**Collection**

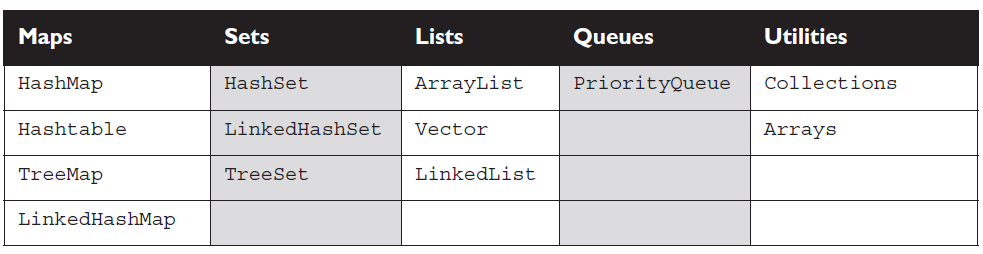
Collection is a data structure which holds only objects. (It doesn’t store primitive data type).

**So What Do You Do with a Collection?**

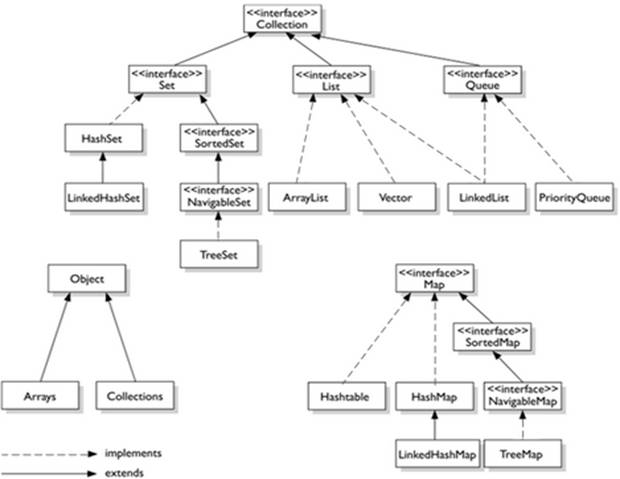
There are a few basic operations you'll normally use with collections:

* Add objects to the collection.
* Remove objects from the collection.
* Find out if an object (or group of objects) is in the collection.
* Retrieve an object from the collection (without removing it).
* Iterate through the collection, looking at each element (object) one

after another.



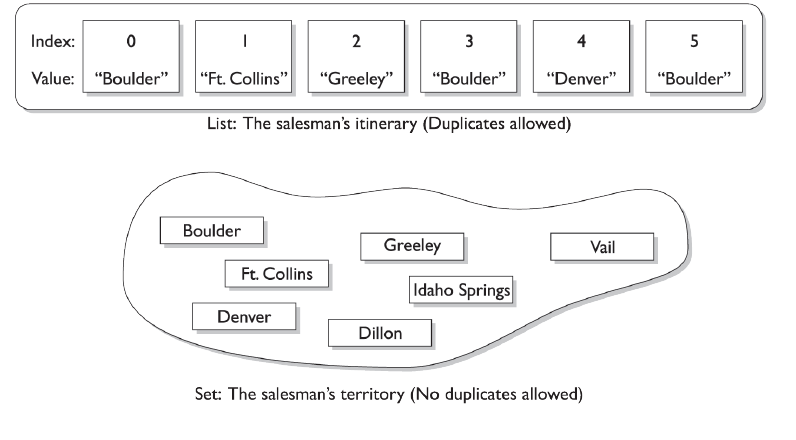
**Collection Hierarchy**

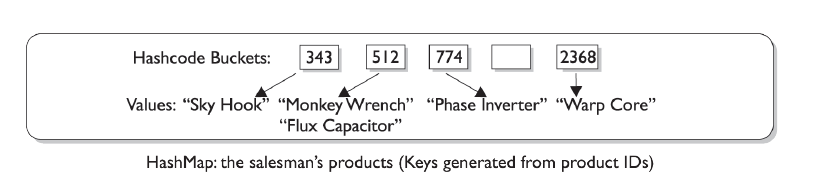


Collections come in four basic flavours:

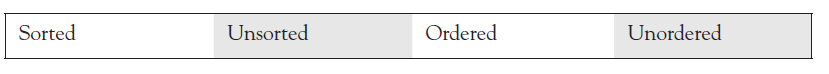
* **Lists** *Lists* of things (classes that implement List).
* **Sets** *Unique* things (classes that implement Set).
* **Maps** Things with a *unique* ID (classes that implement Map).
* **Queues** Things arranged by the order in which they are to be processed.

