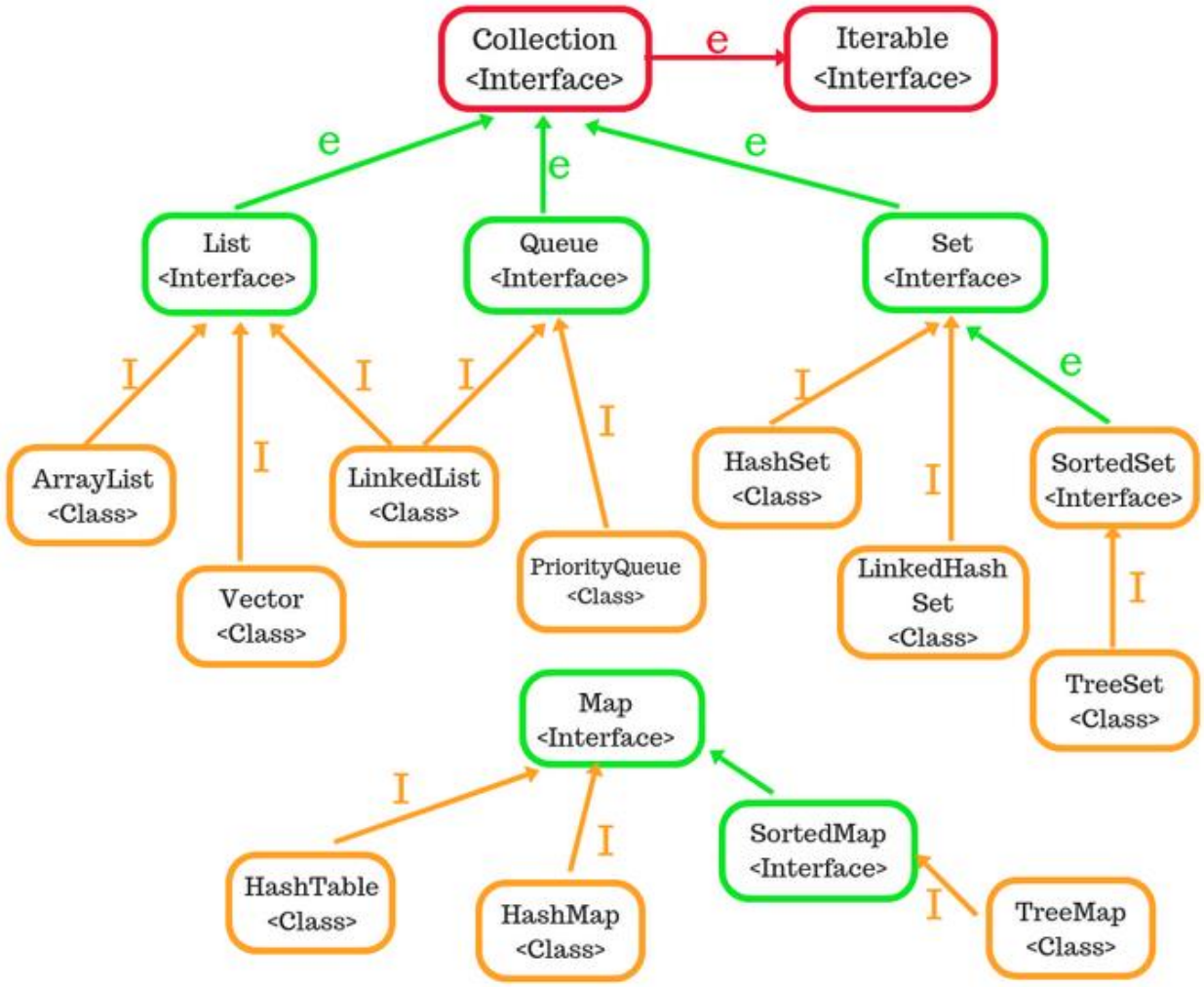
- **TreeSet**
  Keeps the elements sorted and prevents duplicates.

