

JAVA ACADEMY - XIDERAL MONTERREY, N.L



WEEK 2 JAVA'S COLLECTION FRAMEWORK

**Made by:
Luis Miguel Sánchez Flores**

Introduction

In some scenarios, programmers are required to manage a large volume of analogous data in order to work out the task at hand. One option is to use multiple variables of the same datatype to store all the data, but depending on the amount of data one has to work with, it can be cumbersome and difficult to access the data we look for, and would span hundreds (if not thousands) of lines of complex and unreadable code, making it even more difficult to maintain it and cause performance issues. Fortunately, and since its beginnings, Java offers a wide range of data structures to store and manipulate such large amounts of data in an efficient and organized way, called **Collection**.

Collection is a Framework that provides an alternative to using native arrays, which allows the storage of various objects of a certain datatype in only one single list.

It differentiates from a normal array by offering both mutable and immutable options capable of storing homogeneous and heterogeneous types of data and integrating special properties and methods, which allows the development of a flexible and proficient structure that can be tailored to suit the developer's needs.

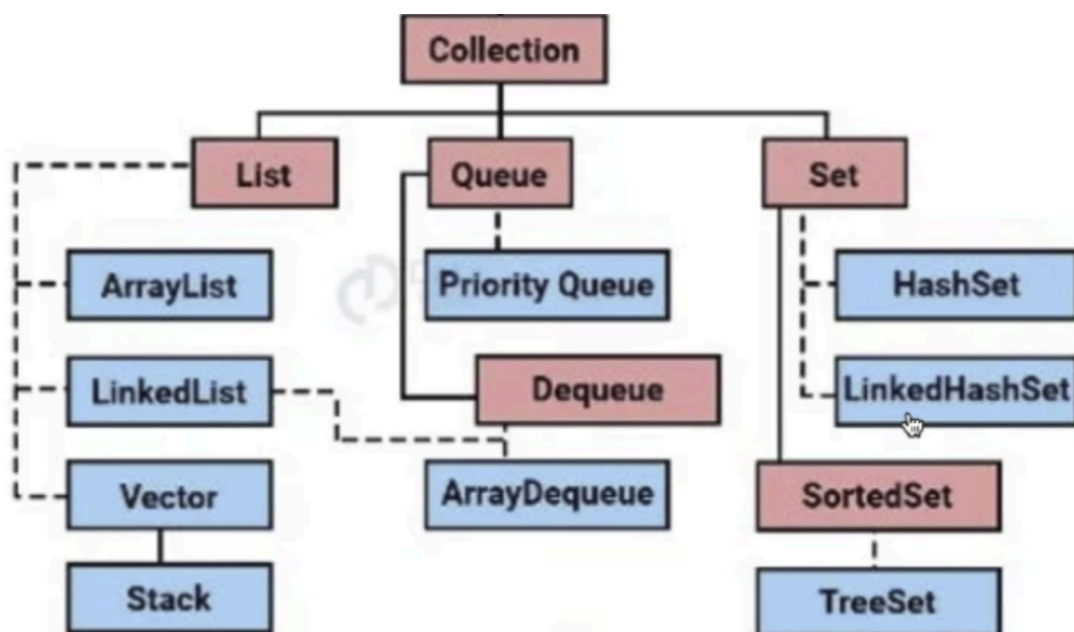
Java provides a variety of Collection classes ready to use, each with their own characteristics and methods, such as **ArrayLists**, **TreeSets** and **HashMaps**. Each class implements a common interface like **Collection**, **List** and **Set**, which defines a series of common functions between the collections.

The Collection hierarchy starts with the **Collection interface**, which establishes the common, generalized methods among each type of Collection, such as **add()**, **remove()** and **size()** for adding, removing and retrieving the

length of the collection, respectively, as well as methods for sorting, searching and filtering elements.

Collections utilize the powers of **Generics** that allows us to specify the data type in which the elements will be stored as. For example, specifying a collection as type Integer will only allow the storage of whole numeric values.

