Programare avansata pe obiecte - laborator 7 (231)

Butan Silvia
silvia.butan@endava.com
butan.silvia@gmail.com

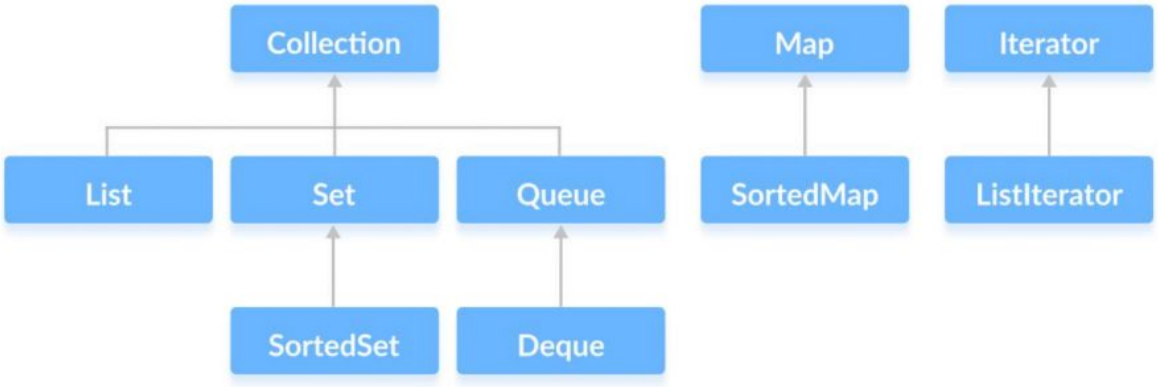
Object oriented programming - lab 7

Collections in Java:

A Collection is a group of individual objects represented as a single unit. Java provides **Collection Framework** which defines several classes and interfaces to represent a group of objects as a single unit.

Java Collections Hierarchy:

The Collections framework is better understood with the help of core interfaces. The collections classes implement these interfaces and provide concrete functionalities.

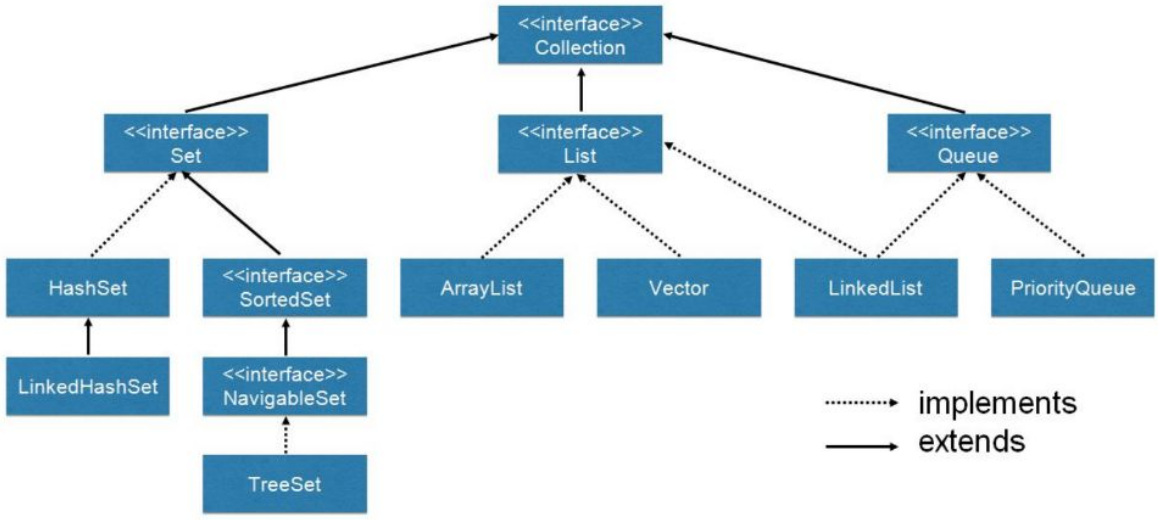


1. **Collection**

   - provides all general purpose methods which all collections classes must support (or throw UnsupportedOperationException), eg: *add(E e), contains(Object o), remove(Object o);*

   - extends Iterable interface which adds support for iterating over collection elements using the “for-each loop” statement;

   - all other collection interfaces and classes (except Map) either extend or implement this interface, Eg: List (indexed, ordered) and Set (sorted) interfaces implement this collection;



2. **List**
   - represents ***an ordered collection of elements***;
   - we can access elements by their integer index (position in the list), and search for elements in the list: index starts with 0, just like an array;
   - useful classes which implement List interface: *ArrayList, CopyOnWriteArrayList, LinkedList, Stack and Vector.*

   Examples:

   ➢ **ArrayList:** represents a resizable list of objects. We can add, remove, find, sort and replace elements in this list. ArrayList is part of Java's collection framework and implements Java's List interface.

