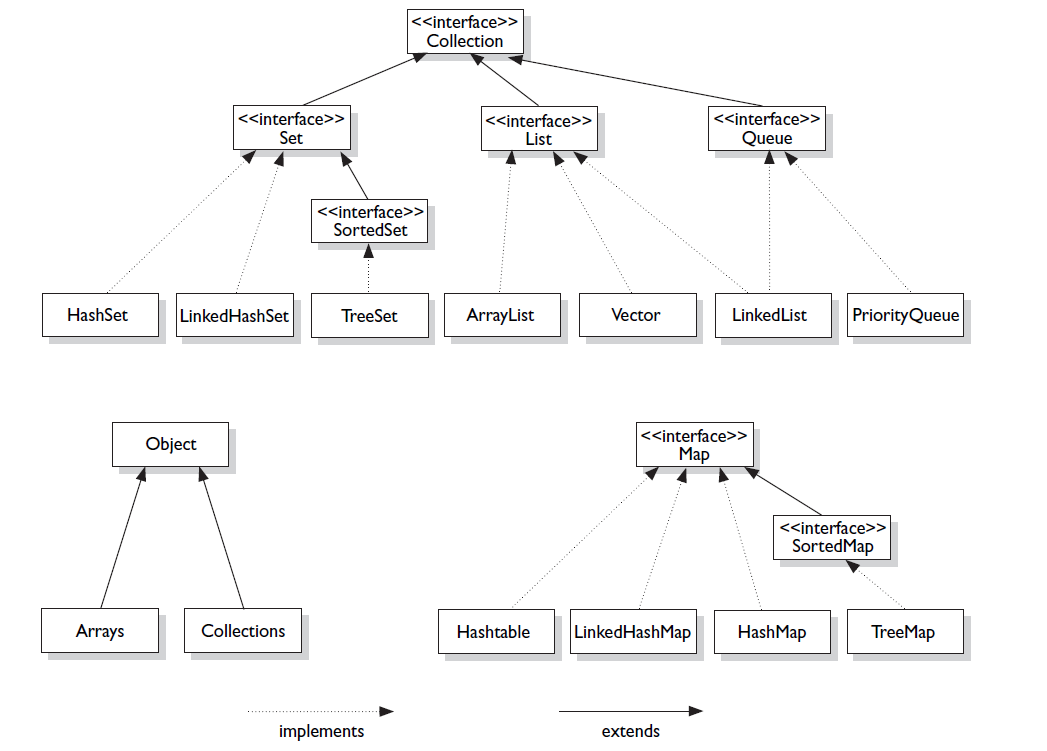
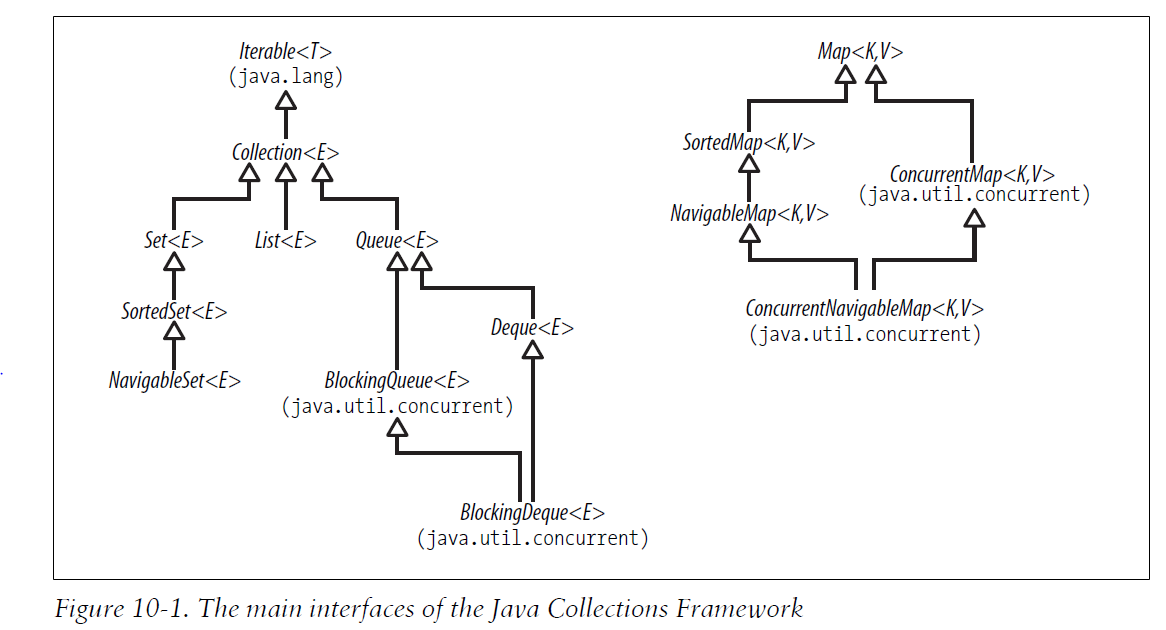
**Collections**