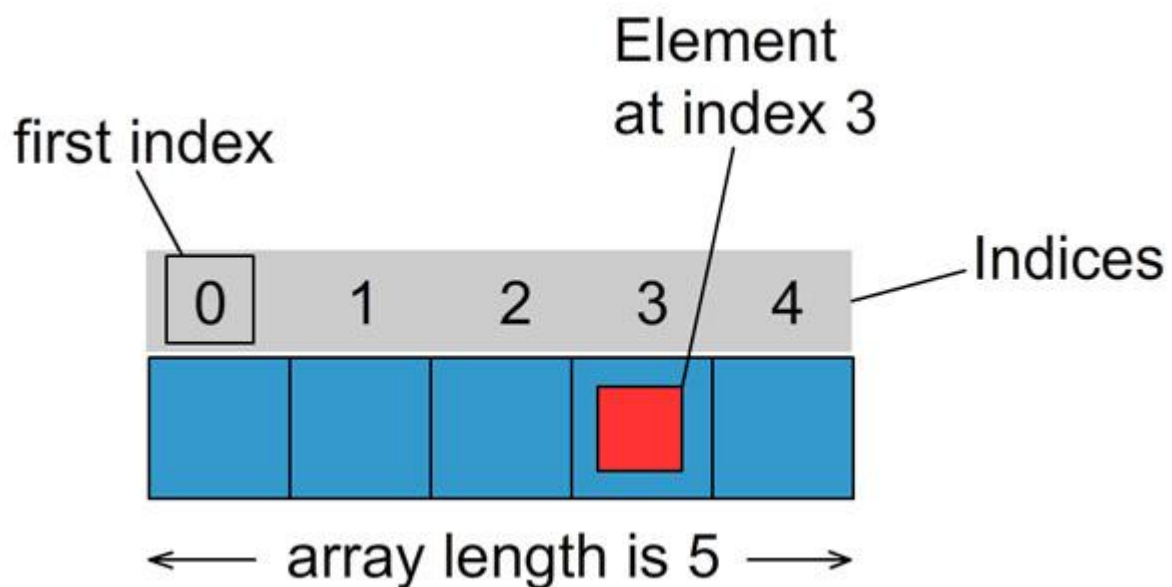
The Collection interface gives place to three types of collections: **List**, **Queue** and **Set**:



LIST Type

The first Collection type included in the framework is List, and it's one of the more commonly used. Lists act similarly like arrays, storing elements of a particular data type while keeping track of their positions with an **index** of integers, starting with 0. Because of this, duplicated elements are allowed. Unlike arrays, some List implementations establish a mutable collection, which allows to add, modify or delete elements in a dynamic set of items.

Lists are used when we need to keep track of a set of objects in an organized manner and when duplicates are allowed.



Some implementations of List include:

- **ArrayList:** A mutable List that grows or shrinks in size when adding or deleting items from it, respectively. It is the best type of List for most scenarios, useful in situations where the number of elements is unknown and it's susceptible to changes. It is worth noting that its resizing process can be costly in terms of memory and time, and can become time-consuming in the case of huge ArrayLists.

- **LinkedList:** It utilizes a double-link structure (by implementing both List and Queue interfaces), which consist of nodes (elements) that only have references for the ones behind and after it, unaware of the other items on the list. This achieves a fast, constant time when adding and deleting elements from the LinkedList, in exchange of additional space in memory and a linear time when accessing the elements, which can require traversing from head to tail of the List. It is best used when frequent updates (adding or removing elements) is necessary.

