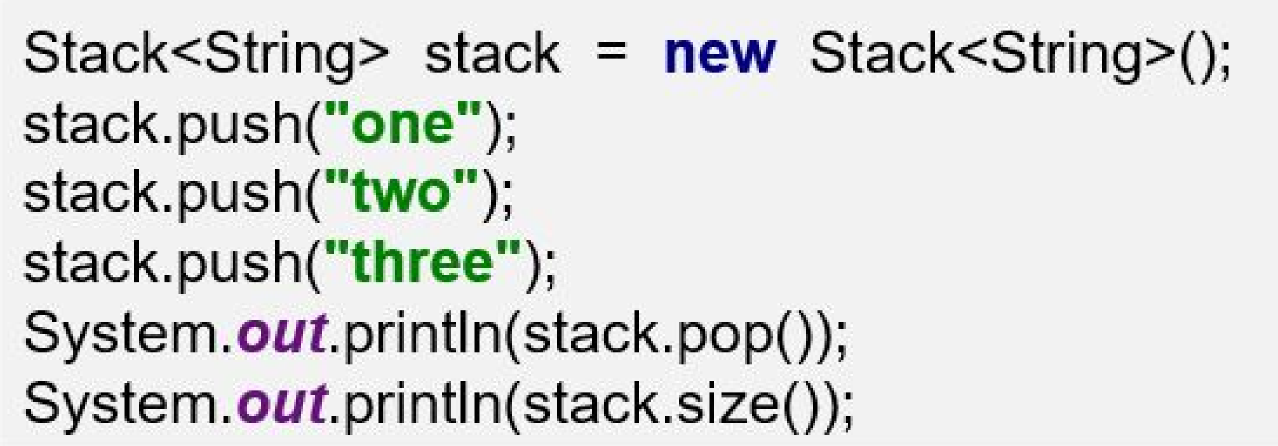
# Lists: LinkedList

```
LinkedList<String> list = new LinkedList<String>();
list.add("one");
list.add("two");
list.addFirst("two");
list.add("two");
list.add("three");
list.addLast("four");
System.out.println(list.size());
for (String s : list) {
        System.out.print(s + " ");
}

System.out.println(list.contains("two"));
```

```
6
two one two two three four true
```

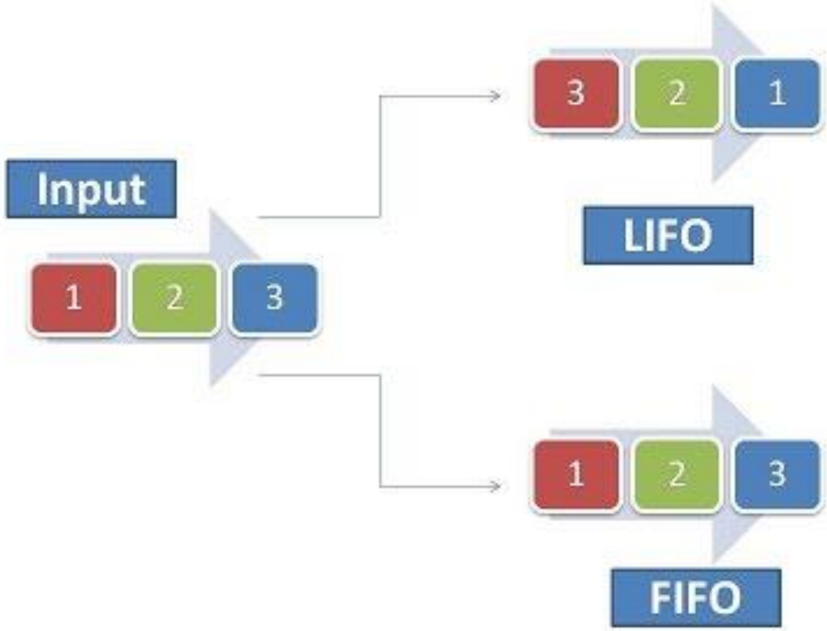

Head
3200
Content
Address(Pointer) of the next node
Last node of LinkedList Points to null
15
3600
3
4000
17
4400
90
null
3200
3600
4000
4400

# Lists: Stack



```
Stack<String> stack = new Stack<String>();
stack.push("one");
stack.push("two");
stack.push("three");
System.out.println(stack.pop());
System.out.println(stack.size());
```

```
three
2
```

