public class **Collections** extends [Object](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html)

This class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends.

The methods of this class all throw a NullPointerException if the collections or class objects provided to them are null.

The documentation for the polymorphic algorithms contained in this class generally includes a brief description of the *implementation*. Such descriptions should be regarded as *implementation notes*, rather than parts of the *specification*. Implementors should feel free to substitute other algorithms, so long as the specification itself is adhered to. (For example, the algorithm used by sort does not have to be a mergesort, but it does have to be *stable*.)

The "destructive" algorithms contained in this class, that is, the algorithms that modify the collection on which they operate, are specified to throw UnsupportedOperationException if the collection does not support the appropriate mutation primitive(s), such as the set method. These algorithms may, but are not required to, throw this exception if an invocation would have no effect on the collection. For example, invoking the sort method on an unmodifiable list that is already sorted may or may not throw UnsupportedOperationException.

This class is a member of the [Java Collections Framework](http://docs.oracle.com/javase/7/docs/technotes/guides/collections/index.html).

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

# Collections Framework Overview

## Introduction

The Java platform includes a *collections framework*. A *collection* is an object that represents a group of objects (such as the classic [Vector](http://docs.oracle.com/javase/7/docs/api/java/util/Vector.html) class). A collections framework is a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details.

The primary advantages of a collections framework are that it:

* **Reduces programming effort** by providing data structures and algorithms so you don't have to write them yourself.
* **Increases performance** by providing high-performance implementations of data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be tuned by switching implementations.
* **Provides interoperability between unrelated APIs** by establishing a common language to pass collections back and forth.
* **Reduces the effort required to learn APIs** by requiring you to learn multiple ad hoc collection APIs.
* **Reduces the effort required to design and implement APIs** by not requiring you to produce ad hoc collections APIs.
* **Fosters software reuse** by providing a standard interface for collections and algorithms with which to manipulate them.

The collections framework consists of:

* **Collection interfaces**. Represent different types of collections, such as sets, lists, and maps. These interfaces form the basis of the framework.
* **General-purpose implementations**. Primary implementations of the collection interfaces.
* **Legacy implementations**. The collection classes from earlier releases, Vector and Hashtable, were retrofitted to implement the collection interfaces.
* **Special-purpose implementations**. Implementations designed for use in special situations. These implementations display nonstandard performance characteristics, usage restrictions, or behavior.
* **Concurrent implementations**. Implementations designed for highly concurrent use.
* **Wrapper implementations**. Add functionality, such as synchronization, to other implementations.
* **Convenience implementations**. High-performance "mini-implementations" of the collection interfaces.
* **Abstract implementations**. Partial implementations of the collection interfaces to facilitate custom implementations.
* **Algorithms**. Static methods that perform useful functions on collections, such as sorting a list.
* **Infrastructure**. Interfaces that provide essential support for the collection interfaces.
* **Array Utilities**. Utility functions for arrays of primitive types and reference objects. Not, strictly speaking, a part of the collections framework, this feature was added to the Java platform at the same time as the collections framework and relies on some of the same infrastructure.

## Collection Interfaces

The *collection interfaces* are divided into two groups. The most basic interface, [java.util.Collection](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html), has the following descendants:

[java.util.Set](http://docs.oracle.com/javase/7/docs/api/java/util/Set.html) [java.util.SortedSet](http://docs.oracle.com/javase/7/docs/api/java/util/SortedSet.html) [java.util.NavigableSet](http://docs.oracle.com/javase/7/docs/api/java/util/NavigableSet.html) [java.util.Queue](http://docs.oracle.com/javase/7/docs/api/java/util/Queue.html)

[java.util.concurrent.BlockingQueue](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html) [java.util.concurrent.TransferQueue](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/TransferQueue.html)

[java.util.Deque](http://docs.oracle.com/javase/7/docs/api/java/util/Deque.html) [java.util.concurrent.BlockingDeque](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingDeque.html)

The other collection interfaces are based on [java.util.Map](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html) and are not true collections. However, these interfaces contain *collection-view* operations, which enable them to be manipulated as collections. Map has the following offspring:

[java.util.SortedMap](http://docs.oracle.com/javase/7/docs/api/java/util/SortedMap.html) [java.util.NavigableMap](http://docs.oracle.com/javase/7/docs/api/java/util/NavigableMap.html)

[java.util.concurrent.ConcurrentMap](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentMap.html) [java.util.concurrent.ConcurrentNavigableMap](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentNavigableMap.html)

Many of the modification methods in the collection interfaces are labeled *optional*. Implementations are permitted to not perform one or more of these operations, throwing a runtime exception (UnsupportedOperationException) if they are attempted. The documentation for each implementation must specify which optional operations are supported. Several terms are introduced to aid in this specification:

* Collections that do not support modification operations (such as add, remove and clear) are referred to as *unmodifiable*. Collections that are not unmodifiable are *modifiable.*
* Collections that additionally guarantee that no change in the Collection object will be visible are referred to as *immutable*. Collections that are not immutable are *mutable*.
* Lists that guarantee that their size remains constant even though the elements can change are referred to as *fixed-size*. Lists that are not fixed-size are referred to as *variable-size*.
* Lists that support fast (generally constant time) indexed element access are known as *random access* lists. Lists that do not support fast indexed element access are known as *sequential access* lists. The [RandomAccess](http://docs.oracle.com/javase/7/docs/api/java/util/RandomAccess.html) marker interface enables lists to advertise the fact that they support random access. This enables generic algorithms to change their behavior to provide good performance when applied to either random or sequential access lists.

Some implementations restrict what elements (or in the case of Maps, keys and values) can be stored. Possible restrictions include requiring elements to:

* Be of a particular type.
* Be not null.
* Obey some arbitrary predicate.

Attempting to add an element that violates an implementation's restrictions results in a runtime exception, typically a ClassCastException, an IllegalArgumentException, or a NullPointerException. Attempting to remove or test for the presence of an element that violates an implementation's restrictions can result in an exception. Some restricted collections permit this usage.

## Collection Implementations

Classes that implement the collection interfaces typically have names in the form of <Implementation-style><Interface>. The general purpose implementations are summarized in the following table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Interface** | **Hash Table** | **Resizable Array** | **Balanced Tree** | **Linked List** | **Hash Table + Linked List** |
| Set | [HashSet](http://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html) |  | [TreeSet](http://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html) |  | [LinkedHashSet](http://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashSet.html) |
| List |  | [ArrayList](http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html) |  | [LinkedList](http://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html) |  |
| Deque |  | [ArrayDeque](http://docs.oracle.com/javase/7/docs/api/java/util/ArrayDeque.html) |  | [LinkedList](http://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html) |  |
| Map | [HashMap](http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html) |  | [TreeMap](http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html) |  | [LinkedHashMap](http://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashMap.html) |

The general-purpose implementations support all of the *optional operations* in the collection interfaces and have no restrictions on the elements they may contain. They are unsynchronized, but the Collections class contains static factories called [synchronization wrappers](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#synchronizedCollection%28java.util.Collection%29) that can be used to add synchronization to many unsynchronized collections. All of the new implementations have *fail-fast iterators*, which detect invalid concurrent modification, and fail quickly and cleanly (rather than behaving erratically).

The AbstractCollection, AbstractSet, AbstractList, AbstractSequentialList and AbstractMap classes provide basic implementations of the core collection interfaces, to minimize the effort required to implement them. The API documentation for these classes describes precisely how each method is implemented so the implementer knows which methods must be overridden, given the performance of the basic operations of a specific implementation.

## Concurrent Collections

Applications that use collections from more than one thread must be carefully programmed. In general, this is known as *concurrent programming*. The Java platform includes extensive support for concurrent programming. See [Java Concurrency Utilities](http://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/index.html) for details.

Collections are so frequently used that various concurrent friendly interfaces and implementations of collections are included in the APIs. These types go beyond the synchronization wrappers discussed previously to provide features that are frequently needed in concurrent programming.

These concurrent-aware interfaces are available:

[BlockingQueue](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html) [TransferQueue](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/TransferQueue.html) [BlockingDeque](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingDeque.html) [ConcurrentMap](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentMap.html) [ConcurrentNavigableMap](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentNavigableMap.html)

The following concurrent-aware implementation classes are available. See the API documentation for the correct usage of these implementations.

[LinkedBlockingQueue](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingQueue.html) [ArrayBlockingQueue](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ArrayBlockingQueue.html) [PriorityBlockingQueue](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/PriorityBlockingQueue.html) [DelayQueue](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/DelayQueue.html) [SynchronousQueue](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/SynchronousQueue.html)

[LinkedBlockingDeque](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingDeque.html) [LinkedTransferQueue](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedTransferQueue.html) [CopyOnWriteArrayList](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArrayList.html) [CopyOnWriteArraySet](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArraySet.html)

[ConcurrentSkipListSet](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListSet.html) [ConcurrentHashMap](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html) [ConcurrentSkipListMap](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListMap.html)

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

**public interface **Set<E>** extends** [**Collection**](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)**<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.

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

**public interface **List<E>** extends** [**Collection**](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)**<E>**

An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

Unlike sets, lists typically allow duplicate elements. More formally, lists typically allow pairs of elements e1 and e2 such that e1.equals(e2), and they typically allow multiple null elements if they allow null elements at all. It is not inconceivable that someone might wish to implement a list that prohibits duplicates, by throwing runtime exceptions when the user attempts to insert them, but we expect this usage to be rare.

The List interface places additional stipulations, beyond those specified in the Collection interface, on the contracts of the iterator, add, remove, equals, and hashCode methods. Declarations for other inherited methods are also included here for convenience.

The List interface provides four methods for positional (indexed) access to list elements. Lists (like Java arrays) are zero based. Note that these operations may execute in time proportional to the index value for some implementations (the LinkedList class, for example). Thus, iterating over the elements in a list is typically preferable to indexing through it if the caller does not know the implementation.

The List interface provides a special iterator, called a ListIterator, that allows element insertion and replacement, and bidirectional access in addition to the normal operations that the Iterator interface provides. A method is provided to obtain a list iterator that starts at a specified position in the list.

The List interface provides two methods to search for a specified object. From a performance standpoint, these methods should be used with caution. In many implementations they will perform costly linear searches.

The List interface provides two methods to efficiently insert and remove multiple elements at an arbitrary point in the list.

Note: While it is permissible for lists to contain themselves as elements, extreme caution is advised: the equals and hashCode methods are no longer well defined on such a list.