EXAMPLE CODE OF ARRAYLIST / LINKEDLIST

```
// LIST IMPLEMENTATIONS:
ArrayList<Integer> arraylist = new ArrayList<>(); // Empty list with a initial capacity of 10 items.
LinkedList<Double> linkedlist = new LinkedList<>(List.of(4.15, 2.12, 4.15, 6.3)); // List containing the elements of the provided Collection.

// Add some elements to the ArrayList....
arraylist.add(4);
arraylist.add(10);
arraylist.add(105);
arraylist.add(32);

// Print out the contents and size of each List....
System.out.println("ARRAYLIST : " + arraylist + " - SIZE : " + arraylist.size()
    + "\nLINKEDLIST : " + linkedlist + " - SIZE : " + linkedlist.size()
);

// Print out the element located at index 2 of the ArrayList:
System.out.println("\nVALUE AT INDEX #2 OF ARRAYLIST : " + arraylist.get(2));

// Where is the number 10 located at?
System.out.println("Number 10 is located at index #" + arraylist.indexOf(10));

// Print out each element of the ArrayList individually:
for(Integer i : arraylist)
    System.out.print(i + " ");
```

```
// -----

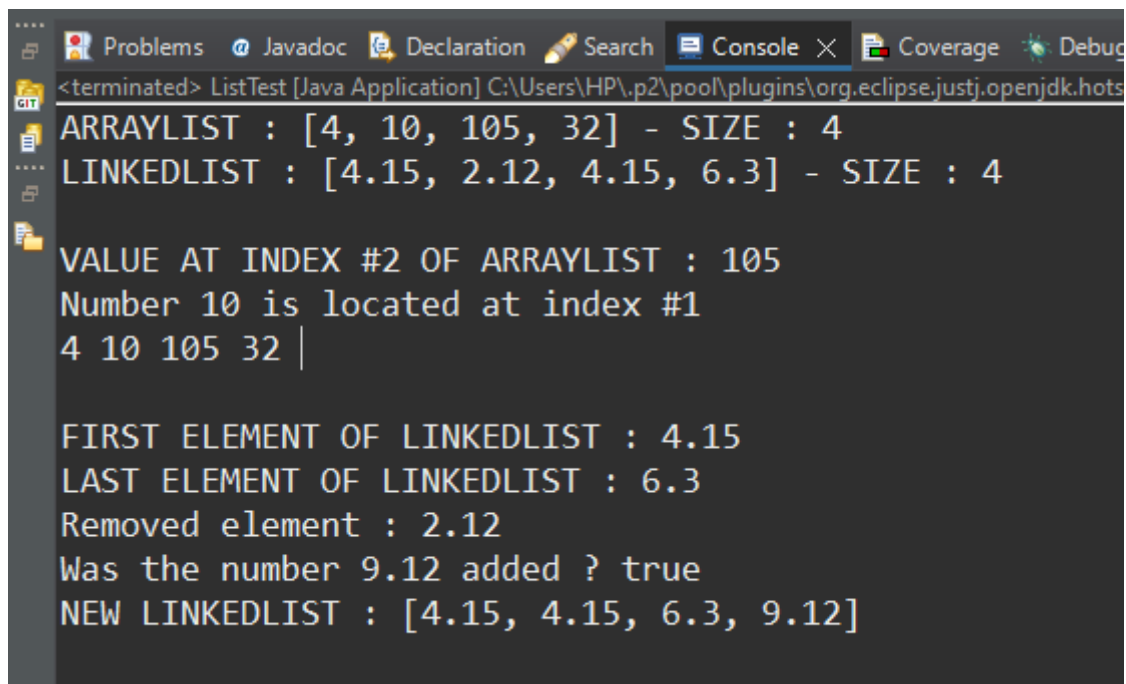
// Take a peek to both the first (HEAD) and last (TAIL) elements in the LinkedList:
System.out.println("FIRST ELEMENT OF LINKEDLIST : " + linkedlist.peekFirst());
System.out.println("LAST ELEMENT OF LINKEDLIST : " + linkedlist.peekLast());

// Remove the element at index #1:
Double removed = linkedlist.remove(1);
System.out.println("Removed element : " + removed);

// Add a element to the end of the linkedlist....
boolean wasAdded = linkedlist.offerLast(9.12);
System.out.println("Was the number 9.12 added ? " + wasAdded);

// Show the modified linkedlist:
System.out.println("NEW LINKEDLIST : " + linkedlist);
```

OUTPUT :



```
<terminated> ListTest [Java Application] C:\Users\HP\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full\jre\bin\java.exe
ARRAYLIST : [4, 10, 105, 32] - SIZE : 4
LINKEDLIST : [4.15, 2.12, 4.15, 6.3] - SIZE : 4

VALUE AT INDEX #2 OF ARRAYLIST : 105
Number 10 is located at index #1
4 10 105 32 |

FIRST ELEMENT OF LINKEDLIST : 4.15
LAST ELEMENT OF LINKEDLIST : 6.3
Removed element : 2.12
Was the number 9.12 added ? true
NEW LINKEDLIST : [4.15, 4.15, 6.3, 9.12]
```

QUEUE Type

Queues are collections that **insert or delete its elements in a specific order**, typically in a **FIFO** (First In, First Out) or **LIFO** (Last in, First Out) fashion.