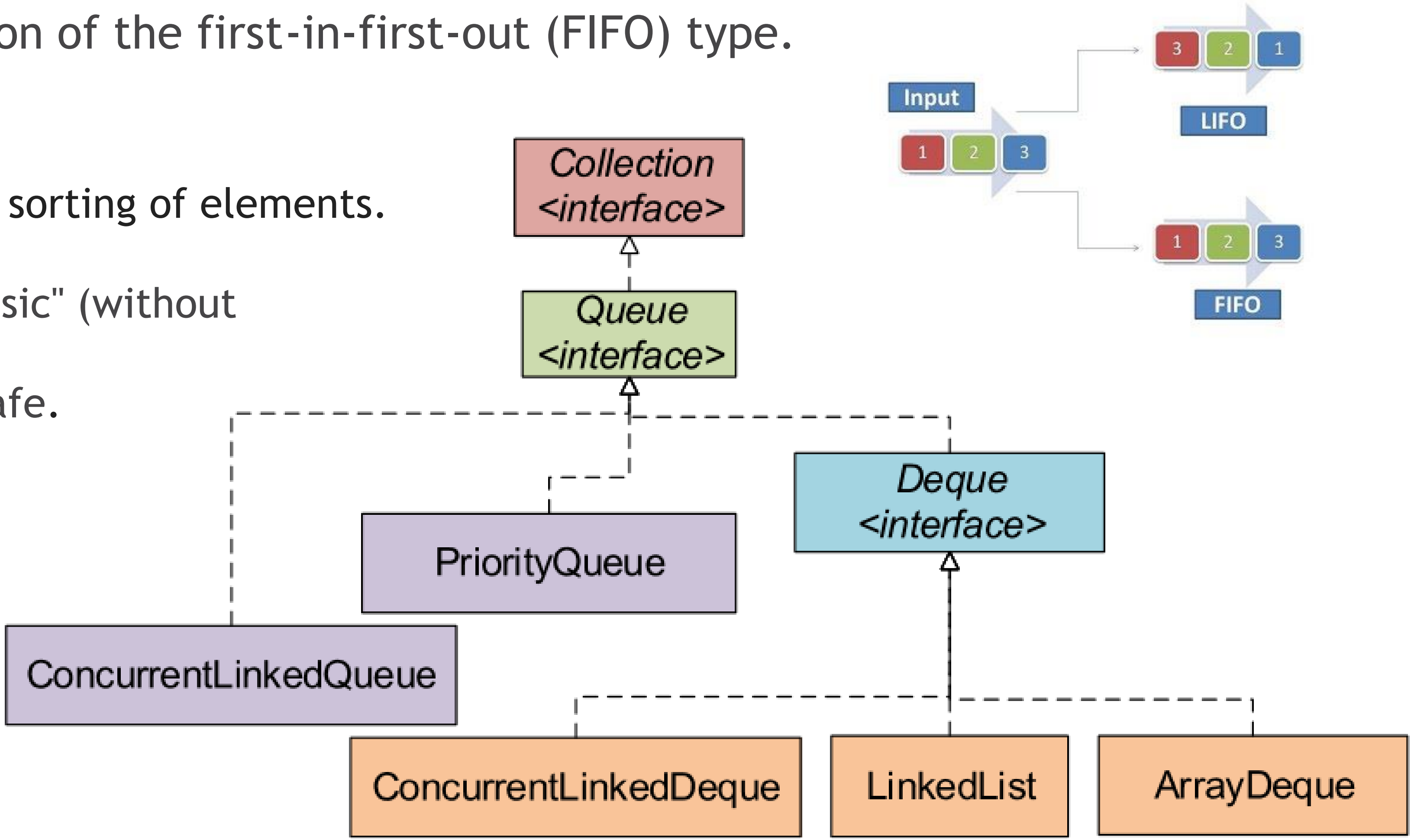


# Queues

- **Queue** is an ordered collection of the first-in-first-out (FIFO) type.

**PriorityQueue** – queue based on sorting of elements.

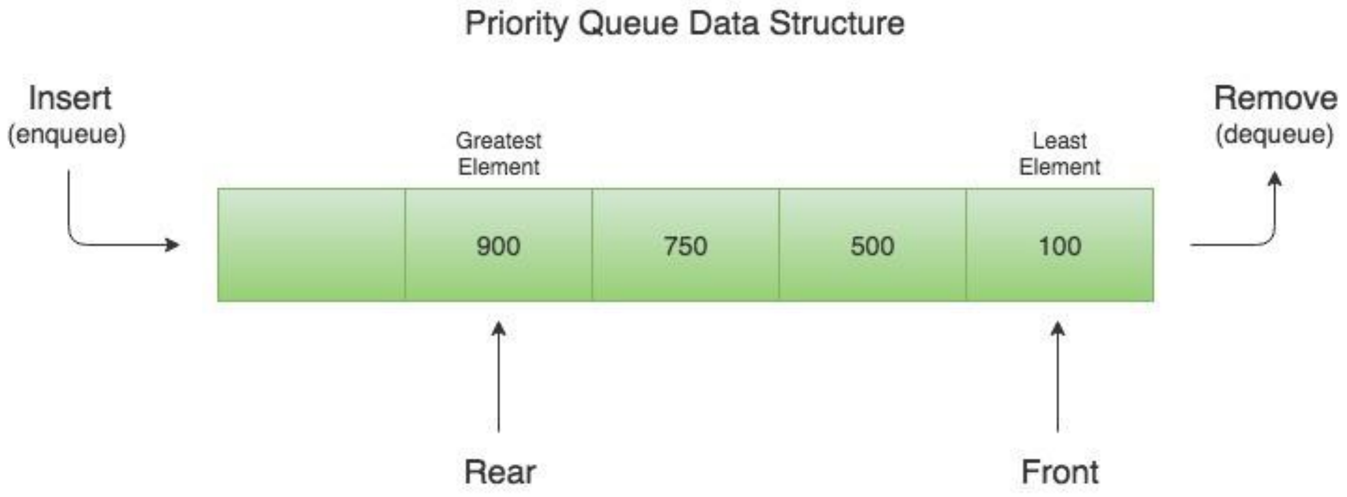
**ConcurrentLinkedQueue** – “classic" (without internal
reorganization) queue, thread-safe.