Some list 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 list 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.

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

**public interface **Map<K,V>****

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

This interface takes the place of the Dictionary class, which was a totally abstract class rather than an interface.

The Map interface provides three *collection views*, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The *order* of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the TreeMap class, make specific guarantees as to their order; others, like the HashMap class, do not.

Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is a key in the map. A special case of this prohibition is that it is not permissible for a map to contain itself as a key. While it is permissible for a map to contain itself as a value, extreme caution is advised: the equals and hashCode methods are no longer well defined on such a map.

All general-purpose map implementation classes should provide two "standard" constructors: a void (no arguments) constructor which creates an empty map, and a constructor with a single argument of type Map, which creates a new map with the same key-value mappings as its argument. In effect, the latter constructor allows the user to copy any map, producing an equivalent map of the desired class. There is no way to enforce this recommendation (as interfaces cannot contain constructors) but all of the general-purpose map implementations in the JDK comply.

The "destructive" methods contained in this interface, that is, the methods that modify the map on which they operate, are specified to throw UnsupportedOperationException if this map does not support the operation. If this is the case, these methods may, but are not required to, throw an UnsupportedOperationException if the invocation would have no effect on the map. For example, invoking the [putAll(Map)](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html#putAll%28java.util.Map%29) method on an unmodifiable map may, but is not required to, throw the exception if the map whose mappings are to be "superimposed" is empty.

Some map implementations have restrictions on the keys and values they may contain. For example, some implementations prohibit null keys and values, and some have restrictions on the types of their keys. Attempting to insert an ineligible key or value throws an unchecked exception, typically NullPointerException or ClassCastException. Attempting to query the presence of an ineligible key or value 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 key or value whose completion would not result in the insertion of an ineligible element into the map 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](http://docs.oracle.com/javase/7/docs/technotes/guides/collections/index.html).

Many methods in Collections Framework interfaces are defined in terms of the [equals](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals%28java.lang.Object%29) method. For example, the specification for the [containsKey(Object key)](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html#containsKey%28java.lang.Object%29) method says: "returns true if and only if this map contains a mapping for a key k such that (key==null ? k==null : key.equals(k))." This specification should *not* be construed to imply that invoking Map.containsKey with a non-null argument key will cause key.equals(k) to be invoked for any key k. Implementations are free to implement optimizations whereby the equals invocation is avoided, for example, by first comparing the hash codes of the two keys. (The [Object.hashCode()](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode%28%29) specification guarantees that two objects with unequal hash codes cannot be equal.) More generally, implementations of the various Collections Framework interfaces are free to take advantage of the specified behavior of underlying [Object](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html) methods wherever the implementor deems it appropriate.

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

public class **HashSet<E>** extends [AbstractSet](http://docs.oracle.com/javase/7/docs/api/java/util/AbstractSet.html)<E> implements [Set](http://docs.oracle.com/javase/7/docs/api/java/util/Set.html)<E>, [Cloneable](http://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html), [Serializable](http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html)

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](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#synchronizedSet%28java.util.Set%29) 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](http://docs.oracle.com/javase/7/docs/api/java/util/ConcurrentModificationException.html). 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.*

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

**public class **TreeSet<E>** extends** [**AbstractSet**](http://docs.oracle.com/javase/7/docs/api/java/util/AbstractSet.html)**<E> implements** [**NavigableSet**](http://docs.oracle.com/javase/7/docs/api/java/util/NavigableSet.html)**<E>,** [**Cloneable**](http://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html)**,** [**Serializable**](http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html)

A [NavigableSet](http://docs.oracle.com/javase/7/docs/api/java/util/NavigableSet.html) implementation based on a [TreeMap](http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html). The elements are ordered using their [natural ordering](http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html), or by a [Comparator](http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html) provided at set creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the basic operations (add, remove and contains).

Note that the ordering maintained by a set (whether or not an explicit comparator is provided) must be *consistent with equals* if it is to correctly implement the Set interface. (See Comparable or Comparator for a precise definition of *consistent with equals*.) This is so because the Set interface is defined in terms of the equals operation, but a TreeSet instance performs all element comparisons using its compareTo (or compare) method, so two elements that are deemed equal by this method are, from the standpoint of the set, equal. The behavior of a set *is* well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the Set interface.

**Note that this implementation is not synchronized.** If multiple threads access a tree 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.synchronizedSortedSet](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#synchronizedSortedSet%28java.util.SortedSet%29) method. This is best done at creation time, to prevent accidental unsynchronized access to the set:

SortedSet s = Collections.synchronizedSortedSet(new TreeSet(...));

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](http://docs.oracle.com/javase/7/docs/api/java/util/ConcurrentModificationException.html). 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.*

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

**public class **LinkedHashSet<E>** extends** [**HashSet**](http://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html)**<E> implements** [**Set**](http://docs.oracle.com/javase/7/docs/api/java/util/Set.html)**<E>,** [**Cloneable**](http://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html)**,** [**Serializable**](http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html)

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](http://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html), without incurring the increased cost associated with [TreeSet](http://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html). 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](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#synchronizedSet%28java.util.Set%29) 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](http://docs.oracle.com/javase/7/docs/api/java/util/ConcurrentModificationException.html). 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.*

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

**public class **HashMap<K,V>** extends** [**AbstractMap**](http://docs.oracle.com/javase/7/docs/api/java/util/AbstractMap.html)**<K,V> implements** [**Map**](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html)**<K,V>,** [**Cloneable**](http://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html)**,** [**Serializable**](http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html)

Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

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.

An instance of HashMap has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The *load factor* is a measure of how full the hash table 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.

As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the HashMap class, including get and put). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.

If many mappings are to be stored in a HashMap instance, creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table.

**Note that this implementation is not synchronized.** If multiple threads access a hash map concurrently, and at least one of the threads modifies the map structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the [Collections.synchronizedMap](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#synchronizedMap%28java.util.Map%29) method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

Map m = Collections.synchronizedMap(new HashMap(...));

The iterators returned by all of this class's "collection view methods" are *fail-fast*: if the map is structurally 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](http://docs.oracle.com/javase/7/docs/api/java/util/ConcurrentModificationException.html). 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.*

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

**public class **TreeMap<K,V>** extends** [**AbstractMap**](http://docs.oracle.com/javase/7/docs/api/java/util/AbstractMap.html)**<K,V> implements** [**NavigableMap**](http://docs.oracle.com/javase/7/docs/api/java/util/NavigableMap.html)**<K,V>,** [**Cloneable**](http://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html)**,** [**Serializable**](http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html)

A Red-Black tree based [NavigableMap](http://docs.oracle.com/javase/7/docs/api/java/util/NavigableMap.html) implementation. The map is sorted according to the [natural ordering](http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html) of its keys, or by a [Comparator](http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html) 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. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's Introduction to Algorithms.

Note that the ordering maintained by a tree map, like any sorted map, and whether or not an explicit comparator is provided, must be consistent with *equals* if this sorted map is to correctly implement the Map interface. (See Comparable or Comparator for a precise definition of consistent with equals.) This is so because the Map interface is defined in terms of the equals operation, but a sorted map performs all key comparisons using its compareTo (or compare) method, so two keys that are deemed equal by this method are, from the standpoint of the sorted map, equal. The behavior of a sorted map is well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the Map interface.

**Note that this implementation is not synchronized.** If multiple threads access a map concurrently, and at least one of the threads modifies the map structurally, it must be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with an existing key is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the [Collections.synchronizedSortedMap](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#synchronizedSortedMap%28java.util.SortedMap%29) method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));

The iterators returned by the iterator method of the collections returned by all of this class's "collection view methods" are fail-fast: if the map is structurally 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](http://docs.oracle.com/javase/7/docs/api/java/util/ConcurrentModificationException.html). 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.

All Map.Entry pairs returned by methods in this class and its views represent snapshots of mappings at the time they were produced. They do **not** support the Entry.setValue method. (Note however that it is possible to change mappings in the associated map using put.)

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

**public interface **NavigableMap<K,V>** extends** [**SortedMap**](http://docs.oracle.com/javase/7/docs/api/java/util/SortedMap.html)**<K,V>**

A [SortedMap](http://docs.oracle.com/javase/7/docs/api/java/util/SortedMap.html) extended with navigation methods returning the closest matches for given search targets. Methods lowerEntry, floorEntry, ceilingEntry, and higherEntry return Map.Entry objects associated with keys respectively less than, less than or equal, greater than or equal, and greater than a given key, returning null if there is no such key. Similarly, methods lowerKey, floorKey, ceilingKey, and higherKey return only the associated keys. All of these methods are designed for locating, not traversing entries.

A NavigableMap may be accessed and traversed in either ascending or descending key order. The descendingMap method returns a view of the map with the senses of all relational and directional methods inverted. The performance of ascending operations and views is likely to be faster than that of descending ones. Methods subMap, headMap, and tailMap differ from the like-named SortedMap methods in accepting additional arguments describing whether lower and upper bounds are inclusive versus exclusive. Submaps of any NavigableMap must implement the NavigableMap interface.

This interface additionally defines methods firstEntry, pollFirstEntry, lastEntry, and pollLastEntry that return and/or remove the least and greatest mappings, if any exist, else returning null.

Implementations of entry-returning methods are expected to return Map.Entry pairs representing snapshots of mappings at the time they were produced, and thus generally do not support the optional Entry.setValue method. Note however that it is possible to change mappings in the associated map using method put.

Methods [subMap(K, K)](http://docs.oracle.com/javase/7/docs/api/java/util/NavigableMap.html#subMap%28K,%20K%29), [headMap(K)](http://docs.oracle.com/javase/7/docs/api/java/util/NavigableMap.html#headMap%28K%29), and [tailMap(K)](http://docs.oracle.com/javase/7/docs/api/java/util/NavigableMap.html#tailMap%28K%29) are specified to return SortedMap to allow existing implementations of SortedMap to be compatibly retrofitted to implement NavigableMap, but extensions and implementations of this interface are encouraged to override these methods to return NavigableMap. Similarly, [SortedMap.keySet()](http://docs.oracle.com/javase/7/docs/api/java/util/SortedMap.html#keySet%28%29) can be overriden to return NavigableSet.

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

**public interface **SortedMap<K,V>** extends** [**Map**](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html)**<K,V>**

A [Map](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html) that further provides a total ordering on its keys. The map is ordered according to the [natural ordering](http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html) of its keys, or by a [Comparator](http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html) typically provided at sorted map creation time. This order is reflected when iterating over the sorted map's collection views (returned by the entrySet, keySet and values methods). Several additional operations are provided to take advantage of the ordering. (This interface is the map analogue of [SortedSet](http://docs.oracle.com/javase/7/docs/api/java/util/SortedSet.html).)

