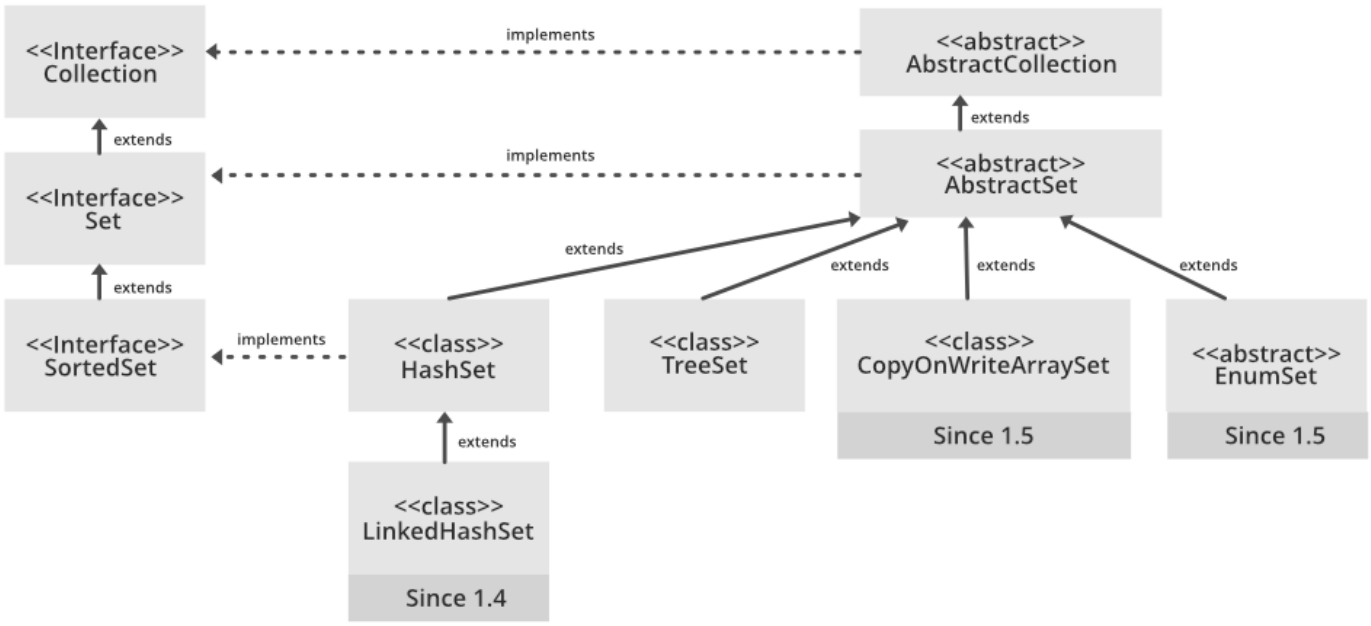
HashSet in Java

The **HashSet** class implements the Set interface, backed by a hash table which is actually a HashMap instance. No guarantee is made as to the iteration order of the set which means that the class does not guarantee the constant order of elements over time. This class permits the null element. The class also offers constant time performance for the basic operations like add, remove, contains, and size assuming the hash function disperses the elements properly among the buckets, which we shall see further in the article.

A few important features of HashSet are:

- Implements Set Interface.
- The underlying data structure for HashSet is Hashtable.
- As it implements the Set Interface, duplicate values are not allowed.
- Objects that you insert in HashSet are not guaranteed to be inserted in the same order. Objects are inserted based on their hash code.
- NULL elements are allowed in HashSet.
- HashSet also implements **Serializable** and **Cloneable** interfaces.

The Hierarchy of HashSet is as follows:



HashSet extends Abstract Set<E> class and implements Set<E>, Cloneable, and Serializable interfaces where E is the type of elements maintained by this set. The directly known subclass of HashSet is LinkedHashSet.