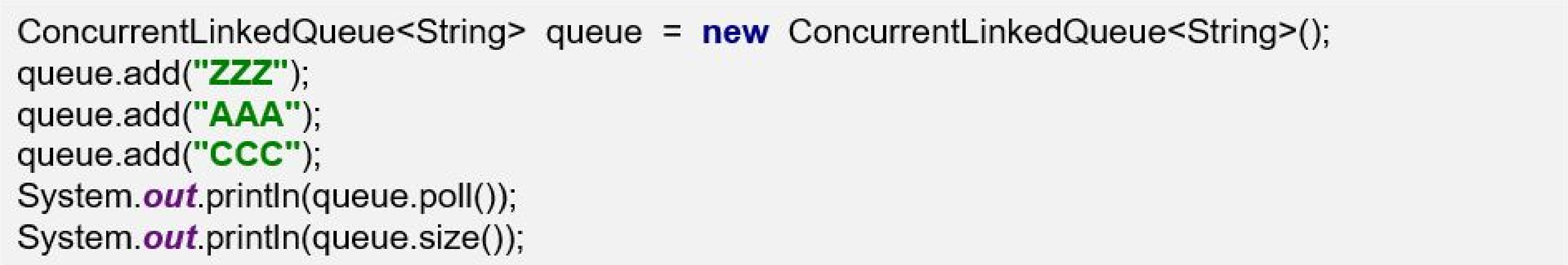
# Queues: PriorityQueue

```
PriorityQueue<String>  queue  =  new  PriorityQueue<String>();
queue.add("ZZZ");
queue.add("AAA");
queue.add("CCC");
System.out.println(queue.poll());
System.out.println(queue.size());
```

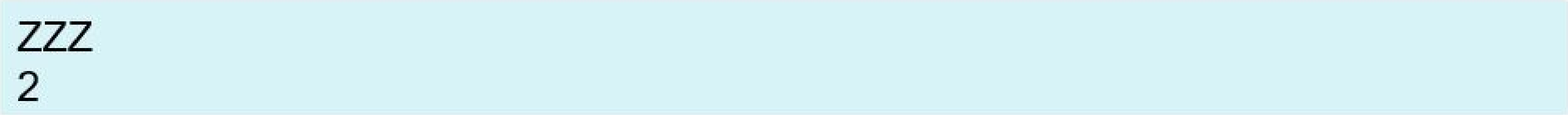
```
AAA
2
```



# Queues: ConcurrentLinkedQueue



```java
ConcurrentLinkedQueue<String> queue = new ConcurrentLinkedQueue<String>();
queue.add("ZZZ");
queue.add("AAA");
queue.add("CCC");
System.out.println(queue.poll());
System.out.println(queue.size());
```

