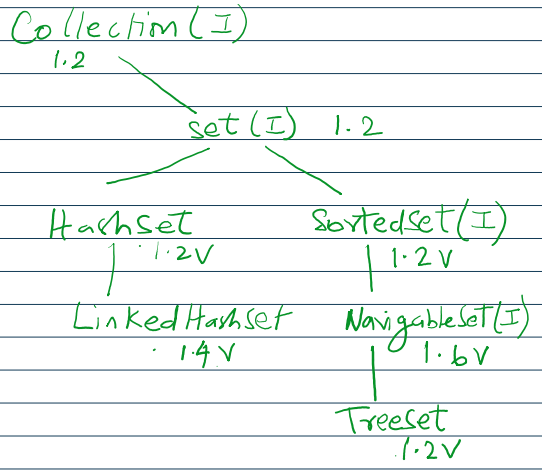
# Set interface

28 March 2025 09:38

Set is a child interface of Collection. If we want to represent a group of individual object as a single entity where duplicates are not allowed and insertion order is not preserved.

Set interface doesn't contain any new method and have to used Collection interface methods.




## Official Documentation

```
public interface Set<E>
extends Collection<E>
```

A collection that contains no duplicate elements. More formally, sets contain no pair of elements `e1` and `e2` such that `e1.equals(e2)`, and at most one null element. As implied by its name, this interface models the mathematical *set* abstraction.

The `Set` interface places additional stipulations, beyond those inherited from the `Collection` interface, on the contracts of all constructors and on the contracts of the `add`, `equals` and `hashCode` methods. Declarations for other inherited methods are also included here for convenience. (The specifications accompanying these declarations have been tailored to the `Set` interface, but they do not contain any additional stipulations.)

The additional stipulation on constructors is, not surprisingly, that all constructors must create a set that contains no duplicate elements (as defined above).

Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects `equals` comparisons while the object is an element in the set. A special case of this prohibition is that it is not permissible for a set to contain itself as an element.

Some set implementations have restrictions on the elements that they may contain. For example, some implementations prohibit null elements, and some have restrictions on the types of their elements. Attempting to add an ineligible element throws an unchecked exception, typically `NullPointerException` or `ClassCastException`. Attempting to query the presence of an ineligible element may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible element whose completion would not result in the insertion of an ineligible element into the set may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

This interface is a member of the Java Collections Framework.

**Since:**
1.2

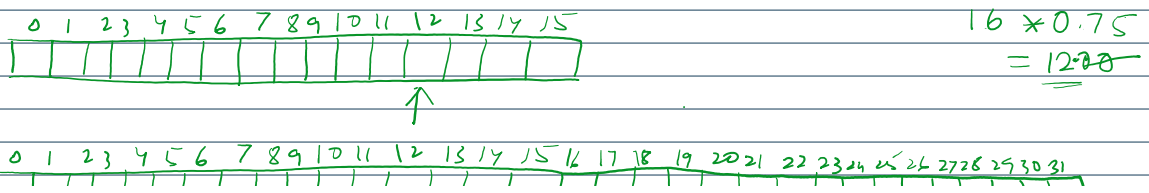
## HashSet

- **The underlying data structure is hash table.**
- **Duplicate objects are not allowed.**
- **Insertion order is not preserved.**
- **And it is based on hash code of object.**
- **Null insertion is possible only once.**
- **Heterogeneous objects are allowed.**
- **Implements Serializable and Cloneable interfaces, but not random access interface.**
- **Headset is the best choice if our frequent operation is search operation.**

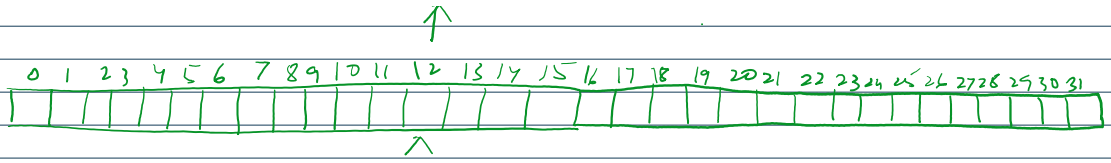
**Note: In HashSet duplicates are not allowed. If we trying to insert duplicate then we won't get any compile time or runtime error. And add() method simply returns false.**

**Constructors:-**

**1. HashSet h = new HashSet(); => Creates an empty hash set object with default initial capacity 16 and default fill ratio 0.75.**




8
16 * 0.75
= 12



2. HashSet h = new HashSet(int initialCapacity); => Creates an empty hash set object with specified initialCapacity and default fill ratio 0.75.