Now for the maintenance of constant time performance, iterating over HashSet requires time proportional to the sum of the HashSet instance's size (the number of elements) plus the "capacity" of the backing HashMap instance (the number of buckets). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

- **Initial Capacity:** The initial capacity means the number of buckets when hashtable (HashSet internally uses hashtable data structure) is created. The number of buckets will be automatically increased if the current size gets full.

- **Load Factor:** The load factor is a measure of how full the HashSet is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

$$\text{Load Factor} = \frac{\text{Number of stored elements in the table}}{\text{Size of the hash table}}$$

Example: *If internal capacity is 16 and the load factor is 0.75 then the number of buckets will automatically get increased when the table has 12 elements in it.*

Effect on performance: Load factor and initial capacity are two main factors that affect the performance of HashSet operations. A load factor of 0.75 provides very effective performance with respect to time and space complexity. If we increase the load factor value more than that then memory overhead will be reduced (because it will decrease internal rebuilding operation) but, it will affect the add and search operation in the hashtable. To reduce the rehashing operation we should choose initial capacity wisely. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operation will ever occur.

Note: *The implementation in a HashSet is not synchronized, in the sense that if multiple threads access a hash set concurrently, and at least one of the threads modifies the set, it must be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be "wrapped" using the Collections.synchronizedSet method. This is best done at creation time, to prevent accidental unsynchronized access to the set as shown below:*

```
Set s = Collections.synchronizedSet(new HashSet(...));
```

Syntax: Declaration of HashSet

```
public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable,
Serializable
```

where **E** is the type of elements stored in a HashSet.

Internal working of a HashSet: All the classes of Set interface are internally backed up by Map. HashSet uses HashMap for storing its object internally. You must be wondering that to enter a value in HashMap we need a key-value pair, but in HashSet, we are passing only one value.

Storage in HashMap: Actually the value we insert in HashSet acts as a key to the map Object and for its value, java uses a constant variable. So in the key-value pair, all the values will be the same.

Implementation of HashSet in Java doc

```
private transient HashMap map;

// Constructor - 1
// All the constructors are internally creating HashMap Object.
public HashSet()
{
    // Creating internally backing HashMap object
    map = new HashMap();
}

// Constructor - 2
public HashSet(int initialCapacity)
{
    // Creating internally backing HashMap object
    map = new HashMap(initialCapacity);
```

```
}

// Dummy value to associate with an Object in Map
private static final Object PRESENT = new Object();
```
If we look at the ***add()*** method of HashSet class:
```
public boolean add(E e)
{
   return map.put(e, PRESENT) == null;
}
```
We can notice that, add() method of HashSet class internally calls the ***put()*** method of backing the HashMap object by passing the element you have specified as a key and constant "PRESENT" as its value. ***remove()*** method also works in the same manner. It internally calls the remove method of the Map interface.
```
public boolean remove(Object o)
{
  return map.remove(o) == PRESENT;
}
```
HashSet not only stores unique Objects but also a unique Collection of Objects like ArrayList<E>, LinkedList<E>, Vector<E>,..etc.

Implementation:

- Java

```
// Java program to illustrate the concept
// of Collection objects storage in a HashSet
import java.io.*;
import java.util.*;

class CollectionObjectStorage {

    public static void main(String[] args)
    {
        // Instantiate an object of HashSet
        HashSet<ArrayList> set = new HashSet<>();

        // create ArrayList list1
        ArrayList<Integer> list1 = new ArrayList<>();

        // create ArrayList list2
        ArrayList<Integer> list2 = new ArrayList<>();

        // Add elements using add method
        list1.add(1);
        list1.add(2);
        list2.add(1);
        list2.add(2);
        set.add(list1);
        set.add(list2);

        // print the set size to understand the
        // internal storage of ArrayList in Set
        System.out.println(set.size());
    }
}
```

Output:

1

Before storing an Object, HashSet checks whether there is an existing entry using hashCode() and equals() methods. In the above example, two lists are considered equal if they have the same elements in the same order. When you invoke the hashCode() method on the two lists, they both would give the same hash since they are equal.