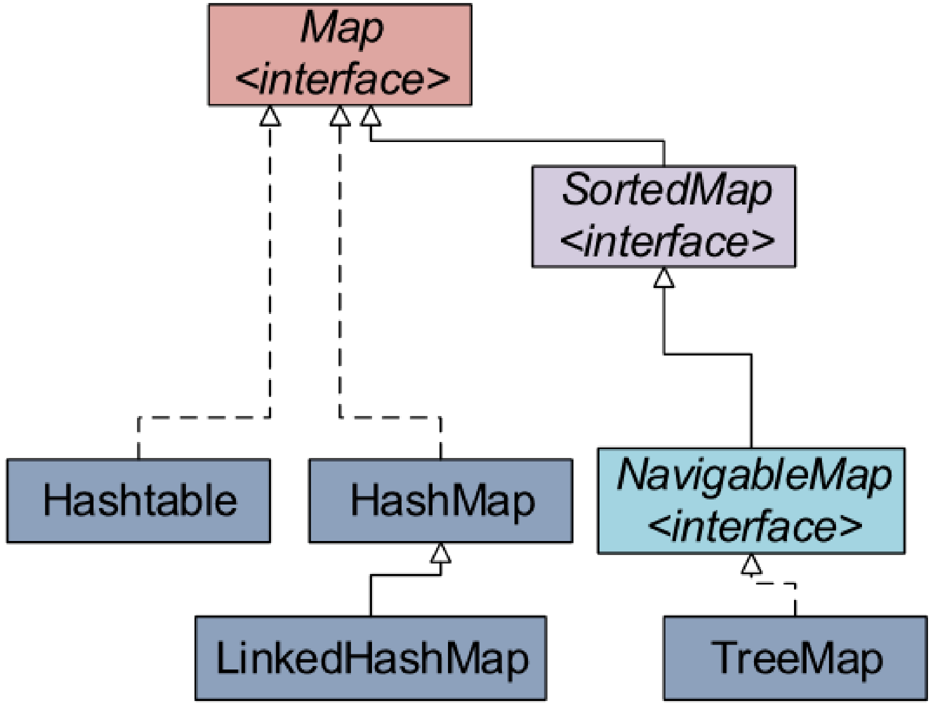


```
ZZZ
2
```

# Maps

- **A map** is a collection that works with sets of pairs of key-value objects (in many other programming languages such a structure is called a "dictionary" or an "associative array".

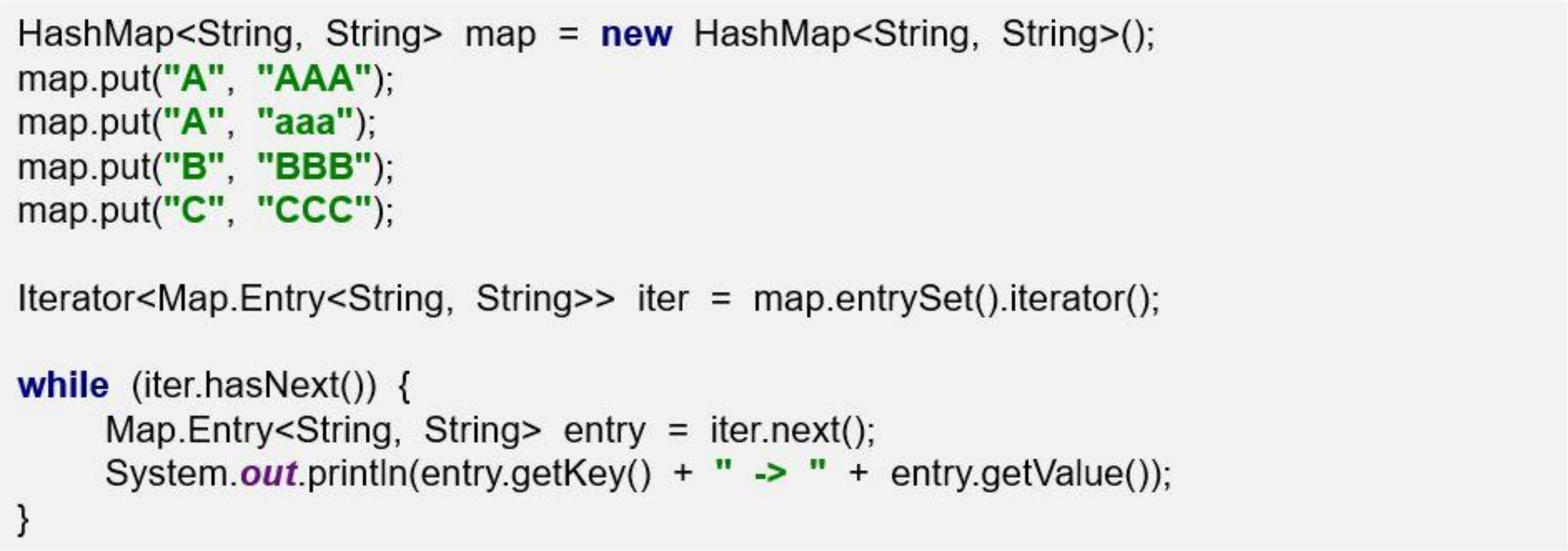
All the keys in the maps are unique. Reusing already existing key value "wipes" an element previously  associated with such a key.



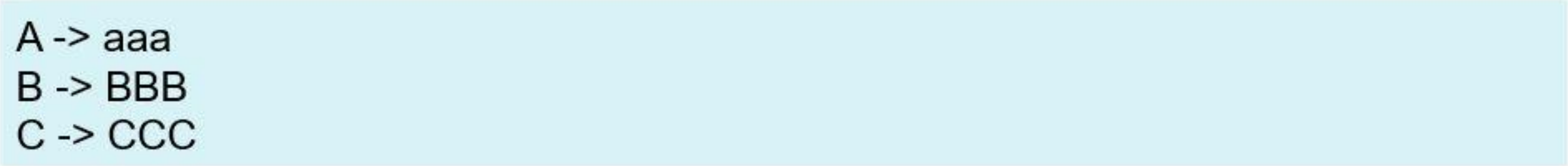
# Basic Maps

- **HashMap** – unsorted and disordered map.
- **LinkedHashMap** – the order in which the elements are stored is determined by the order in which they are added.
- **TreeMap** – stores items in sorted order.
- **Hashtable** – thread-safe collection (analogous to HashMap), the order of the elements in which it is not defined.

