Set

In Java, a Set is an interface that extends the Collection interface. A Set is a collection of unique elements, meaning that no element can be repeated within the Set. Sets are typically used when you want to store a collection of items, but you only need to keep track of unique items, and you don't care about the order in which they were added.

HashSet is one of the most commonly used implementation class of Set. Here's an example of how to create a Set in Java:

```java
Set<String> mySet = new HashSet<>();
```

You can also create a Set from an existing Collection, like this. This creates a new HashSet object that contains all the unique elements in myList.

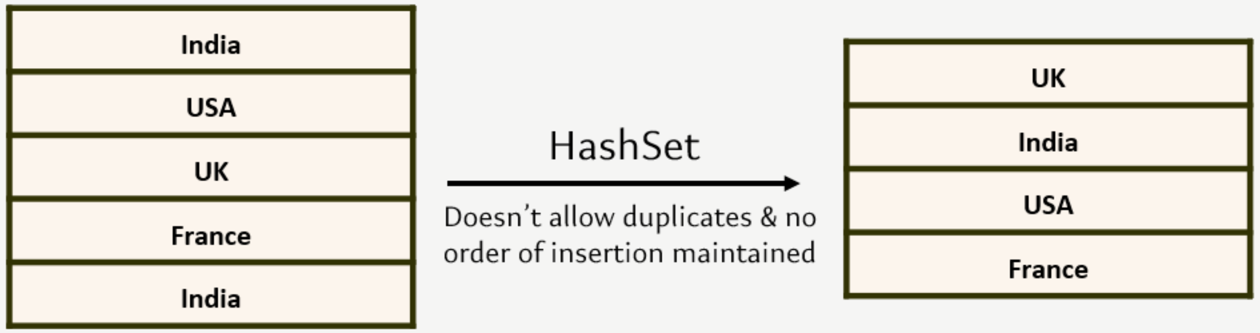
```java
List<String> myList = new ArrayList<>();
// add some elements to the list
Set<String> mySet = new HashSet<>(myList);
```

HashSet

eazy bytes

Creating HashSet

```
// Create a new HashSet
Set< String> countries = new HashSet<>();
countries.add("India");
countries.add("USA");
countries.add("UK");
countries.add("France");
countries.add("India");
```



The elements inside HashSet are stored using hashing mechanism & hash table. It allows storing only one null value. HashSet class is non-synchronized & is useful for search operations.

How HashSet Maintains Uniqueness

When an element is added to the HashSet, its hash code is computed using the hashing function, and the HashSet checks if an element with the same hash code already exists in the set. If an element with the same hash code is not already present, the element is added to the HashSet.

If an element with the same hash code is already present in the HashSet, the HashSet uses the equals() method to compare the new element with the existing one. If the equals() method returns true, then the new element is considered a duplicate and is not added to the HashSet. If the equals() method returns false, then the new element is considered unique and is added to the HashSet.

By using both the hashCode() and the equals() method to determine uniqueness, the HashSet ensures that only unique elements are stored in the set.

How HashSet works internally ?

Many Java developers may not be aware that HashSet is internally implemented using HashMap in Java. Understanding how HashMap works internally can provide insights into how HashSet operates. One might wonder why HashSet, which stores only one object, utilizes HashMap, typically designed for key-value pairs.

As HashSet implements the Set interface, it must ensure the uniqueness of elements. This is accomplished by consistently storing elements as keys with the same value inside the internal HashMap. Do you remember that HashMap allows only unique keys ? The same is exploited to build HashSet

When we create an object of HashSet in Java behind the scenes a internal HashMap created. Below is the code of the constructor inside HashSet class,

```
 public HashSet() {
  map = new HashMap<>();
}
```

But you may have a question, what value is going to be stored inside HashMap against a given key. For the same let see the add method of HashSet,

