Collections

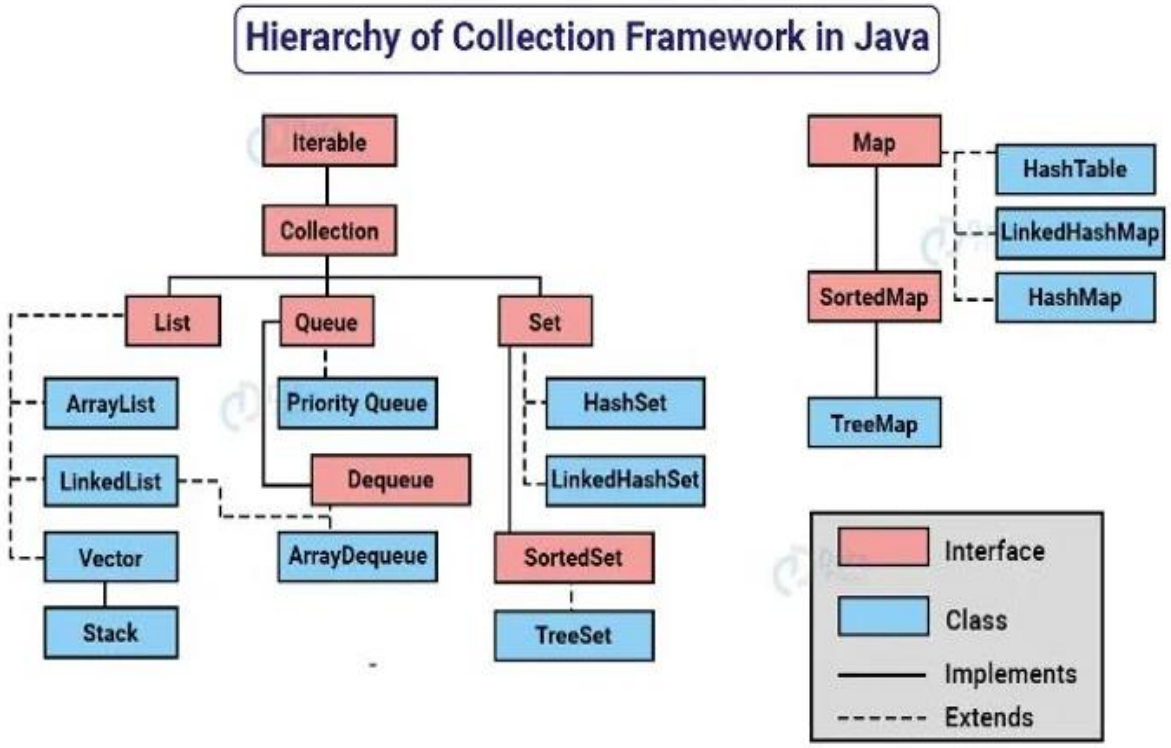
Contents

- Introduction
- Core IFCs
- Collection IFC
- List
- Set
- Queue
- Map
- Vector
- Advanced Collections

Collections

Introduction

Hierarchy



Collections

Introduction

Collections

- Contains prepackaged data structures, interfaces, algorithms for manipulating those data structures.
- Examples of collections – hand of cards, software engineers working on same project, etc.
- Collections – Use existing data structures without concern for how they are implemented

Collections

Introduction

- **Collection:** Root interface of the collection hierarchy.
- **List:** Ordered collection (sequence).
- **Set:** Collection that contains no duplicate elements.
- **Queue:** Collection used to hold multiple elements prior to processing.
- **Map:** Object that maps keys to values, not a true collection.

