

# Java Collection

Press Space for next page →

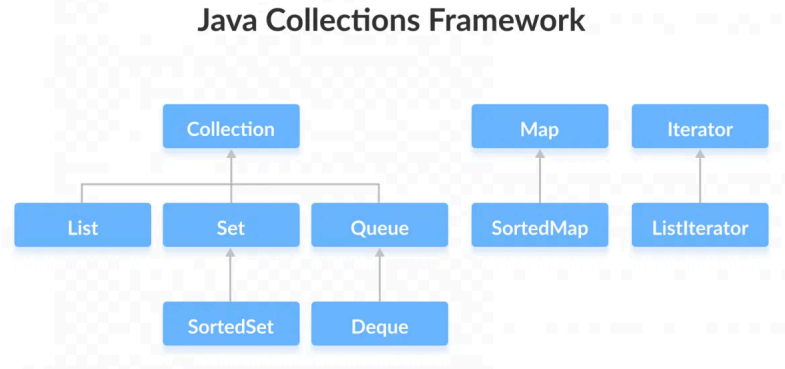


# Agenda

1. Java Collections Framework
2. Limitations of Arrays
3. Importance of collections
4. Array vs Collection
5. Difference between Collection & Collections
6. Collection Framework Hierarchy
7. Key Interfaces of Collection Framework
8. Collection Interface
9. List Interface
10. Set Interface
11. SortedSet Interface
12. NavigableSet Interface
13. Queue Interface
14. Map Interface

# Java Collections Framework

The Java collections framework provides a set of interfaces and classes to implement various data structures and algorithms.



# Limitations of Arrays

- Arrays are fixed in size i.e. once we created an array with some size there is no chance of increasing or decreasing its size based on our requirement. Hence to use arrays we should know the size in advance which may not be possible always.
- Arrays can hold only homogeneous data elements. Example:

```
1  Student []s = new Student [10000];  
2  s[0] = new Student(); // correct  
3  s[1] = new Customer(); // wrong
```

- But We can resolve this problem by using object Arrays.

```
1  Object[] obj = new Object [10000];  
2  obj [0] = new Student();  
3  obj [1] = new Customer();
```

- Arrays are not built using any standard data structure, so they don't come with ready-to-use methods for common tasks like adding, removing, or sorting elements. You have to handle these things on your own or use other classes to help.

# Importance of collections

- Collections are growable in nature. i.e. Based on our requirement we can increase (or) Decrease the size.
- Collections can hold both **homogeneous** (*same type*) & **Heterogeneous** (*different types*) elements.
- Every Collection class is implemented based on some standard data structure. Hence readymade method support is available for every requirement. Being a programmer we have to use this method and we are not responsible to provide implementation.

# Array vs Collection

FEATURE	ARRAYS	COLLECTIONS
Size	Fixed	Growable
Memory Recommendation	Not Recommended	Recommended
Performance	Recommended	Not Recommended
Element Types	Homogeneous	Homogeneous and Heterogeneous
Underlying Data Structure	None	Standard Data Structures (e.g., List, Set, Map)
Readymade Method Support	Limited	Comprehensive
Supported Data Types	Primitive and Objects	Only Objects

# Difference between Collection & Collections

- **Collection** is an **interface** which can be used to represent a group of individual objects as a single entity.

Example: List, Set, Queue, Deque

```
1 List<String> studentNames = new ArrayList<>();
2 studentNames.add("Alice");
3 studentNames.add("Bob");
```

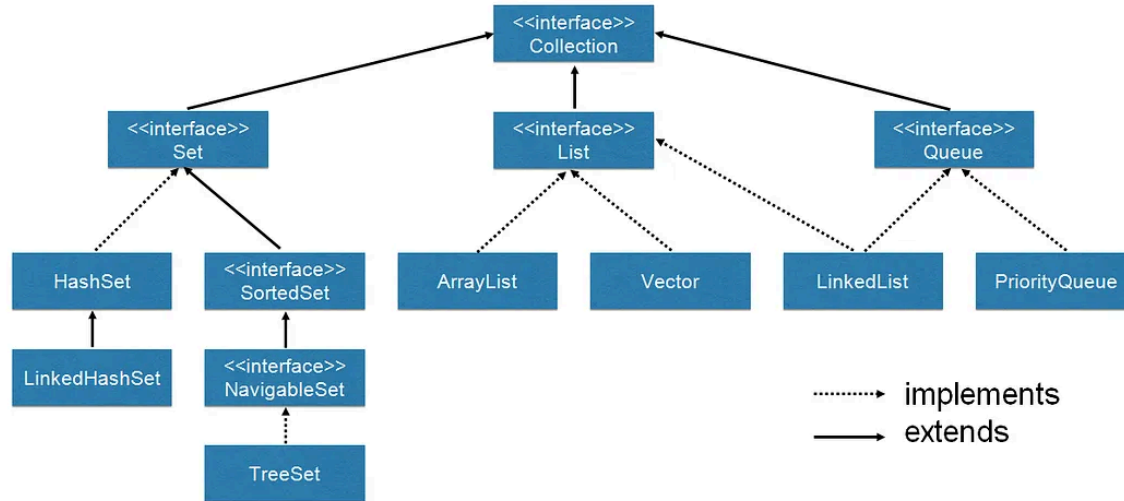
- **Collections** is an **utility class** present in `java.util.package` to define several utility methods (like Sorting, Searching...) for Collection objects.