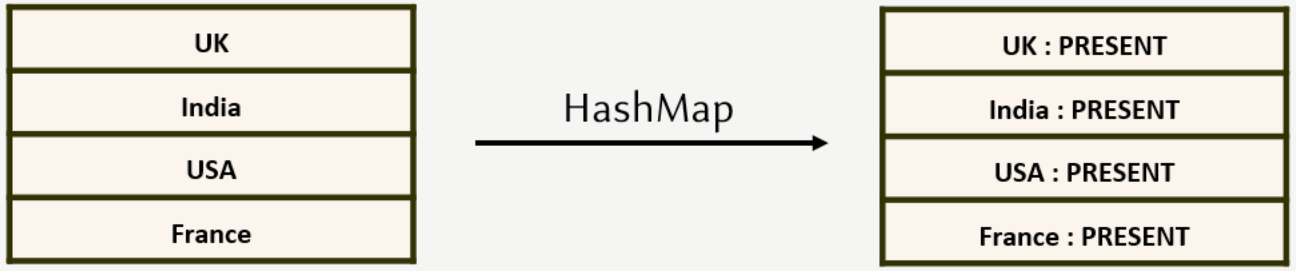
```
public boolean add(E e) {
  return map.put(e, PRESENT)==null;
}
```

As evident in the code snippet above, invoking the `add(Object)` method is essentially a delegation to the internal `put(Key, Value)` operation. In this context, the key is the object provided, and the corresponding value is another object denoted as PRESENT. It's noteworthy that PRESENT is a constant defined in the java.util.HashSet class.

How HashSet works internally ?

How HashSet stores internally ?

```
// Create a new HashSet
Set< String> countries = new HashSet<>();
countries.add("India");
countries.add("USA");
countries.add("UK");
countries.add("France");
```



HashSet does not offer a direct method for retrieving objects, such as get(Key key) in HashMap or get(int index) in List. The only way to retrieve objects from the HashSet is through iterating the entire HashSet. This can be achieved by using iterator, for, for-each ,etc. Below is the code inside the iterator() method of HashSet,

```
public Iterator<E> iterator() {
        return map.keySet().iterator();
    }
```

The presence of an object in a HashSet can be verified using the contains() method, which employs the equals() method for comparison. Additionally, the remove() method can be employed to eliminate objects from the HashSet.

It's important to note that as the elements of the HashSet serve as keys in the underlying HashMap, they are required to implement both the equals() and hashCode() methods.

eazy
bytes

Iterating HashSet

We can iterate the elements of HashSet using any of the below approaches,

1 Using a for-each loop: This is the simplest and most commonly used method to iterate over a HashSet. It involves using a for-each loop to iterate over the elements of the HashSet.

```java
Set<String> superpowers = new HashSet<>();
// Superheroes and their unique abilities
superpowers.add("Invisibility");
superpowers.add("Teleportation");
superpowers.add("Mind Reading");
superpowers.add("Super Strength");
superpowers.add("Time Travel");

// Villains and their mischievous powers
superpowers.add("Laser Vision");
superpowers.add("Weather Manipulation");
superpowers.add("Telekinesis");
superpowers.add("Shape-Shifting");
superpowers.add("Chaos Induction");

for (String superPower: superpowers) {
    System.out.println(superPower.toUpperCase());
}
```

2 Using an iterator: The iterator provides a way to iterate over the elements of a HashSet while removing elements from the set if required. The iterator() method returns an iterator object that can be used to iterate over the elements of the HashSet.

```java
Set<String> superpowers = new HashSet<>();
// Superheroes and their unique abilities
superpowers.add("Invisibility");
superpowers.add("Teleportation");
superpowers.add("Mind Reading");
superpowers.add("Super Strength");
superpowers.add("Time Travel");

// Villains and their mischievous powers
superpowers.add("Laser Vision");
superpowers.add("Weather Manipulation");
superpowers.add("Telekinesis");
superpowers.add("Shape-Shifting");
superpowers.add("Chaos Induction");

Iterator<String> iterator = superpowers.iterator();

while (iterator.hasNext()) {
    String superPower = iterator.next();
    System.out.println(superPower.toLowerCase());
}
```

Set operations

eazy bytes

The various methods of the HashSet class can also be used to perform various set operations.

U

Union of Sets using addAll()

```
HashSet1: [7, 4]
HashSet2: [1, 6]
hashset1.addAll(hashset2);
Output : [6, 1, 7, 4]
```

I

Intersection of Sets using retainAll()

```
HashSet1: [7, 4]
HashSet2: [1, 7]
hashset1.retainAll(hashset2);
Output : [7]
```

D

Difference of Sets using removeAll()

```
HashSet1: [7, 4]
HashSet2: [1, 7]
hashset1.removeAll(hashset2);
Output : [4]
```