# Collection

Collection class hierarchy





* The Java Collections Framework is a set of interfaces and classes in the packages **java.util** and **java.util.concurrent**.

# Iterable and Iterators

The purpose of iterators is to provide a uniform way of accessing collection elements sequentially, so whatever kind of collection you are dealing with, and however it is implemented, you always know how to process its elements in turn.

Implementing this interface allows an object to be the target of the "foreach" statement.

**public** **interface** Iterable<T> {

/\*\*

\* Returns an iterator over a set of elements of type T.

\*

\* **@return** an Iterator.

\*/

Iterator<T> iterator();

}

**public** **interface** Iterator<E> {

**boolean hasNext(); //**Returns true if the iteration has more elements.

**E next(); //**Returns the next element in the iteration.

/\*Removes from the underlying collection the last element returned by this iterator (optional operation). This method can be called only once per call to [next](eclipse-javadoc:%E2%98%82=cacheInvalidator/C:%5C/Program%20Files%5C/Java%5C/jdk1.7.0_51%5C/jre%5C/lib%5C/rt.jar%3Cjava.util(Iterator.class%E2%98%83Iterator~remove%E2%98%82%E2%98%82next). The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

**Throws:**

[UnsupportedOperationException](eclipse-javadoc:%E2%98%82=cacheInvalidator/C:%5C/Program%20Files%5C/Java%5C/jdk1.7.0_51%5C/jre%5C/lib%5C/rt.jar%3Cjava.util(Iterator.class%E2%98%83Iterator~remove%E2%98%82UnsupportedOperationException) - if the remove operation is not supported by this iterator

[IllegalStateException](eclipse-javadoc:%E2%98%82=cacheInvalidator/C:%5C/Program%20Files%5C/Java%5C/jdk1.7.0_51%5C/jre%5C/lib%5C/rt.jar%3Cjava.util(Iterator.class%E2%98%83Iterator~remove%E2%98%82IllegalStateException) - if the next method has not yet been called, or the remove method has already been called after the last call to the next method

\*/

**void** remove();

}

The iterators throw the ConcurrentModificationException whenever they detect that the collection from which they were derived has been structurally changed. The general-purpose Collections Framework iterators are fail-fast. The methods of a fail-fast iterator check that the underlying collection has not been structurally changed (by another iterator, or by the methods of the collection itself) since the last iterator method call. If they detect a change, they throw ConcurrentModificationException.

The 4 main structures, described below:

***Arrays***

These are the structures familiar from the Java language—and just about every other programming language since Fortran. Because arrays are implemented directly in hardware, they have the properties of random-access memory: **very fast for accessing elements by position** and **for iterating over them**, but **slower for inserting and removing elements at arbitrary positions** (because that may require adjusting the position of other elements). Arrays are used in the Collections Framework as the backing structure for ArrayList, CopyOnWriteArrayList, EnumSet and EnumMap, and for many of the Queue and Deque implementations. They also form an important part of the mechanism for implementing hash tables.