Example: If you have a collection (like a list of numbers) and want to sort it or search for an item, you can use the Collections class to do this without writing the logic yourself.

```
1 List<Integer> numbers = Arrays.asList(3, 1, 4, 2);
2 Collections.sort(numbers); // Sorts the list in ascending order
3 System.out.println(numbers); // Output: [1, 2, 3, 4]
```

# Collection Framework Hierarchy

## Collection Interface





# Key Interfaces of Collection Framework

1. Collection
2. List
3. Set
4. SortedSet
5. NavigableSet
6. Queue
7. Map

# Collection Interface

- If we want to represent a group of individual objects as a single entity then we should go for Collection.
- Collection interface defines the most common methods which are applicable for any Collection object.
- Collection interface is considered as the root interface of the Collection framework.

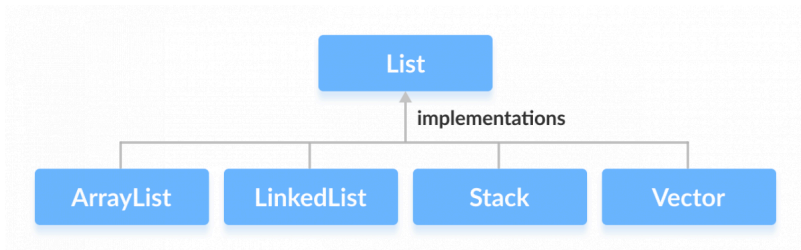
There is no concrete class which implements collection interface directly.

```
1  Collection<String> myCollection = new ArrayList<>();
2  myCollection.add("Apple");
3  myCollection.add("Banana");
4  System.out.println(myCollection.size());
```

# List Interface

- List is child interface of Collection.
- If we want to represent a group of individual objects as a single entity where **duplicates are allowed** and **insertion order** preserved then we should go for List.

```
1 List<String> myList = new ArrayList<>();  
2 myList.add("Apple");  
3 myList.add("Banana");  
4 myList.add("Apple"); // Duplicates are allowed
```

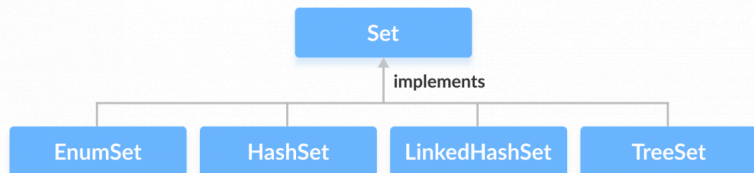


Learn more: [Java List](#)

# Set Interface

- It is the child interface of Collection.
- If we want to represent a group of individual objects as a single entity where **duplicates are not allowed** and **insertion order not preserved** then we should go for Set.

```
1 Set<String> mySet = new HashSet<>();
2 mySet.add("Apple");
3 mySet.add("Banana");
4 mySet.add("Apple"); // Duplicate element
5 System.out.println(mySet); // Output: [Apple, Banana]
```



# SortedSet Interface

- It is the child interface of Set.
- If we want to represent a group of individual objects as a single entity where **duplicates are not allowed** , **insertion order is not preserved** but all objects **should be inserted according to some sorting order** then we should go for SortedSet.

```
1 SortedSet<Integer> numbers = new TreeSet<>();
2 numbers.add(3);
3 numbers.add(1);
4 numbers.add(2);
5 System.out.println(numbers); // Output: [1, 2, 3]
```

```
1 SortedSet<Integer> treeSetDesc = new TreeSet<>(Collections.reverseOrder());
2 treeSetDesc.add(5);
3 treeSetDesc.add(2);
4 treeSetDesc.add(7);
5 System.out.println(treeSetDesc); // Output: [7, 5, 2]
```

# NavigableSet Interface

- It is the child interface of Sorted Set
- It defines several methods for navigation purposes.

```
1  NavigableSet<Integer> set = new TreeSet<>();
2  set.add(10);
3  set.add(20);
4  set.add(30);
5  System.out.println(set.lower(25)); // Output: 20
6  System.out.println(set.floor(20)); // Output: 20
```

# Queue Interface

- It is child interface of Collection.
- If we want to represent a group of individual objects prior to processing then we should go for Queue.

Ex: before sending a mail all mail id's we have to store somewhere and in which order we saved in the same order mail's should be delivered (**First in First out**) for this requirement Queue concept is the best choice.

```
1 Queue<String> queue = new LinkedList<>();
2 queue.add("A");
3 queue.add("B");
4 queue.add("C");
5 System.out.println(queue.poll()); // Output: A (first added, first removed)
6 queue.offer("D"); // Adds "D" to the end of the queue
```

# Map Interface

- Map is not the child interface of Collection.
- If we want to represent a group of individual objects as **key-value pairs** then should go for Map.

Ex: Roll No Name

101 → Lamia

102 → Sudipto

103 → Abir


104 → Sudipto

- Both key and value are objects, **duplicated keys are not allowed** but **values can be duplicated**.

```
1 Map<Integer, String> myMap = new HashMap<>();
2 myMap.put(101, "Lamia");
3 myMap.put(102, "Sudipto");
4 myMap.put(103, "Abir");
5 myMap.put(104, "Sudipto");
6 System.out.println(studentMap.get(101)); // Output: Lamia
```



Thank you 

 [qa-june-2024-automation-with-java-slides](#)