All keys inserted into a sorted map must implement the Comparable interface (or be accepted by the specified comparator). Furthermore, all such keys must be mutually comparable: k1.compareTo(k2) (or comparator.compare(k1, k2)) must not throw a ClassCastException for any keys k1 and k2 in the sorted map. Attempts to violate this restriction will cause the offending method or constructor invocation to throw a ClassCastException.

Note that the ordering maintained by a sorted map (whether or not an explicit comparator is provided) must be consistent with equals if the sorted map is to correctly implement the Map interface. (See the Comparable interface or Comparator interface for a precise definition of consistent with equals.) This is so because the Map interface is defined in terms of the equals operation, but a sorted map performs all key comparisons using its compareTo (or compare) method, so two keys that are deemed equal by this method are, from the standpoint of the sorted map, equal. The behavior of a tree map is well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the Map interface.

All general-purpose sorted map implementation classes should provide four "standard" constructors. It is not possible to enforce this recommendation though as required constructors cannot be specified by interfaces. The expected "standard" constructors for all sorted map implementations are:

1. A void (no arguments) constructor, which creates an empty sorted map sorted according to the natural ordering of its keys.
2. A constructor with a single argument of type Comparator, which creates an empty sorted map sorted according to the specified comparator.
3. A constructor with a single argument of type Map, which creates a new map with the same key-value mappings as its argument, sorted according to the keys' natural ordering.
4. A constructor with a single argument of type SortedMap, which creates a new sorted map with the same key-value mappings and the same ordering as the input sorted map.

**Note**: several methods return submaps with restricted key ranges. Such ranges are half-open, that is, they include their low endpoint but not their high endpoint (where applicable). If you need a closed range (which includes both endpoints), and the key type allows for calculation of the successor of a given key, merely request the subrange from lowEndpoint to successor(highEndpoint). For example, suppose that m is a map whose keys are strings. The following idiom obtains a view containing all of the key-value mappings in m whose keys are between low and high, inclusive:

SortedMap<String, V> sub = m.subMap(low, high+"\0");

A similar technique can be used to generate an open range (which contains neither endpoint). The following idiom obtains a view containing all of the key-value mappings in m whose keys are between low and high, exclusive:

SortedMap<String, V> sub = m.subMap(low+"\0", high);

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

**public class **LinkedHashMap<K,V>** extends** [**HashMap**](http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html)**<K,V> implements** [**Map**](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html)**<K,V>**

Hash table and linked list implementation of the Map interface, with predictable iteration order. This implementation differs from HashMap in that 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 insertion order is not affected if a key is *re-inserted* into the map. (A key k is reinserted into a map m if m.put(k, v) is invoked when m.containsKey(k) would return true immediately prior to the invocation.)

This implementation spares its clients from the unspecified, generally chaotic ordering provided by [HashMap](http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html) (and [Hashtable](http://docs.oracle.com/javase/7/docs/api/java/util/Hashtable.html)), without incurring the increased cost associated with [TreeMap](http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html). It can be used to produce a copy of a map that has the same order as the original, regardless of the original map's implementation:

void foo(Map m) {

Map copy = new LinkedHashMap(m);

...

}

This technique is particularly useful if a module takes a map 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.)

A special [constructor](http://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashMap.html#LinkedHashMap%28int,%20float,%20boolean%29) is provided to create a linked hash map whose order of iteration is the order in which its entries were last accessed, from least-recently accessed to most-recently (*access-order*). This kind of map is well-suited to building LRU caches. Invoking the put or get method results in an access to the corresponding entry (assuming it exists after the invocation completes). The putAll method generates one entry access for each mapping in the specified map, in the order that key-value mappings are provided by the specified map's entry set iterator. *No other methods generate entry accesses.* In particular, operations on collection-views do *not* affect the order of iteration of the backing map.

The [removeEldestEntry(Map.Entry)](http://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashMap.html#removeEldestEntry%28java.util.Map.Entry%29) method may be overridden to impose a policy for removing stale mappings automatically when new mappings are added to the map.

This class provides all of the optional Map operations, and permits null elements. Like HashMap, 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 HashMap, due to the added expense of maintaining the linked list, with one exception: Iteration over the collection-views of a LinkedHashMap requires time proportional to the *size* of the map, regardless of its capacity. Iteration over a HashMap is likely to be more expensive, requiring time proportional to its *capacity*.

A linked hash map has two parameters that affect its performance: *initial capacity* and *load factor*. They are defined precisely as for HashMap. Note, however, that the penalty for choosing an excessively high value for initial capacity is less severe for this class than for HashMap, as iteration times for this class are unaffected by capacity.

**Note that this implementation is not synchronized.** If multiple threads access a linked hash map concurrently, and at least one of the threads modifies the map structurally, it must be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the [Collections.synchronizedMap](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#synchronizedMap%28java.util.Map%29) method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

Map m = Collections.synchronizedMap(new LinkedHashMap(...));

A structural modification is any operation that adds or deletes one or more mappings or, in the case of access-ordered linked hash maps, affects iteration order. In insertion-ordered linked hash maps, merely changing the value associated with a key that is already contained in the map is not a structural modification. **In access-ordered linked hash maps, merely querying the map with get is a structural modification.**)

The iterators returned by the iterator method of the collections returned by all of this class's collection view methods are fail-fast: if the map is structurally 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](http://docs.oracle.com/javase/7/docs/api/java/util/ConcurrentModificationException.html). 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.*

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

**public class **ArrayList<E>** extends** [**AbstractList**](http://docs.oracle.com/javase/7/docs/api/java/util/AbstractList.html)**<E>**

**implements** [**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)**<E>,** [**RandomAccess**](http://docs.oracle.com/javase/7/docs/api/java/util/RandomAccess.html)**,** [**Cloneable**](http://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html)**,** [**Serializable**](http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html)

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in *amortized constant time*, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.

Each ArrayList instance has a *capacity*. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.

An application can increase the capacity of an ArrayList instance before adding a large number of elements using the ensureCapacity operation. This may reduce the amount of incremental reallocation.

**Note that this implementation is not synchronized.** If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements, or explicitly resizes the backing array; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be "wrapped" using the [Collections.synchronizedList](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#synchronizedList%28java.util.List%29) method. This is best done at creation time, to prevent accidental unsynchronized access to the list:

List list = Collections.synchronizedList(new ArrayList(...));

The iterators returned by this class's [iterator](http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html" \l "iterator%28%29) and [listIterator](http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html#listIterator%28int%29) methods are fail-fast: if the list is structurally modified at any time after the iterator is created, in any way except through the iterator's own [remove](http://docs.oracle.com/javase/7/docs/api/java/util/ListIterator.html#remove%28%29) or [add](http://docs.oracle.com/javase/7/docs/api/java/util/ListIterator.html#add%28E%29) methods, the iterator will throw a [ConcurrentModificationException](http://docs.oracle.com/javase/7/docs/api/java/util/ConcurrentModificationException.html). 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.*

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

**public class **LinkedList<E>** extends** [**AbstractSequentialList**](http://docs.oracle.com/javase/7/docs/api/java/util/AbstractSequentialList.html)**<E>**

**implements** [**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)**<E>,** [**Deque**](http://docs.oracle.com/javase/7/docs/api/java/util/Deque.html)**<E>,** [**Cloneable**](http://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html)**,** [**Serializable**](http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html)

Doubly-linked list implementation of the List and Deque interfaces. Implements all optional list operations, and permits all elements (including null).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

**Note that this implementation is not synchronized.** If multiple threads access a linked list concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be "wrapped" using the [Collections.synchronizedList](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#synchronizedList%28java.util.List%29) method. This is best done at creation time, to prevent accidental unsynchronized access to the list:

List list = Collections.synchronizedList(new LinkedList(...));

The iterators returned by this class's iterator and listIterator methods are *fail-fast*: if the list is structurally modified at any time after the iterator is created, in any way except through the Iterator's own remove or add methods, the iterator will throw a [ConcurrentModificationException](http://docs.oracle.com/javase/7/docs/api/java/util/ConcurrentModificationException.html). 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.*

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

**public abstract class **AbstractSequentialList<E>** extends** [**AbstractList**](http://docs.oracle.com/javase/7/docs/api/java/util/AbstractList.html)**<E>**

This class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "sequential access" data store (such as a linked list). For random access data (such as an array), AbstractList should be used in preference to this class.

This class is the opposite of the AbstractList class in the sense that it implements the "random access" methods (get(int index), set(int index, E element), add(int index, E element) and remove(int index)) on top of the list's list iterator, instead of the other way around.

To implement a list the programmer needs only to extend this class and provide implementations for the listIterator and size methods. For an unmodifiable list, the programmer need only implement the list iterator's hasNext, next, hasPrevious, previous and index methods.

For a modifiable list the programmer should additionally implement the list iterator's set method. For a variable-size list the programmer should additionally implement the list iterator's remove and add methods.

The programmer should generally provide a void (no argument) and collection constructor, as per the recommendation in the Collection interface specification.

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

**public class **ArrayDeque<E>** extends** [**AbstractCollection**](http://docs.oracle.com/javase/7/docs/api/java/util/AbstractCollection.html)**<E>**

**implements** [**Deque**](http://docs.oracle.com/javase/7/docs/api/java/util/Deque.html)**<E>,** [**Cloneable**](http://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html)**,** [**Serializable**](http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html)

Resizable-array implementation of the [Deque](http://docs.oracle.com/javase/7/docs/api/java/util/Deque.html) interface. Array deques have no capacity restrictions; they grow as necessary to support usage. They are not thread-safe; in the absence of external synchronization, they do not support concurrent access by multiple threads. Null elements are prohibited. This class is likely to be faster than [Stack](http://docs.oracle.com/javase/7/docs/api/java/util/Stack.html) when used as a stack, and faster than [LinkedList](http://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html) when used as a queue.