***Linked lists***

As the name implies, these consist of chains of linked cells. Each cell contains a reference to data and a reference to the next cell in the list (and, in some implementations, the previous cell). Linked lists perform quite differently from arrays: **accessing elements by position is slow, because you have to follow the reference chain from the start of the list**, but **insertion and removal operations can be performed in constant time by rearranging the cell references**. Linked lists are the primary backing structure used for the classes ConcurrentLinkedQueue, LinkedBlockingQueue, and LinkedList, and the new skip list collections ConcurrentSkipListSet and ConcurrentSkipListMap. They are also used in implementing HashSet and LinkedHashSet.

***Hash tables***

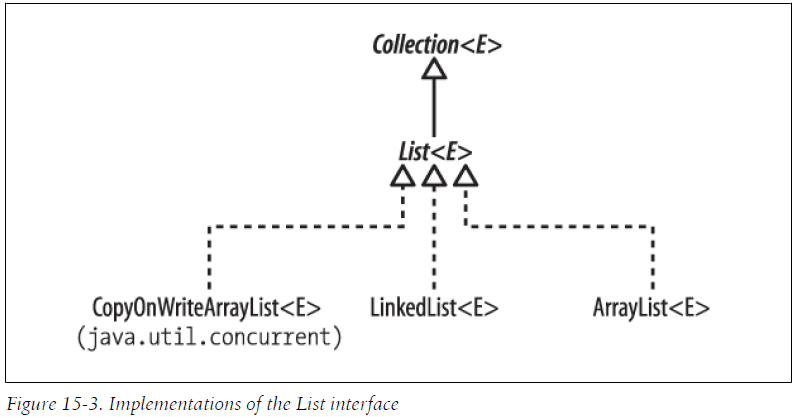
These provide a way of **storing elements indexed on their content rather than on an integer-valued index**, as with lists. In contrast to arrays and linked lists, hashtables **provide no support for accessing elements by position**, but **access by content is usally very fast, as are insertion and removal**. Hash tables are the backing structure for many Set and Map implementations, including HashSet and LinkedHashSet together with the corresponding maps HashMap and LinkedHashMap, as well as WeakHashMap, IdentityHashMap and ConcurrentHashMap.

***Trees***

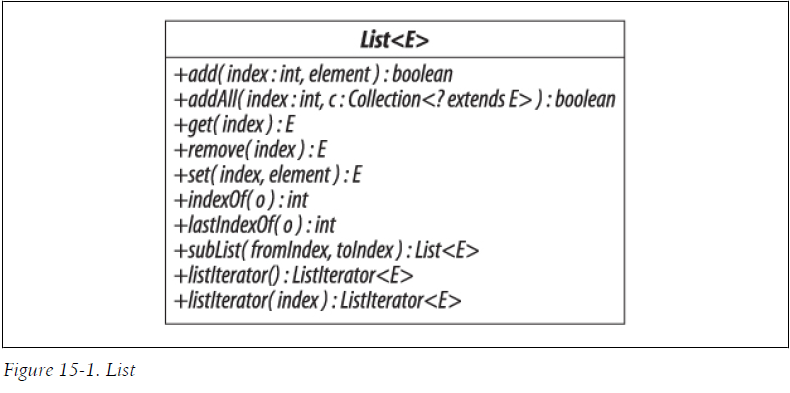
These also organize their elements by content, but with the important difference that **they can store and retrieve them in sorted order**. They are relatively **fast for the operations of inserting and removing elements**, **accessing them by content and iterating over them**. Trees are the backing structures for TreeSet and TreeMap. Priorityheaps, used in the implementation of PriorityQueue and PriorityBlockingQueue, are tree related structures.

# LIST

Ordered, duplicates allowed



Methods in List interface:



## ARRAYLIST

-ArrayList extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, java.io.Serializable

- AbstractList<E> extends AbstractCollection<E> implements List<E>

- AbstractCollection<E> implements Collection<E>

### Algorithm

when an arraylist is made then a Transient array of Object class is made, named elementData, with the initial capacity that is 10.