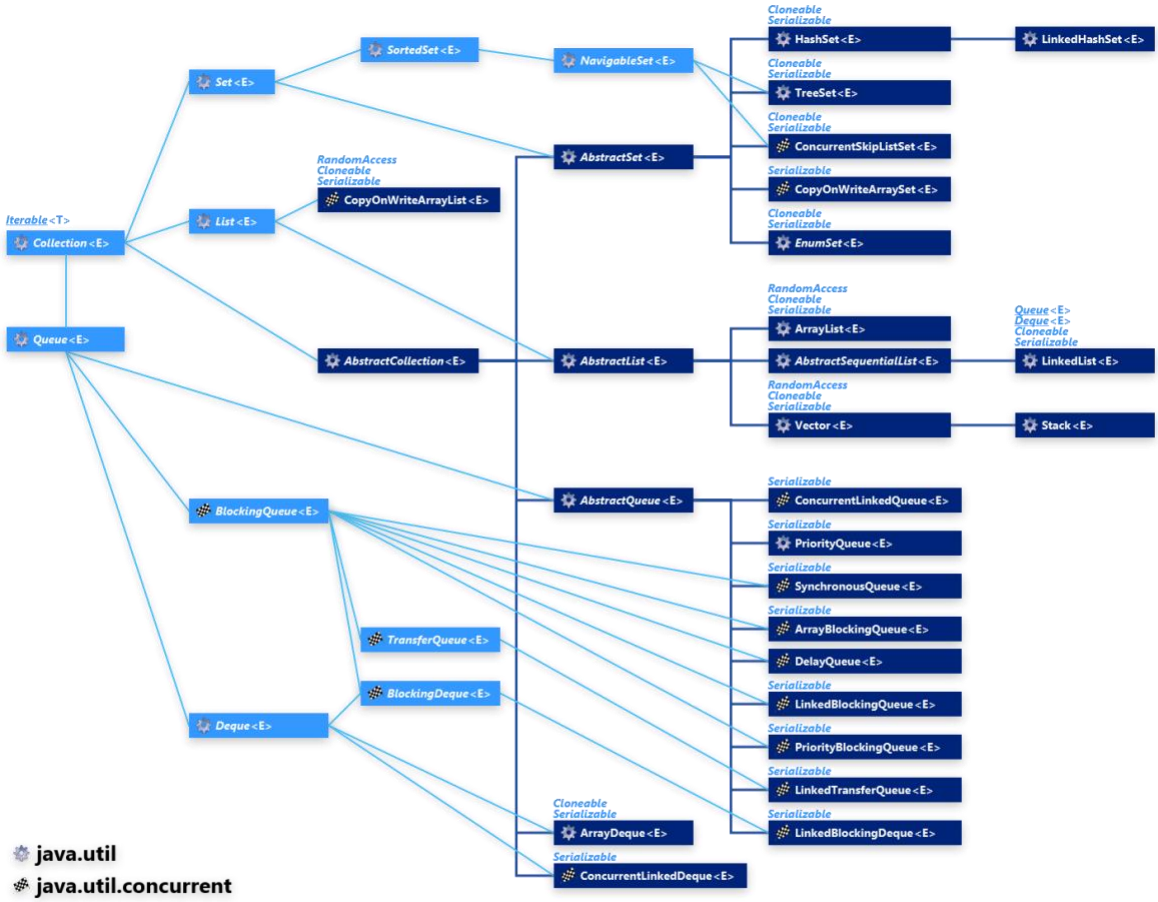
- **HashMap**
  Let's you store and access elements as name/value pairs.

- **LinkedList**
  Designed to give better performance when you insert or delete elements from the middle of the collection. (In practice, an ArrayList is still usually what you want.)

- **HashSet**
  Prevents duplicates in the collection, and given an element, can find that element in the collection quickly.

- **LinkedHashMap**
  Like a regular HashMap, except it can remember the order in which elements (name/value pairs) were inserted, or it can be configured to remember the order in which elements were last accessed.