Most ArrayDeque operations run in amortized constant time. Exceptions include [remove](http://docs.oracle.com/javase/7/docs/api/java/util/ArrayDeque.html#remove%28java.lang.Object%29), [removeFirstOccurrence](http://docs.oracle.com/javase/7/docs/api/java/util/ArrayDeque.html#removeFirstOccurrence%28java.lang.Object%29), [removeLastOccurrence](http://docs.oracle.com/javase/7/docs/api/java/util/ArrayDeque.html#removeLastOccurrence%28java.lang.Object%29), [contains](http://docs.oracle.com/javase/7/docs/api/java/util/ArrayDeque.html#contains%28java.lang.Object%29), [iterator.remove()](http://docs.oracle.com/javase/7/docs/api/java/util/ArrayDeque.html#iterator%28%29), and the bulk operations, all of which run in linear time.

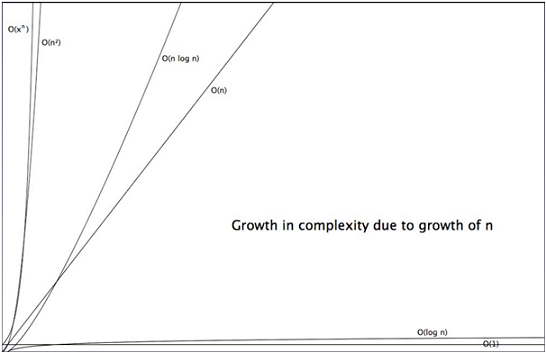
The iterators returned by this class's iterator method are *fail-fast*: If the deque is modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will generally throw a [ConcurrentModificationException](http://docs.oracle.com/javase/7/docs/api/java/util/ConcurrentModificationException.html). 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 and its iterator implement all of the optional methods of the [Collection](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html) and [Iterator](http://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html) interfaces.

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

**“Big O” notation**. This notation approximately describes how the time to do a given task grows with the size of the input. Roughly speaking, on one end we have *O(1)* which is “constant time” and on the opposite end we have *O(xn)* which is “exponential time”. The following chart summarizes the growth in complexity due to growth of input (*n*). In our data structure walk-through we sometimes use the symbol *h* to signify the Hash Table capacity.



**List** *A list is an ordered collection of elements.*

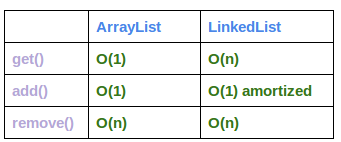
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Add** | **Remove** | **Get** | **Contains** | **Data  Structure** |
| **ArrayList** | O(**1**/**n?**) | O(n) | **O(1)** | O(n) | Array |
| **LinkedList** | **O(1)** | **O(1)** | O(n) | O(n) | Linked List |
| **CopyonWriteArrayList** | O(n) | O(n) | O(1) | O(n) | Array |

<http://www.programcreek.com/2013/09/top-10-questions-for-java-collections/>

ArrayList is essentially an array. Its elements can be accessed directly by index. But if the array is full, a new larger array is needed to allocate and moving all elements to the new array will take O(n) time. Also adding or removing an element needs to move existing elements in an array. This might be the most disadvantage to use ArrayList.

LinkedList is a double linked list. Therefore, to access an element in the middle, it has to search from the beginning of the list. On the other hand, adding and removing an element in LinkedList is quicklier, because it only changes the list locally.

In summary, the worst case of time complexity comparison is as follows:

 | Arraylist | LinkedList

------------------------------------------

get(index) | **O(1)** | O(n)

add(E) | O(n) | **O(1)**

add(E, index) | O(n)+ | O(n)?

remove(index) | O(n)+ | O(n)?

Iterator.remove() | O(n) | **O(1)**

Iterator.add(E) | O(n) | **O(1)**

<http://www.programcreek.com/2013/03/arraylist-vs-linkedlist-vs-vector/>

6. Performance of ArrayList vs. LinkedList - The time complexity comparison is as follows:

\* add() in the table refers to add(E e), and remove() refers to remove(int index)

ArrayList has O(n) time complexity for arbitrary indices of add/remove, but O(1) for the operation at the end of the list.

LinkedList has O(n) time complexity for arbitrary indices of add/remove, but O(1) for operations at end/beginning of the List.

**Set** *A collection that contains no duplicate elements.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Add** | **Contains** | **Next** | **Data Structure** |
| **HashSet** | O(1) | O(1) | O(h/n) | Hash Table |
| **LinkedHashSet** | O(1) | O(1) | O(1) | Hash Table + Linked List |
| **EnumSet** | O(1) | O(1) | O(1) | Bit Vector |
| **TreeSet** | O(log n) | O(log n) | O(log n) | Red-black tree |
| **CopyonWriteArraySet** | O(n) | O(n) | O(1) | Array |
| **ConcurrentSkipList** | O(log n) | O(log n) | O(1) | Skip List |

<http://www.programcreek.com/2013/03/hashset-vs-treeset-vs-linkedhashset/>

HashSet vs. TreeSet vs. LinkedHashSet

**HashSet** is Implemented using a hash table. Elements are not ordered. The add, remove, and contains methods have constant time complexity **O(1).**

**TreeSet** is implemented using a tree structure(red-black tree in algorithm book). The elements in a set are sorted, but the add, remove, and contains methods has time complexity of **O(log (n)).** It offers several methods to deal with the ordered set like first(), last(), headSet(), tailSet(), etc.

**LinkedHashSet** is between HashSet and TreeSet. It is implemented as a hash table with a linked list running through it, so it provides the order of insertion. The time complexity of basic methods is **O(1).**

**Queue** *A collection designed for holding elements prior to processing.*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Offer** | **Peak** | **Poll** | **Size** | **Data Structure** |
| **PriorityQueue** | O(log n ) | O(1) | O(log n) | O(1) | Priority Heap |
| **LinkedList** | O(1) | O(1) | O(1) | O(1) | Array |
| **ArrayDequeue** | O(1) | O(1) | O(1) | O(1) | Linked List |
| **ConcurrentLinkedQueue** | O(1) | O(1) | O(1) | O(n) | Linked List |
| **ArrayBlockingQueue** | O(1) | O(1) | O(1) | O(1) | Array |
| **PriorirityBlockingQueue** | O(log n) | O(1) | O(log n) | O(1) | Priority Heap |
| **SynchronousQueue** | O(1) | O(1) | O(1) | O(1) | None! |
| **DelayQueue** | O(log n) | O(1) | O(log n) | O(1) | Priority Heap |
| **LinkedBlockingQueue** | O(1) | O(1) | O(1) | O(1) | Linked List |

**Map** *An object that maps keys to values. A map cannot duplicate keys; each key can map to at most one value.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Get** | **ContainsKey** | **Next** | **Data Structure** |
| **HashMap** | O(1) | O(1) | O(h / n) | Hash Table |
| **LinkedHashMap** | O(1) | O(1) | O(1) | Hash Table + Linked List |
| **IdentityHashMap** | O(1) | O(1) | O(h / n) | Array |
| **WeakHashMap** | O(1) | O(1) | O(h / n) | Hash Table |
| **EnumMap** | O(1) | O(1) | O(1) | Array |
| **TreeMap** | O(log n) | O(log n) | O(log n) | Red-black tree |
| **ConcurrentHashMap** | O(1) | O(1) | O(h / n) | Hash Tables |
| **ConcurrentSkipListMap** | O(log n) | O(log n) | O(1) | Skip List |

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

<http://javarevisited.blogspot.com/2011/11/collection-interview-questions-answers.html>

**1. How HashMap works in Java?**

This is *Classical Java Collection interview questions* which I have also discussed in [How HashMap works in Java](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html). This collection interview questions is mostly asked during AVP Role interviews on Investment-Banks and has lot of follow-up questions based on response of interviewee e.g. Why HashMap keys needs to be [immutable](http://javarevisited.blogspot.com/2010/10/why-string-is-immutable-in-java.html), what is race conditions on HashMap and how HashMap resize in Java. For explanation and answers of these questions Please see earlier link.

**2. What is difference between poll() and remove() method of Queue interface?**

Though both poll() and remove() method from Queue is used to remove object and returns head of the queue, there is subtle difference between them. If Queue is empty() then a call to remove() method will throw Exception, while a call to poll() method returns null. By the way, exactly which element is removed from the queue depends upon queue's ordering policy and varies between different implementation, for example PriorityQueue keeps lowest element as per Comparator or Comparable at head position.

**3. What is difference between fail-fast and fail-safe Iterators?**

This is relatively *new collection interview questions* and can become trick if you hear the term fail-fast and fail-safe first time. Fail-fast Iterators throws ConcurrentModificationException when one [Thread](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) is iterating over collection object and other thread structurally modify Collection either by adding, removing or modifying objects on underlying collection. They are called fail-fast because they try to immediately throw Exception when they encounter failure. On the other hand [fail-safe Iterators](http://javarevisited.blogspot.com/2011/10/java-iterator-tutorial-example-list.html) works on copy of collection instead of original collection

**4. How do you remove an entry from a Collection? and subsequently what is difference between remove() method of Collection and remove() method of Iterator, which one you will use, while removing elements during iteration?**

Collection interface defines remove(Object obj) method to remove objects from Collection. List interface adds another method remove(int index), which is used to remove object at specific index. You can use any of these method to remove an entry from Collection, while not iterating. Things change, when you iterate. Suppose you are traversing a List and removing only certain elements based on logic, then you need to use Iterator's remove() method. This method removes current element from Iterator's perspective. If you use Collection's or List's remove() method during iteration then your code will throw ConcurrentModificationException. That's why it's advised to use Iterator remove() method to remove objects from Collection.

**5. What is difference between Synchronized Collection and Concurrent Collection?**

Java 5 has added several new Concurrent Collection classes e.g. ConcurrentHashMap, CopyOnWriteArrayList, BlockingQueue etc, which has made Interview questions on Java Collection even trickier. Java Also provided way to get Synchronized copy of collection e.g. ArrayList, HashMap by using Collections.synchronizedMap() Utility function.One Significant difference is that Concurrent Collections has better performance than synchronized Collection because they lock only a portion of Map to achieve concurrency and Synchronization. See [Difference between Synchronized Collection and Concurrent Collection in Java](http://javarevisited.blogspot.com/2011/04/difference-between-concurrenthashmap.html) for more details.

**6. What is difference between Iterator and Enumeration?**

This is a beginner level collection interview questions and mostly asked during interviews of Junior Java developer up to experience of 2 to 3 years Iterator duplicate functionality of Enumeration with one addition of remove() method and both provide navigation functionally on objects of Collection.Another difference is that Iterator is more safe than Enumeration and doesn't allow another thread to modify collection object during iteration except remove() method and throws ConcurrentModificaitonException. See [Iterator vs Enumeration in Java](http://javarevisited.blogspot.com/2010/10/what-is-difference-between-enumeration.html) for more differences.

