Xideral

Java Academy

Week 2 -Day 1

Collections

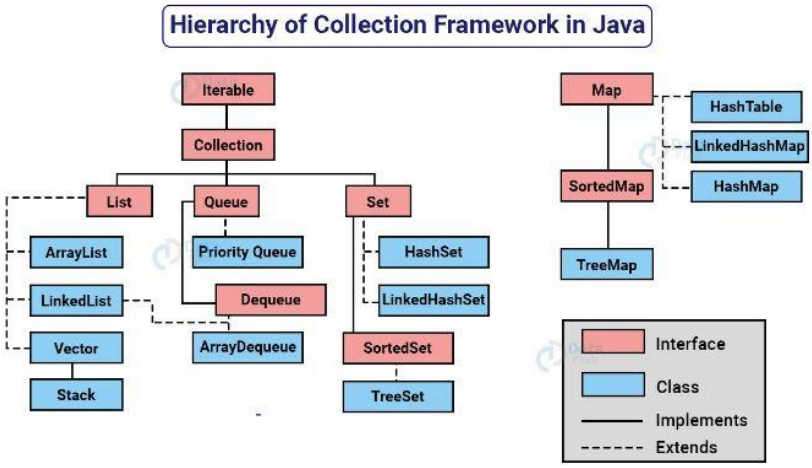
Presented by:

Edgar Itzak Sánchez Rogers

Monterrey Nuevo León México at 15 august 2024

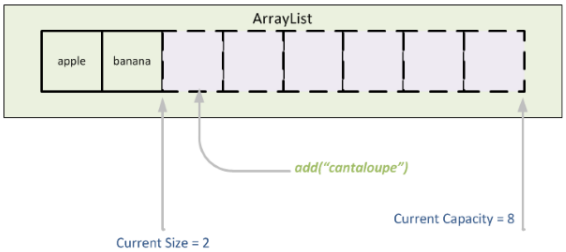
Introduction:

Collections are a group of classes that implement the “Collection” interface. Collections allow to group an indeterminate number of variables of the same type in different types of “containers” each collection has different characteristics depending on the type of the collection and implements methods to manipulate the values of the collection. This document explains the functionality of some of the most commonly used collections and gives an example of each one.



ArrayList :

An ArrayList is one of the most used collections, it is a resizable array and its use is to lists elements, assigning each element an index to be able to access it.



Some of the methods of ArrayList are:

`add(E e)`

- `Appends the specified element to the end of this list.`

clear()

- Removes all of the elements from this list.

remove(Object o)

- Removes the first occurrence of the specified element from this list, if it is present.

get(int index)

- Returns the element at the specified position in this list.

ArrayList Example:

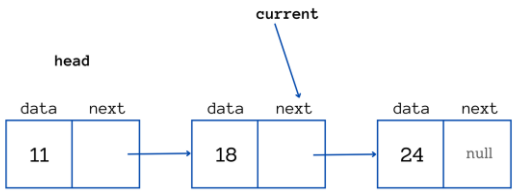
```
//---------------ARRAYLIST--------------------

List<String> names = new ArrayList<>(); //create list
//add elements
names.add("Edgar");
names.add("Jose");
names.add("Sofia");
names.add("Vanesa");
System.out.println(names); //show list

names.remove(0); //remove "Edgar" from the list
System.out.println(names);
System.out.println(names.get(1));//print "Sofia"
```

Output:

```
[Edgar, Jose, Sofia, Vanesa]
[Jose, Sofia, Vanesa]
Sofia
```

LinkedList:

A linked list is similar like a list, the main difference is the way is made connection between elements, while ArrayList organize its elements by index, LinkedList hold data in individual objects called nodes. These nodes hold both the data and a reference to the next node in the list



Some of the methods of ArrayList are:

`add(int index, E element)`

- Inserts the specified element at the specified position in this list.

`addFirst(E e)`

- Inserts the specified element at the beginning of this list.