When an element is added to the arraylist: first ensureCapacity is done, that is if the size has increased 10, then a new array of object is made with increased size and the old one is copied into it. If the size is less than 10, then the element is put into the elemenetData array.

The increasing of size is done by calculation : int newCapacity = (oldCapacity \* 3)/2 + 1; Eg: newCapacity= (10\*3)/2 + 1 = 16

The iterators of ArrayList are fail-fast.

### Advantage and disadvantages

-Since Array is an index based data-structure searching or getting element from Array with index is pretty fast

-but remove is costly in ArrayList as you need to rearrange all elements

-to synchronize a list : Collection.synchronizedList(List list)

-It is a resizable array implementation. The size of the ArrayList can be increased dynamically also operations like add,remove and get can be formed once the object is created. It also ensures that the data is retrieved in the manner it was stored. The ArrayList is not thread-safe.

-resizable array implementation, ordered, search and getting objects using index is easy, insertion and deletion is costly, unsynchronized

-The performance of ArrayList reflects array performance for "random-access” operations: set and get take constant time. The downside of an array implementation is in inserting or removing elements at arbitrary positions, because that may require adjusting the position of other elements.

### Q) How is searching performed in arrayList:

The whole list is traversed, using a for loop and the search object is compared with every object one by one

### Q) Why is search faster in arrayList

while searching in arrayList, the list is traversed using the index number,

but while searching in linkedList the list is traversed using the element.next reference.

### Q) How to convert Array to ArrayList:

ArrayList<Element> arrayList = new ArrayList<Element>(Arrays.asList(array));

or

List<Element> list = Arrays.asList(array);

-BUG in ArrayList : It implements same interface as its superClass

### Q) Can null be added into arrayList ?

yes

### Q) if we add an element into the index where element exists, will it be overridden ?

no, the old element will be moved to the next index.

## LINKEDLIST

### Algorithm

When a linked list is made nothing is initialized.

When an element is added into linkedList, an object is made of Node class. Node/Entry class has 3 instance variables item, next, previous.

The element to be added is put in the item, the last element is put in previous. After doing all this the existing element is saved in the "last" instance variable (transient variable of Node type)

Where LinkedList does have a performance advantage over ArrayList is in adding and removing elements anywhere other than at the end of the list; for LinkedList this takes constant time, against the linear time required for noncircular array implementations.

-doubly linkedList

-LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable, java.io.Serializable

-AbstractSequentialList<E> extends AbstractList<E>

-LinkedList doesn't provide Random or index based access and you need to iterate over linked list to retrieve any element which is of order O(n).

-Insertions are easy and fast in LinkedList as compared to ArrayList because there is no risk of resizing array and copying content to new array if array gets full

-removal is also easy in linkedlist

-LinkedList has more memory overhead than ArrayList because in ArrayList each index only holds actual object (data) but in case of LinkedList each node holds both data and address of next and previous node.

-null can be added into LinkedList

## VECTOR

Synchronized arrayList

## CopyOnWriteArrayList

CopyOnWriteArrayList vs Array List in Java

CopyOnWriteArrayList is a concurrent Collection class introduced in Java 5 Concurrency API along with its popular cousin ConcurrentHashMap in Java.

CopyOnWriteArrayList implements List interface like ArrayList, Vector and LinkedList but its a thread-safe collection and it achieves its thread-safety in a slightly different way than Vector or other thread-safe collection class. As name suggest CopyOnWriteArrayList creates copy of underlying ArrayList with every mutation operation e.g. add or set. Normally CopyOnWriteArrayList is very expensive because it involves costly Array copy with every write operation but its very efficient if you have a List where Iteration outnumber mutation e.g. you mostly need to iterate the ArrayList and don't modify it too often. Iterator of CopyOnWriteArrayList is fail-safe and doesn't throw ConcurrentModificationException even if underlying CopyOnWriteArrayList is modified once Iteration begins because Iterator is operating on separate copy of ArrayList. Consequently all the updates made on CopyOnWriteArrayList is not available to Iterator. In this Java Collection tutorial we will see What is CopyOnWriteArrayList in Java, Difference between ArrayList and CopyOnWriteArrayList in Java and One simple Java program example on How to use CopyOnWriteArrayList in Java.