**7. How does HashSet is implemented in Java, How does it uses Hashing ?**

This is a tricky question in Java, because for hashing you need both key and value and there is no key for store it in a bucket, then how exactly HashSet store element internally. Well, HashSet is built on top of HashMap. If you look at source code of java.util.HashSet class, you will find that that it uses a HashMap with same values for all keys, as shown below :

private transient HashMap map;

// Dummy value to associate with an Object in the backing Map

private static final Object PRESENT = new Object();

When you call add() method of HashSet, it put entry in HashMap :

public boolean add(E e) {

  return map.put(e, PRESENT)==null;

}

Since keys are unique in a HashMap, it provides uniqueness guarantee of Set interface.

**8. What do you need to do to use a custom object as key in Collection classes like Map or Set?**

Answer is : If you are using any custom object in Map as key, you need to override equals() and hashCode() method, and make sure they follow there contract. On the other hand if you are storing a custom object in Sorted Collection e.g. SortedSet or SortedMap, you also need to make sure that your equals() method is consistent to compareTo() method, otherwise those collection will not follow there contacts e.g. Set may allow duplicates.

**9. Difference between HashMap and Hashtable?**

This is another Classical Java Collection interview asked on beginner’s level and most of Java developer has a predefined answer for this interview questions e.g. HashMap is not synchronized while Hashtable is not or hashmap is faster than hash table etc. What could go wrong is that if he placed another follow-up question like how hashMap works in Java or can you replace Hashtable with ConcurrentHashMap etc. See [Hashtable vs HashMap in Java](http://javarevisited.blogspot.com/2010/10/difference-between-hashmap-and.html) for detailed answer of this interview question.

**10. When do you use ConcurrentHashMap in Java?**

This is another advanced level collection interview questions in Java which normally asked to check whether interviewer is familiar with optimization done on ConcurrentHashMap or not. ConcurrentHashMap is better suited for situation where you have multiple readers and one

Writer or fewer writers since Map gets locked only during write operation. If you have equal number of reader and writer than [ConcurrentHashMap](http://javarevisited.blogspot.com/2011/04/difference-between-concurrenthashmap.html) will perform in line of Hashtable or synchronized HashMap.

**11. What is difference between Set and List in Java?**

Another classical Java Collection interview popular on telephonic round or first round of interview. Most of Java programmer knows that Set doesn't allowed duplicate while List does and List maintains insertion order while Set doesn't. What is key here is to show interviewer that you can decide which collection is more suited based on requirements.

**12. How do you Sort objects on collection?**

This Collection interview question serves two purpose it not only test an important programming concept Sorting but also utility class like Collections which provide several methods for creating synchronized collection and sorting. Sorting is implemented using Comparable and Comparator in Java and when you call Collections.sort() it gets sorted based on natural order specified in compareTo() method while Collections.sort(Comparator) will sort objects based on compare() method of Comparator. See [Sorting in Java using Comparator and Comparable](http://javarevisited.blogspot.com/2011/06/comparator-and-comparable-in-java.html) for more details.

**13. What is difference between Vector and ArrayList?**

One more beginner level collection interview questions, this is still very popular and mostly asked in telephonic round. [ArrayList in Java](http://javarevisited.blogspot.com/2011/05/example-of-arraylist-in-java-tutorial.html) is one of the most used Collection class and most interviewer asked questions on ArrayList. See Difference between Vector and ArrayList for answer of this interview question.

**14. What is difference between HashMap and HashSet?**

This collection interview questions is asked in conjunction with HashMap vs Hashtable. HashSet implements java.util.Set interface and that's why only contains unique elements, while HashMap allows duplicate values.  In fact, HashSet is actually implemented on top of java.util.HashMap. If you look internal implementation of java.util.HashSet, you will find that it adds element as key on internal map with same values. For a more detailed answer, see [HashMap vs HashSet](http://javarevisited.blogspot.com/2011/09/difference-hashmap-vs-hashset-java.html).

**15) What is NavigableMap in Java ? What is benefit over Map?**

NavigableMap Map was added in Java 1.6, it adds navigation capability to Map data structure. It provides methods like lowerKey() to get keys which is less than specified key, floorKey() to return keys which is less than or equal to specified key, ceilingKey() to get keys which is greater than or equal to specified key and higherKey() to return keys which is greater specified key from a Map. It also provide similar methods to get entries e.g. lowerEntry(), floorEntry(), ceilingEntry() and higherEntry(). Apart from navigation methods, it also provides utilities to create sub-Map e.g. creating a Map from entries of an exsiting Map like tailMap, headMap and subMap. headMap() method returns a NavigableMap whose keys are less than specified, tailMap() returns a NavigableMap whose keys are greater than the specified and subMap() gives a NavigableMap between a range, specified by toKey to fromKey.

**16) Which one you will prefer between Array and ArrayList for Storing object and why?**Though ArrayList is also backed up by array, it offers some usability advantage over array in Java. Array is fixed length data structure, once created you can not change it's length. On the other hand, ArrayList is dynamic, it automatically allocate a new array and copies content of old array, when it resize. Another reason of using ArrayList over Array is support of Generics. Array doesn't support Generics, and if you store an Integer object on a String array, you will only going to know about it at runtime, when it throws ArrayStoreException. On the other hand, if you use ArrayList, compiler and IDE will catch those error on the spot. So if you know size in advance and you don't need re-sizing than use array, otherwise use ArrayList.

**17) Can we replace Hashtable with ConcurrentHashMap?**

Answer 3 : Yes we can replace Hashtable with ConcurrentHashMap and that's what suggested in Java documentation of ConcurrentHashMap. but you need to be careful with code which relies on locking behavior of Hashtable. Since Hashtable locks whole Map instead of portion of Map, compound operations like if(Hashtable.get(key) == null) put(key, value) works in Hashtable but not in concurrentHashMap. instead of this use putIfAbsent() method of ConcurrentHashMap

**18) What is CopyOnWriteArrayList, how it is different than ArrayList and Vector?**

Answer : CopyOnWriteArrayList is new List implementation introduced in Java 1.5 which provides better concurrent access than Synchronized List. better concurrency is achieved by Copying ArrayList over each write and replace with original instead of locking. Also CopyOnWriteArrayList doesn't throw any ConcurrentModification Exception. Its different than ArrayList because its thread-safe and ArrayList is not thread safe and its different than Vector in terms of Concurrency. CopyOnWriteArrayList provides better Concurrency by reducing contention among readers and writers.

**19) Why ListIterator has add() method but Iterator doesn't or Why add() method is declared in ListIterator and not on Iterator.**

Answer : ListIterator has add() method because of its ability to traverse or iterate in both direction of collection. it maintains two pointers in terms of previous and next call and in position to add new element without affecting current iteration.

**20) When does ConcurrentModificationException occur on iteration?**

When you remove object using Collection's or List's remove method e.g. remove(Object element) or remove(int index), instead of Iterator's remove() method than ConcurrentModificationException occur. As per Iterator's contract, if it detect any structural change in Collection e.g. adding or removing of element, once Iterator begins, it can throw ConcurrentModificationException.

**21) Difference between Set, List and Map Collection classes?**

java.util.Set, java.util.List and java.util.Map defines three of most popular data structure support in Java. Set provides uniqueness guarantee i.e.g you can not store duplicate elements on it, but it's not ordered. On the other hand List is an ordered Collection and also allowes duplicates. Map is based on hashing and stores key and value in an Object called entry. It provides O(1) performance to get object, if you know keys, if there is no collision. Popular impelmentation of Set is HashSet, of List is ArrayList and LinkedList, and of Map are HashMap, Hashtable and ConcurrentHashMap. Another key difference between Set, List and Map are that Map doesn't implement Collection interface, while other two does. For a more detailed answer, see Set vs List vs Map in Java

**22) What is BlockingQueue, how it is different than other collection classes?**

BlockingQueue is a Queue implementation available in java.util.concurrent package. It's one of the concurrent Collection class added on Java 1.5, main difference between BlockingQueue and other collection classes is that apart from storage, it also provides flow control. It can be used in inter thread communication and also provides built-in thread-safety by using happens-before guarantee. You can use BlockingQueue to solve Producer Consumer problem, which is what is needed in most of concurrent applications.

Few more questions for practice, try to find answers of these question by yourself :

23) How does LinkedList is implemented in Java, is it a Singly or Doubly linked list?

hint : LinkedList in Java is a doubly linked list.

24) How do you iterator over Synchronized HashMap, do you need to lock iteration and why ?

25) What is Deque? when do you use it ?

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

<http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html>

HashMap in Java works on hashing principle. It is a data structure which allows us to store object and retrieve it in constant time O(1) provided we know the key. In hashing, hash functions are used to link key and value in HashMap. Objects are stored by calling put(key, value) method of HashMap and retrieved by calling get(key) method. When we call put method, hashcode() method of key object is called so that hash function of map can find a bucket location to store value object, which is actually index of internal array, known as table. HashMap internally store mapping in form of Map.Entry object which contains both key and value object. When you want to retrieve the object, you call get() method and again pass key object. This time again key object generate same hash code (it's mandatory for it to do so to retrieve object and that's why HashMap keys are immutable e.g. String) and we end up at same bucket location. If there is only one object then it is returned and that's your value object which you have stored earlier. Things get little tricky when collisions occurs. Since internal array of HashMap is of fixed size, and if you keep storing objects, at some point of time hash function will return same bucket location for two different keys, this is called collision in HashMap. In this case, a linked list is formed at that bucket location and new entry is stored as next node. If we try to retrieve object from this linked list, we need an extra check to search correct value, this is done by equals() method. Since each node contains an entry, HashMap keep comparing entry's key object with passed key using equals() and when it return true, Map returns corresponding value. Since searching in lined list is O(n) operation, in worst case hash collision reduce a map to linked list. This issue is recently addressed in Java 8 by replacing linked list to tree to search in O(logN) time. By the way, you can easily verify how HashMap work by looking at code of HashMap.java in your Eclipse IDE, if you know [how to attach source code of JDK in Eclipse](http://javarevisited.blogspot.com/2012/12/how-to-attach-source-in-eclipse-Jar-JDK-debugging.html).

