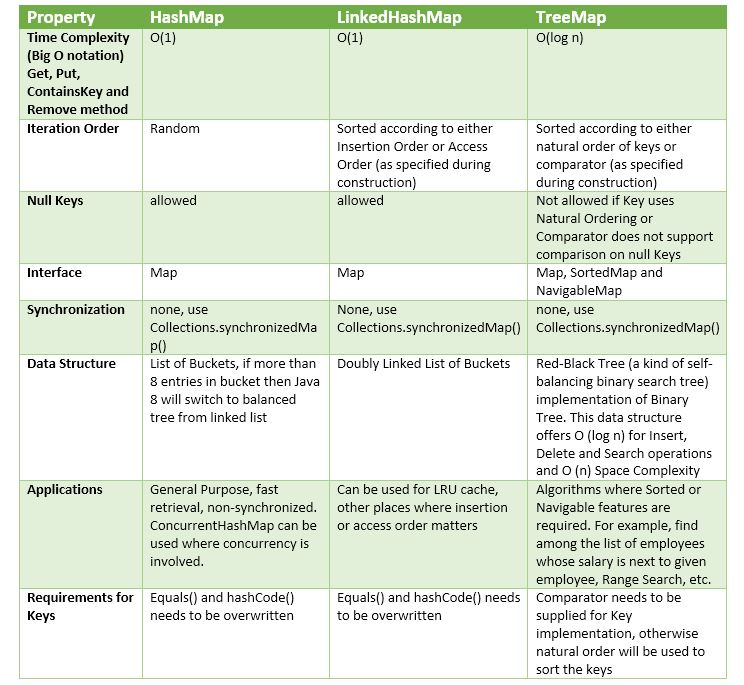


[](https://i.stack.imgur.com/clp27.jpg)

### Which Java List to use?

A list is the simplest of structures. It keeps its elements in the same order in which they are inserted and allows duplicates. There are essentially three underlying list classes:

|  |  |  |
| --- | --- | --- |
| **Class** | **Features/implementation** | **When to use** |
| ArrayList | * Allows elements to be efficiently read by index. * Adding/removing the last element is efficient. * Not synchronized in any way. | In most cases. |
| LinkedList | * First and last elements can be accessed efficiently; * Other elements *cannot* be efficiently accessed by index; * Not synchronized in any way. | Effectively, functions as a non-synchronized *queue*. In practice, rarely used: when you need a queue, you often need it to be concurrent or to provide other functionality; other implementations are often more useful. |
| CopyOnWriteArrayList | * Allows safe concurrent access; * Reads are efficient and non-blocking; * Modifications are not efficient (since a brand new copy of the list is taken each time). | Where you need concurrent access and where frequency of reads far outweights frequency of modifications. |

If you need concurrent access to a list and CopyOnWriteArrayList is not appropriate (either because the list is large or because reads don't outnumber writes), then the best you can really do is place a synchronized wrapper around an ordinary ArrayList:

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

Note that this gives you *thread-safe* access, but it's not truly *concurrent* in the sense that each access to the list will lock the entire list during the access.

Remember if you do this that you must always **synchronize on the list while *iterating* over it** (and in some cases this could be bad for concurrency). In practice, you should think carefully whether this type of list makes much sense. If a list is being continually altered by different threads, for example, what value do the individual index numbers really have?

### Which Java Map to use?

The JDK provides various Map implementations depending on:

* whether you need to maintain the *keys* in **sorted order**, in some **fixed order** or whether no particular order is required;
* whether you require efficient **concurrent access** (i.e. where multiple threads can access the map efficiently and can perform **atomic updates** on the map).

Depending on these requirements, the various Map implementations are as follows:

|  |  |  |
| --- | --- | --- |
| **Ordering of keys** | **Non-concurrent** | **Concurrent** |
| *No particular order* | HashMap | [ConcurrentHashMap](http://www.javamex.com/tutorials/synchronization_concurrency_8_hashmap.shtml) |
| *Sorted* | TreeMap | ConcurrentSkipListMap |
| *Fixed* | LinkedHashMap | — |

### Which Java Set to use?

Conceptually, a **set** serves to record *whether or not* an object belongs to a particular group. But a set is usually *implemented* as a degenerate type of map, in which each item added to the set is mapped to some special object meaning "present in the set". Therefore, especially in the non-current case, the choices in deciding which set implementation to use are largely similar to in the previous section: do you need a predictable iteration order and/or concurrent access? The available Set implementations are then as follows:

|  |  |  |
| --- | --- | --- |
| **Ordering of keys** | **Non-concurrent** | **Concurrent** |
| *No particular order* | HashSet | — |
| *Sorted* | TreeSet | ConcurrentSkipListSet |
| *Fixed* | LinkedHashSet | CopyOnWriteArraySet |

Perhaps surprisingly, Java doesn't provide a set implementation based on ConcurrentHashMap, though you could make such a subclass trivially yourself. Another option that requires no code is to use a ConcurrentSkipListSet, although you will be paying (in efficiency) for sorting that you don't really need1.