Difference between CopyOnWriteArrayList and ArrayList in Java.

In last section we have seen What is CopyOnWriteArrayList in Java and How it achieves thread-safety by creating a separate copy of List for each write operation. Now let's see Some difference between ArrayList and CopyOnWriteArrayList in Java , which is another implementation of List interface :

1) First and foremost difference between CopyOnWriteArrayList and ArrayList in Java is that CopyOnWriteArrayList is a thread-safe collection while ArrayList is not thread-safe and can not be used in multi-threaded environment.

2) Second difference between ArrayList and CopyOnWriteArrayList is that Iterator of ArrayList is fail-fast and throw ConcurrentModificationException once detect any modification in List once iteration begins but Iterator of CopyOnWriteArrayList is fail-safe and doesn't throw ConcurrentModificationException.

3) Third difference between CopyOnWriteArrayList vs ArrayList is that Iterator of former doesn't support remove operation while Iterator of later supports remove() operation.

--

Along with ConcurrentHashMap, Java 5 introduces a copy-on-write implementation of a list: CopyOnWriteArrayList. It also introduces CopyOnWriteArraySet, essentially a convenience wrapper around the list.

CopyOnWriteArrayList

An instance of CopyOnWriteArrayList behaves as a List implementation that allows multiple concurrent reads, and for reads to occur concurrently with a write. The way it does this is to make a brand new copy of the list every time it is altered.

Reads do not block, and effectively pay only the cost of a volatile read;

Writes do not block reads (or vice versa), but only one write can occur at once;

Unlike ConcurrentHashMap, write operations that write or access multiple elements in the list (such as addAll(), retainAll()) will be atomic.

During a write operation, the array must be locked completely against other writes. (The standard implementation uses a ReentrantLock.) However, this does mean that as mentioned, operations affecting multiple locations can be atomic. That is, if one thread adds several items to the list with addAll() while another thread calls size(), the thread reading the size will get a value that either reflects or not the number of elements added in addAll(): there'll be no possibility of an intermediate value returned (provided of course that these are the only two threads accessing the list!).

CopyOnWriteArrayList is designed for cases where:

reads hugely outnumber writes;

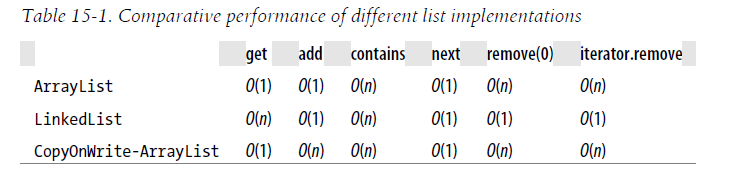
the array is small (or writes are very infrequent);

the caller genuinely needs the functionality of a list rather than an array.

The first two are fairly obvious: if too great, then either the size of the array or the frequency of writes can outweigh the benefit of not synchrozing.

The last point is possibly more subtle, but what it boils down to is this: if you only need the functionality of an array (that is, you don't need the list to grow arbitrarily, you don't need inserts etc), then the atomic array classes generally give better throughput.

## Comparison



# MAP

-key value pair

-duplicate keys not allowed

## HashMap

### Algorithm:

**When a hashmap is made:** A transient array of Entry class is made, named table, with initial size of 16.

**Entry class is inner class with fields:** final K key, V value, Entry<K,V> next, final int hash.

When elements are put into the hashmap : first the hashcode of the key is calculated. Using the hashcode and the table length the bucketIndex is calculated.

**How bucketIndex is calculated:** the key.hashCode is sent to hash function and the result returned as i. [i & (table.length-1)] operation is performed and this operation reduces the big hashcode to an index that is less than or equals to the table array size. After getting the location, checks if that table[index] already contains some element at that location. An object of Entry class is made using the hashCode, key, value, old element at that bucketIndex. This element is saved in the table[bucketIndex]. If bucketIndex is calculated same as that of some old element already present in the hashMap, the old element is saved in the "old element at the bucketIndex" and the new element has refernce to the old element. The new element is saved in that bucketIndex. Whenever the table array size reaches the threshold (table\_size \* load\_factory). Null key is saved at indexLocation 0.