$0.0 < \text{fillRatio} < 1.0$

3. HashSet h = new HashSet(int initialCapacity, float fillRatio);

4. HashSet h = new HashSet(Collection c); => Creates an equivalent HashSet for the given collection. This constructor is meant for inter conversion between Collection object.

Fill Ratio or Load Factor: After filling how much ratio a new hash set object will be created? This ratio is called fill ratio or load factor. For example, fill ratio 0.75 means after filling 75% a new HashSet object will be created.

```java
package Chapter1;
import java.util.*;


/**
 * Write a description of class HashSetDemo here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class HashSetDemo
{
    public static void main(String args[]){
        HashSet h = new HashSet();
        System.out.print("\fHashSet capacity: " + h.size());
        h.add("H");
        h.add("A");
        h.add("S");
        h.add("H");
        h.add("null");
        System.out.print("\n"+h.add("S"));
        h.add("E");
        h.add("T");
        h.add(10);
        h.add(3.14);
        System.out.print("\n" + h);
    }
}
```

Class compiled - no syntax errors

BlueJ: Terminal Window - CollectionFramework

Options

```
HashSet capacity: 0
false
[A, S, null, T, E, 3.14, H, 10]
```

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

This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element.

This class offers constant time performance 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 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.

**Note that this implementation 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. 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:

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

The iterators returned by this class's iterator method are *fail-fast*: if the set is modified at any time after the iterator is created, in any way except through the iterator's own remove method, the Iterator throws a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

This class is a member of the Java Collections Framework.

**Since:**

1.2

All Methods | Instance Methods | Concrete Methods

| Modifier and Type | Method and Description |
|---|---|
| boolean | add(E e)<br>Adds the specified element to this set if it is not already present. |
| void | clear()<br>Removes all of the elements from this set. |
| Object | clone()<br>Returns a shallow copy of this HashSet instance: the elements themselves are not cloned. |
| boolean | contains(Object o)<br>Returns true if this set contains the specified element. |
| boolean | isEmpty()<br>Returns true if this set contains no elements. |
| Iterator<E> | iterator()<br>Returns an iterator over the elements in this set. |
| boolean | remove(Object o)<br>Removes the specified element from this set if it is present. |
| int | size()<br>Returns the number of elements in this set (its cardinality). |
| Spliterator<E> | spliterator()<br>Creates a *late-binding* and *fail-fast* Spliterator over the elements in this set. |

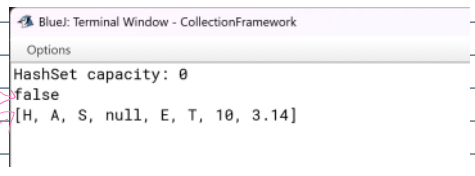
## LinkedHashSet

| HashSet | LinkedHashSet |
|---|---|
| Underlying data structure is Hash Table. | Underline data structure is linked list + hash table. |
| Insertion order is not preserved. | Insertion order is preserved. |
| Introduced in version 1.2. | Introduced in version 1.4. |

**LinkedHashSet** Is a child class of hash set? It is exactly same as hash said. (including. constructors and methods) except the following difference:

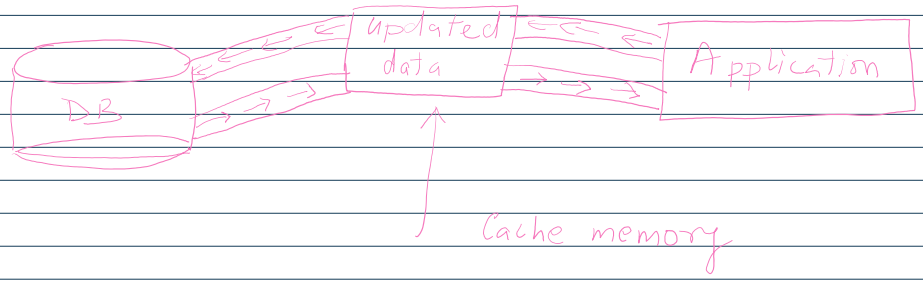
```
package Chapter1;
import java.util.*;


/**
 * Write a description of class HashSetDemo here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class HashSetDemo
{
    public static void main(String args[]){
        LinkedHashSet h = new LinkedHashSet();
        System.out.print("\fHashSet capacity: " + h.size());
        h.add("H");
        h.add("A");
        h.add("S");
        h.add("H");
        h.add("null");
        System.out.print("\n"+h.add("S"));
        h.add("E");
        h.add("T");
        h.add(10);
        h.add(3.14);
        System.out.print("\n" + h);
    }
}
```

Duplicates are not allowed




Insertion order is preserved

In general we can use LinkedHashSet to develop cache based application where duplicates are not allowed and insertion order is preserved




Official Documentation

public class LinkedHashSet<E>
extends HashSet<E>
implements Set<E>, Cloneable, Serializable
Hash table and linked list implementation of the Set interface, with predictable iteration order. This implementation differs from HashSet in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is the order in which elements were inserted into the set (*insertion-order*). Note that insertion order is *not* affected if an element is *re-inserted* into the set. (An element e is reinserted into a set s if s.add(e) is invoked when s.contains(e) would return true immediately prior to the invocation.)
This implementation spares its clients from the unspecified, generally chaotic ordering provided by HashSet, without incurring the increased cost associated with TreeSet. It can be used to produce a copy of a set that has the same order as the original, regardless of the original set's implementation:

```
void foo(Set s) {
    Set copy = new LinkedHashSet(s);
    ...
}
```

This technique is particularly useful if a module takes a set on input, copies it, and later returns results whose order is determined by that of the copy. (Clients generally appreciate having things returned in the same order they were presented.)
This class provides all of the optional Set operations, and permits null elements. Like HashSet, it provides constant-time performance for the basic operations (add, contains and remove), assuming the hash function disperses elements properly among the buckets. Performance is likely to be just slightly below that of HashSet, due to the added expense of maintaining the linked list, with one exception: Iteration over a LinkedHashSet requires time proportional to the *size* of the set, regardless of its capacity. Iteration over a HashSet is likely to be more expensive, requiring time proportional to its *capacity*.
A linked hash set has two parameters that affect its performance: *initial capacity* and *load factor*. They are defined precisely as for HashSet. Note, however, that the penalty for choosing an excessively high value for initial capacity is less severe for this class than for HashSet, as iteration times for this class are unaffected by capacity.
**Note that this implementation is not synchronized.** If multiple threads access a linked 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:

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

The iterators returned by this class's iterator method are *fail-fast*: if the set is modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-

deterministic behavior at an undetermined time in the future.
Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*
This class is a member of the Java Collections Framework.
Since:
1.4

From <https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html>

***Constructor Summary***

**Constructors**

| Constructor and Description |
| --- |
| `LinkedHashSet()`<br>Constructs a new, empty linked hash set with the default initial capacity (16) and load factor (0.75). |
| `LinkedHashSet(Collection<? extends E> c)`<br>Constructs a new linked hash set with the same elements as the specified collection. |
| `LinkedHashSet(int initialCapacity)`<br>Constructs a new, empty linked hash set with the specified initial capacity and the default load factor (0.75). |
| `LinkedHashSet(int initialCapacity, float loadFactor)`<br>Constructs a new, empty linked hash set with the specified initial capacity and load factor. |

Methods are same as HashSet