HashSet does not store duplicate items, if you give two Objects that are equal then it stores only the first one, here it is list1.

Constructors of HashSet class

In order to create a HashSet, we need to create an object of the HashSet class. The HashSet class consists of various constructors that allow the possible creation of the HashSet. The following are the constructors available in this class.

1. HashSet(): This constructor is used to build an empty HashSet object in which the default initial capacity is 16 and the default load factor is 0.75. If we wish to create an empty HashSet with the name hs, then, it can be created as:

```
HashSet<E> hs = new HashSet<E>();
```

2. HashSet(int initialCapacity): This constructor is used to build an empty HashSet object in which the initialCapacity is specified at the time of object creation. Here, the default loadFactor remains 0.75.

```
HashSet<E> hs = new HashSet<E>(int initialCapacity);
```

3. HashSet(int initialCapacity, float loadFactor): This constructor is used to build an empty HashSet object in which the initialCapacity and loadFactor are specified at the time of object creation.

```
HashSet<E> hs = new HashSet<E>(int initialCapacity, float loadFactor);
```

4. HashSet(Collection): This constructor is used to build a HashSet object containing all the elements from the given collection. In short, this constructor is used when any conversion is needed from any Collection object to the HashSet object. If we wish to create a HashSet with the name hs, it can be created as:

```
HashSet<E> hs = new HashSet<E>(Collection C);
```

Implementation:

- Java

```
// Java program to Demonstrate Working of HashSet Class

// Importing required classes
import java.util.*;

// Main class
// HashSetDemo
class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        // Creating an empty HashSet
        HashSet<String> h = new HashSet<String>();

        // Adding elements into HashSet
        // using add() method
        h.add("India");
        h.add("Australia");
        h.add("South Africa");
```

```
        // Adding duplicate elements
        h.add("India");

        // Displaying the HashSet
        System.out.println(h);
        System.out.println("List contains India or not:"
                           + h.contains("India"));

        // Removing items from HashSet
        // using remove() method
        h.remove("Australia");
        System.out.println("List after removing Australia:"
                           + h);

        // Display message
        System.out.println("Iterating over list:");

        // Iterating over hashSet items
        Iterator<String> i = h.iterator();

        // Holds true till there is single element remaining
        while (i.hasNext())

            // Iterating over elements
            // using next() method
            System.out.println(i.next());
    }
}
```

Output:

```
[South Africa, Australia, India]
List contains India or not:true
List after removing Australia:[South Africa, India]
Iterating over list:
South Africa
India
```

Performing Various Operations on HashSet

Let's see how to perform a few frequently used operations on the HashSet.

Operation 1: Adding Elements

In order to add an element to the HashSet, we can use the add() method. However, the insertion order is not retained in the HashSet. We need to keep a note that duplicate elements are not allowed and all the duplicate elements are ignored.

Example

- Java

```
// Java program to Adding Elements to HashSet

// Importing required classes
import java.io.*;
import java.util.*;

// Main class
```

```
// AddingElementsToHashSet
class GFG {

    // Method 1
    // Main driver method
    public static void main(String[] args)
    {
        // Creating an empty HashSet of string entities
        HashSet<String> hs = new HashSet<String>();

        // Adding elements using add() method
        hs.add("Geek");
        hs.add("For");
        hs.add("Geeks");

        // Printing all string el=ntries inside the Set
        System.out.println("HashSet elements : " + hs);
    }
}
```

Output:

```
HashSet elements : [Geek, For, Geeks]
```

Operation 2: Removing Elements

The values can be removed from the HashSet using the remove() method.