Hierarchy

Collections

Interfaces

Implementations

- **Collection**
- **List Interface**
  ArrayList, LinkedList
- **Set Interface**
  HashSet, LinkedHashSet, TreeSet
- **Queue Interface**
  PriorityQueue, LinkedList
- **Map Interface**
  HashMap, LinkedHashMap, TreeMap

Collections

Core IFC

Classes & special IFCs

Collection Classes:

- Legacy Classes
  - Vector

Specialized Interfaces:

- Deque Interface
- Map.Entry Interface

Collections

Collection

Collection

Common Methods:

- **add(E e):** Adds an element to the collection.
- **remove(Object o):** Removes an element from the collection.
- **size():** Returns the number of elements in the collection.
- **clear():** Removes all elements from the collection.
- **isEmpty():** Checks if the collection is empty.
- **contains(Object o)**: Checks if the collection contains a specific element.

Collections

List

Common Implementations: **ArrayList** , **LinkedList**

Common Methods:

List

- **get(int index):** Returns the element at the specified position.
- **set(int index, E element):** Replaces the element at the specified position.
- **add(int index, E element):** Inserts an element at the specified position.
- **remove(int index):** Removes the element at the specified position.
- **indexOf(Object o):** Returns the index of the first occurrence of the specified element.

Collections

List

ArrayList

Characteristics:

- Resizable array.
- Fast random access.

Performance:

- O(1) time complexity for the get and set operations.
- O(n) for add and remove operations (amortized O(1) for adding at the end).

Use Case:

Best for frequent access and modification of elements by index.

Example:

```
List<String> arrayList = new ArrayList<>();
arrayList.add("Apple");
```

Collections

Set

Common Implementations:

HashSet , **LinkedHashSet** , **TreeSet**

Common Methods:

- All methods from Collection.

Set

Collections

Set

HashSet

Characteristics:

- Backed by a hash table.
- No guaranteed order of elements.

Performance:

- O(1) time complexity for basic operations (add, remove, contains).

Use Case:

Best for high-performance operations with no need for order.

Example:

```
Set<String> hashSet = new HashSet<>();
hashSet.add("Orange");
```

Collections

Set

LinkedHashSet

Characteristics:

- Hash table with linked list running through it.
- Maintains insertion order.

Performance:

- O(1) time complexity for basic operations.

Use Case:

Best for when iteration order matters.

Example:

```
Set<String> linkedHashSet = new LinkedHashSet<>();
linkedHashSet.add("Grapes");
```

Collections

Set

TreeSet

Characteristics:

- Navigable set backed by a TreeMap.
- Elements sorted in natural order or by a comparator.

Performance:

- O(log n) time complexity for basic operations.

Use Case:

Best for sorted data and range view.

Example:

```
Set<String> treeSet = new TreeSet<>();
treeSet.add("Pineapple");
```

Collections

Queue

Queue

Common Implementations:

LinkedList , PriorityQueue

Common Methods:

- **offer(E e):** Inserts the specified element into this queue.
- **poll():** Retrieves and removes the head of this queue.
- **peek():** Retrieves, but does not remove, the head of this queue.

Collections

Map

Common Implementations:

HashMap , LinkedHashMap , TreeMap , Hashtable

Common Methods:

- **put(K key, V value):** Associates the specified value with the specified key.
- **get(Object key):** Returns the value to which the specified key is mapped.
- **remove(Object key):** Removes the mapping for a key.
- **containsKey(Object key):** Checks if the map contains a mapping for the specified key.
- **keySet():** Returns a set view of the keys.

Map

Collections

Map

HashMap

Characteristics:

- Hash table-based implementation.
- No guaranteed order.

Performance:

- O(1) time complexity for basic operations.

Use Case:

Best for high-performance key-value mapping.

Example:

```
Map<String, Integer> hashMap = new HashMap<>();
hashMap.put("Apple", 1);
```

Collections

Map

LinkedHashMap

Characteristics:

- Hash table with a linked list running through it.
- Maintains insertion order.

Performance:

- O(1) time complexity for basic operations.

Use Case:

Best for iteration order needs.

Example:

```
Map<String, Integer> linkedHashMap = new LinkedHashMap<>();
linkedHashMap.put("Banana", 2);
```

Collections

Map

TreeMap

Characteristics:

- Red-black tree-based implementation.
- Sorted according to natural order or by comparator.