Collection
<Interface>
e
Iterable
<Interface>
e
e
e
List
<Interface>
Queue
<Interface>
Set
<Interface>
I
I
I
I
I
I
e
ArrayList
<Class>
Vector
<Class>
LinkedList
<Class>
PriorityQueue
<Class>
HashSet
<Class>
LinkedHash
Set
<Class>
SortedSet
<Interface>
TreeSet
<Class>
Map
<Interface>
I
I
I
HashTable
<Class>
HashMap
<Class>
SortedMap
<Interface>
I
TreeMap
<Class>


Iterable <T>
Collection <E>
Set <E>
SortedSet <E>
NavigableSet <E>
HashSet <E>
LinkedHashSet <E>
TreeSet <E>
ConcurrentSkipListSet <E>
CopyOnWriteArraySet <E>
EnumSet <E>
AbstractSet <E>
List <E>
RandomAccess
Cloneable
Serializable
CopyOnWriteArrayList <E>
Queue <E>
AbstractCollection <E>
AbstractList <E>
ArrayList <E>
AbstractSequentialList <E>
LinkedList <E>
Queue <E>
Deque <E>
Vector <E>
Stack <E>
AbstractQueue <E>
BlockingQueue <E>
ConcurrentLinkedQueue <E>
PriorityQueue <E>
SynchronousQueue <E>
ArrayBlockingQueue <E>
DelayQueue <E>
LinkedBlockingQueue <E>
PriorityBlockingQueue <E>
LinkedTransferQueue <E>
LinkedBlockingDeque <E>
TransferQueue <E>
BlockingDeque <E>
Deque <E>
ArrayDeque <E>
ConcurrentLinkedDeque <E>
java.util
java.util.concurrent
The Java™ Tutorials: The Collection Interface
The Java™ Tutorials: Lesson: Implementations
www.falkhausen.de

We can pass the ArrayList<Song> to the sort() method only if the Song class implements Comparable, since that's the way the sort() method was declared. A quick check of the API docs shows the Comparable interface is really simple, with only one method to implement:

java.lang.Comparable

```
public interface Comparable<T> {
   int compareTo(T o);
}
```

And the method documentation for compareTo() says

Returns:
a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Using a custom Comparator

An element in a list can compare *itself* to another of its own type in only one way, using its compareTo() method. But a Comparator is external to the element type you're comparing—it's a separate class. So you can make as many of these as you like! Want to compare songs by artist? Make an ArtistComparator. Sort by beats per minute? Make a BPMComparator.

Then all you need to do is call the overloaded sort() method that takes the List and the Comparator that will help the sort() method put things in order.

The sort() method that takes a Comparator will use the Comparator instead of the element's own compareTo() method, when it puts the elements in order. In other words, if your sort() method gets a Comparator, it won't even *call* the compareTo() method of the elements in the list. The sort() method will instead invoke the **compare**() method on the Comparator.

So, the rules are:

java.util.Comparator

```
public interface Comparator<T> {
   int compare(T o1, T o2);
}
```

If you pass a Comparator to the sort() method, the sort order is determined by the Comparator rather than the element's own compareTo() method.

- **Invoking the one-argument sort(List o) method means the list element's compareTo() method determines the order. So the elements in the list MUST implement the Comparable interface.**

- **Invoking sort(List o, Comparator c) means the list element's compareTo() method will NOT be called, and the Comparator's compare() method will be used instead. That means the elements in the list do NOT need to implement the Comparable interface.**

Q: So does this mean that if you have a class that doesn't implement Comparable, and you don't have the source code, you could still put the things in order by creating a Comparator?

A: That's right. The other option (if it's possible) would be to subclass the element and make the subclass implement Comparable.

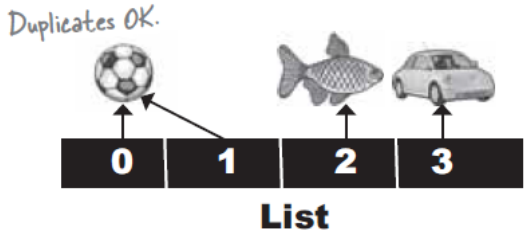
Q: But why doesn't *every* class implement Comparable?