How HashMap works in Java or sometime how get method work in HashMap is a very common question on Java interviews now days. Almost everybody who worked in Java knows about HashMap, where to use HashMap and difference between Hashtable and HashMap then why this interview question becomes so special? Because of the depth it offers. It has become very popular Java interview question in almost any senior or mid-senior level Java interviews. Investment banks mostly prefer to ask this question and some time even ask you to implement your own HashMap based upon your coding aptitude. Introduction of [ConcurrentHashMap](http://javarevisited.blogspot.co.uk/2013/02/concurrenthashmap-in-java-example-tutorial-working.html) and other concurrent collections has also made this questions as starting point to delve into more advanced feature. let's start the journey.

## How HashMap Internally Works in Java

Questions start with simple statement :

**Have you used HashMap before**or**What is HashMap? Why do you use it**

Almost everybody answers this with yes and then interviewee keep talking about common facts about HashMap like HashMap accept null while Hashtable doesn't, [HashMap is not synchronized](http://javarevisited.blogspot.com/2010/10/difference-between-hashmap-and.html), HashMap is fast and so on along with basics like its stores key and value pairs etc. This shows that person has used HashMap and quite familiar with the functionality it offers, but interview takes a sharp turn from here and next set of follow-up questions gets more detailed about fundamentals involved with HashMap in Java . Interviewer strike back with questions like :

**Do you Know how HashMap works in Java** or **How does get () method of HashMap works in Java**

And then you get answers like,  I don't bother its standard Java API, you better look code on Java source or Open JDK; I can find it out in Google at any time etc. But some interviewee definitely answer this and will say **HashMap works on principle of hashing**, we have put(key, value) and get(key) method for storing and retrieving Objects from HashMap. When we pass Key and Value object  to put() method on Java HashMap, HashMap implementation calls [hashCode method](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html) on Key object and applies returned hashcode into its own hashing function to find a bucket location for storing Entry object, important point to mention is that HashMap in Java stores both key and value object as Map.Entry in bucket which is essential to understand the retrieving logic. If people fails to recognize this and say it only stores Value in the bucket they will fail to explain the retrieving logic of any object stored in Java HashMap . This answer is very much acceptable and does make sense that interviewee has fair bit of knowledge on how hashing works and how HashMap  works in Java. But this is just start of story and confusion increases when you put interviewee on scenarios faced by Java developers on day by day basis. Next question could be about collision detection and collision resolution in Java HashMap  e.g.

**What will happen if two different objects have same hashcode?**

Now from here onwards real confusion starts, Some time candidate will say that since hashcode is equal, both objects are equal and HashMap  will throw exception or not store them again etc, Then you might want to remind them about [equals() and hashCode() contract](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html) that two unequal object in Java can have same hash code. Some will give up at this point and few will move ahead and say "Since hashcode is same, bucket location would be same and collision will occur in HashMap, Since HashMap use LinkedList to store object, this entry (object of Map.Entry comprise key and value )  will be stored in [LinkedList](http://javarevisited.blogspot.sg/2012/02/difference-between-linkedlist-vs.html). Great this answer make sense though there are many collision resolution methods available  like linear probing and chaining, this is simplest and HashMap in Java does follow this. But story does not end here and interviewer asks

--------

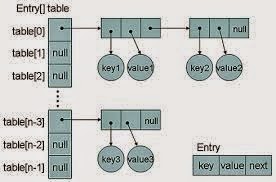
**How will you retrieve Value object  if two Keys will have same hashcode?**

Interviewee will say we will call get() method and then HashMap uses Key Object's hashcode to find out bucket location and retrieves Value object but then you need to remind him that there are two Value objects are stored in same bucket , so they will say about [traversal in LinkedList](http://javarevisited.blogspot.sg/2010/10/how-do-you-find-length-of-singly-linked.html) until we find the value object , then you ask *how do you identify value object because you don't  have value object to compare* ,Until they know that HashMap  stores both Key and Value in LinkedList node or as Map.Entry they won't be able to resolve this issue and will try and fail.

-------

But those bunch of people who remember this key information will say that after finding bucket location , we will **call keys.equals() method** to identify correct node in LinkedList and return associated value object for that key in Java HashMap . Perfect this is the correct answer.

In many cases interviewee fails at this stage because they get confused between[hashCode()](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html) and equals(**)** or keys and values object in Java HashMap  which is pretty obvious because they are dealing with the hashcode() in all previous questions and equals() come in picture only in case of retrieving value object from HashMap in Java. Some good developer point out here that using immutable, [final object](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) with proper equals() and hashcode() implementation would act as perfect Java HashMap  keys and **improve performance of Java HashMap  by reducing collision**. Immutability *also allows caching there hashcode of different keys* which makes overall retrieval process very fast and suggest that [String](http://javarevisited.blogspot.sg/2011/07/string-vs-stringbuffer-vs-stringbuilder.html) and various wrapper classes e.g. Integer very good keys in Java HashMap.



Now if you clear this entire Java HashMap interview,  You will be surprised by this very interesting question "**What happens On HashMap in Java if the size of the HashMap  exceeds a given threshold defined by load factor ?"**. Until you know how HashMap  works exactly you won't be able to answer this question. If the size of the Map exceeds a given threshold defined by load-factor e.g. if load factor is .75 it will act to re-size the map once it filled 75%. Similar to other collection classes like [ArrayList](http://javarevisited.blogspot.sg/2011/05/example-of-arraylist-in-java-tutorial.html),  Java HashMap re-size itself by creating a new bucket array of size twice of previous size of HashMap , and then start putting every old element into that new bucket array. This process is called rehashing because it also applies hash function to find new bucket location.

If you manage to answer this question on HashMap in Java you will be greeted by **"do you see any problem with resizing of HashMap  in Java"** , you might not be able to pick the context and then he will try to give you hint about multiple thread accessing the Java HashMap and potentially looking for **race condition on HashMap  in Java**.

So the answer is Yes there is potential [race condition](http://javarevisited.blogspot.sg/2012/02/what-is-race-condition-in.html) exists while resizing HashMap in Java, if two [thread](http://javarevisited.blogspot.sg/2011/02/how-to-implement-thread-in-java.html) at the same time found that now HashMap needs resizing and they both try to resizing. on the process of resizing of HashMap in Java , the element in bucket which is stored in linked list get reversed in order during there migration to new bucket because Java HashMap  doesn't append the new element at tail instead it append new element at head *to avoid tail traversing*. If race condition happens then you will end up with an infinite loop. Though this point you can potentially argue that what the hell makes you think to use HashMap  in multi-threaded environment to interviewer :)

## Some more Hashtable and HashMap Questions

Few more question on HashMap in Java which is contributed by readers of Javarevisited blog :

**1) Why String, Integer and other wrapper classes are considered good keys ?**

String, Integer and other wrapper classes are natural candidates of HashMap key, and String is most frequently used key as well because [String is immutable and final](http://javarevisited.blogspot.sg/2010/10/why-string-is-immutable-in-java.html),and overrides equals and hashcode() method. Other wrapper class also shares similar property. Immutabiility is required, in order to prevent changes on fields used to calculate hashCode() because if key object return different hashCode during insertion and retrieval than it won't be possible to get object from HashMap. Immutability is best as it offers other advantages as well like [thread-safety](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html), If you can keep your hashCode same by only making certain fields final, then you go for that as well. Since equals() and hashCode() method is used during reterival of value object from HashMap, its important that key object correctly override these methods and follow contact. If unequal object return different hashcode than chances of collision will be less which subsequently improve performance of HashMap.

**2) Can we use any custom object as key in HashMap ?**

This is an extension of previous questions. Ofcourse you can use any Object as key in Java HashMap provided it follows equals and hashCode contract and its hashCode should not vary once the object is inserted into [Map](http://javarevisited.blogspot.sg/2011/12/how-to-traverse-or-loop-hashmap-in-java.html). If custom object is Immutable than this will be already taken care because you can not change it once created.

**3) Can we use ConcurrentHashMap in place of Hashtable ?**

This is another question which getting popular due to increasing popularity of ConcurrentHashMap. Since we know Hashtable is synchronized but ConcurrentHashMap provides better concurrency by only locking portion of map determined by concurrency level. ConcurrentHashMap is certainly introduced as Hashtable and can be used in place of it but Hashtable provide stronger thread-safety than ConcurrentHashMap. See my post [difference between Hashtable and ConcurrentHashMap](http://javarevisited.blogspot.sg/2011/04/difference-between-concurrenthashmap.html) for more details.

Personally, I like this question because of its depth and number of concept it touches indirectly, if you look at questions asked during interview this HashMap  questions has verified

* Concept of hashing
* Collision resolution in HashMap
* Use of equals () and hashCode () and there importance in HashMap?
* Benefit of immutable object?
* Race condition on HashMap  in Java
* Resizing of Java HashMap

Just to summarize here are the answers which does makes sense for above questions

**How HashMap  works in Java**

HashMap  works on principle of hashing, we have put() and get() method for storing and retrieving object form HashMap .When we pass an both key and value to put() method to store on HashMap , it uses key object hashcode() method to calculate hashcode and they by applying hashing on that hashcode it identifies bucket location for storing value object. While retrieving it uses key object equals method to find out correct key value pair and return value object associated with that key. HashMap  uses linked list in case of collision and object will be stored in next node of linked list. Also [HashMap stores both key and value tuple](http://java67.blogspot.com/2013/02/10-examples-of-hashmap-in-java-programming-tutorial.html) in every node of linked list in form of Map.Entry object.

**What will happen if two different HashMap  key objects have same hashcode?**

They will be stored in same bucket but no next node of linked list. And keys equals () method will be used to identify correct key value pair in HashMap .

**How null key is handled in HashMap? Since equals() and hashCode() are used to store and retrieve values, how does it work in case of null key?**

Null key is handled specially in HashMap, there are two separate method for that putForNullKey(V value) and getForNullKey(). Later is offloaded version of get() to look up null keys.  Null keys always map to index 0.  This null case is split out into separate methods for the sake of performance in the two most commonly used operations (get and put), but incorporated with conditionals in others. In short, equals() and hashcode() method are not used in case of null keys in HashMap.

here is how nulls are retreived from HashMap

**private** V **getForNullKey**() {

**if** (size == **0**) {

**return** **null**;

}

**for** (Entry<K,V> e = table[**0**]; e != **null**; e = e.next) {

**if** (e.key == **null**)

**return** e.value;

}

**return** **null**;

}

In terms of usage Java HashMap is very versatile and I have mostly used HashMap as cache in electronic trading application I have worked . Since finance domain used Java heavily and due to performance reason we need caching HashMap and ConcurrentHashMap  comes as very handy there. You can also check following articles form Javarevisited to learn more about HashMap and Hashtable in Java :

## HashMap Changes in JDK 1.7 and JDK 1.8

There is some [performance improvement done on HashMap and ArrayList from JDK 1.7](http://javarevisited.blogspot.com/2014/07/java-optimization-empty-arraylist-and-Hashmap-cost-less-memory-jdk-17040-update.html), which reduce memory consumption. Due to this empty Map are lazily initialized and will cost you less memory. Earlier, when you create HashMap e.g. new HashMap() it automatically creates array of default length e.g. 16. After some research, Java team founds that most of this Map are temporary and never use that many elements, and only end up wasting memory. Also, From JDK 1.8 onwards HashMap has introduced an improved strategy to deal with high collision rate. Since a poor hash function e.g. which always return location of same bucket, can turn a HashMap into linked list, i.e. converting get() method to perform in O(n) instead of O(1) and someone can take advantage of this fact, Java now internally replace linked list to a binary true once certain threshold is breached. This ensures performance or order O(log(n)) even in worst case where hash function is not distributing keys properly.