Example

- Java

```
// Java program Illustrating Removal Of Elements of HashSet

// Importing required classes
import java.io.*;
import java.util.*;

// Main class
// RemoveElementsOfHashSet
class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        // Creating an
        HashSet<String> hs = new HashSet<String>();

        // Adding elements to above Set
        // using add() method
        hs.add("Geek");
        hs.add("For");
        hs.add("Geeks");
        hs.add("A");
        hs.add("B");
        hs.add("Z");

        // Printing the elements of HashSet elements
        System.out.println("Initial HashSet " + hs);

        // Removing the element B
```

```
        hs.remove("B");

        // Printing the updated HashSet elements
        System.out.println("After removing element " + hs);

        // Returns false if the element is not present
        System.out.println("Element AC exists in the Set : "
                           + hs.remove("AC"));
    }
}
```

Output:

```
Initial HashSet [A, B, Geek, For, Geeks, Z]
After removing element [A, Geek, For, Geeks, Z]
Element AC exists in the Set : false
```

Operation 3: Iterating through the HashSet

Iterate through the elements of HashSet using the iterator() method. Also, the most famous one is to use the enhanced for loop.

Example

- Java

```
// Java Program to Illustrate Iteration Over HashSet

// Importing required classes
import java.io.*;
import java.util.*;

// Main class
// IterateTheHashSet
class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        // Creating an empty HashSet of string entries
        HashSet<String> hs = new HashSet<String>();

        // Adding elements to above Set
        // using add() method
        hs.add("Geek");
        hs.add("For");
        hs.add("Geeks");
        hs.add("A");
        hs.add("B");
        hs.add("Z");

        // Iterating though the HashSet using iterators
        Iterator itr = hs.iterator();

        // Holds true till there is single element
        // remaining in Set
        while (itr.hasNext())

            // Traversing elements and printing them
```

```
            System.out.print(itr.next() + ", ");
        System.out.println();

        // Using enhanced for loop for traversal
        for (String s : hs)

            // Traversing elements and printing them
            System.out.print(s + ", ");
        System.out.println();
    }
}
```

Output

```
A, B, Geek, For, Geeks, Z,
A, B, Geek, For, Geeks, Z,
```

Time Complexity of HashSet Operations: *The underlying data structure for HashSet is hashtable. So amortize (average or usual case) time complexity for add, remove and look-up (contains method) operation of HashSet takes* ***O(1)*** *time.*

Methods in HashSet

METHOD	DESCRIPTION
add(E e)	Used to add the specified element if it is not present, if it is present then return false.
clear()	Used to remove all the elements from set.
contains(Object o)	Used to return true if an element is present in set.
remove(Object o)	Used to remove the element if it is present in set.
iterator()	Used to return an iterator over the element in the set.
isEmpty()	Used to check whether the set is empty or not. Returns true for empty and false for a non-empty condition for set.
size()	Used to return the size of the set.
clone()	Used to create a shallow copy of the set.

Methods inherited from class java.util.AbstractSet

METHOD	DESCRIPTION
equals()	Used to verify the equality of an Object with a HashSet and compare them. The list returns true only if both HashSet contains same elements, irrespective of order.
hashcode()	Returns the hash code value for this set.
removeAll(collection)	This method is used to remove all the elements from the collection which are present in the set. This method returns true if this set changed as a result of the call.

Methods inherited from class java.util.AbstractCollection

METHOD	DESCRIPTION
addAll(collection)	This method is used to append all of the elements from the mentioned collection to the existing set. The elements are added randomly without following any specific order.
containsAll(collection)	This method is used to check whether the set contains all the elements present in the given collection or not. This method returns true if the set contains all the elements and returns false if any of the elements are missing.
retainAll(collection)	This method is used to retain all the elements from the set which are mentioned in the given collection. This method returns true if this set changed as a result of the call.
toArray()	This method is used to form an array of the same elements as that of the Set.
toString()	The toString() method of Java HashSet is used to return a string representation of the elements of the HashSet Collection.