As with CopyOnWriteArrayList (on which it is actually based), the CopyOnWriteArraySet class is suited to cases where the set is relatively small and reads far outweight writes. Any modification of the set is expensive (since a brand new copy is created each time), but reads are non-blocking.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Interface | HasDuplicates? | Implementations | | | | | Historical |
| Set | no | [HashSet](http://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html) | ... | [**LinkedHashSet**](http://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashSet.html) | ... | [TreeSet](http://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html) | ... |
| List | yes | ... | [**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) | ... | [Vector](http://docs.oracle.com/javase/7/docs/api/java/util/Vector.html), [Stack](http://docs.oracle.com/javase/7/docs/api/java/util/Stack.html) |
| Map | no duplicate keys | [HashMap](http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html) | ... | [**LinkedHashMap**](http://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashMap.html) | ... | [TreeMap](http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html) | [Hashtable](http://docs.oracle.com/javase/7/docs/api/java/util/Hashtable.html), [Properties](http://docs.oracle.com/javase/7/docs/api/java/util/Properties.html) |

Principal features of non-primary implementations:

* HashMap has slightly better performance than LinkedHashMap, but its iteration order is *undefined*
* HashSet has slightly better performance than LinkedHashSet, but its iteration order is *undefined*
* TreeSet is ordered and sorted, but slower
* TreeMap is ordered and sorted, but slower
* LinkedList has fast adding to the start of the list, and fast deletion from the interior via iteration

Iteration order for above implementations:

* HashSet - *undefined*
* HashMap - *undefined*
* LinkedHashSet - insertion order
* LinkedHashMap - insertion order of keys (by default), or 'access order'
* ArrayList - insertion order
* LinkedList - insertion order
* TreeSet - ascending order, according to Comparable / Comparator
* TreeMap - ascending order of keys, according to Comparable / Comparator

# **How to choose which Java collection class to use?**

The Java Collections API provides a whole host of data structures, especially since the API was expanded in Java 5 (and again slightly in Java 6) to include concurrent collections. At first, the array of choices can be a little daunting: should I use a HashMap or a LinkedHashMap? When should I use a list or a HashSet? When should I use a TreeMap rather than a HashMap? But with a bit of guidance, the choice needn't be quite so daunting. There are also a few cases where it's difficult to decide because the choice is very arguable. And in other cases, having a clear set of rules of thumb can guide you to an appropriate decision.

## **Basic approach to choosing a collection**

The overall approach I'd suggest for choosing is as follows:

1. choose the **general type** of organisation that your data needs to have (e.g. map or list); without too much thought, this is usually fairly clear;
2. then, choose the implementation of that type that has the **minimum functionality** that you actually require (e.g. don't choose a sorted structure if you don't actually need the data to be sorted).

In general, the algorithm that underlies each collection class is designed to be a good tradeoff between efficiency and certain minimal requirements. So as long as you understand the *minimal requirements* that a given class is designed to provide, you shouldn't need to get too bogged down in the actual algorithms (though if you *are* interested in algorithms, the source code to all the collections classes is available and they make fascinating case studies, of course).

## **1. Basic collection types**

The first part of the decision is choosing what *"basic category"* of organisation or functionality your data needs to have. The broad types are as follows:

|  |  |  |
| --- | --- | --- |
| **Collection type** | **Functionality** | **Typical uses** |
| **List** | * Essentially a variable-size array; * You can usually add/remove items at any arbitrary position; * The order of the items is well defined (i.e. you can say what position a given item goes in in the list). | Most cases where you just need to store or iterate through a "bunch of things" and later iterate through them. |
| **Set** | * Things can be "there or not"— when you add items to a set, there's no notion of *how many times* the item was added, and usually no notion of ordering. | * Remembering "which items you've already processed", e.g. when doing a web crawl; * Making other *yes-no decisions* about an item, e.g. "is the item a word of English", "is the item in the database?" , "is the item in this category?" etc. |
| **Map** | * Stores an *association* or mapping between "keys" and "values" | Used in cases where you need to say "for a given X, what is the Y"? It is often useful for implementing in-memory caches or indexes. For example:   * For a given user ID, what is their cached name/User object? * For a given IP address, what is the cached country code? * For a given string, how many instances have I seen? |
| **Queue** | * Like a list, but where you **only ever access the *ends*** of the list (typically, you add to one end and remove from the other). | * Often used in **managing tasks** performed by different threads in an application (e.g. one thread receives incomming connections and puts them on a queue; other "worker" threads take connections off the queue for processing); * For **traversing hierarchical structures** such as a filing system, or in general where you need to remember **"what data to process next"**, whilst also adding to that list of data; * Related to the previous point, queues crop up in various algorithms, e.g. build the encoding tree for *Huffman compression*. |

All three classes HashMap, TreeMap and LinkedHashMap implements java.util.Map interface, and represents mapping from unique key to values.

[**HashMap**](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/HashMap.java#HashMap)

1. A HashMap contains values based on the key.
2. It contains only unique elements.
3. It may have one null key and multiple null values.
4. It maintains ***no order***.

public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable

[**LinkedHashMap**](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/LinkedHashMap.java#LinkedHashMap)

1. A LinkedHashMap contains values based on the key.
2. It contains only unique elements.
3. It may have one null key and multiple null values.
4. It is same as HashMap instead maintains ***insertion order***. *//See class deceleration below*

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

[**TreeMap**](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/TreeMap.java#TreeMap)

1. A TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
2. It contains only unique elements.
3. It cannot have null key but can have multiple null values.
4. It is same as HashMap instead maintains ***ascending order***(Sorted using the natural order of its key.).

public class TreeMap<K,V> extends AbstractMap<K,V> implements NavigableMap<K,V>, Cloneable, Serializable

[**Hashtable**](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/Hashtable.java#Hashtable)

1. A Hashtable is an array of list. Each list is known as a bucket. The position of bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.
2. It contains only unique elements.
3. It may have not have any null key or value.
4. It is ***synchronized***.
5. It is a legacy class.

public class Hashtable<K,V> extends Dictionary<K,V> implements Map<K,V>, Cloneable, Serializable

Example:

<https://github.com/dineshmetkari/java-collections.git>

<https://github.com/dineshmetkari/Collections>