FIFO prioritizes the elements that came in first, while LIFO places first the most recently added elements of the queue (LIFO is also known as Stacks).

Queues can be viewed as a waiting list of items, each of them awaiting to be the next one removed or peeked at. The next element to be removed or peeked at is called the **HEAD**, while the last element to be looked at is called the **TAIL**. Unlike the List type, Queues don't allow access to the intermediate elements, only the ones in front or back of the list.



Some types of Queue include:

- **Priority Queue:** Type of Queue where the natural ordering of the elements is used to determine the order de las mismas through a Comparable, where the element with the least value gets the highest priority of the queue. A custom Comparable can be used to establish the desired order of the elements.
- **Deque:** Short for “double-ended queue”, it allows the insertion and removal of elements at both ends, implementing a double-link structure. Useful for when developers want to leverage both Queue and Stack functionalities into one to process the elements from both ways.

EXAMPLE CODE OF PRIORITY QUEUE

```
// PRIORITY QUEUE EXAMPLE

// Create new Priority Queue with a Comparator to order the strings by their length in a descending order:
PriorityQueue<String> pq = new PriorityQueue<>((s1, s2)-> s2.length() - s1.length());

// Add some elements to the end of the queue:
pq.offer("Andy");
pq.offer("Laurel");

// How is the queue right now?
System.out.println("PRIORITY QUEUE #1 : " + pq);

// Add more elements to the queue:
pq.offer("Kimi");
pq.offer("Strovanovich");
pq.offer("Eder");

System.out.println("PRIORITY QUEUE #2 : " + pq);

// Remove the first element from the queue:
String removed = pq.poll();
System.out.println("Removed " + removed + " from the queue.");
```

EXAMPLE CODE OF DEQUEUE

```
// Check out the modified queue:
System.out.println("FINAL PRIORITY QUEUE : " + pq);

System.out.println("\n");

// EXAMPLE OF DEQUE (DOUBLE-ENDED QUEUE):

// Create a simple Deque object with no items:
Deque<Integer> dq = new ArrayDeque<>();

// We can add items through the common function that is add()...
dq.add(1);
dq.add(5);

System.out.println("DEQUE #1 : " + dq);

// ...or we can specify if the items should be placed at the HEAD or TAIL of the queue:
dq.addFirst(10);
dq.addLast(20);

System.out.println("DEQUE #2 : " + dq);

// Now remove the last element of the queue...
Integer removedNum = dq.pollLast();
System.out.println("Number " + removedNum + " was removed");
```

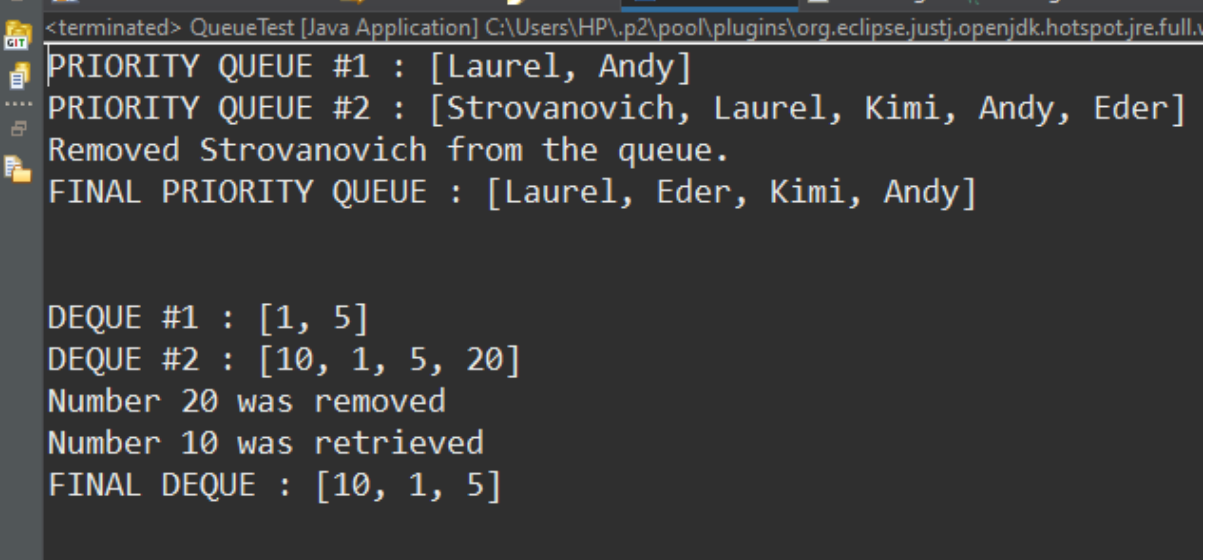


```
// Retrieve, but don't delete, the first element of the queue...
Integer firstNum = dq.peekFirst();
System.out.println("Number " + firstNum + " was retrieved");

// Check the new Dequeue...

System.out.println("FINAL DEQUE : " + dq);
```