A: Do you really believe that *everything* can be ordered? If you have element types that just don't lend themselves to any kind of natural ordering, then you'd be misleading other programmers if you implement Comparable. And you aren't taking a huge risk by not implementing Comparable, since a programmer can compare anything in any way that he chooses using his own custom Comparator.

- **LIST - when *sequence* matters**

  Collections that know about ***index position.***

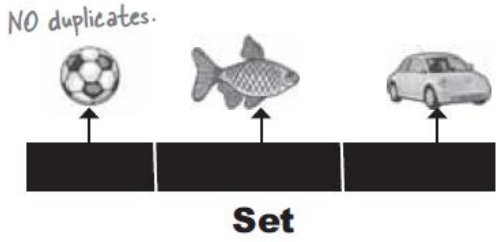
  Lists know where something is in the list. You can have more than one element referencing the same object.



➤ SET - when *uniqueness* matters

Collections that ***do not allow duplicates.***

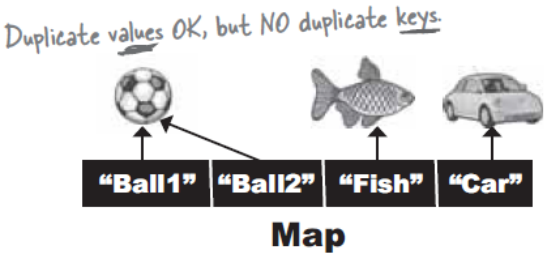
Sets know whether something is already in the collection. You can never have more than one element referencing the same object (or more than one element referencing two objects that are considered equal—we'll look at what object equality means in a moment).



➤ MAP - when *finding something by key* matters

Collections that use ***key-value pairs.***

Maps know the value associated with a given key. You can have two keys that reference the same value, but you cannot have duplicate keys. Although keys are typically String names (so that you can make name/value property lists, for example), a key can be any object.

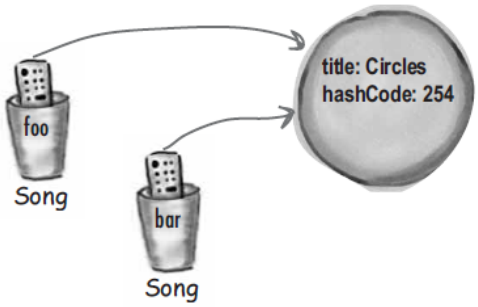


➤ Reference equality

Two references, one object on the heap.

Two references that refer to the same object on the heap are equal. Period. If you call the **hashCode()** method on both references, you'll get the same result. If you don't override the hashCode() method, the default behavior (remember, you inherited this from class Object) is that each object will get a unique number (most versions of Java assign a hashcode based on the object's memory address on the heap, so no two objects will have the same hashcode).

If you want to know if two *references* are really referring to the same object, use the == operator, which (remember) compares the bits in the variables. If both references point to the same object, the bits will be identical.



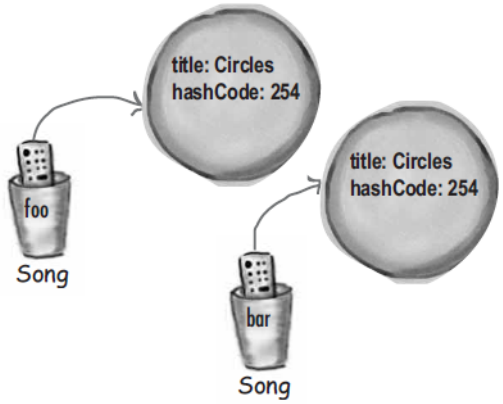

```
if (foo == bar) {
   // both references are referring
   // to the same object on the heap
}
```

➤ Object equality

Two references, two objects on the heap, but the objects are considered *meaningfully equivalent.*

If you want to treat two different Song objects as equal (for example if you decided that two Songs are the same if they have matching *title* variables), you must override *both* the **hashCode()** and **equals()** methods inherited from class Object.