`addLast(E e)`

- Appends the specified element to the end of this list.

`getFirst()`

- Returns the first element in this list.

`getLast()`

- Returns the last element in this list.

Example:

```
//---------------LINKEDLIST--------------------
LinkedList<String> lastname = new LinkedList<>();

lastname.add("Johnson");
lastname.addLast("Roberts");
lastname.addFirst("Lee");
lastname.addLast("Brown");
lastname.addLast("Harris");
lastname.add(4,"Smith");
System.out.println(lastname);//print last names


lastname.removeFirst(); //remove Lee and Harris
lastname.removeLast(); //print last names
System.out.println(lastname);//print last names

System.out.println(lastname.get(1));//print "Roberts"
```

Output:

```
[Lee, Johnson, Roberts, Brown, Smith, Harris]
[Johnson, Roberts, Brown, Smith]
Roberts
```

Deque:

Deque is a Double-Ended Queue that represents a queue that allows insertion, removal, and inspection of elements from both ends. Dequeue has methods for operations at both ends of the queue, which distinguishes it from the standard Queue that only supports operations at one end.

Some of the methods of Deque are:

DEQUE
size()
isEmpty()
Insert_First()
Insert_Last()
Remove_First()
Remove_Last()

Example:

```
//---------------DEQUEUE--------------------
Deque<String> lastname = new ArrayDeque<String>();

lastname.add("Johnson");
lastname.addFirst("Smith");
lastname.addFirst("Lee");
lastname.addLast("Roberts");
lastname.addLast("Brown");
lastname.addLast("Harris");
System.out.println(lastname);//print last names


lastname.pollFirst(); //remove Lee
lastname.pollLast(); //remove Harris
lastname.addFirst("Sanchez"); //add "Sanchez" at first
System.out.println(lastname);//print last names
```

Output:

```
[Lee, Smith, Johnson, Roberts, Brown, Harris]
[Sanchez, Smith, Johnson, Roberts, Brown]
```

TreeSet:

A TreeSet is a collection class that implements the SortedSet interface. TreeSet elements are sorted in ascending order by default. Like all implementations of the Set interface, TreeSet does not allow duplicate elements.

Some of the methods of Deque are:

Example:

```
//---------------TREESET--------------------

TreeSet phoneNumbers = new TreeSet(); //Create TreeSet

//add numbers
phoneNumbers.add("8126280959");
phoneNumbers.add("8172714816");
phoneNumbers.add("8159974499");
phoneNumbers.add("8159212275");
phoneNumbers.add("8172795045"); //contains this number
phoneNumbers.add("8100298380");
phoneNumbers.add("8157996990");
phoneNumbers.add("8140279655");
phoneNumbers.add("8113084918");

System.out.println(phoneNumbers.contains("8172795045")); //Return if contains a number
System.out.println(phoneNumbers.size()); //show size
phoneNumbers.remove("8126280959"); //delete an item
System.out.println(phoneNumbers.size()); //show size

System.out.println(phoneNumbers); //Show sorted numbers
```

Output:

```
true
9
8
[8100298380, 8113084918, 8140279655, 8157996990, 8159212275, 8159974499, 8172714816, 8172795045]
```

HashSet

A HashSet can store multiple values in a collection using a hash table. The hash table stores the values in an unordered method with the help of hashing mechanism. A HashSet ensures that all elements in the set are unique.

Some of the methods of Deque are:

`add (E e)`

- Adds the specified element to this set if it is not already present.

clear ()

- Removes all of the elements from this set.

remove (Object o)

- Removes the specified element from this set if it is present.

size ()

- Returns the number of elements in this set (its cardinality)

Example:

```
//---------------HASHSET--------------------

HashSet<String> inventory = new HashSet<>();

inventory.add("Grapes");
inventory.add("Apples");
inventory.add("Pears");
inventory.add("Oranges");
inventory.add("Banana");
inventory.add("Watermelon");


System.out.println("HashSet size: "+inventory.size()); //get HashSet size
inventory.add("Apples");  //Attempts to add apples again but doesn't add to hashMap

System.out.println("HashSet size: "+inventory.size()); //get HashSet size
System.out.println(inventory);
```

Output:

```
HashSet size: 6
HashSet size: 6
[Apples, Grapes, Watermelon, Oranges, Pears, Banana]
```

HashTable:

A HashTable is a data structure that implements the Map interface and is used to store key and value pairs. The hash code of the keys determines where in the table the values are stored.