Performance:

- O(log n) time complexity for basic operations.

Use Case:

Best for sorted key-value pairs.

Example:

```
Map<String, Integer> treeMap = new TreeMap<>();
treeMap.put("Orange", 3);
```

Collections

Vector

Vector

```
public static void main(String[] args) {
    Vector<Integer> arr = new Vector<Integer>();

    System.out.println("There are " + arr.capacity() + " spaces in the vector" +
            " only " + arr.size() + " are in use");
    arr.add(5);
    arr.add(2);
    System.out.println("There are " + arr.capacity() + " spaces in the vector" +
            " only " + arr.size() + " are in use");
    arr.add(3);
    arr.add(4);
    arr.add(1);
    arr.add(7);
    arr.add(2);
    arr.add(9);
    arr.add(0);
    arr.add(4);
    arr.add(7);

    System.out.println("There are " + arr.capacity() + " spaces in the vector" +
    " only " + arr.size() + " are in use");

    System.out.println("The array is " + arr);

    System.out.println("Does 13 exists? " + arr.contains(13));
    System.out.println("Does 2 exists?  " + arr.contains(2));
}
```

הגודל הפיזי ההתחלתי של המערך הוא 10

ניתן לראות כי ברגע שהמקום במערך נגמר המחלקה מגדילה אותו פי 2

Collections

Vector

Vector

```java
import java.util.Iterator;
import java.util.Vector;

public class Program {

    public static void main(String[] args) {
        Vector<Integer> arr = new Vector<Integer>();

        arr.add(3);
        arr.add(4);
        arr.add(1);
        arr.add(7);
        arr.add(2);

        System.out.println("The array using loop: ");
        for (int i=0 ; i < arr.size() ; i++)
            System.out.print(arr.get(i) + " ");

        System.out.println("\nThe array using iterator: ");
        Iterator<Integer> itr = arr.iterator();
        while (itr.hasNext())
            System.out.print(itr.next() + " ");
    }
}
```

הגדרת איטרטור מטיפוס איברי האוסף

קבלת איטרטור לאיבר הראשון באוסף

כל עוד יש איברים באוסף

קבלת האיבר הבא

Collections

Vector

Vector

```
public static void main(String args[]) {
      Enumeration<String> days;
      Vector<String> dayNames = new Vector<String>();

      dayNames.add("Sunday");
      dayNames.add("Monday");
      dayNames.add("Tuesday");
      dayNames.add("Wednesday");
      dayNames.add("Thursday");
      dayNames.add("Friday");
      dayNames.add("Saturday");
      days = dayNames.elements();

      while (days.hasMoreElements()) {
         System.out.println(days.nextElement());
      }
   }
```

Collections

Advanced Collections

EnumMap

Characteristics:

- A specialized Map implementation for use with enum keys.
- All keys must be of the same enum type.

Performance:

Highly efficient, faster than HashMap when working with enum keys.

Use Case:

Best for maps with enum keys.

Example:

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY }
EnumMap<Day, String> map = new EnumMap<>(Day.class);
map.put(Day.MONDAY, "Start of work week");
```

Collections

Advanced Collections

ConcurrentMap

Characteristics:

- A thread-safe implementation of HashMap optimized for concurrent access.
- Divides the map into segments to reduce contention.

Performance:

High throughput for concurrent read and write operations.

Use Case:

Best for high-concurrency scenarios.

Example:

```
Map<String, Integer> concurrentMap = new ConcurrentHashMap<>();
concurrentMap.put("key1", 1);
```

Collections

Advanced Collections

CopyOnWriteArrayList

Characteristics:

- A thread-safe variant of ArrayList where all mutative operations (add, set, remove) are implemented by making a fresh copy of the underlying array.

Performance:

Suitable for situations where reads vastly outnumber writes.

Use Case:

Best for lists where traversal operations are more frequent than updates.

Example:

```
List<String> list = new CopyOnWriteArrayList<>();
list.add("Item1");
```

Collections

Thank You !!