S

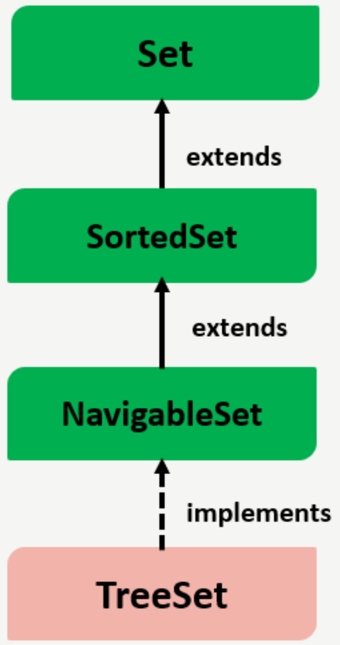
Check if subset using containsAll()

```
HashSet1: [7, 4, 1]
HashSet2: [1, 7]
boolean output = hashset1.containsAll(hashset2);
Output : true
```

TreeSet

In Java, TreeSet is a collection that stores elements in a sorted and ascending order. It implements the SortedSet interface and is based on the TreeMap implementation. TreeSet uses a red-black tree data structure to maintain the sorted order of elements.

TreeSet does not allow null elements, as it relies on the natural ordering or custom Comparator to maintain the sorted order.



Some common methods provided by the TreeSet class include:

add(E e): Adds the specified element to the TreeSet.

clear(): Removes all the elements from the TreeSet.

contains(Object o): Returns true if the specified element is present in the TreeSet.

first(): Returns the first (lowest) element in the TreeSet.

last(): Returns the last (highest) element in the TreeSet.

remove(Object o): Removes the specified element from the TreeSet.

size(): Returns the number of elements in the TreeSet.

TreeSet

```java
// Creating a TreeSet of strings
TreeSet<String> treeSet = new TreeSet<>();

// Adding elements to the TreeSet
treeSet.add("India");
treeSet.add("USA");
treeSet.add("Germany");

// Printing the TreeSet
System.out.println("TreeSet: " + treeSet);

// Removing an element from the TreeSet
treeSet.remove("Germany");

// Printing the TreeSet after removal
System.out.println("TreeSet after removal: " + treeSet);

// Getting the size of the TreeSet
System.out.println("Size of TreeSet: " + treeSet.size());

// Checking if an element is present in the TreeSet
System.out.println("Is India present in TreeSet? " + treeSet.contains("India"));

// Getting the first and last elements of the TreeSet
System.out.println("First element of TreeSet: " + treeSet.first());
System.out.println("Last element of TreeSet: " + treeSet.last());
```

Here is an example of creating and using a TreeSet in Java:

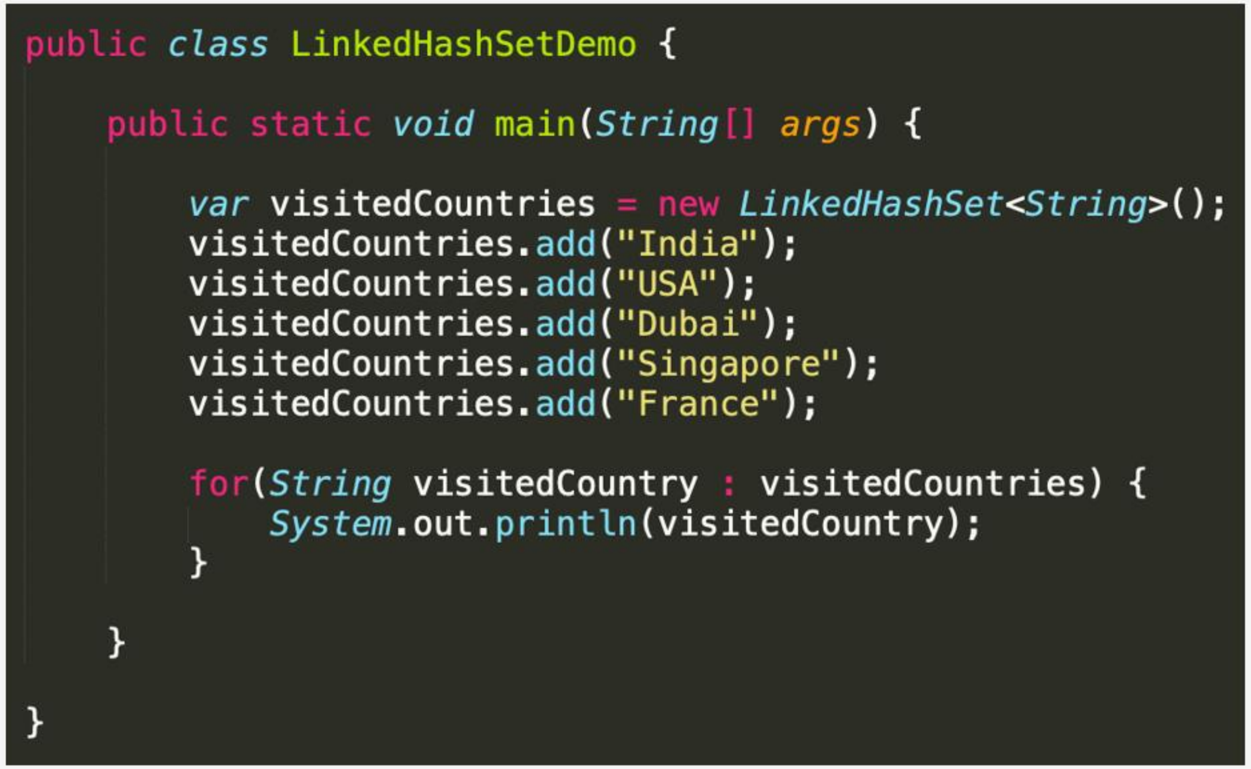
output

```
TreeSet: [Germany, India, USA]
TreeSet after removal: [India, USA]
Size of TreeSet: 2
Is India present in TreeSet? true
First element of TreeSet: India
Last element of TreeSet: USA
```