# Maps: HashMap

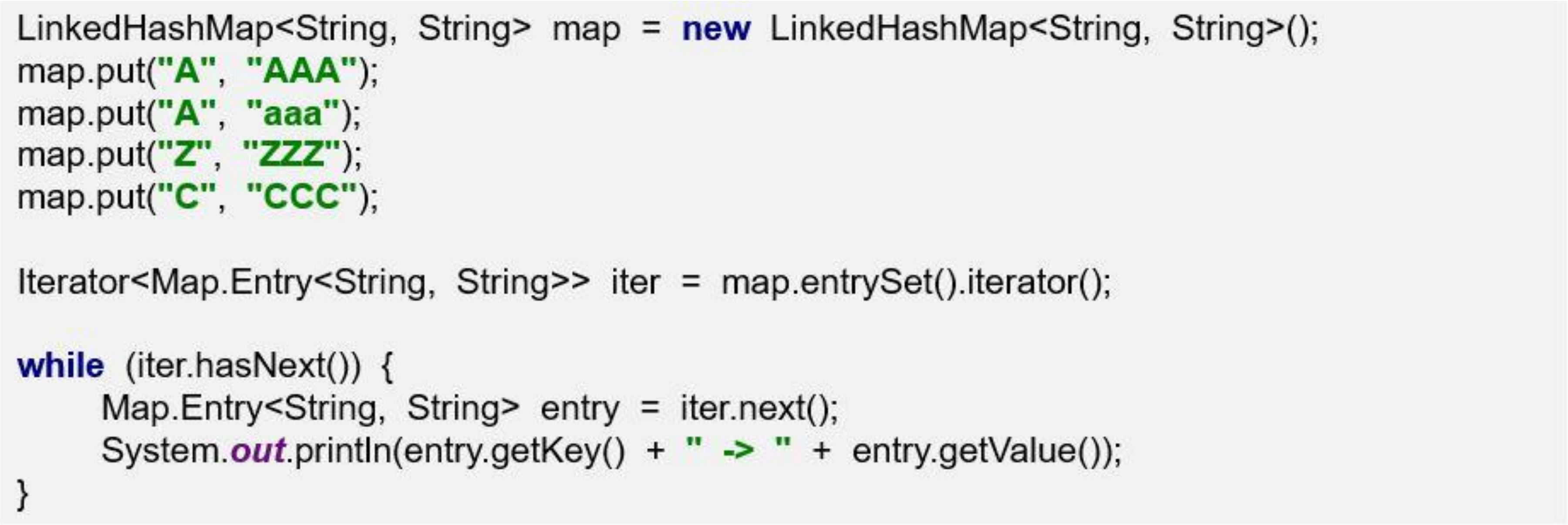


```
HashMap<String, String> map = new HashMap<String, String>();
map.put("A", "AAA");
map.put("A", "aaa");
map.put("B", "BBB");
map.put("C", "CCC");

Iterator<Map.Entry<String, String>> iter = map.entrySet().iterator();

while (iter.hasNext()) {
        Map.Entry<String, String> entry = iter.next();
        System.out.println(entry.getKey() + " -> " + entry.getValue());
}
```



```
A -> aaa
B -> BBB
C -> CCC
```

