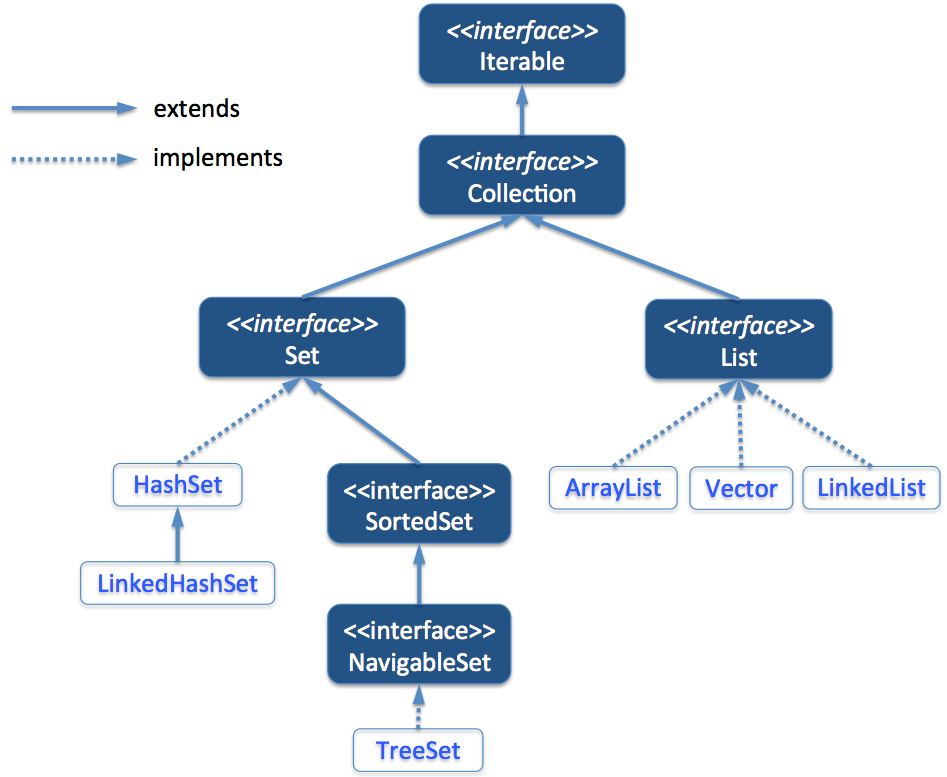
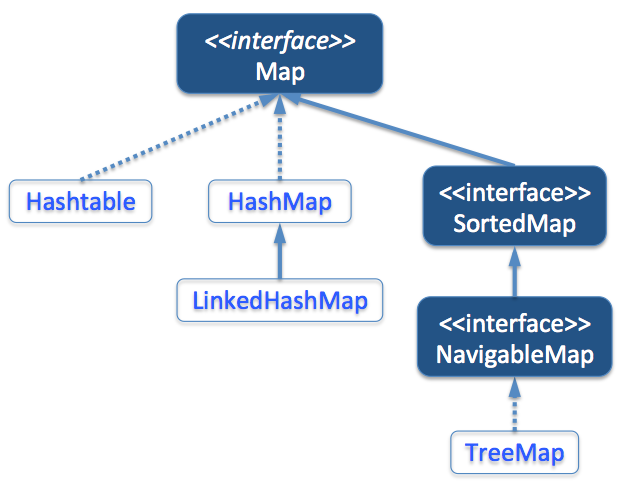
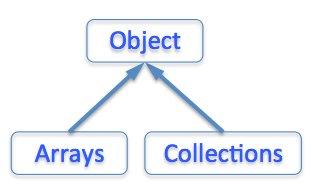
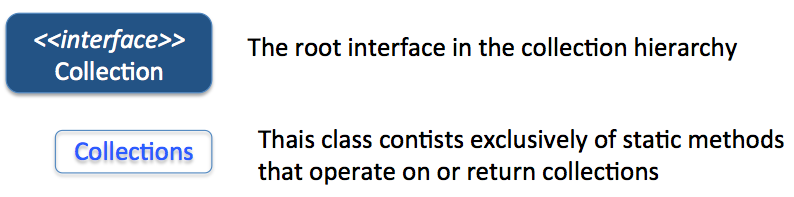
**Java Collections Definition**

Java Collections provide developers with a set of classes and interfaces to easily handle collections of objects. Collections works a bit like arrays, except their size can change dynamically, and they have more advanced functionality than arrays.

