L59

Monday, February 7, 2022 6:07 PM

Data Structures - Set, HashSet, TreeSet, LinkedHashSet

Saurav Hathi

https://www.youtube.com/channel/UCp6MFWao5vWRnyRCxBsKnfw

https://www.instagram.com/saurav_hathi/

Understanding Set interface

Q1. A Set represents a collection which will not contain duplicate elements.

For example, if we add the same element twice to a **Set**, it will not include it for the second time and will simply ignore the second addition request. Whenever such an addition does not change the contents of the **Set**, the add method returns a boolean value of false.

Below are some of the concrete classes which implement Set interface:

1. HashSet - most commonly used class whenever we want a Set behaviour

(the order of iteration over the elements is not guaranteed to be the same as the insertion order).

1. TreeSet - is used when we want the elements to be sorted on the natural ordering of its elements, or by a custom Comparator.
2. LinkedHashSet - is used when we want the retrieval (during iteration) of elements to be in the order in which they were inserted.

Some of the most commonly used methods in Set interface are :

1. **add**(E e) - adds the given element into the set.
2. **contains**(Object obj) - returns true if set contains the object, false otherwise.
3. **remove**(Object obj) - removes the given element if present and returns true, false otherwise.
4. **clear**()
5. **size**()

There are many more methods in the Set interface, above mentioned are some of the most commonly used ones.

See and retype the below code to learn the usage of few methods.

```
1  package q11378;
2  import java.util.*;
3  public class SetDemo {
4      public static void main(String[] args) {
5          Set namesSet = new HashSet();
6          namesSet.add("Tokyo");
7          namesSet.add("Athens");
8          namesSet.add("New York");
9          System.out.println(namesSet);
10         namesSet.add("Tokyo");
11         System.out.println(namesSet);
12         System.out.println("Above result shows how Set does not include the duplicate Tokyo");
13         boolean removeStatus = namesSet.remove("Delhi");
14         System.out.println("Delhi removeStatus : " + removeStatus);
15         boolean containsFlag = namesSet.contains("Athens");
16         System.out.println("Athens containsFlag : " + containsFlag);
17     }
18 }
19
20
```

HashSet

Q1. Whenever we want to store distinct elements and also be able to verify if some element exists in the collection quickly, we use a HashSet.

HashSet internally uses a HashMap to provide all the functionality.

HashSet has 4 constructors.

1. HashSet() - the default constructor creates an empty HashSet with initialCapacity as 16 and a default load factor of 0.75
2. HashSet(int initialCapacity) - it creates an empty HashSet with the given initial capacity and a default load factor of 0.75.
3. HashSet(int initialCapacity, float loadFactor) - it creates an empty HashSet with the given initial capacity and load factor.
4. HashSet(Collection c) - it creates a HashSet with elements already present in the Collection c passed as parameter. Note that duplicates if present in the collection c will be automatically removed in the set which is being created.

HashSet internally uses a HashMap instance to store the elements as keys in that HashMap. Hence, the meaning and behaviour of **initialCapacity** and **load factor** in a HashSet are the same as in a HashMap.

These elements in a HashSet are placed in buckets/slots. **Capacity** is number of such slots/buckets. The capacity at the time of creation of a **HashSet** is called **initialCapacity**.

Note that **size** of the **HashSet** is different from the **capacity**. **Size** is the total number of elements inserted into the **HashSet**.

The **load factor** determines at what level of fullness the HashSet's capacity should be automatically increased.

The increase in the capacity is performed when the size becomes greater than (load factor x current capacity).

Note that whenever we call the size() method on an HashSet, it always returns the current count of elements it holds.

See and retype the below code.
You will notice that the size of cSet is 0, even though we create it with an **initialCapacity** of 20. This is because we have not added any elements to cSet.

When we know the count of elements we will be storing in an HashSet (assuming they are greater than **16**), it is efficient to provide it as the initialCapacity so that

the HashSet can avoid frequent internal capacity adjustments during insertions.

```java
package q11379;
import java.util.*;
public class HashSetDemo {
    public static void main(String[] args) {
        Set aSet = new HashSet();
        System.out.println("aSet.size() = " + aSet.size());
        System.out.println("aSet = " + aSet);
        aSet.add("Cuprum");
        aSet.add("Aurum");
        aSet.add("Argentum");
        System.out.println("aSet.size() = " + aSet.size());
        System.out.println("aSet = " + aSet);
        Set bSet = new HashSet(aSet);
        System.out.println("bSet.size() = " + bSet.size());
        System.out.println("bSet = " + bSet);
        Set cSet = new HashSet(20);
        System.out.println("cSet.size() = " + cSet.size());
        System.out.println("cSet = " + cSet);
    }
}

```

Q2. See and retype the below code to familiarize yourself with some of the commonly used methods in HashSet.