**Structure of a List, a Set, and a Map.**

****

****

But there are sub-flavors within those four flavors of collections:



**Ordered** When a collection is ordered, it means you can iterate through the

collection in a specific (not-random) order.

**Sorted** A *sorted* collection means that the order in the collection is determined

according to some rule or rules, known as the sort order. Most commonly, the sort order used is something called the *natural* order.

**List Interface**

A List cares about the index. The one thing that List has that non-lists don't have

is a set of methods related to the index. Those key methods include things like

get(int index), indexOf(Object o), add(int index, Object obj), and so

on. All three List implementations are ordered by index position—a position that

you determine either by setting an object at a specific index or by adding it without

specifying position, in which case the object is added to the end. The three List

implementations are described in the following sections.

**ArrayList :**Think of this as a growable array. It gives you fast iteration and fast

random access. To state the obvious: it is an ordered collection (by index), but not sorted. You might want to know that as of version 1.4, ArrayList now implements the new RandomAccess interface—a marker interface (meaning it has no methods)

that says, "this list supports fast (generally constant time) random access." Choose this over a LinkedList when you need fast iteration but aren't as likely to be doing a lot of insertion and deletion.

**Vector:** Vector is a holdover from the earliest days of Java; Vector and Hashtable were the two original collections, the rest were added with Java 2 versions 1.2 and 1.4. A Vector is basically the same as an ArrayList, but Vector methods are synchronized for thread safety. You'll normally want to use ArrayList instead of Vector because the synchronized methods add a performance hit you might not need. And if you do need thread safety, there are utility methods in class Collections that can help. Vector is the only class other than ArrayList to implement RandomAccess.

**LinkedList :** A LinkedList is ordered by index position, like ArrayList, except

that the elements are doubly-linked to one another. This linkage gives you new

methods (beyond what you get from the List interface) for adding and removing from the beginning or end, which makes it an easy choice for implementing a stack or queue. Keep in mind that a LinkedList may iterate more slowly than an ArrayList, but it's a good choice when you need fast insertion and deletion. As of Java 5, the LinkedList class has been enhanced to implement the java.util.Queue interface. As such, it now supports the common queue methods: peek(), poll(), and offer().

**Set Interface**

A Set cares about uniqueness—it doesn't allow duplicates. Your good friend the equals() method determines whether two objects are identical (in which case only one can be in the set). The three Set implementations are described in the following sections.

**HashSet:** A HashSet is an unsorted, unordered Set. It uses the hashcode

of the object being inserted, so the more efficient your hashCode() implementation the better access performance you'll get. Use this class when you want a collection with no duplicates and you don't care about order when you iterate through it.

**LinkedHashSet :** A LinkedHashSet is an ordered version of HashSet that

maintains a doubly-linked List across all elements. Use this class instead of HashSet when you care about the iteration order. When you iterate through a HashSet the order is unpredictable, while a LinkedHashSet lets you iterate through the elements in the order in which they were inserted.

**TreeSet :** The TreeSet is one of two sorted collections (the other being

TreeMap). It uses a Red-Black tree structure (but you knew that), and guarantees that the elements will be in ascending order, according to natural order. Optionally, you can construct a TreeSet with a constructor that lets you give the collection your own rules for what the order should be (rather than relying on the ordering defined by the elements' class) by using a Comparable or Comparator.