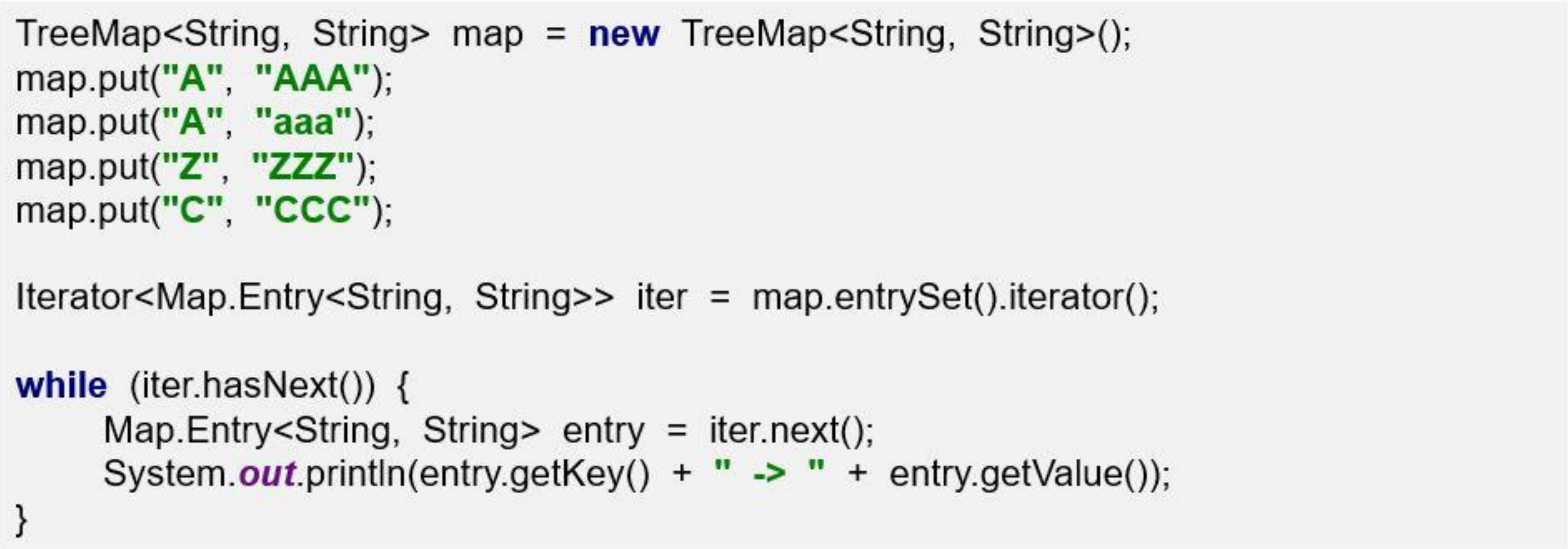
# Maps: LinkedHashMap



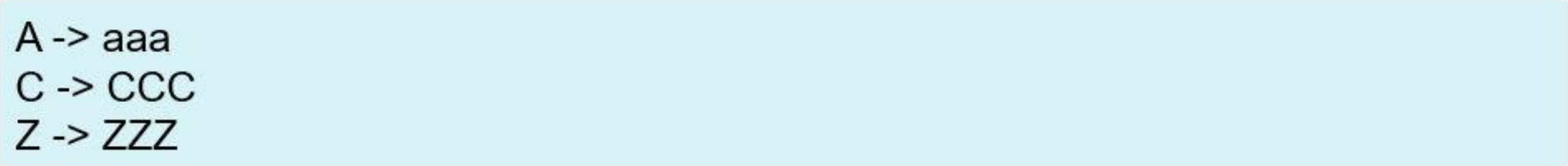
```
LinkedHashMap<String, String> map = new LinkedHashMap<String, String>();
map.put("A", "AAA");
map.put("A", "aaa");
map.put("Z", "ZZZ");
map.put("C", "CCC");

Iterator<Map.Entry<String, String>> iter = map.entrySet().iterator();

while (iter.hasNext()) {
      Map.Entry<String, String> entry = iter.next();
      System.out.println(entry.getKey() + " -> " + entry.getValue());
}
```

```
A -> aaa
Z -> ZZZ
C -> CCC
```

# Maps: TreeMap

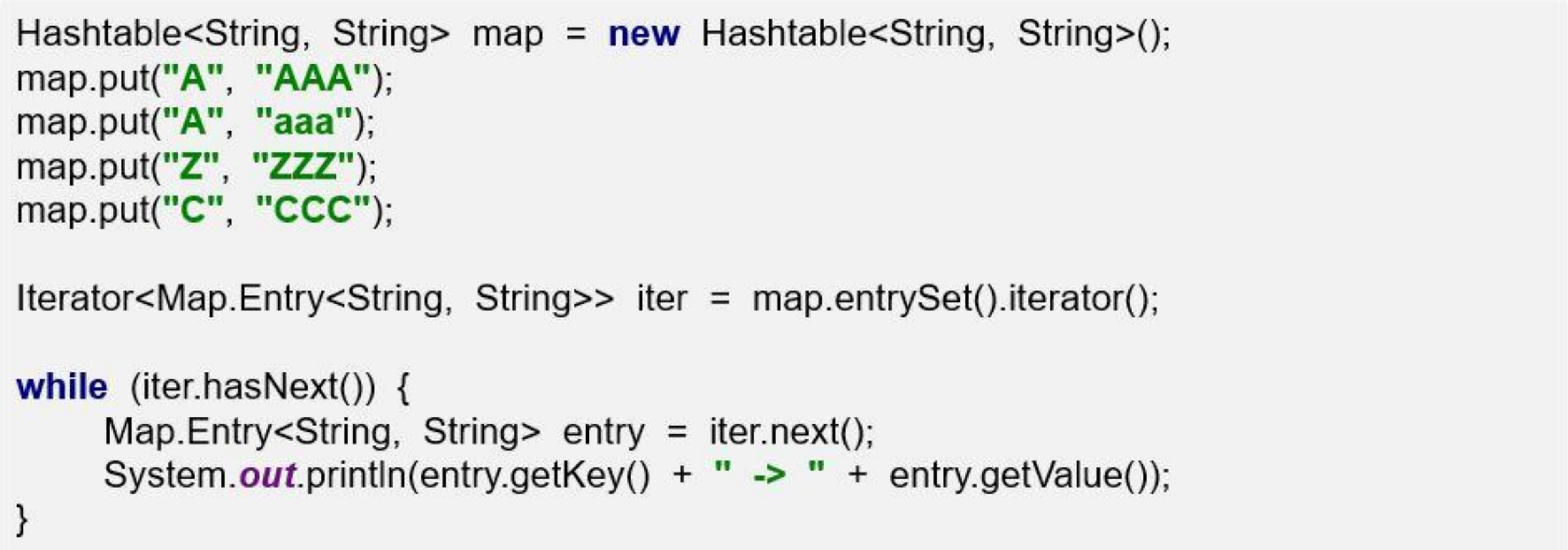


```
TreeMap<String, String> map = new TreeMap<String, String>();
map.put("A", "AAA");
map.put("A", "aaa");
map.put("Z", "ZZZ");
map.put("C", "CCC");

Iterator<Map.Entry<String, String>> iter = map.entrySet().iterator();

while (iter.hasNext()) {
        Map.Entry<String, String> entry = iter.next();
        System.out.println(entry.getKey() + " -> " + entry.getValue());
}
```