OUTPUT:

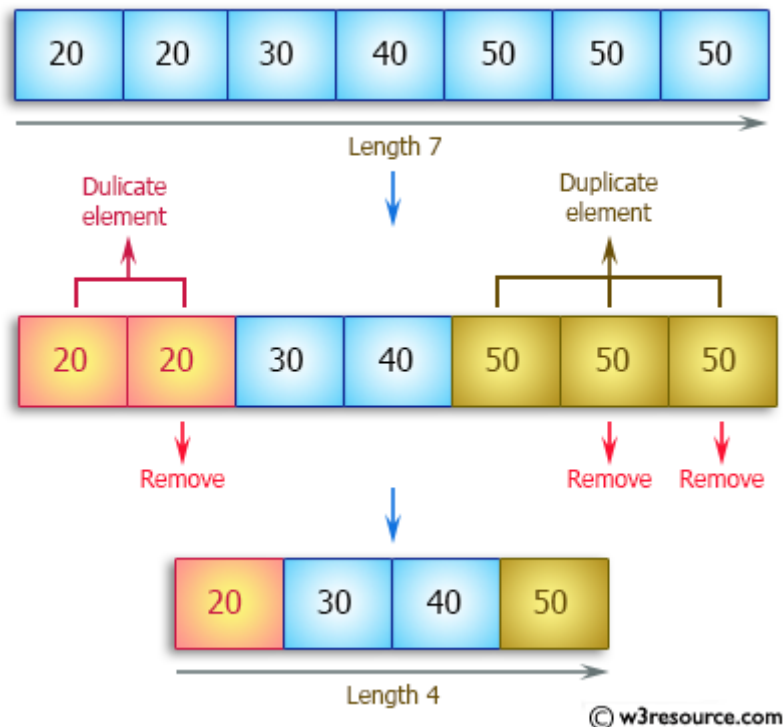


```
<terminated> QueueTest [Java Application] C:\Users\HP\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.v
PRIORITY QUEUE #1 : [Laurel, Andy]
PRIORITY QUEUE #2 : [Strovanovich, Laurel, Kimi, Andy, Eder]
Removed Strovanovich from the queue.
FINAL PRIORITY QUEUE : [Laurel, Eder, Kimi, Andy]

DEQUE #1 : [1, 5]
DEQUE #2 : [10, 1, 5, 20]
Number 20 was removed
Number 10 was retrieved
FINAL DEQUE : [10, 1, 5]
```

SET Type

Another type of collection included in the framework is **Set**, which stores a series of unique objects without a particular order, meaning that, unlike Lists or Queues, a Set **does not store duplicate items in its collection**. As such, Sets establishes additional stipulations on inherited Collection methods that prohibits the addition of duplicates, as well as onto the equals() and hashCode() methods that strictly verifies if two elements are identical or not.



Java offers concrete Set implementations such as:

- **HashSet:** Applies hash values from the **hashCode()** methods to store its elements and offers a constant time performance for basic operations, making it the most commonly used Set implementations thanks to this performance. It is with great importance to set up an appropriate size for the HashSet during its initialization, so as to not affect the iteration process for a prolonged amount of time.
- **LinkedHashSet:** Variant of the HashSet that establishes a double-link list that goes through all the entries and defines an order of insertion. Although they're not often used, it can be ideal to keep a set of unique elements ordered by insertion. but they can become costly in terms of performance, compared to the HashSet.
- **TreeSet:** Stores the unique elements in a **value-based order**, either by the natural ordering or through the use of a Comparator when initializing

the TreeSet. This implementation guarantees a performance of $\log(N)$ in its basic operations thanks to the tree structure selected to store the elements.