As we said above, if you *don't* override hashCode(), the default behavior (from Object) is to give each object a unique hashcode value. So you must override hashCode() to be sure that two equivalent objects return the same hashcode. But you must also override equals() so that if you call it on *either* object, passing in the other object, always returns ***true***.




```
if (foo.equals(bar) && foo.hashCode() == bar.hashCode()) {
   // both references are referring to either a
   // a single object, or to two objects that are equal
}
```

How a HashSet checks for duplicates: hashCode() and equals()

When you put an object into a Hashset, it uses the object's hashcode value to determine where to put the object in the Set. But it also compares the object's hashcode to the hashcode of all the other objects in the HashSet, and if there's no matching hashcode, the HashSet assumes that this new object is not a duplicate.

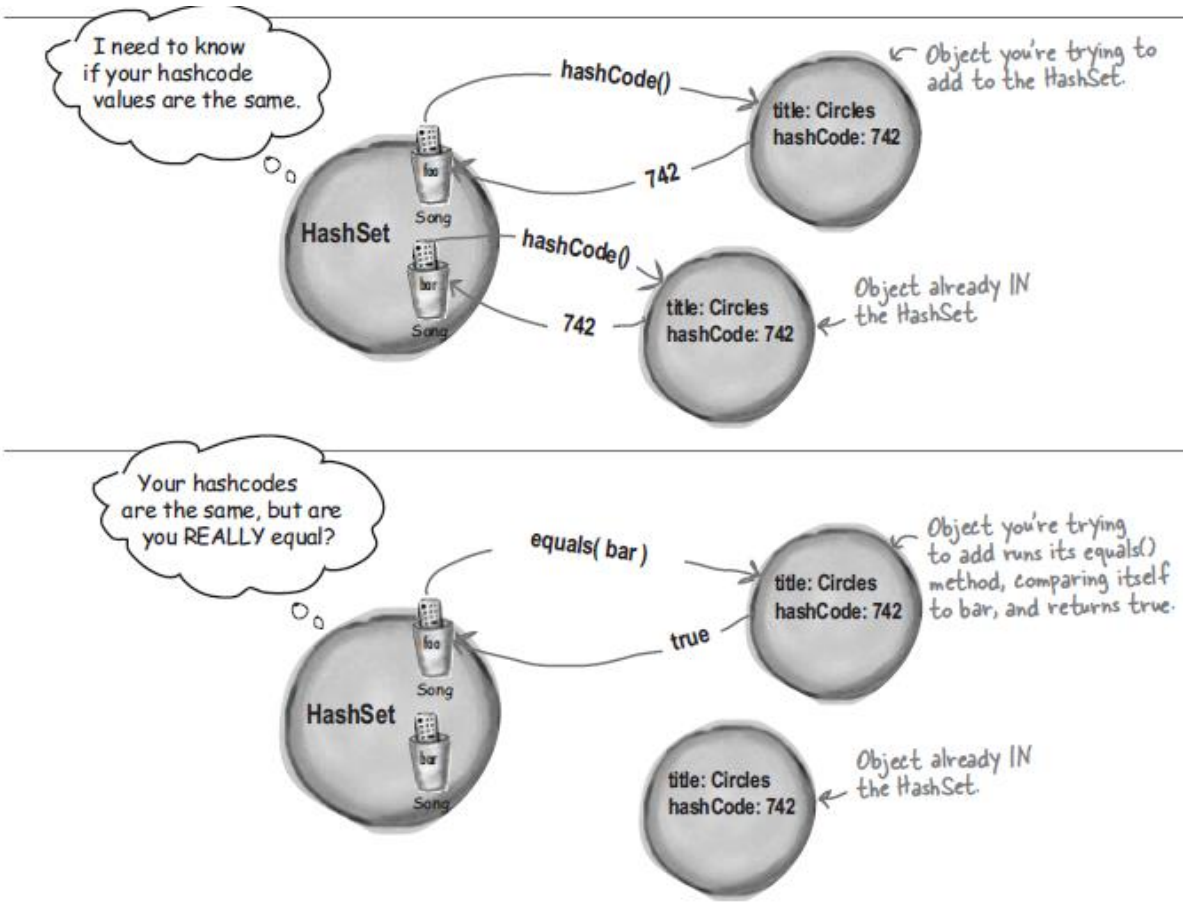
In other words, if the hashcodes are different, the HashSet assumes there's no way the objects can be equal!

