TELUSKO

13.2-ArrayList

👉Introduction to Collections

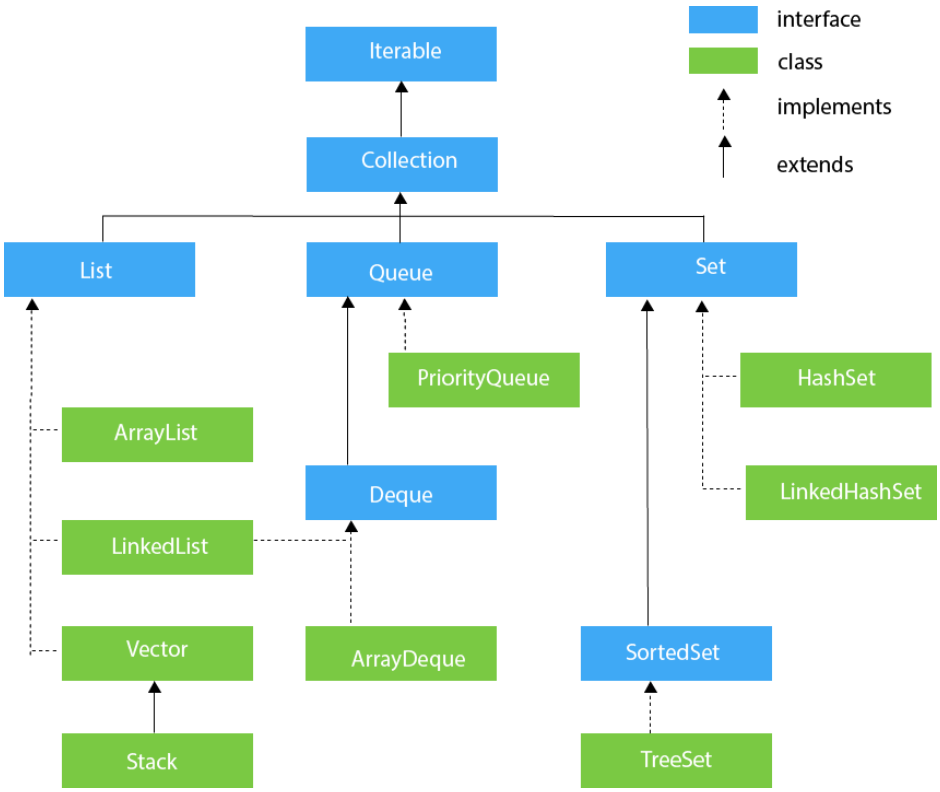
In Java, starting from version 1.2, the Collections Framework was introduced to enhance array operations and provide a set of standard interfaces and classes for handling data. The Collections API includes a variety of interfaces and classes with many built-in methods for data manipulation. To use the Collections Framework, you need to import the ***java.util package***, which contains these interfaces and classes.

Previously, most code did not require importing external packages because the core classes were included in the ***java.lang package***. This package automatically imports classes like Object, the superclass of all classes in Java.

Similarly, the Collections API is part of the ***java.util package***, which is also known as the **utility package** because it provides methods for various tasks such as **sorting** and data **insertion**.

👉Hierarchy of the Collection Interface

The top-level interface in the Collection Framework is **Iterable**, followed by the Collection interface. The Collection interface is then extended by the most



TELUSKO

commonly used interfaces, such as **List, Queue**, and **Set**. Each of these interfaces has its own set of implementing classes, which provide specific functionalities.

- **List Interface:** Implemented by classes such as ArrayList, LinkedList, and Vector.
- **Queue Interface:** Implemented by classes such as PriorityQueue. The **Deque interface**, which extends Queue, is implemented by classes like ArrayDeque.
- **Set Interface:** Commonly used in more advanced topics. It is implemented by classes such as HashSet and LinkedHashSet.
- **Map Interface:** Although not directly under the Collection hierarchy, the Map interface is part of the Java Collections Framework. It stores data in key-value pairs and is implemented by classes like HashMap and LinkedHashMap.

👉Example

Let's look at a simple example using the Collection & List interface with the ArrayList class.

```
import java.util.Collection;
import java.util.ArrayList;

public class Demo {
    public static void main(String[] args) {
        // Using Collection interface with ArrayList implementation
        Collection<Integer> nums = new ArrayList<Integer>();
        nums.add(3);
        nums.add(4);
        nums.add(5);
        nums.add(9);
        nums.add(8);
        nums.add(7);

        // Printing the ArrayList
        System.out.println(nums);
    }
}
```

👉**Output:**

```
[3, 4, 5, 9, 8, 7]
```

👉**Explanation**

In the above example, we use the Collection interface and the ArrayList class. When writing code in an IDE, you might see a warning suggesting using generics for type safety. This is done by specifying the type of data inside angular brackets (< >), which ensures that only a particular data type can be added to the collection. Generics help to prevent errors by avoiding the insertion of incompatible data types.

Since Java treats everything as an object, collections store elements as objects. For primitive types, you should use their corresponding wrapper classes (e.g., Integer for int).

The **add()** method is used to insert elements into the ArrayList. There are several other methods available for collection, which should be explore based on our need as each method as different working.

After adding elements, the ArrayList can be printed directly without using a loop, as the **toString()** method of the collection handles this internally. However, if you want to print each element separately, you can use an enhanced for loop.

Printing Separate Values Using a Loop

```
for (Integer num : nums) {
    System.out.println(num);
}
```

👉**Accessing Elements by Index**

If you need to access elements by their index, use the List interface as the Collection interface doesn't have methods to work with indexing which List provides, an get(int index) method to retrieve the element at a specific position.

```
List<Integer> numList = new ArrayList<>(nums);
System.out.println("Element at index 2: " + numList.get(2));
```