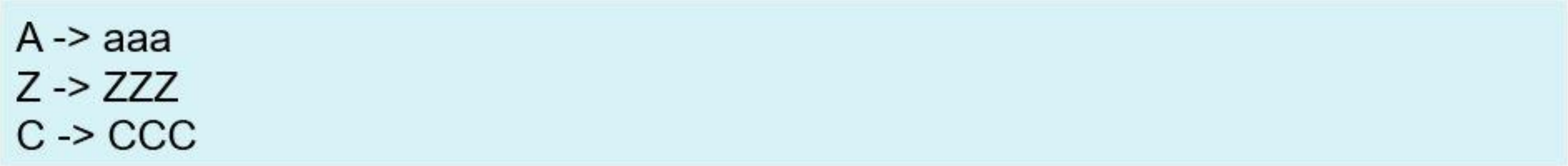


```
A -> aaa
C -> CCC
Z -> ZZZ
```

# Maps: Hashtable



```java
Hashtable<String, String> map = new Hashtable<String, String>();
map.put("A", "AAA");
map.put("A", "aaa");
map.put("Z", "ZZZ");
map.put("C", "CCC");

Iterator<Map.Entry<String, String>> iter = map.entrySet().iterator();

while (iter.hasNext()) {
        Map.Entry<String, String> entry = iter.next();
        System.out.println(entry.getKey() + " -> " + entry.getValue());
}
```

