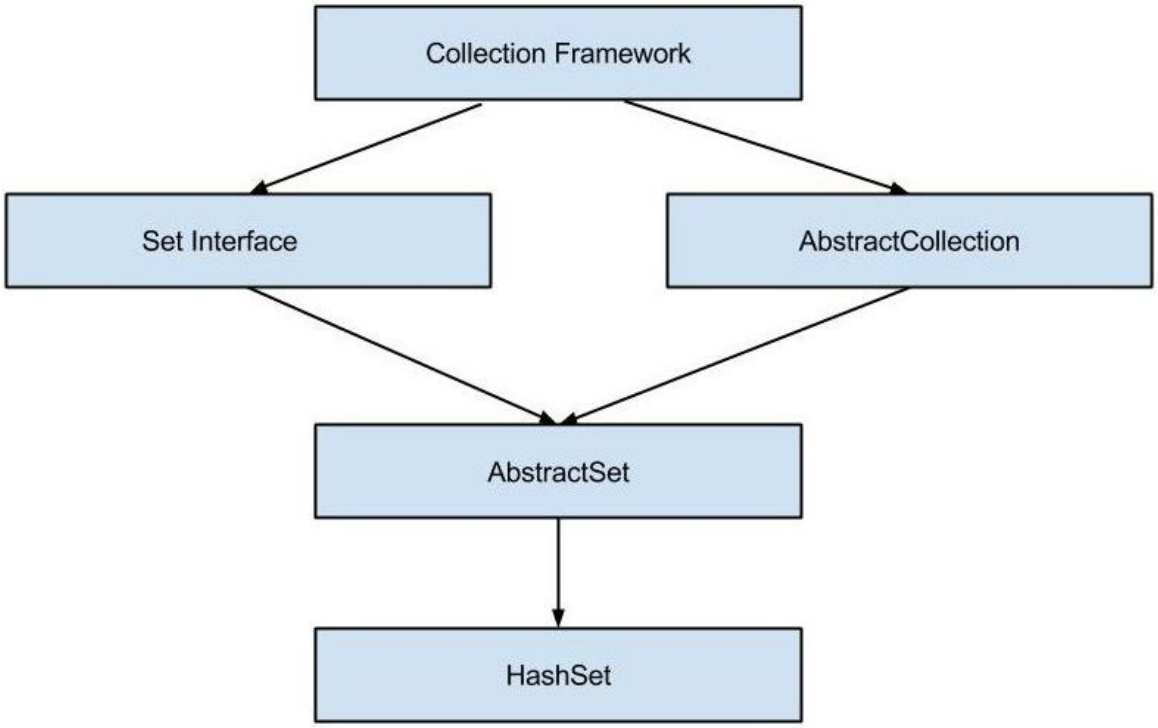
Java HashSet class

Java HashSet class implements the `Set` interface, backed by a hash table(actually a `HashMap` instance). If does not offer any guarantees as to the iteration order, and allows `null` element.



2. HashSet Features

- It implements `Set` Interface.
- Duplicate values are not allowed in HashSet.
- One NULL element is allowed in HashSet.
- It is un-ordered collection and makes no guarantees as to the iteration order of the set.
- This class offers constant time performance for the basic operations(add, remove, contains and size).
- HashSet is not synchronized. If multiple threads access a hash set concurrently, and at least one of the threads modifies the set, it must be synchronized externally.
- Use Collections.synchronizedSet(new HashSet()) method to get the synchronized hashset.

- The iterators returned by this class's iterator method are fail-fast and may throw `ConcurrentModificationException` if the set is modified at any time after the iterator is created, in any way except through the iterator's own `remove()` method.
- HashSet also implements Searlizable and Cloneable interfaces.

2.1. Initial Capacity

The initial capacity means the number of buckets (in backing HashMap) when hashset is created. The number of buckets will be automatically increased if the current size gets full.

Default initial capacity is 16. We can override this default capacity by passing default capacity in it's constructor HashSet(int initialCapacity).

```
1. HashSet<String> hs=new HashSet<>();
```

Where <String> is the generic type parameter. It represents the type of element storing in the HashSet.

HashSet implements Set interface. It guarantees uniqueness. It is achieved by storing elements as keys with the same value always.

When we create an object of HashSet, it internally creates an instance of HashMap with default initial capacity 16.

Working :

```
1. public class HashSet<E> extends AbstractSet<E>
2. {
3. private transient HashMap<E,Object> map;
4. // Dummy value to associate with an Object in the backing Map
5. private static final Object PRESENT = new Object();
6. public HashSet()
7. {
8. map = new HashMap<>();
9. }
10. public boolean add(E e)
```

```
11. {
12. return map.put(e, PRESENT)==null;
13. }
14. }
```

In the above code a call to add(object) is delegated to put(key, value) internally. Where key is the object we have passed and the value is another object, called PRESENT. It is a constant in java.util.HashSet.

We are achieving uniqueness in Set internally through HashMap. When we create an object of HashSet, it will create an object of HashMap. We know that each key is unique in the HashMap. So, we pass the argument in the add(E e) method. Here, we need to associate some value to the key. It will associate with Dummy value that is (new Object()) which is referred by Object reference PRESENT.

When we add an element in HashSet like hs.add("India"), Java does internally is that it will put that element E here "India" as a key into the HashMap (generated during HashSet object creation). It will also put some dummy value that is Object's object is passed as a value to the key.

The important points about put(key, value) method is that:

- If the Key is unique and added to the map, then it will return null
- If the Key is duplicate, then it will return the old value of the key.

When we invoke add() method in HashSet, Java internally checks the return value of map.put(key, value) method with the null value.

```
1. public boolean add(E e)
2. {
3. return map.put(e, PRESENT==null);
4. }
```

- If the method map.put(key, value) returns null, then the method map.put(e, PRESENT)==null will return true internally, and the element added to the HashSet.
- If the method map.put(key, value) returns the old value of the key, then the method map.put(e, PRESENT)==null will return false internally, and the element will not add to the HashSet.

2.2. Load Factor

The load factor is a measure of how full the HashSet is allowed to get before its capacity is automatically increased. Default load factor is 0.75.

This is called threshold and is equal to (DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY). When HashSet elements count exceed this threshold, HashSet is resized and new capacity is double the previous capacity.

With default HashSet, the internal capacity is 16 and the load factor is 0.75. The number of buckets will automatically get increased when the table has 12 elements in it.

3. HashSet Constructors

The HashSet has four types of constructors:

1. HashSet(): initializes a default HashSet instance with the default initial capacity (16) and default load factor (0.75).
2. HashSet(int capacity): initializes a HashSet with a specified capacity and default load factor (0.75).
3. HashSet(int capacity, float loadFactor): initializes HashSet with specified initial capacity and specified load factor.
4. HashSet(Collection c): initializes a HashSet with same elements as the specified collection.

HashSet Usecases

HashSet is very much like ArrayList class. It additionally restrict the duplicate values. So when we have a requirement where we want to store only distinct elements, we can choose HashSet.

A real-life usecase for HashSet can be storing data from stream where the stream may contain duplicate records, and we are only interested in distinct records.

Another usecase can be finding distinct words in a given sentence.

Java HashSet Performance

- HashSet class offers constant time performance of O(1)for the basic operations(add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets.
- Iterating over this set 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 ofbuckets). Thus, it’s very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

From above discussion, it is evident that HashSet is very useful collection class in cases where we want to handle duplicate records. It provided predictable performance for basic operations.