So you must override hashCode() to make sure the objects have the same value.

But two objects with the same hashCode() might *not* be equal (more on this on the next page), so if the HashSet finds a matching hashcode for two objects—one you're inserting and one already in the set—the HashSet will then call one of the object's equals() methods to see if these hashcode-matched objects really *are* equal.

And if they're equal, the HashSet knows that the object you're attempting to add is a duplicate of something in the Set, so the add doesn't happen.

You don't get an exception, but the HashSet's add() method returns a boolean to tell you (if you care) whether the new object was added. So if the add() method returns *false*, you know the new object was a duplicate of something already in the set.


I need to know if your hashcode values are the same.
HashSet
foo
Song
bar
Song
hashCode()
742
title: Circles
hashCode: 742
Object you're trying to add to the HashSet.
hashCode()
742
title: Circles
hashCode: 742
Object already IN the HashSet.

Your hashcodes are the same, but are you REALLY equal?
HashSet
foo
Song
bar
Song
equals(bar)
true
title: Circles
hashCode: 742
Object you're trying to add runs its equals() method, comparing itself to bar, and returns true.
title: Circles
hashCode: 742
Object already IN the HashSet.

Java Object Law For HashCode() and equals()

The API docs for class Object state the rules you MUST follow:

- **If two objects are equal, they MUST have matching hashcodes.**
- **If two objects are equal, calling equals() on either object MUST return true. In other words, if (a.equals(b)) then (b.equals(a)).**
- **If two objects have the same hashcode value, they are NOT required to be equal. But if they're equal, they MUST have the same hashcode value.**
- **So, if you override equals(), you MUST override hashCode().**
- **The default behavior of hashCode() is to generate a unique integer for each object on the heap. So if you don't override hashCode() in a class, no two objects of that type can EVER be considered equal.**

- **The default behavior of equals() is to do an == comparison. In other words, to test whether the two references refer to a single object on the heap. So if you don't override equals() in a class, no two objects can EVER be considered equal since references to two different objects will always contain a different bit pattern.**

a.equals(b) must also mean that _a.hashCode() == b.hashCode()_

But _a.hashCode() == b.hashCode()_ does NOT have to mean _a.equals(b)_

TreeSet elements MUST be comparable

TreeSet can't read the programmer's mind to figure out how the object's should be sorted. You have to tell the TreeSet *how*.

To use a TreeSet, one of these things must be true:

- **The elements in the list must be of a type that implements _Comparable_**

```
class Book implements Comparable {
   String title;
   public Book(String t) {
     title = t;
   }
   public int compareTo(Object b) {
     Book book = (Book) b;
     return (title.compareTo(book.title));
   }
}
```

OR

You use the TreeSet's overloaded constructor that takes a *Comparator*

TreeSet works a lot like the sort() method—you have a choice of using the element's compareTo() method, assuming the element type implemented the Comparable interface, OR you can use a custom Comparator that knows how to sort the elements in the set. To use a custom Comparator, you call the TreeSet constructor that takes a Comparator.

```
public class BookCompare implements Comparator<Book>{
   public int compare(Book one, Book two) {
      return (one.title.compareTo(two.title));
   }
}

class Test {
     public void go() {
     Book b1 = new Book("How Cats Work");
     Book b2 = new Book("Remix your Body");
     Book b3 = new Book("Finding Emo");
     BookCompare bCompare = new BookCompare();
     TreeSet<Book> tree = new TreeSet<Book>(bCompare);
     tree.add(new Book("How Cats Work");
     tree.add(new Book("Finding Emo");
     tree.add(new Book("Remix your Body");
     System.out.println(tree);
   }
}
```