EXAMPLE CODE OF LINKED HASH SET

```
// LINKED HASH SET EXAMPLE

// Initialize the hashSet with unique floating values...
LinkedHashSet<Double> numbers = new LinkedHashSet<>(List.of(3.14, 59.23, 31.675, 21.90, 1.09, 0.521, -5.4));

// Print the Set...
System.out.println(numbers);

// Check if PI is in there somewhere...
System.out.println("Is PI (3.14) there ? " + numbers.contains(3.14));

// What about all these numbers?
List<Double> numbersToCheck = List.of(21.90, 31.695, 59.23, 0.52);
System.out.println("Are all the following numbers : " + numbersToCheck + ", available in the set ? " + numbers.containsAll(numbersToCheck));

// Which is the last number ?
System.out.println("LAST NUMBER #1: " + numbers.getLast());

// Get that negative number out of here!
numbers.removeLast();

System.out.println("LAST NUMBER #2: " + numbers.getLast());

// How about in reverse...
System.out.println("REVERSED SET : " + numbers.reversed());
```

EXAMPLE CODE OF TREESET

```
// TREESET EXAMPLE

// Creating a TreeSet for unique items in a sorted way (This case, order by Strings.length()):
TreeSet<String> ts = new TreeSet<>((s1, s2) -> s1.length() - s2.length());

// Add some names...
ts.add("Dante");
ts.add("Sergio");
ts.add("Mat");

System.out.println("TREESET #1 : " + ts);

// What if we add another name with the same length?
ts.add("Erick");
System.out.println("TREESET #2 : " + ts); // <- Erick gets ignored :(

// But Vinicius (length 8) is not a problem:
ts.add("Vinicius");
System.out.println("TREESET #3 : " + ts);

// Print out the elements between "Dante" (inclusive) and "Vinicius" (exclusive):
System.out.println("TREESET BETWEEN DANTE AND VINICIUS : " + ts.subSet("Dante", "Vinicius"));
```

OUTPUT :

```
<terminated> SetTest [Java Application] C:\Users\HP\AppData\Local\Temp\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.4.v20240802-1551\jre\bin\javaw.exe (24 a
LINKED HASH SET : [3.14, 59.23, 31.675, 21.9, 1.09, 0.521, -5.4]
Is PI (3.14) there ? true
Are all the following numbers : [21.9, 31.695, 59.23, 0.52], available in the set ? false
LAST NUMBER #1: -5.4
LAST NUMBER #2: 0.521
REVERSED LINKED HASH SET : [0.521, 1.09, 21.9, 31.675, 59.23, 3.14]

TREESET #1 : [Mat, Dante, Sergio]
TREESET #2 : [Mat, Dante, Sergio]
TREESET #3 : [Mat, Dante, Sergio, Vinicius]
TREESET BETWEEN DANTE AND VINICIUS : [Dante, Sergio]
```

MAP Type

While Map is part of the Collection framework, as it is capable of storing elements, **it is not a subtype of the Collection interface** because Map has the special property of working with key-value pairs instead of single values. All the other collections that extend the Collection interface only work with single elements, so Maps are considered to be a whole new category of collections.

Nevertheless, Map is a great collection type to use thanks to its powerful key-value association.

Maps work like a dictionary: a person looks up the terms (keys) to learn more about the one or many meanings (values) they hold.

Keys must be unique between each other and avoid duplicates, while Values can be duplicated without problems. Each key must be associated with only one value, but we can apply lists to allow "multiple values" to be associated with a single key.



The Map interface has three main implementations : **HashMap**, **TreeMap** and **Hashtable**.

- **HashMap**: The most popular Map implementation that stores its keys in a **hash table**, using the key's hashCode to determine where the pair should be stored, and allows it to achieve fast performance when iterating the collection, providing constant times at simple operations as long as the hash functions properly distributes the elements within the hash table.
- **Hashtable**: Works similarly like a HashMap, except that a Hashtable doesn't allow null objects as a key or values , and are inherently synchronized, making them safe to use in threads.
- **TreeMap**: Based on the Red-Black data structure, this type of Map stores the key-value pairs in a **key-based order**, either by the natural ordering of the keys or by a custom logic through a Comparator interface when creating it. Useful when it's necessary to maintain the key-value pairs in a specific order.