**Map Interface**

A Map cares about unique identifiers. You map a unique key (the ID) to a specific value, where both the key and the value are, of course, objects. You're probably quite familiar with Maps since many languages support data structures that use a key/value or name/value pair. The Map implementations let you do things like search for a value based on the key, ask for a collection of just the values, or ask for a collection of just the keys. Like Sets, Maps rely on the equals() method to determine whether two keys are the same or different.

**HashMap :** The HashMap gives you an unsorted, unordered Map. When you

need a Map and you don't care about the order (when you iterate through it), then HashMap is the way to go; the other maps add a little more overhead. Where the keys land in the Map is based on the key's hashcode, so, like HashSet, the more efficient your hashCode() implementation, the better access performance you'll get. HashMap allows one null key and multiple null values in a collection.

**Hashtable :** Like Vector, Hashtable has existed from prehistoric Java times.

For fun, don't forget to note the naming inconsistency: HashMap vs. Hashtable.

Where's the capitalization of *t*? Oh well, you won't be expected to spell it. Anyway,just as Vector is a synchronized counterpart to the sleeker, more modern ArrayList,Hashtable is the synchronized counterpart to HashMap. Remember that you don't synchronize a class, so when we say that Vector and Hashtable are synchronized, we just mean that the key methods of the class are synchronized. Another difference,though, is that while HashMap lets you have null values as well as one null key, a Hashtable doesn't let you have anything that's null.

**LinkedHashMap :** Like its Set counterpart, LinkedHashSet, the LinkedHash-

Map collection maintains insertion order (or, optionally, access order). Although it will be somewhat slower than HashMap for adding and removing elements, you can expect faster iteration with a LinkedHashMap.

**TreeMap :** You can probably guess by now that a TreeMap is a sorted Map. And you already know that by default, this means "sorted by the natural order of the elements."

Like TreeSet, TreeMap lets you define a custom sort order (via a Comparable

or Comparator) when you construct a TreeMap, that specifies how the elements should be compared to one another when they're being ordered.

**Queue Interface**

A Queue is designed to hold a list of "to-dos," or things to be processed in some way. Although other orders are possible, queues are typically thought of as FIFO (first-in, first-out). Queues support all of the standard Collection methods and they also add methods to add and subtract elements and review queue elements.

**PriorityQueue :**This class is new with Java 5. Since the LinkedList class has

been enhanced to implement the Queue interface, basic queues can be handled with a LinkedList. The purpose of a PriorityQueue is to create a "priority-in, priority out" queue as opposed to a typical FIFO queue. A PriorityQueue's elements are ordered either by natural ordering (in which case the elements that are sorted first will be accessed first) or according to a Comparator. In either case, the elements' ordering represents their relative priority.

**Summary : Collection Interface Concrete Implementation Classes**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Collection** | **Ordering** | **Random Access** | **Key-Value** | **Duplicate Elements** | **Null Element** | **Thread Safety** |
| ArrayList | Yes | Yes | No | Yes | Yes | No |
| LinkedList | Yes | No | No | Yes | Yes | No |
| HashSet | No | No | No | No | Yes | No |
| TreeSet | Yes | No | No | No | No | No |
| HashMap | No | Yes | Yes | No | Yes | No |
| TreeMap | Yes | Yes | Yes | No | No | No |
| Vector | Yes | Yes | No | Yes | Yes | Yes |
| Hashtable | No | Yes | Yes | No | No | Yes |
| Properties | No | Yes | Yes | No | No | Yes |
| Stack | Yes | No | No | Yes | Yes | Yes |
| CopyOnWriteArrayList | Yes | Yes | No | Yes | Yes | Yes |
| ConcurrentHashMap | No | Yes | Yes | No | No | Yes |
| CopyOnWriteArraySet | No | No | No | No | Yes | Yes |