-only one null key allowed, multiple null values allowed

-unsorted,unordered: because the elements are stored in the array index and the index doesnt depends on the order in which the key/value is added

-Fast insertion and deletion:

- Note that if a key is inserted again the old one is removed and the new one is added

-This implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

what happens if hashmap is shared between multiple threads

How is collision of bucket handled in hashMap

How to sort hashMap

-Make a treeMap from hashMap, if the key object overrides comparable interface it will be sorted. Or pass comparator object while making a TreeMap from hashMap

## HashTable

-synchronized form of hashmap

-doesnt adds anything that is null (Why so)

Q)Why Hashtable doesnot accepts null as key

-HashTable implementation takes out hashCode of the key and then adds the key,value pair accordingly.

If null is provided as key for hashtable, it tries to take out hashCode of null and null pointer exception is thrown.

-while in case of HashMap, if null is provided as key, hashCode is not taken out and that key,value pair will always be added at index[0]

## LinkedHashMap

-ALGO: on initialization linkedHashMap makes a hashMap, that is A transient array of entry class is made,named table, with initial size of 16.

- It maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion-order)

- Note that if a key is inserted again the old one is removed and the new one is added at the end

-class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V>

-ordered,faster iteration

## TreeMap

-class TreeMap<K,V> extends AbstractMap<K,V> implements NavigableMap<K,V>, Cloneable, java.io.Serializable

-A Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map

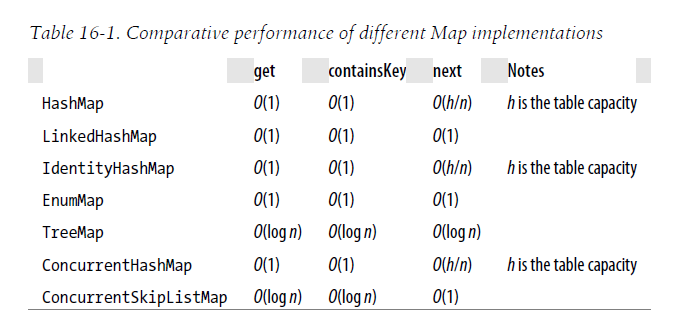
creation time, depending on which constructor is used.

-This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations.

## ConcurrentHashMap

The ConcurrentHashMap is very similar to the java.util.HashTable class, except that ConcurrentHashMap offers better concurrency than HashTable does. ConcurrentHashMap does not lock the Map while you are reading from it. Additionally, ConcurrentHashMap does not lock the entire Map when writing to it. It only locks the part of the Map that is being written to, internally.

Another difference is that ConcurrentHashMap does not throw ConcurrentModificationException if the ConcurrentHashMap is changed while being iterated. The Iterator is not designed to be used by more than one thread though.

