Index

1. ArrayList

Return to the index

1.1 Definition and creation a collection

ArrayList in Java is an implementation of the List Interface that use an dynamic method to store elements. Provides methods for handle elements dynamically.

Useful Contructors

```
// Void constructor
ArrayList<String> A = new ArrayList<>();

// Constructor with initial capacity
// ArrayList<T> A = new ArrayList<>(n) --> A = {null, null, ...} repeating
until reach n times
ArrayList<String> A = new ArrayList<>(3); // A = {null, null, null}

// Constructor that accepts other collection
ArrayList<String> B = new ArrayList<>();
ArrayList<String> A = new ArrayList<>(B);  // A = B
```

Return to the index

1.2 Main methods and properties

Size

```
// size() is a method that returns the size of the collection (regardless
// of whether your element is a collection, only counts the "first" dimension)
ArrayList<String> A = new ArrayList<>();

List<String> B = new List<>();
B.add("A"); // B = {"A"}
B.add("B"); // B = {"A", "B"}

A.add(B);
int sizeA = A.size(); // --> sizeA = 1
int sizeB = B.size(); // --> sizeB = 2
```

Access by index/key

```
ArrayList<String> A = new ArrayList<>();
A.add("A"); // A = {"A"}

// String elementA = A.get(indexElementA);
String elementA = A.get(0); // Returns "A"
```

Return to the index

1.3. Add data to the collection

Add elements by the constructor

```
ArrayList<String> A = new ArrayList<>(Arrays.asList("A","B","C")); // A = {"A"
, "B", "C"}
```

Add elements using other collections

```
ArrayList<String> B = new ArrayList<>();
B.add("A"); // B = {"A"}
B.add("B"); // B = {"A", "B"}

ArrayList<String> A = new ArrayList<>();
A.addAll(B); // A = {"A" , "B"}
```

Add elements by code

```
ArrayList<String> A = new ArrayList<>();
A.add("A") // A = {"A"}

ArrayList<ArrayList<String>> B = new ArrayList<>();
B.add(A) // B = {{"A"}}
```

Return to the index

1.4 Delete elements of the collection

Delete elements by code

```
ArrayList<String> A = new ArrayList<>();
A.add("A"); // A = {"A"}
A.add("B"); // A = {"A", "B"}
A.add("C"); // A = {"A", "B", "C"}

// Delete by the element that contains
A.remove("B"); // A = {"A", "C"}

// Delete by the index of the element
A.remove(1); // A = {"A"}
```

> **Important Note:** Removal through the **element only removes the first element** that matches the element:
> `A = {"A", "B", "C", "B"} ---> A.remove("B") ---> A = {"A", "C", "B"}`

Delete elements using another collection

Making use of the method ***removeAll*** we delete **all** of the elements that math with the collection that introduce in the method.

```
ArrayList<String> B = new ArrayList<>();
B.add("A"); // B = {"A"}
B.add("C"); // B = {"A", "C"}

ArrayList<String> A = new ArrayList<>();
A.addAll(B); // A = {"A", "C"}
A.add("B"); // A = {"A", "C", "B"}

A.removeAll(B); // A = {"B"}
```

Delete elements using a Predicate

```
ArrayList<int> A = new ArrayList<>();
A.add(1); // A = {1}
A.add(3); // A = {1, 3}
A.add(5); // A = {1, 3, 5}
A.add(7); // A = {1, 3, 5, 7}

// A.removeIf(element -> condicion) ---> condition = true ---> delete "element"
A.removeIf(element -> element > 5) // A = {1, 3}
```

Return to the index

1.5 Go through the collection

Go through using a for loop

```
ArrayList<String> A = new ArrayList<>();
A.add("A"); // A = {"A"}
A.add("B"); // A = {"A", "B"}
A.add("C"); // A = {"A", "B", "C"}
A.add("D"); // A = {"A", "B", "C", "D"}

for(int i = 0; i < A.size(); i++){
    String element = A.get(i); // Being String element returned of the
collection
    System.out.println(element);
}
```

Go through using a foreach loop

```
ArrayList<String> A = new ArrayList<>();
A.add("A"); // A = {"A"}
A.add("B"); // A = {"A", "B"}
A.add("C"); // A = {"A", "B", "C"}
A.add("D"); // A = {"A", "B", "C", "D"}

for(String s : A){
    System.out.println(s); // Beeing s the String returned by A
}
```