The class HashSetMethodsDemo iterates through all the arguments passed to the **main** method, and stores them into an HashSet, which is later manipulated using its methods.

Correlate the code and output to understand the usage of the methods.

```java
package q11380;
import java.util.*;
public class HashSetMethodsDemo {
    public static void main(String[] args) {
        Set aSet = new HashSet(args.length);
        for (String argument : args) {
            aSet.add(argument);
        }
        System.out.println("aSet = " + aSet);
        System.out.println("aSet.size() = " + aSet.size());
        boolean removedFlag = aSet.remove("Larry Page");
        System.out.println("Larry Page removedFlag = " + removedFlag);
        System.out.println("aSet = " + aSet);
        aSet.add("Steve Jobs");
        System.out.println("aSet = " + aSet);
        aSet.add("Bill Gates");
        System.out.println("aSet = " + aSet);
    }
}

```

Q3. See the code and retype the same to learn how to iterate over the elements stored in a HashSet.

Note the usage of HashSet class and the for-each statement.

The class scans through all the arguments passed to the **main** method, and stores them into a HashSet if the argument's first char is in uppercase.

The program first uses the **for-each** loop to print all the stored names from the HashSet one name on each line.

Note that HashSet **does not** maintain the any order while iterating on the elements. If we want to have the insertion order to be maintained during iteration, we should use LinkedHashSet, about which we will learn later.

```java
package q11381;
import java.util.*;
public class HashSetIterationDemo {
    public static void main(String[] args) {
        Set namesSet = new HashSet();
        for (String argument : args) {
            if (Character.isUpperCase(argument.charAt(0))) {
                namesSet.add(argument);
            }
        }
        for (Object name : namesSet) {
            System.out.println(name);
        }
    }
}

```

TreeSet

Q1. The TreeSet class implements NavigableSet. NavigableSet extends the SortedSet and the SortedSet in turn extends Set interface.

Unlike a HashSet, implementations of a SortedSet interface guarantee a sorted order (ascending) on the keys. The sort order can also be controlled by providing a custom Comparator implementation.

A NavigableSet interface extends SortedSet, and additionally provides navigation methods for navigating on the sorted entries.

TreeSet is a concrete implementation of SortedSet and NavigableSet interfaces.

See and retype the below code. You will notice that the entries in the TreeSet always remain sorted on the ascending order of their keys.

```java
package q11382;
import java.util.*;
public class TreeSetDemo {
    public static void main(String[] args) {
        Set namesSet = new TreeSet();
        namesSet.add("Ravi");
        namesSet.add("Soma");
        namesSet.add("Budha");
        namesSet.add("Budha");
        System.out.println("namesSet = " + namesSet);
        namesSet.add("Budha");
        namesSet.add("Mangal");
        System.out.println("namesSet = " + namesSet);
        namesSet.remove("Ravi");
        System.out.println("namesSet = " + namesSet);
    }
}

```

LinkedHashSet

Q1. The LinkedHashSet is a subclass of HashSet.

Unlike a **HashSet** which does not maintain order of the added entries, a LinkedHashSet orders the entries in their insertion order by default.

See and retype the below code. You will notice that the entries in the LinkedHashSet always maintain their insertion order.

```java
package q11383;
import java.util.*;
public class LinkedHashSetDemo {
    public static void main(String[] args) {
        Set namesSet = new LinkedHashSet();
        namesSet.add("Samos");
        namesSet.add("Hyderabad");
        namesSet.add("Dallas");
        namesSet.add("Dallas");
        System.out.println("namesSet = " + namesSet);
        namesSet.add("Dallas");
        namesSet.add("Bangalore");
        System.out.println("namesSet = " + namesSet);
        namesSet.add("Athens");
        System.out.println("namesSet = " + namesSet);
    }
}

```

Practice Programs on Sets

Q1. Write a program to understand how to insert elements to a set using the add method. Create a class SetDemo with a main method. Create a set instance and add the given elements to the set.