   Java ArrayList class extends AbstractList class which implements List interface. The List interface extends Collection and Iterable interfaces in hierarchical order.

   Features:

   - **ordered** (elements preserve their ordering which is by default the order in which they were added to the list)
   - **index based** (elements can be randomly accessed using index positions. Index start with '0'.)
   - **dynamic resizing** (grows dynamically when more elements need to be added)
   - **duplicates allowed** (we can add duplicate elements)

Methods:

- add(), addAll(), clear(), clone(), contains(), forEach(), get(), indexOf(), remove(), removeAll(), etc
- You can find more on: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/ArrayList.html

```
package com.paolabs.lab7.ex1;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class ArrayListExamples {

   private static ArrayListExamples instance = new ArrayListExamples();

   //Empty ArrayList
   private final List<String> names = new ArrayList<>();

   private ArrayListExamples() {
   }

   public static ArrayListExamples getInstance() {
       return instance;
   }

   public void addNameElement(String name) {
       // add elements to the list
       names.add(name);
   }

   public void removeNameElement(int index) {
       // remove element with given index from the list
       names.remove(index);
   }

   public void updateNameElement(int index, String name) {
       // update element at given index with given value
       names.set(index, name);
   }
```

```
    public void displayList() {
        Iterator<String> iterator = names.iterator();

        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }

    public void anotherDisplayList() {
        for (String name : names) {
            System.out.println(name);
        }
    }

    public void yetAnotherDisplayList() {
        for (int i = 0; i < names.size(); i++) {
            System.out.println(names.get(i));
        }
    }
}
```

➢ **LinkedList:** is doubly-linked list implementation of the List and Deque interfaces. It implements all optional list operations, and permits all elements (including null).

Features:

- **doubly linked** list implementation which implements List and Deque interfaces. Therefore, It can also be used as a Queue, Deque or Stack.
- **permits all elements** including duplicates and NULL.
- **maintains** the insertion **order** of the elements
- we can use ListIterator to **iterate** LinkedList elements
- we can **access elements in sequential order only**. It does not support accessing elements randomly.

Constructors:

- LinkedList() : initializes an empty LinkedList implementation.
- LinkedListExample(Collection c) : initializes a LinkedList containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Methods: add(Object o), add(int index, Object element), addFirst(Object o),

addLast(Object o), size(), contains(), getFirst(), getLast(), iterator(), etc

```
package com.paolabs.lab7.ex1;

import java.util.Arrays;
import java.util.Collections;
import java.util.LinkedList;
import java.util.ListIterator;

public class LinkedListExample {

   private static LinkedListExample instance = new LinkedListExample();

   //Create linked list
   LinkedList<String> linkedList = new LinkedList<>();

   private LinkedListExample() {
   }

   public static LinkedListExample getInstance() {
       return instance;
   }

   public void addElement(String element) {
       //Add elements
       linkedList.add(element);
   }

   public void removeElementBy(int index) {
       //Remove element by index
       linkedList.remove(index);
   }

   public void removeElementBy(String value) {
       //Remove element by value
       linkedList.remove(value);
   }

   public String getFirst() {
       // Return first element
       return linkedList.getFirst();
   }
```

```
    public String getLast() {
        // Return last element
        return linkedList.getLast();
    }

    public String getElementAtIndex(int index) {
        // Return element at index
        return linkedList.get(index);
    }

    public void displayLinkedList() {
        ListIterator<String> iterator = linkedList.listIterator();

        while (itrator.hasNext()) {
            System.out.println(iterator.next());
        }
    }

    public void anotherDisplayLinkedList() {
        for (String s : linkedList) {
            System.out.println(s);
        }
    }

    public String[] getLinkedListAsArray() {
        String[] array = new String[linkedList.size()];
        return linkedList.toArray(array);
    }

    public LinkedList<String> getArrayAsLinkedList(String[] array) {
        LinkedList<String> linkedListNew = new
LinkedList<>(Arrays.asList(array));
        return linkedListNew;
    }

    public void sort() {
        Collections.sort(linkedList);
    }

    public void reverseOrder() {
        Collections.sort(linkedList, Collections.reverseOrder());
```

```
    }

}
```

3. **Set**
    - represents a collection of **sorted elements;**
    - do **not allow duplicate** elements;
    - some useful classes which implement Set interface are: ConcurrentSkipListSet, CopyOnWriteArraySet, EnumSet, HashSet, LinkedHashSet and TreeSet.

Examples:

➢ **HashSet:** extends AbstractSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

Features:

- **Duplicate values are not allowed.**
- **One NULL element** is **allowed** in HashSet.
- It is an **unordered collection** and makes no guarantees as to the iteration order of the set.