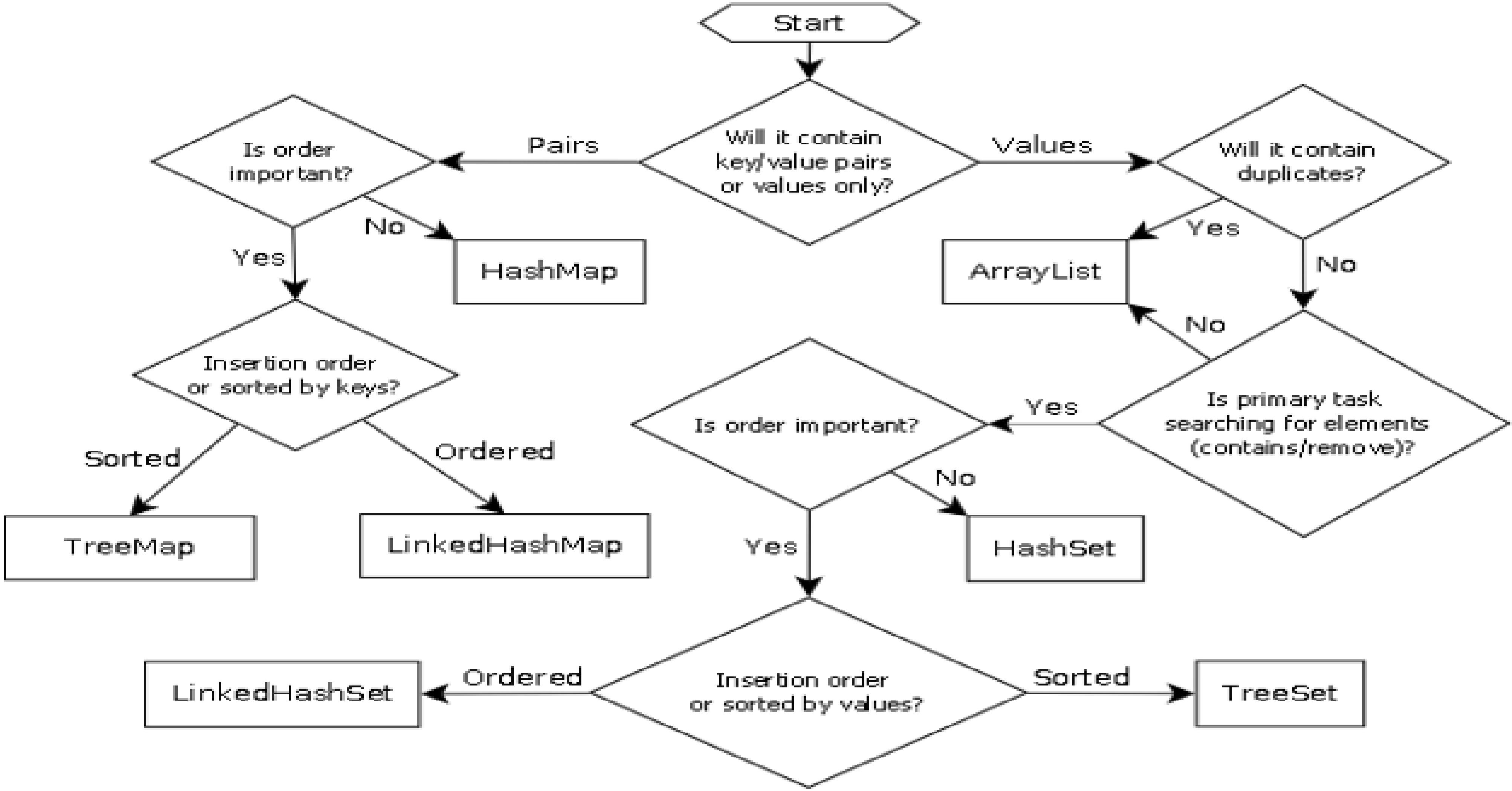


```
A -> aaa
Z -> ZZZ
C -> CCC
```

# Typical operations with collections

- Adding an element;
- Extraction of the element
- Removing an item;
- Pass collection around;
- Sorting the collection.

# Choosing the right collection - WHEN



Start
Will it contain key/value pairs or values only?
Pairs
Values
Is order important?
No
HashMap
Yes
Insertion order or sorted by keys?
Sorted
TreeMap
Ordered
LinkedHashMap
Will it contain duplicates?
Yes
ArrayList
No
Is primary task searching for elements (contains/remove)?
No
Yes
Is order important?
No
HashSet
Yes
Insertion order or sorted by values?
Ordered
LinkedHashSet
Sorted
TreeSet

# Why do we need collection

- When input size is dynamic.
  - When data grows and shrinks frequently.
- Collection framework is nothing but the data structure in Java. So we can use its functionality instead of writing too much code.
- The framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hashtables) are highly efficient.
- The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
  - Whenever you are required to store heterogeneous data.
  - Extending and/or adapting a collection had to be easy.

# Best Practices for Collection

- **Code for Interface, not for Implementation**

By declaring a collection using an interface type, the code would be more flexible as you can change the concrete implementation easily when needed, for example:

```
// Better
 List<String> list = new ArrayList<>();

List<String> list = new LinkedList<>();

// Avoid
ArrayList<String> list = new ArrayList<>();
```

- **Prefer isEmpty() over a size()**

There’s no performance difference between *isEmpty()* and *size()*. The reason is for the readability of the code.

Avoid checking the emptiness of a collection like this:

```
if (listOfEmployees.size() > 0) {
 // dos something if the list is not empty
}
```

Instead, you should use the *isEmpty()* method:

```
if (!listOfEmployees.isEmpty()) {
// dos something if the list is not empty
 }
```

# Best Practices for Collection

- **Return empty collections or arrays, not nulls**

Some APIs intentionally return a null reference to indicate that instances are unavailable. This practice can lead to denial-of-service vulnerabilities when the client code fails to explicitly handle the null return value case.

If a method is designed to return a collection, it should not return *null* in case there’s no element in the collection. Consider the following method:

```
public List<Student> findStudents(String className)
 {
List<Student> listStudents = null;
 if (//students are found//)
{
// add students to the lsit
}
return listStudents;
}
```

Here, the method returns `null` if no student is found. The key point here is a `null` value should not be used to indicate no result. The best practice is, returning an empty collection to indicate no result. The above code can be easily corrected by initializing the collection:

```
List<Student> listStudents = new ArrayList<>;
```

# References

"Collections", Java tutorial

http://java.sun.com/docs/books/tutorial/collections/index.html

# HomeTask

TRAINING CENTER

## Task: Implement a custom collection List with the following features.

- List may grow from zero to infinite size .
- List will be initialized with minimum 10 elements at the time of creation.
- List will provide methods for fetching, adding, removing and printing the list at any state in its lifecycle.

Reference link for creating custom collection:

https://docs.oracle.com/javase/tutorial/collections/custom-implementations/index.html

Task link : https://epa.ms/EPAMPEP2020S6TaskSubmission

**Deadline : 06-March-2020**

# THANK YOU