

same as the Map interface, plus the keys in the Map are sorted.

Set or List or Queue

There is no direct implementation for the `java.util.Collection` interface. The `Collection` interface has three sub interfaces. Those are:

`java.util.Set<E>`

contains unique elements, so duplicates not allowed. It is similar to a mathematical Set.

`java.util.List<E>`

elements are put in the list in a certain order, and can be accessed by an index. Duplicates are allowed, the same element can be added to a list.

`java.util.SortedSet<E>`

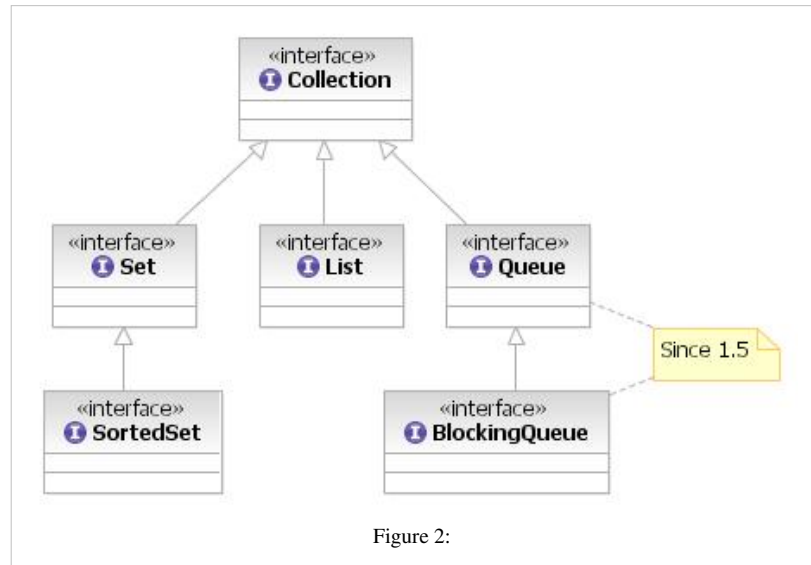
same as the Set interface; plus the elements in the SortedSet are sorted

`java.util.Queue<E>`

queues provide additional insertion, extraction, and inspection operations. There are FIFO (first in, first out) and LIFO (last in, first out) queues.

`java.util.BlockingQueue<E>`

waits for the queue to become non-empty when retrieving an element, and waits for space to become available in the queue when storing an element. Best used for producer-consumer queues.



Set

The basic implementation of the Set interface is the HashSet.

`java.util.TreeSet<E>`

Elements are sorted, not synchronized. **null** not allowed

`java.util.HashSet<E>`

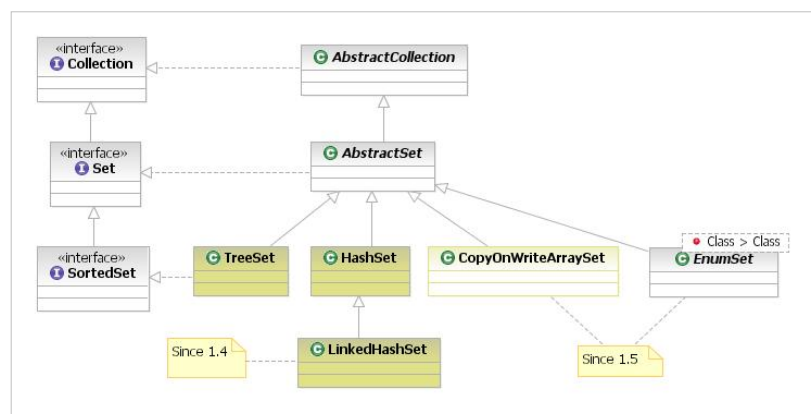
Not synchronized. Allows the **null** elements

`java.util.CopyOnWriteArraySet<E>`

Thread safe, a fresh copy is created during modification operation. Add, update, delete are expensive.