Add the following elements to the set:

- Ganga
- Krishna
- Godavari
- Yamuna
- Krishna

In the given list of elements **Krishna** is given two times try to add it both times but set does not allow to add duplicate elements and observe the output.

```java
package q11975;
import java.util.*;
public class SetDemo
{
  public static void main (String[]args)
  {
    Set < String > namesSet = new HashSet < String > ();

    namesSet.add ("Ganga");
    namesSet.add ("Krishna");
    namesSet.add ("Godavari");
    namesSet.add ("Yamuna");
    namesSet.add ("Krishna");

    System.out.println (namesSet);

    // Write your code here

    namesSet.add ("Ganga");
    namesSet.add ("Krishna");
    namesSet.add ("Godavari");
    namesSet.add ("Yamuna");
    namesSet.add ("Krishna");

    /*// Write your code here

       System.out.println(namesSet);
     */

    // Write your code here

    // add Krishna again here and observe the output

    System.out.println (namesSet);
    System.out.println("Above result shows how Set does not include the duplicate elements");
  }
}

```

Q2. Write a program to understand the working of the method contains. Create a class SetDemo with a main method. Create a set instance and add the given elements to the set.

Add the following elements to the set:

- Ganga
- Krishna
- Godavari
- Yamuna

Find if the list contains the elements **Godavari** and **Sindhu** if the element present in the list it returns **true** else returns **false** and observe the output.

```java
package q11976;
import java.util.*;
public class SetDemo {
    public static void main(String[] args) {
        Set<String> namesSet = new HashSet <String> ();

        // write  your code here
        namesSet.add("Ganga");
        namesSet.add("Krishna");
        namesSet.add("Godavari");
        namesSet.add("Yamuna");

        System.out.println(namesSet);

        // write your code here
        System.out.println(namesSet.contains("Godavari")+"\n"+namesSet.contains("Sindhu"));

    }
}

```

Q3. Write a program to understand how to remove an element in the Set using the remove method. Create a class SetDemo with a main method and insert the numbers from **one** to **five** and remove the elements **two** and **three** from the set and print the result.

Note: Set does not maintain the order of insertion.

```java
import java.util.*;
public class SetDemo {
    public static void main(String[] args) {
        Set<String> namesSet = new HashSet <String> ();


        // write your code here

        namesSet.add("One");
        namesSet.add("Two");
        namesSet.add("Three");
        namesSet.add("Four");
        namesSet.add("Five");

        System.out.println(namesSet);

        // write your code here
        namesSet.remove("Two");
        namesSet.remove("Three");


        System.out.println(namesSet);
    }
}
```

Q4. Write a program to understand the Set methods clear and size. Create a class SetDemo with a main method and insert the days of week from Sunday through Saturday.

```java
import java.util.*;
public class SetDemo
{
  public static void main (String[]args)
  {
    Set < String > namesSet = new HashSet < String > ();

    // add days of week to the set using the method add

    namesSet.add ("Sunday");

    namesSet.add ("Monday");

    namesSet.add ("Tuesday");

    namesSet.add ("Wednesday");

    namesSet.add ("Thursday");

    namesSet.add ("Friday");
    namesSet.add ("Saturday");


    System.out.println (namesSet);
    System.out.println ("Size of the set is " + namesSet.size ());  // print the size of the set

    // clear all elements in the set
    namesSet.clear ();

    System.out.println ("The set after removing all the elements " +namesSet);
    System.out.println ("Size of the set after removing the elements is " +namesSet.size ());
  }
}
```

Q5. Create a class SetDemo with a main method. The method takes inputs from the command line arguments. Create a set instance and add these inputs to the set. Fill the missing code in the below program and print the output as shown in the example.

Sample Input and Output:
Cmd Args : Hyderabad Kolkata Hyderabad Delhi Mumbai
Delhi
Kolkata
Mumbai
Hyderabad
From the example we can observe that the set does not allow duplicate elements, also does not maintain the insertion order

```java
package q24095;
import java.util.*;
public class SetDemo
{
  public static void main (String[]args)
  {
    Set < String > namesSet = new HashSet < String > ();
// write your code here

// iterate over the set using iterator and print the output
    for (int i = 0; i < args.length; i++)
      {

    String name = args[i];

      namesSet.add (name);

      }

    // iterate over the set using iterator and print the output

    for (String name:namesSet)
      {

    System.out.println (name);

      }
  }
}

```