Go through using an Iterator

```
ArrayList<String> A = new ArrayList<>();
A.add("A"); // A = {"A"}
A.add("B"); // A = {"A", "B"}
```

```
A.add("C"); // A = {"A", "B", "C"}

Iterator<String> iteratorA = A.iterator(); // Create the iterator
while (iteratorA.hasNext()) {
    String element = iteratorA.next(); // Being element the String returned by
A
    System.out.println(element);
}
```

Go through using functional programming

```
ArrayList<String> A = new ArrayList<>();
A.add("A"); // A = {"A"}
A.add("B"); // A = {"A", "B"}
A.add("C"); // A = {"A", "B", "C"}

A.forEach(elemento -> {
    System.out.println(elemento); // Being the element the String returned by A
});
```

> **Important anotation:** if we do any modifitacion in the list or any of the elements while is going through using a *foreach* or *Iterator*, is going to throw a ***ConcurrentModificationException***.

Return to the index

1.6 Search an element

Doing a search using a loop (For/Foreach/Iterator)

```
ArrayList<String> A = new ArrayList<>();
A.add("A"); // A = {"A"}
A.add("B"); // A = {"A", "B"}
A.add("C"); // A = {"A", "B", "C"}

for(String element : A){
    if(element.equals("C")){
        // Element found!
    }
}
```

Doing a search using Lambda Expressions (Stream)

```
ArrayList<String> A = new ArrayList<>();
A.add("A"); // A = {"A"}
```

```
A.add("B"); // A = {"A", "B"}
A.add("C"); // A = {"A", "B", "C"}

boolean encontrado = A.stream().anyMatch(elemento -> elemento.equals("C"));
// Devuelve true si hay alguna coincidencia
```

> **Important Anotation:** the ***Stream*** type, from which it becomes the *ArrayList* can make use of all the functional programming. In the case of we want to return to the ArrayList we need to use the collect method specifying by a parameter the type we want using the conversion static methods int the class *Collectors*.

Doing a search using the Stream API (filtering/Collect)

```
ArrayList<String> A = new ArrayList<>();
A.add("A"); // A = {"A"}
A.add("B"); // A = {"A", "B"}
A.add("C"); // A = {"A", "B", "C"}
A.add("A"); // A = {"A", "B", "C", "A"}

List<String> B = A.stream().filter(element ->
element.equals("A")).collect(Collectors.toList()); // A = {"A", "A"}
```

Return to the index

1.7 Order the elements

Order by Collection methods

```
ArrayList<String> A = new ArrayList<>();
A.add("A"); // A = {"A"}
A.add("C"); // A = {"A", "C"}
A.add("B"); // A = {"A", "C", "B"}
A.add("A"); // A = {"A", "C", "B", "A"}

Collections.sort(A); // A = {"A", "A", "B", "C"}
```

Order by Lambda Expresions

```
ArrayList<Integer> A = new ArrayList<>();
A.add(1); // A = {1}
A.add(3); // A = {1, 3}
A.add(2); // A = {1, 3, 2}
```

```
// ArrayList.sort((current, next) -> current.compareTo(next))
A.sort((a, b) -> a.compareTo(b)); // A = {1, 2, 3}
```

Order by using the API Stream

```
ArrayList<String> A = new ArrayList<>();
A.add("A"); // A = {"A"}
A.add("B"); // A = {"A", "B"}
A.add("C"); // A = {"A", "B", "C"}
A.add("A"); // A = {"A", "B", "C", "A"}

List<String> B = A.stream().filter(element ->
element.equals("A")).collect(Collectors.toList()); // A = {"A", "A"}
```

Return to the index

1.9 Others aspects

- **Generic uses:** ArrayList in Java use generics to ensure the type of the stored elements.
- **Dynamic Capacity:** ArrayList adjust automatically its internal capacity if its needed.
- **Insertions/Removal Efficiency:** The insertions and removals in the middle of the list can be slow, because needs to move all the elements.
- **Using of Equals and HashCode methods:** ArrayList uses the equals() method to compare elements and hashCode() to improve performance in certain operations.
- **Synchronization:** *ArrayList* is asynchronized, it means is not safe to use in subprocesses. It if need, we can use Collections.synchronizedList(list) method to make it synchronized.

Return to the index

HashMap