`java.util.EnumSet<E extends Enum<E>>`

All of the elements in an enum set must come from a single enum type that is specified, explicitly or implicitly, when the set is created. Enum sets are represented internally as bit vectors.



java.util.LinkedHashSet<E>

Same as HashSet, plus defines the iteration ordering, which is the order in which elements were inserted into the set.

Set cannot have duplicates in it. You may wonder how duplicates are detected when we are adding an object to the Set. We have to see if that object exists in the Set or not. It is not enough to check the object references, the objects' values have to be checked as well.

To do that, fortunately, each java object has the boolean `equal(Object obj)` method available inherited from `Object`. You need to override it. That method will be called by the Set implementation to compare the two objects to see if they are equal or not.

There is a problem, though. What if I put two different type of objects to the Set. I put an Apple and an Orange. They can not be compared. Calling the `equal()` method would cause a `ClassCastException`. There are two solutions to this:

- **Solution one** : Override the `int hashCode()` method and return the same values for the same type of objects and return different values for different type of objects. The `equal()` method is used to compare objects only with the same value of `hashCode`. So before an object is added, the Set implementation needs to:
 - find all the objects in the Set that has the same `hashCode` as the candidate object `hashCode`
 - and for those, call the `equal()` methods passing in the candidate object
 - if any of them returns true, the object is not added to the Set.
- **Solution two** : Create a super class for the Apple and Orange, let's call it Fruit class. Put Fruits in the Set. You need to do the following:
 - Do not override the `equals()` and `hashCode()` methods in the Apple and Orange classes
 - Create `appleEquals()` method in the Apple class, and create `orangeEquals()` method in the Orange class
 - Override the `hashCode()` method in the Fruit class and return the same value, so the `equals()` is called by the Set implementation
 - Override the `equals()` method in the Fruit class for something like this.

```
public boolean equals( Object obj )
{
    boolean ret = false;
    if ( this instanceof Apple &&
        obj instanceof Apple )
    {
        ret = this.appleEquals(obj);
    }
    else if ( this instanceof Orange &&
             obj instanceof Orange )
    {
        ret = this.orangeEquals(obj);
    }
    else
    {
        // --- Can not compare Orange to Apple ---
        ret = false;
    }
    return ret;
}
```

Note:

- only the objects that have the same hashCode will be compared.
- you are responsible to override the equal() and hashCode() methods. The default implementations in Object won't work.
- Only override the hashCode() method if you want to eliminate value duplicates.
- Do not override the hashCode() method if you know that the values of your objects are different, or if you only want to prevent adding the exactly same object.
- Beware that the hashCode() may be used in other collection implementations, like in a Hashtable to find an object fast. Overriding the default hashCode() method may affect performance there.
- the default hashCodes are unique for each object created, so if you decide not to override the hashCode() method, there is no point overriding the equal() method, as it won't be called.

SortedSet

The SortedSet interface extends the Set Interface. All elements in the SortedSet must implement the Comparable Interface, further more all elements must be mutually comparable.

Note that the ordering maintained by a sorted set must be consistent with equals if the sorted set is to correctly implement the Set interface. This is so because the Set interface is defined in terms of the equals operation, but a sorted set 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 sorted set, equal.

The SortedSet interface has additional methods due to the sorted nature of the 'Set'. Those are:

- E first(); -- returns the first element
- E last(); -- returns the last element
- SortedSet headSet(E toElement); -- returns from the first, to the exclusive toElement
- SortedSet tailSet(E fromElement); -- returns from the inclusive fromElement to the end
- SortedSet subSet(E fromElement, E toElement); -- returns elements range from fromElement, inclusive, to toElement, exclusive. (If fromElement and toElement are equal, the returned sorted set is empty.)

List

The List has the following implementations:

java.util.Vector<E>

Synchronized, use in multiple thread access, otherwise use ArrayList

java.util.Stack<E>

It extends class Vector with five operations that allow a vector to be treated as a stack. It represents a last-in-first-out (LIFO) stack of objects.

java.util.ArrayList<E>

Non-synchronized, use in single thread environment, otherwise use Vector

java.util.LinkedList<E>

Non-synchronized, update operation is faster than other lists, easy to use for stacks, queues, double-ended queues.