**hashcode() and equals() method in java**

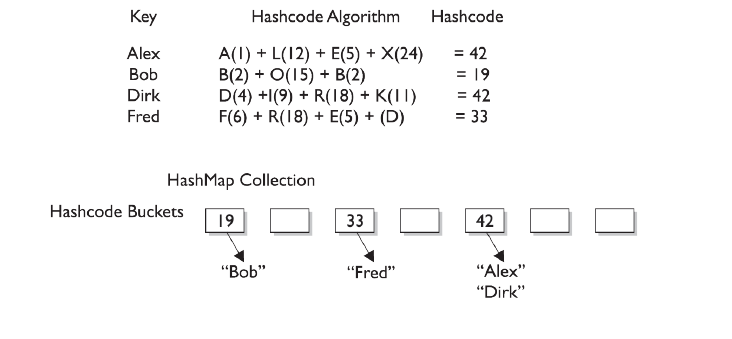
These methods can be found in the Object class and hence available to all java classes.Using these two methods, an object can be stored or retrieved from a Hashtable, HashMap or HashSet.

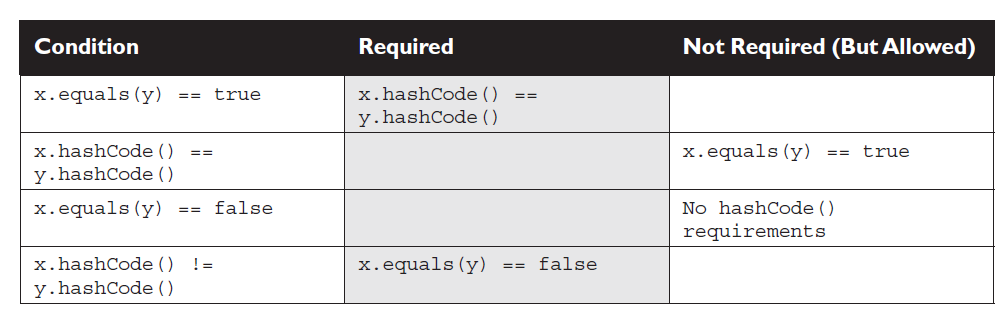
**hashcode():**You might know if you put entry in HashMap, first hashcode is calculated and this hashcode used to find bucket(index) where this entry will get stored in hashMap. What if you don't override hashcode method, it will  return integer representation of memory address.

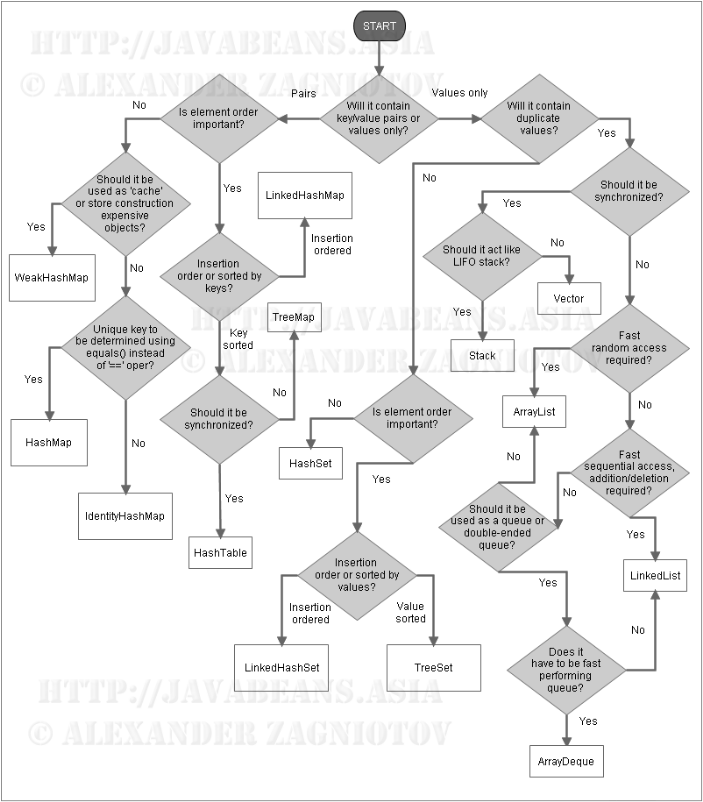
**equals():**You have to override equals method, when you want to define equality between two object. If you don't override this method, it will check for reference equality(==) i.e. if two reference refers to same object or not

**If you are overriding equals method then you should override hashcode() also.**

**Example:**

****

****

****