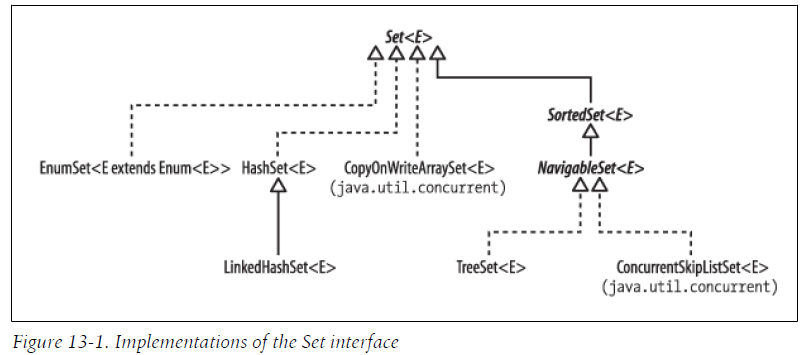


# QUEUE

\*\*\*\*\*\*

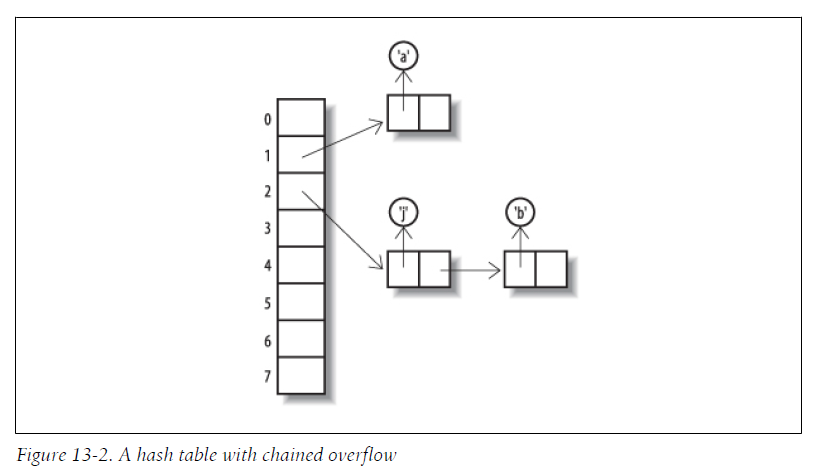
# SET

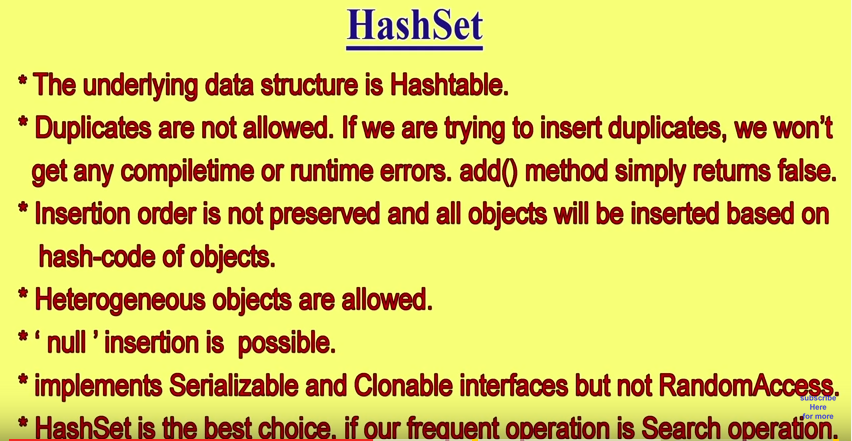
- cannot contain duplicates; adding an item if it is already present in the set has no effect.

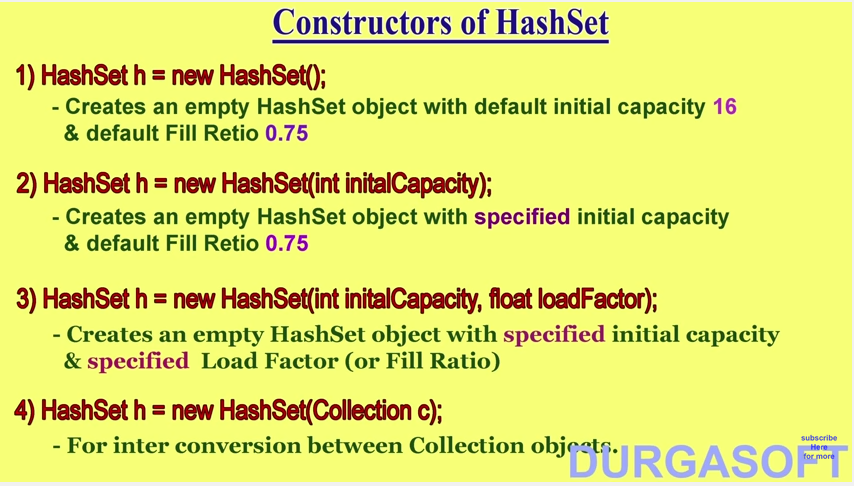


## HashSet

As the name implies, it is implemented by a *hash table*, an array in which elements are stored at a position derived from their contents. Since hash tables store and retrieve elements by their content, they are well suited to implementing the operations of Set







We can minimize the problem with a good hash function—one which spreads the elements out equally in the table—but, when collisions do occur, we have to have a way of keeping the colliding elements at the same table location or *bucket*. This is often done by storing them in a linked list.