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

**HashMap vs Hashtable in Java**

Though both Hashtable and HashMap are data-structure based upon hashing and implementation of Map interface, main difference between them is that HashMap is not thread-safe but Hashtable is thread-safe. Which means you cannot use HashMap in multi-threaded Java application without external synchronization. Another difference is HashMap allows one null key and null values but Hashtable doesn't allow null key or values. Also thread-safety of hash table is achieved using internal synchronization, which makes it slower than HashMap. By the way *difference between HashMap and Hashtable in Java* is one of the frequently asked in core Java interviews to check whether candidate understand correct usage of collection classes and aware of alternative solutions available. Along with [How HashMap internally works in Java](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html) and [ArrayList vs Vector](http://javarevisited.blogspot.sg/2011/09/difference-vector-vs-arraylist-in-java.html), this  is one of the oldest question from Collection framework in Java. Hashtable is a legacy Collection class and it's there in Java API from long time but it got re-factored to implement Map interface in Java 4 and from there Hashtable became part of Java Collection framework. Hashtable vs HashMap in Java is so popular a question that it can top any list of [Java Collection interview Question](http://javarevisited.blogspot.sg/2011/11/collection-interview-questions-answers.html). You just can't afford not to prepare HashMap vs Hashtable before going to any Java programming interview. In this Java article we will not only see some important differences between HashMap and Hashtable but also some similarities between these two collection classes. Let's first see How different they are :

## Difference between HashMap and Hashtable in Java

Both HashMap and Hashtable implements Map interface but there are some significant difference between them which is important to remember before deciding whether to use HashMap or Hashtable in Java. Some of them is thread-safety, synchronization and speed. here are those differences :

1.The HashMap class is roughly equivalent to Hashtable, except that it is non synchronized and permits nulls. (HashMap allows null values as key and value whereas [Hashtable](http://javarevisited.blogspot.sg/2012/01/java-hashtable-example-tutorial-code.html) doesn't allow nulls).

2. One of the major **differences between HashMap and Hashtable** is that HashMap is non synchronized whereas Hashtable is synchronized, which means Hashtable is thread-safe and can be shared between multiple threads but HashMap can not be shared between multiple threads without proper synchronization. Java 5 introduces [ConcurrentHashMap](http://javarevisited.blogspot.sg/2011/04/difference-between-concurrenthashmap.html) which is an alternative of Hashtable and provides better scalability than Hashtable in Java.

3. Another significant difference between HashMap vs Hashtable is that Iterator in the HashMap is  a fail-fast iterator  while the enumerator for the Hashtable is not and throw ConcurrentModificationException if any other Thread modifies the map structurally  by adding or removing any element except Iterator's own remove() method. But this is not a guaranteed behavior and will be done by JVM on best effort. This is also an important [difference between Enumeration and Iterator in Java](http://javarevisited.blogspot.sg/2010/10/what-is-difference-between-enumeration.html).

4. One more notable *difference between Hashtable and HashMap* is that because of thread-safety and synchronization Hashtable is much slower than HashMap if used in Single threaded environment. So if you don't need synchronization and HashMap is only used by one thread, it out perform Hashtable in Java.

5. HashMap does not guarantee that the order of the map will remain constant over time.

### HashMap and Hashtable : note on Some Important Terms

1)Synchronized means only one Thread can modify a hash table at one point of time. Basically, it means that any thread before performing an update on a Hashtable will have to acquire a lock on the object while others will wait for lock to be released.

2) Fail-safe is relevant from the context of iterators. If an [Iterator or ListIterator](http://javarevisited.blogspot.sg/2011/10/java-iterator-tutorial-example-list.html) has been created on a collection object and some other thread tries to modify the collection object "structurally", a concurrent modification exception will be thrown. It is possible for other threads though to invoke "set" method since it doesn't modify the collection "structurally". However, if prior to calling "set", the collection has been modified structurally, "IllegalArgumentException" will be thrown.

3) Structurally modification means deleting or inserting element which could effectively change the structure of map.

HashMap can be synchronized by

Map m = Collections.synchronizeMap(hashMap);

In Summary there are significant *differences between Hashtable and HashMap in Jav*a e.g. thread-safety and speed and based upon that only use Hashtable if you absolutely need thread-safety, if you are running Java 5 consider using ConcurrentHashMap in Java.

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

# [Difference between HashMap, LinkedHashMap and TreeMap in Java](http://javarevisited.blogspot.com/2015/08/difference-between-HashMap-vs-TreeMap-vs-LinkedHashMap-Java.html)

Map is one of the most important data structure from Java Collection Framework.  It provides hash table data structure functionality with it's rich implementations like HashMap, Hashtable, LinkedHashMap and little bit of sorting with TreeMap. So if you are looking to store key value pairs in Java program,  you have wide range of choices available depending upon your requirement. Main difference between LinkedHashMap, TreeMap and HashMap comes in there internal implementation and specific features, which makes them useful in certain scenarios. For example, [HashMap](http://java67.blogspot.sg/2013/02/10-examples-of-hashmap-in-java-programming-tutorial.html) is a general purpose Map (hash table data structure), which should be used whenever you need a hashing based data structure for storing your mappings (key value pairs). **TreeMap** provides you sorting, on top of hashing offered by Map interface, which means you can not only retrieve elements in constant time i.e. O(1) time, but also iterate through those mapping in a predefined sorted order, but you need to pay heavy price to keep mappings in sorted order. On the other hand, **LinkedHashMap** is a compromise between these two, it doesn't provide sorting but unlike HashMap, it provides ordering e.g. maintaining mappings in a order they are inserted into Map, known as *insertion order* or order on which they are accessed, called *access order*. Apart from these three popular Map implementation, you also have some special purpose Map implementations e.g. [EnumMap](http://javarevisited.blogspot.sg/2012/09/what-is-enummap-in-java-example-tutorial.html) for storing mapping with enum constants as keys,  it is highly optimized for enum constants. You also have a special map called [WeakHashMap](http://javarevisited.blogspot.sg/2014/03/difference-between-weakreference-vs-softreference-phantom-strong-reference-java.html) for creating a Garbage Collector friendly Cache, where values become eligible for garbage collection as soon as there is no other reference to them apart from keys in WeakHashMap.Then there is [IdentityHashMap](http://javarevisited.blogspot.sg/2013/01/difference-between-identityhashmap-and-hashmap-java.html) for creating a Map which uses identity instead of equality for comparing keys, since identity equality is rare, you get less number of collision on this Map and finally JDK 5 introduced [ConcurrentHashMap](http://javarevisited.blogspot.sg/2013/02/concurrenthashmap-in-java-example-tutorial-working.html) for better scalability in multi-threaded environment, where number of reader threads clearly out numbers number of writer threads.

## LinkedHashMap vs TreeMap vs HashMap

Though all three classes implements java.util.Map interface and follows general contract of a Map interface, defined in terms of [equals() and hashCode()](http://javarevisited.blogspot.sg/2015/01/why-override-equals-hashcode-or-tostring-java.html) method, they also have several differences in terms of Ordering, Sorting, permitting null elements, Iteration, Performance, Speed and internal implementation. Let's have a quick look on each of these property.

#### Ordering and Sorting

HashMap doesn't provide any ordering guarantee for entries, which means, you can not assume any order while [iterating over keys and values of HashMap](http://java67.blogspot.sg/2013/08/best-way-to-iterate-over-each-entry-in.html). This behavior of HashMap is similar to Hashtable, while other two Map implementation provides ordering guarantee.

LinkedHashMap can be used to maintain insertion-order, on which keys are inserted into Map or it can also be used to maintain an access-order, on which keys are accessed. This provides LinkedHashMap an edge over HashMap without compromising too much performance.

TreeMap provides you complete control over sorting elements by passing [custom Comparator](http://javarevisited.blogspot.sg/2014/01/java-comparator-example-for-custom.html) of your choice, but with expense of some performance. Since entries are stored in a tree based data structure, it provides lower performance than HashMap and LinkedHashMap.

#### Null keys and Values

HashMap allows one null key and multiple null values. It keeps null key based entries on index[0] on internal bucket. If you look at put() method of HashMap, you can see, it doesn't throw [NullPointerException for null keys](http://javarevisited.blogspot.sg/2012/06/common-cause-of-javalangnullpointerexce.html). Since LinkedHashMap is a sub class of HashMap, it also allows null keys and values.

On the other hand TreeMap, which sorts elements in natural order doesn't allow null keys because compareTo() method throws NullPointerException if compared with null. If you are using TreeMap with [user defined Comparator](http://java67.blogspot.sg/2014/11/java-8-comparator-example-using-lambda-expression.html), than it depends upon implementation of compare() method.

#### Iterators

Iterators returned by all these Map's collection view methods e.g. values() or keySet() is [fail-fast iterators](http://java67.blogspot.sg/2015/06/what-is-fail-safe-and-fail-fast-iterator-in-java.html), which means they will throw ConcurrentModificatoinException, if Collection is modified structurally once Iteration begins, except by using remove() method of Iterator.

By the way, it's worth remembering that apart from adding or removing more mappings, it can also be any operation which affects iteration order of LinkedHashMap. In access-ordered LinkedHashMap, even querying the Map with get() method is a structural modification, because it changes the iteration order, on the other hand updating value in a insertion-ordered linked hash map is not a structural modification.

Finally, fail-fast behavior is not guaranteed, and they throw ConcurrentModificationException on best-effort basis, which means do not write code, which depends upon this behavior. It should only be used to detect programming bugs.