**Hierarchy of the most commonly used Java Collections**







|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Interface** | **Implementation** | **Allow duplicate elements?** | **Indexed?** | **Ordered?** | **Sorted?** | **Nulls allowed?** | **Syncronised?** | **Efficiency issues** | **Average time complexity** | **Other considerations** |
| List | ArrayList | **Yes** | **Yes** | **Yes** | **No** | **Yes** | **No** | Quick on retrieving, but slow on inserting/removing | Indexed access: O(1)  Add/Remove/Update/Search: O(n) | **ArrayList** are slower than primitive **Arrays**. So, if you need efficiency try to use **Arrays** if possible (you’d need the size in advance) |
| LinkedList | **Yes** | **Yes** | **Yes** | **No** | **Yes** | **No** | Slow on retrieving, but quick on inserting/removing | Read/Search/Update: O(n)  Add/Remove at beginning: O(1)  Add/Remove in middle/at end: O(n) |  |
| Vector | **Yes** | **Yes** | **Yes** | **No** | **Yes** | **Yes** | Similar to ArrayList, but syncronized, and then often slower. | Indexed access: O(1)  Add/Remove/Update/Search: O(n) | If you don't need a thread-safe collection, forgive Vector and use ArrayList. If you need a syncronised collection, and you know the rate at which data grow, use Vector |
| Set | HashSet | **No** | **No** | **No** | **No** | **Yes** | **No** | The fastest Set implementation, but slower than ArrayList | Read/Search/Update/Remove/Add: O(1) |  |
| LinkedHashSet | **No** | **No** | **Yes** | **No** | **Yes** | **No** | A bit slower than HashSet | Read/Search/Update/Delete: O(1)  Add at beginning: O(1)  Add in middle/at end: O(n) |  |
| TreeSet | **No** | **No** | **Yes** | **Yes** | **No** | **No** | Much slower than HashSet | Read/Search/Update/Remove/Add: O(log n) |  |
| Map | HashMap | **No** | **No** | **No** | **No** | **Yes** | **No** | The fastest Map implementation | Read/Search/Update/Remove/Add: O(1) |  |
| LinkedHashMap | **No** | **No** | **Yes** | **No** | **Yes** | **No** | A bit slower than HashMap | Read/Search/Update/Remove : O(1)  Add at beginning: O(1)  Add in middle/at end: O(n) |  |
| HashTable | **No** | **No** | **No** | **No** | **No** | **Yes** | Similar to HashMap, but syncronized, and then often slower. | Read/Search/Update/Add/Remove: O(1) |  |
| TreeMap | **No** | **No** | **Yes** | **Yes** | Only in values, not in keys | **No** | Much slower than HashMap | Read/Search/Update/Remove/Add: O(log n) |  |

**Syncronized Collections**

Vector and Hashtable are **synchronized** implementations and slow.

The remaining collections are **non-syncronized** but it is easy to create synchronized collections from them.

All you need is to create the object

Collections.synchronizedCollection()

from java.util.Collections utility class as shown below:

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

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

Set s = Collections.synchronizedSet(new TreeSet(...));

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

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

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

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

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

***These syncronised collections are unsuitable for use in highly concurrent applications. The lock used for the sycronization is an impediment to scalability and it often becomes necessary to lock a collection for a considerable time during iteration to prevent ConcurrentModificationException.***

**Concurrent collections**

In package java.util.concurrent, there are several new concurrent alternatives of standard syncronized ArrayList, Hashtable and syncronized HashMap. They can be utilized to build more scalable and high performance Java application.

* ConcurrentHashMap: It provides a concurrent alternative to Hashtable and Syncronized Map with better scalability than the syncronized version.
* CopyOnWriteArrayList and CopyOnWriteArraySet: They provide a concurrent alternative to syncronized List and Set, respectively with a better performance.
* ConcurrentSkipListMap and ConcurrentSkipListSet: Provides a concurrent alternative of syncronized HashMap.

**Rule of thumb**

* Type declarations should always be an interface (i.e. List, Set, Map). Afterwards, you can change the implementation without breaking anything. For example:

//Set<String> myCollection = new HashSet<String>();

List<String> myCollection = new ArrayList<String>();

myCollection.add("A");

myCollection.add("B");

**Highly Scalable Java**

**We should avoid using this library unless strictly necessary**

A collection of Concurrent and Highly Scalable Utilities. These are intended as direct replacements for the java.util.\* or java.util.concurrent.\* collections but with better performance when many CPUs are using the collection concurrently.

Website: <http://high-scale-lib.sourceforge.net>

You need to add the libraries to the project and

import org.cliffc.high\_scale\_lib.\*;

Classes:

* AbstractEntry
* ConcurrentAutoTable
* Counter
* NonBlockingHashMap
* NonBlockingHashMapLong
* NonBlockingHashSet
* NonBlockingHashtable
* NonBlockingIdentityHashMap
* NonBlockingSetInt
* UtilUnsafe