LinkedHashSet

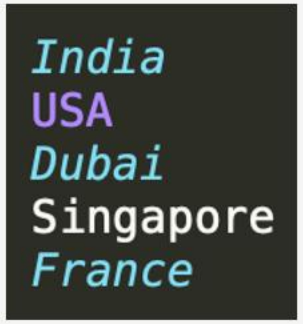
In Java, LinkedHashSet is a class that extends HashSet, implements SequencedSet and maintains the order of elements based on the order of their insertion. It means that when you iterate over the entries of a LinkedHashSet, they are returned in the order in which they were added. LinkedHashSet combines the features of a hash set and a linked list.

```
public class LinkedHashSet<E> extends HashSet<E>
    implements SequencedSet<E>{}
```



```java
public class LinkedHashSetDemo {

    public static void main(String[] args) {

        var visitedCountries = new LinkedHashSet<String>();
        visitedCountries.add("India");
        visitedCountries.add("USA");
        visitedCountries.add("Dubai");
        visitedCountries.add("Singapore");
        visitedCountries.add("France");

        for(String visitedCountry : visitedCountries) {
            System.out.println(visitedCountry);
        }

    }

}
```

Output:



```
India
USA
Dubai
Singapore
France
```

From Java 9

Creating Immutable Sets

eazy bytes

Immutable Set Static Factory Methods

The Set.of static factory methods provide a convenient way to create immutable sets. A set is a collection that does not contain duplicate elements. If a duplicate entry is detected, then an IllegalArgumentException is thrown. Null values are not allowed. The syntax of these methods is:

```
Set.of()
Set.of(e1)
Set.of(e1, e2)  // fixed-argument form overloads up to 10 elements
Set.of(elements...)  // varargs form supports an arbitrary number of elements or an array
```

In JDK 8:

```
Set<String> stringSet = new HashSet<>(Arrays.asList("a", "b", "c"));
stringSet = Collections.unmodifiableSet(stringSet);
```

In JDK 9:

```
Set<String> stringSet = Set.of("a", "b", "c");
```

List vs Set

List and Set are both interfaces in Java that represent collections of elements. However, there are several key differences between them.

Lists can contain duplicate elements. If you need to maintain a collection of elements that can contain duplicates, use a List.	Sets doesn’t allow duplicates. If you need to ensure that each element is unique, use a Set.
Lists maintain the order of insertion, while Sets do not. If the order of elements is important, use a List.	Sets does not maintain the order of insertion. If you do not care about the order, use a Set.
Lists allow for positional access to elements, meaning that you can access an element by its index.	Sets do not provide positional access. Instead, you can only check if an element is in the Set or not.
Lists generally have better performance than Sets for positional access. This is because Lists use an index to access elements, which is faster than checking every element in a Set for membership.	Sets have better performance than Lists for testing membership.

In summary, use a List when you need to maintain duplicates and/or preserve the order of elements, and use a Set when you need to ensure uniqueness and/or test for membership quickly.