#### Performance and Speed

Since HashMap is bare bone implementation of java.util.Map interface, it provides constant time performance for get() and put() operation, where put is used to store key value pair and get is used to retrieve value based upon key. BTW, constant time performance is only if mappings are distributed uniformly across bucket location, but HashMap is certainly faster than Hashtable because it's not synchronized. [Iteration over Map](http://java67.blogspot.sg/2014/05/3-examples-to-loop-map-in-java-foreach.html) is directly proportional to the "capacity" + "size" of HashMap, that's why it's important to set the initial capacity high enough, if iteration performance is important. You can further use **initial capacity** and **load factor** to fine tune your HashMap performance, to avoid rehashing of HashMap.

Since TreeMap is based on tree data structure, it provides log(n) time for get(), put(), containsKey() and remove() operation, so it's costlier than HashMap, if order is not concerned.

LinkedHashMap is a trade-off between two, like HashMap it also provides constant time performance for add, contains and remove, though its slightly slower than HashMap, to maintain linked list. By the way Iteration over Map in case of LinkedHashMap is slightly faster than HashMap, because time required is proportional to size only. So if you need insertion order or *access order*, consider using LinkedHashMap over TreeMap in Java.

#### Thread-safety and Synchronization

All three Map implementation are [not thread-safe](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html), which means you can not use them safely in a multi-threaded application. Though you can synchronized them externally by using Collections.synchronizedMap(Map map) method. Alternatively you can also use there concurrent counter part e.g. ConcurrentHashMap which is also a better choice than HashMap in a concurrent Java application.

When using synchronized Map e.g. synchronized LinkedHashMap or SortedMap, you must do at the time or creating map to prevent accidental non synchronized access. You can use following idiom to create Synchronized Map in Java :

**Synchronized LinkedHashMap**

Map<Integer, Integer> numbers **=** Collections**.**synchronizedMap(**new** LinkedHashMap<>());

**Synchronized TreeMap**

SortedMap<Integer, String> sorted **=** Collections**.**synchronizedSortedMap(**new** TreeMap<>());

Remember to use Collections.synchronizedMap() for synchronizing HashMap, LinkedHashMap and Collections.synchronizedSortedMap() method for synchronizing TreeMap. If you are not comfortable then see this guide on [how to synchronize HashMap in Java](http://java67.blogspot.sg/2015/02/how-to-synchronize-hashmap-in-java-with.html).

#### Internal Implementation

TreeMap is Red-Black tree based NavigableMap implementation, while HashMap is internally backed by an array. It uses index[0] to store entries corresponding to null keys. In fact questions related to inner working of HashMap is very popular in Java, for example [How get() method of HashMap works internally](http://java67.blogspot.sg/2013/06/how-get-method-of-hashmap-or-hashtable-works-internally.html) is one of the frequently used questions to Senior Java developers. On the other hand LinkedHashMap extends HashMap and uses linked list to provide insertion order guarantee. It uses doubly linked list running through all of it's entries, which can also be used to maintain access-order. Remember, insertion order is not affected if a key is re-inserted into LinkedHashMap, but access order is affected, if LinkedHashMap is created to maintain access-order.

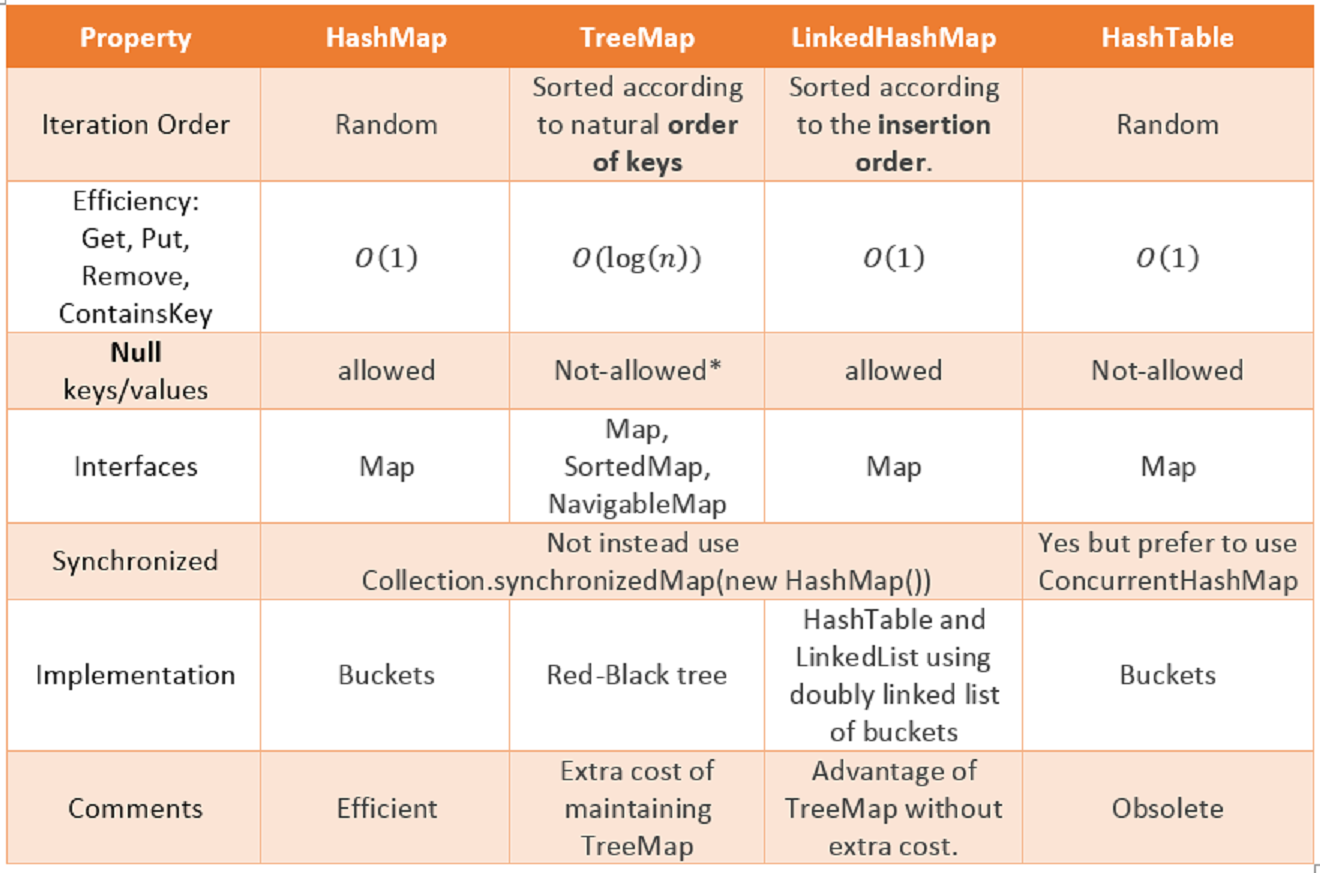
## When to use LinkedHashMap, TreeMap and HashMap in Java

You can use a LinkedHashMap, when you need to keep your mappings in either **insertion order** or **access-order**. LinkedHashMap by default keeps elements in the order, on which they are inserted, and this order is reflected when you [traverse over LinkedHashMap](http://javarevisited.blogspot.sg/2011/12/how-to-traverse-or-loop-hashmap-in-java.html), but it also provides a constructor, which allows you to keep entries in *access-order*, i.e. order in which they are accessed. One of the clever use of Java LinkedHashMap is to use it as Least Recently Use or **LRU Cache**.

**TreeMap** is your go to map implementation if you want to keep keys  in a sorted order, either in there natural order defined by Comparable interface or a custom order imposed by Comparator interface, though it's worth remembering that your compareTo() or compare() method must be [consistent with equals() method](http://java67.blogspot.sg/2013/04/example-of-overriding-equals-hashcode-compareTo-java-method.html), because Map interface is defined in terms of equals and TreeMap uses compareTo for comparing keys. So if keys compare() or compareTo() implementation is not consistent, then it will fail to obey Map's general contract.

**HashMap** is your general purpose hashing based collection, whenever you need to use a hash table data structure in Java to store key value pairs, first choice goes to HashMap in single threaded environment. If you happened to use a Map in a multi-threaded environment consider using [Hashtable, synchronized HashMap or ConcurrentHashMap](http://javarevisited.blogspot.sg/2011/04/difference-between-concurrenthashmap.html) from Java Collection Framework.

Since LinkedHashMap solved problem of chaotic ordering provided by Hashtable and HashMap, without incurring high cost associated with TreeMap, you can also used LinkedHashMap to create a copy of a Map in Java, as shown in below example.



That's all on **difference between LinkedHashMap, TreeMap and HashMap in Java**. Though all three are Map implementation, they have different purpose and used accordingly. Use LinkedHashMap, if you need to maintain insertion or access order of mappings e.g. in LRU Cache. Use TreeMap, if you need to maintain mappings in a sorted order, either in there natural order or a custom order defined by Comparator and use HashMap for all your general purpose hashing based collection requirement. HashMap allows you to retrieve object in O(1) time, if you know key.

**Further Reading**

If you are really serious about mastering various classes from Java Collection framework then I suggest you to read following two books, both of them are classic and highly recommended for experienced Java developer who wants to constantly improve their skill and knowledge of Java API.

* What is difference between HashSet, TreeSet and LinkedHashSet in Java? ([answer](http://javarevisited.blogspot.sg/2012/11/difference-between-treeset-hashset-vs-linkedhashset-java.html))
* What is difference between HashMap and ArrayList in Java? ([answer](http://java67.blogspot.sg/2012/08/difference-between-hashmap-and-ArrayList-in-Java.html))
* What is difference between HashSet and ArrayList in Java? ([answer](http://java67.blogspot.sg/2012/07/difference-between-arraylist-hashset-in-java.html))
* 5 differences between HashMap and Hashtable in Java? ([answer](http://java67.blogspot.sg/2012/08/5-difference-between-hashtable-hashmap-Java-collection.html))
* What is difference between ArrayList and LinkedList in Java? ([answer](http://java67.blogspot.sg/2012/12/difference-between-arraylist-vs-LinkedList-java.html))
* How to use NavigableMap in Java 6? [[example](http://javarevisited.blogspot.sg/2013/01/what-is-navigablemap-in-java-6-example-submap-head-tail.html)]
* How to use BlockingQueue in Java Program? [[example](http://javarevisited.blogspot.sg/2012/02/producer-consumer-design-pattern-with.html)]