Methods declared in interface java.util.Collection

METHOD	DESCRIPTION
parallelStream()	Returns a possibly parallel Stream with this collection as its source.
removeIf(Predicate<? super E> filter)	Removes all of the elements of this collection that satisfy the given predicate.
stream()	Returns a sequential Stream with this collection as its source.
toArray(IntFunction<T[]> generator)	Returns an array containing all of the elements in this collection, using the provided generator function to allocate the returned array.

Methods declared in interface java.lang.Iterable

METHOD	DESCRIPTION
forEach(Consumer<? super T> action)	Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.

Methods declared in interface java.util.Set

METHOD	DESCRIPTION
addAll(Collection<? extends E> c)	Adds all of the elements in the specified collection to this set if they're not already present (optional operation).
containsAll (Collection<?> c)	Returns true if this set contains all of the elements of the specified collection.
equals(Object o)	Compares the specified object with this set for equality.
hashCode()	Returns the hash code value for this set.

removeAll(Collection<?> c)	Removes from this set all of its elements that are contained in the specified collection (optional operation).
retainAll(Collection<?> c)	Retains only the elements in this set that are contained in the specified collection (optional operation).
toArray()	Returns an array containing all of the elements in this set.
toArray(T[] a)	Returns an array containing all of the elements in this set; the runtime type of the returned array is that of the specified array.

HashSet vs HashMap

BASIS	HashSet	HashMap
Implementation	HashSet implements Set interface.	HashMap implements Map interface.
Duplicates	HashSet doesn't allow duplicate values.	HashMap store key, value pairs and it does not allow duplicate keys. If key is duplicate then the old key is replaced with the new value.
Number of objects during storing objects	HashSet requires only one object add(Object o).	HashMap requires two objects put(K key, V Value) to add an element to the HashMap object.
Dummy value	HashSet internally uses HashMap to add elements. In HashSet, the argument passed in add(Object) method serves as key K. Java internally associates dummy value for each value passed in add(Object) method.	HashMap does not have any concept of dummy value.
Storing or Adding mechanism	HashSet internally uses the HashMap object to store or add the objects.	HashMap internally uses hashing to store or add objects
Faster	HashSet is slower than HashMap.	HashMap is faster than HashSet.
Insertion	HashSet uses the add() method for add or storing data.	HashMap uses the put() method for storing data.
Example	HashSet is a set, e.g. {1, 2, 3, 4, 5, 6, 7}.	HashMap is a key -> value pair(key to value) map, e.g. {a -> 1, b -> 2, c -> 2, d -> 1}.

HashSet vs TreeSet

BASIS	HashSet	TreeSet
Speed and internal implementation	For operations like search, insert and delete. It takes constant time for these operations on average.	TreeSet takes O(Log n) for search, insert and delete which is higher than HashSet. But TreeSet keeps sorted data. Also, it supports operations like higher() (Returns least higher element), floor(), ceiling(), etc. These

	HashSet is faster than TreeSet. HashSet is Implemented using a hash table.	operations are also O(Log n) in TreeSet and not supported in HashSet. TreeSet is implemented using a Self Balancing Binary Search Tree (Red-Black Tree). TreeSet is backed by TreeMap in Java.
Ordering	Elements in HashSet are not ordered.	TreeSet maintains objects in Sorted order defined by either Comparable or Comparator method in Java. TreeSet elements are sorted in ascending order by default. It offers several methods to deal with the ordered set like first(), last(), headSet(), tailSet(), etc.
Null Object	HashSet allows the null object.	TreeSet doesn't allow null Object and throw NullPointerException, Why, because TreeSet uses compareTo() method to compare keys and compareTo() will throw java.lang.NullPointerException.
Comparison	HashSet uses equals() method to compare two objects in Set and for detecting duplicates.	TreeSet uses compareTo() method for same purpose. If equals() and compareTo() are not consistent, i.e. for two equal object equals should return true while compareTo() should return zero, then it will break the contract of the Set interface and will allow duplicates in Set implementations like TreeSet

This article is contributed by **Dharmesh Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.