If the size of the hash table is fixed, performance will worsen as more elements are added and the load increases. To prevent this from happening, the table size is increased by *rehashing*—copying to a new and larger table—when the load reaches a specified threshold (its *load factor*).

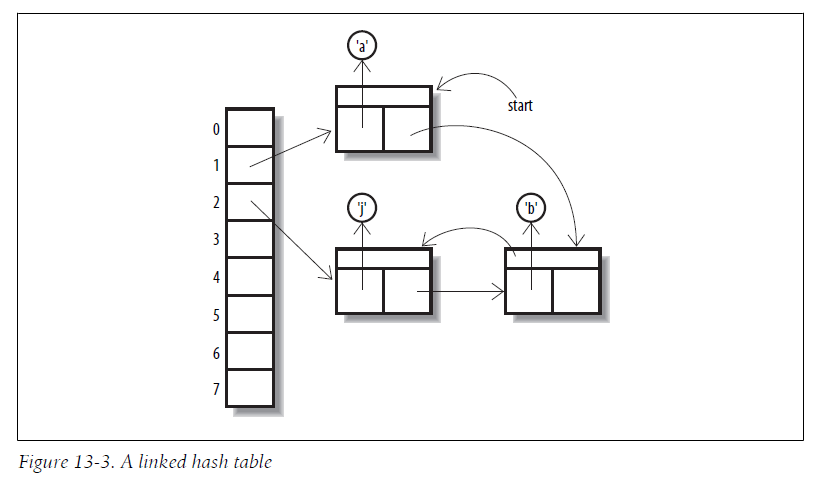
Collections.sort :

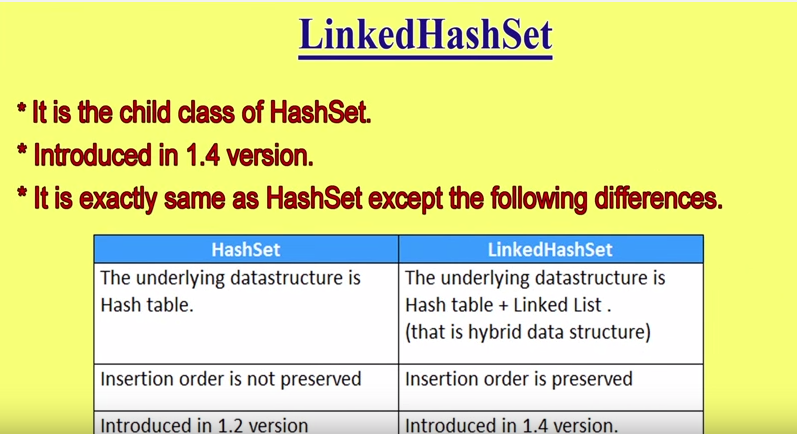
-uses mergeSort :

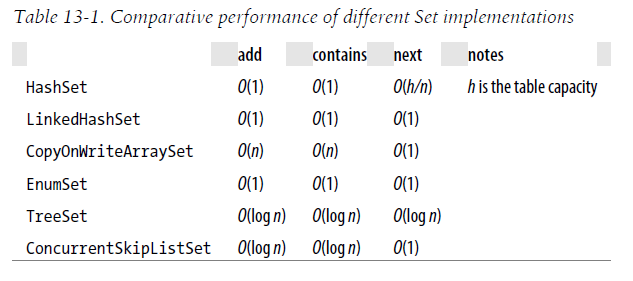
## LinkedHashSet

This class inherits from HashSet, still implementing Set.

It guarantees that its iterators will return their elements in the order in which they were first added. It does this by maintaining a linked list of the set elements.







# Fail-safe vs Fail-fast iterator ??

# Comparator and comparable ??

# Sorting ??

# Searching ??