EXAMPLE CODE OF HASHMAP

```
// HASHMAP EXAMPLE

// Create a new HashMap with a initial capacity of 5 items:
HashMap<String, Integer> examScores = new HashMap<>(5);

// Introduce some scores...
examScores.put("Adrian", 8);
examScores.put("Michael", 7);
examScores.put("Nina", 9);

// Check out the keys and values of the HashMap...
System.out.println("MAP #1: " + examScores + " - COUNT : " + examScores.size());
System.out.println("KEYS : " + examScores.keySet() + " - VALUES : " + examScores.values());

// Adding a element and check if the order is consistent...
examScores.put("Robert", 8);
System.out.println("MAP #2: " + examScores + " - COUNT : " + examScores.size());

// Get the desired value providing its corresponding key:
System.out.println("What was Nina's exam score ? " + examScores.get("Nina"));

// What if we take out a element...
examScores.remove("Nina");
System.out.println("MAP #3: " + examScores + " - COUNT : " + examScores.size());

// CLEAR THE WHOLE MAP
examScores.clear();
System.out.println("Is the list empty ? " + examScores.isEmpty());
```

EXAMPLE CODE OF TREEMAP

```
// TREEMAP EXAMPLE

TreeMap<Character, List<String>> listOfWords = new TreeMap<>(Character::compareTo);

// Associate each list of word with a unique character key, somewhat like a dictionary:
listOfWords.put('b', List.of("bacon", "bathroom", "background", "building"));
listOfWords.put('v', List.of("visitor", "vector", "vigorous", "velocity"));
listOfWords.put('G', List.of("gold", "greek", "guitar", "goose"));

System.out.println("TREEMAP #1 : " + listOfWords);

// Trying to insert a entry with the same 'G' key...
listOfWords.put('G', List.of("game", "give", "gambling", "grapes")); // Overwrites the previous entry...

System.out.println("TREEMAP #2 : " + listOfWords);

// Retrieve and print out the last entry of the TreeMap
var lastList = listOfWords.lastEntry();
System.out.println("LAST ENTRY IN THE TREEMAP : " + lastList);

// Remove the first key-value entry:
Map.Entry<Character, List<String>> first = listOfWords.pollFirstEntry();
System.out.println("First entry removed : " + first);

// Check the modified Treemap:
System.out.println("FINAL TREEMAP : " + listOfWords);
```

OUTPUT :

```
<terminated> MapTest [Java Application] C:\Users\HP\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_21.0.4.v20240802-1551\jre\bin\javaw.exe (24 ago 2024 10:21:37 - 10:21:37) [pid: 8484]
MAP #1: {Nina=9, Adrian=8, Michael=7} - COUNT : 3
KEYS : [Nina, Adrian, Michael] - VALUES : [9, 8, 7]
MAP #2: {Nina=9, Robert=8, Adrian=8, Michael=7} - COUNT : 4
What was Nina's exam score ? 9
MAP #3: {Robert=8, Adrian=8, Michael=7} - COUNT : 3
Is the list empty ? true

TREEMAP #1 : {G=[gold, greek, guitar, goose], b=[bacon, bathroom, background, building], v=[visitor, vector, vigorous, velocity]}
TREEMAP #2 : {G=[game, give, gambling, grapes], b=[bacon, bathroom, background, building], v=[visitor, vector, vigorous, velocity]}
LAST ENTRY IN THE TREEMAP : v=[visitor, vector, vigorous, velocity]
First entry removed : G=[game, give, gambling, grapes]
FINAL TREEMAP : {b=[bacon, bathroom, background, building], v=[visitor, vector, vigorous, velocity]}
```

REFERENCES :

<https://codegym.cc/groups/java-collections>

<https://home.csulb.edu/~pnguyen/cecs277/lecnotes/javacollections.pdf>

https://www3.cs.stonybrook.edu/~pfodor/courses/CSE260/L20_Lists_Stacks_Queues_and_Priority_Queues.pdf