Some of the methods of HashTable are:

put(K key, V value)

- Associates the specified value with the specified key in this map.

remove(Object key)

- Removes the mapping for the specified key from this map if present.

size()

- Returns the number of key-value mappings in this map.

`get(Object key)`

- `Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.`

`values()`

- `Returns a Collection view of the values contained in this map.`

Example:

```
//---------------TREESET--------------------

Hashtable<Integer, String> usernamesId = new Hashtable<>();
usernamesId.put(1, "GtBolt");
usernamesId.put(2, "TekSir");
usernamesId.put(3, "DreamOmega");
usernamesId.put(4, "CrazyHydro");
usernamesId.put(5, "DevilBold");

usernamesId.put(5, "New UserNames"); // add a value with a 5 as key, replace value
usernamesId.remove(4); //removes "CrazyHydro"

System.out.println(usernamesId.size()); //print size
System.out.println(usernamesId.get(3)); //print DreamOmega
System.out.println((usernamesId)); //print Hashtable
```

Output:

```
4
DreamOmega
{5=New UserNames, 3=DreamOmega, 2=TekSir, 1=GtBolt}
```

HashMap:

A HashMap is similar to a HashTable, both store key and value pairs. Some of the difference between and HashMap are:

- HashTable is synchronized, HashMap is not. This makes HashMap faster for single-threaded scenarios.
- HashMap allows one null key and multiple null values, whereas HashTable does not allow null keys or values.

Some of the methods of HashMap are:

`put(K key, V value)`

- `Associates the specified value with the specified key in this map.`

`remove(Object key)`

- Removes the mapping for the specified key from this map if present.

size()

- Returns the number of key-value mappings in this map.

get(Object key)

- Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

values()

- Returns a Collection view of the values contained in this map.

Example:

```
//---------------HASHMAP--------------------
HashMap<Integer, String> EmployeeList = new HashMap<>();

EmployeeList.put(1, "Lorraine Donaldson");
EmployeeList.put(2, "Nick Good");
EmployeeList.put(3, "Emma Jordan");
EmployeeList.put(4, "Stella Fields");
EmployeeList.put(5, null); // allows null values

EmployeeList.put(5, "Katherine Dodson"); // add a value with a 5 as key, replace value
EmployeeList.remove(4); //removes "Stella Fields"

System.out.println(EmployeeList.size()); //print size
System.out.println(EmployeeList.get(3)); //Emma Jordan
System.out.println((EmployeeList)); //print HashMap
```

Output:

```
4
Emma Jordan
{1=Lorraine Donaldson, 2=Nick Good, 3=Emma Jordan, 5=Katherine Dodson}
```

Conclusion:

Collections are a great way to group values; They are even more powerful if they are used to store objects with their own attributes, there is a great variety of collections so surely one of them will help solve almost all problems. Understanding these collections and when and how to properly use them is necessary to write clean, efficient, and maintainable code. Selecting the correct collection can have a major impact on the performance of an application

References:

[1] HashSet in Java | Features, Hierarchy, Constructors, & Methods (simplilearn.com)

[2] HashSet (Java Platform SE 8) (oracle.com)

[3] Deque (Java Platform SE 8) (oracle.com)

[4] Hashtable in Java - GeeksforGeeks

[5] HashMap (Java Platform SE 8) (oracle.com)

[6] Java - The TreeSet Class (tutorialspoint.com)