Constructors:

- HashSet()
- HashSet(int capacity)
- HashSet(Collection c)

```
package com.paolabs.lab7.ex1;

import java.util.HashSet;
import java.util.Iterator;

public class HashSetExample {

   private static HashSetExample instance = new HashSetExample();

   private final HashSet<String> hashSet = new HashSet<>();

   private HashSetExample() {
```

```
    }

    public static HashSetExample getInstance() {
        return instance;
    }

    public void addElement(String element) {
        hashSet.add(element);
    }

    public void removeElement(String element) {
        hashSet.remove(element);
    }

    public boolean containsElement(String element) {
        return hashSet.contains(element);
    }

    public void displayElements() {
        Iterator<String> iterator = hashSet.iterator();

        while (iterator.hasNext()) {
            String value = iterator.next();

            System.out.println("Value: " + value);
        }
    }

    public void anotherDisplayElements() {
        for (String value : hashSet) {
            System.out.println("Value: " + value);
        }
    }

    public String[] convertHashSetToArray() {
        String[] newArray = new String[hashSet.size()];
        return hashSet.toArray(newArray);
    }

}
```

4. **Map**
    - enable us to store data in **key-value pairs** (keys should be immutable).
    - **cannot contain duplicate keys**;
    - each **key can map to** at most **one value;**
    - provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings.
    - some useful classes which implement Map interface are: ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, Properties, TreeMap and WeakHashMap.

Examples:

➢ **HashMap:** It is used to store key and value pairs. Each key is mapped to a single value in the map.

Features:

- cannot contain duplicate keys
- allows multiple null values but only one null key
- is an **unordered** collection. It does not guarantee any specific order of the elements
- a value can be retrieved only using the associated key
- **stores only object references**. So primitives must be used with their corresponding wrapper classes. Eg: int will be stored as Integer

```
package com.paolabs.lab7.ex1;

import java.util.HashMap;
import java.util.Iterator;

public class HashMapExample {

   private static HashMapExample instance = new HashMapExample();

   private final HashMap<Integer, String> map = new HashMap<>();

   private HashMapExample() {
   }
```

```
    public static HashMapExample getInstance() {
        return instance;
    }

    public void addElement(Integer key, String value) {
        map.put(key, value);
    }

    public String getElementBy(Integer key) {
        return map.get(key);
    }

    public String removeElementBy(Integer key) {
        return map.remove(key);
    }

    public void displayMap() {
        Iterator<Integer> iterator = map.keySet().iterator();

        while (iterator.hasNext()) {
            Integer key = iterator.next();
            String value = map.get(key);

            System.out.println("The key is :: " + key + ", and value is :: "
+ value);
        }
    }

    public void anotherDisplayMap() {
        for (Integer key : map.keySet()) {
            String value = map.get(key);

            System.out.println("The key is :: " + key + ", and value is :: "
+ value);
        }
    }
}
```

5. **Queue**
    - **intended to hold the elements** (put by producer threads) **prior to processing**

by consumer thread(s).

- provide additional **insertion**, **extraction**, and **inspection** operations.
- queues typically order elements in a **FIFO** (**first-in-first-out**) manner.
- useful classes: ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue and SynchronousQueue.

6. **Deque**
   - A **double ended queue** (pronounced “deck“) that supports element insertion and removal at both ends.
   - when a deque is used as a **queue, FIFO** (First-In-First-Out) behavior results; when a deque is used as a **stack, LIFO** (Last-In-First-Out) behavior results.
   - some common known classes implementing this interface are ArrayDeque, ConcurrentLinkedDeque, LinkedBlockingDeque and LinkedList.

Example:

➢ **ArrayDeque:** a special kind of array that grows and allows users to add or remove an element from both the sides of the queue.

Features:

- have **no capacity restrictions** and they grow as necessary to support usage
- Null elements are prohibited in the ArrayDeque
- likely to be faster than LinkedList when used as a queue

Constructors:

- ArrayDeque()
- ArrayDeque(Collection c)
- ArrayDeque(int numofElements)

```
package com.paolabs.lab7.ex1;

import java.util.ArrayDeque;
import java.util.Deque;
import java.util.Iterator;

public class DequeExample {
```

```
   private static DequeExample instance = new DequeExample();

   Deque<String> deque = new ArrayDeque<>();

   private DequeExample() {
   }

   public static DequeExample getInstance() {
       return instance;
   }

   public void addElement(String element) {
       deque.add(element);
   }

   public void addFirst(String element) {
       deque.addFirst(element);
   }

   public void addLast(String element) {
       deque.addLast(element);
   }

   public void display() {
       for (Iterator<String> iterator = deque.iterator();
iterator.hasNext(); ) {
           System.out.println(iterator.next());
       }
   }

   public void anotherDisplay() {
       for (String s : deque) {
           System.out.println(s);
       }
   }

   public void displayReverse() {
       for (Iterator<String> dItr = deque.descendingIterator();
dItr.hasNext(); ) {
           System.out.println(dItr.next());
       }
```

```
    }

    public void anotherDisplayReverse() {
        Iterator<String> dItr = deque.descendingIterator();
        while (dItr.hasNext()) {
            System.out.println(dItr.next());
        }
    }
}
```