

# Java Collections Framework

Marcus Biel, Software Craftsman



# Collection(s)

**Unfortunately,  
there are several overloaded uses of the word “collection”.  
Let me clarify the various meanings up-front.**



# Collection(s)

1. a compilation or group of things

**The different use cases are:  
A collection without any IT relevance,  
as a compilation or group of things**



# Collection(s)

1. a compilation or group of things
2. Java Collections Framework

**Second: The “Java Collections Framework” –  
a library of different interfaces and classes**



# Collection(s)

1. a compilation or group of things
2. Java Collections Framework
3. a data structure

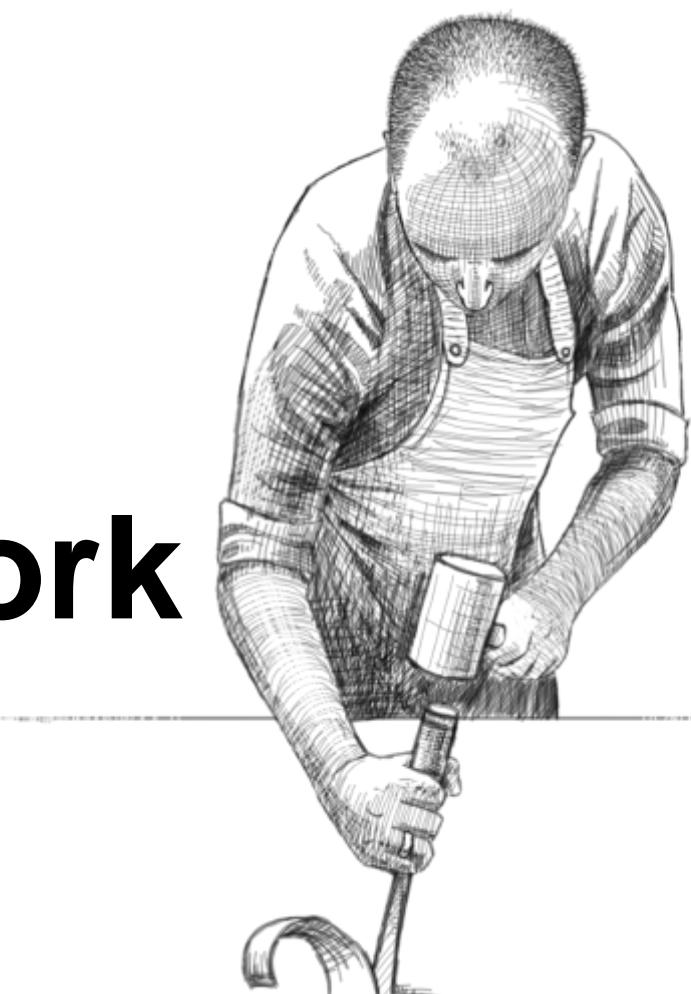
**Third: A collection as a data structure –**  
**Think of a box or container,**  
**that can hold a group of objects like an array for example.**



# Collection(s)

1. a compilation or group of things
2. Java Collections Framework
3. a data structure
4. `java.util.Collection` interface

**Fourth: The `java.util.Collection` interface –  
one of the two main interfaces of the Java Collections Framework**



# Collection(s)

1. a compilation or group of things
2. Java Collections Framework
3. a data structure
4. `java.util.Collection` interface
5. `java.util.Collections`

**And fifth: `java.util.Collections` –  
a utility class that will help you to modify or  
operate on Java collections**



# What is the Java Collections Framework?

**So what is the Java Collections framework,  
from a high level perspective?**



# What is the Java Collections Framework?

A toolbox of generic interfaces and classes

**First of all, it is more like a library,  
a toolbox of generic interfaces and classes.  
This toolbox contains:**



# What is the Java Collections Framework?

A toolbox of generic interfaces and classes

- collection interfaces and classes

**various collection interfaces and classes that serve  
as a more powerful,  
object oriented alternative to arrays.**



# What is the Java Collections Framework?

A toolbox of generic interfaces and classes

- collection interfaces and classes
- collection related utility interfaces classes

**Collection related utility interfaces and classes  
that assist you in using the collections.  
I am going to describe both parts in detail now.**



# Collections interface and class hierarchy

<<interface>>  
Collection

<<interface>>  
Map

**On the next slides you will see the interface and  
class hierarchy for collections.**

**Unlike arrays, all collections can dynamically grow or shrink in size.**



# Collections interface and class hierarchy

<<interface>>  
Collection

<<interface>>  
Map

**As said before a collection can hold a group of objects.  
A Map can store pairs of objects that have some kind of relation  
which ties them together, named key and value.**



# Collections interface and class hierarchy

<<interface>>  
Collection

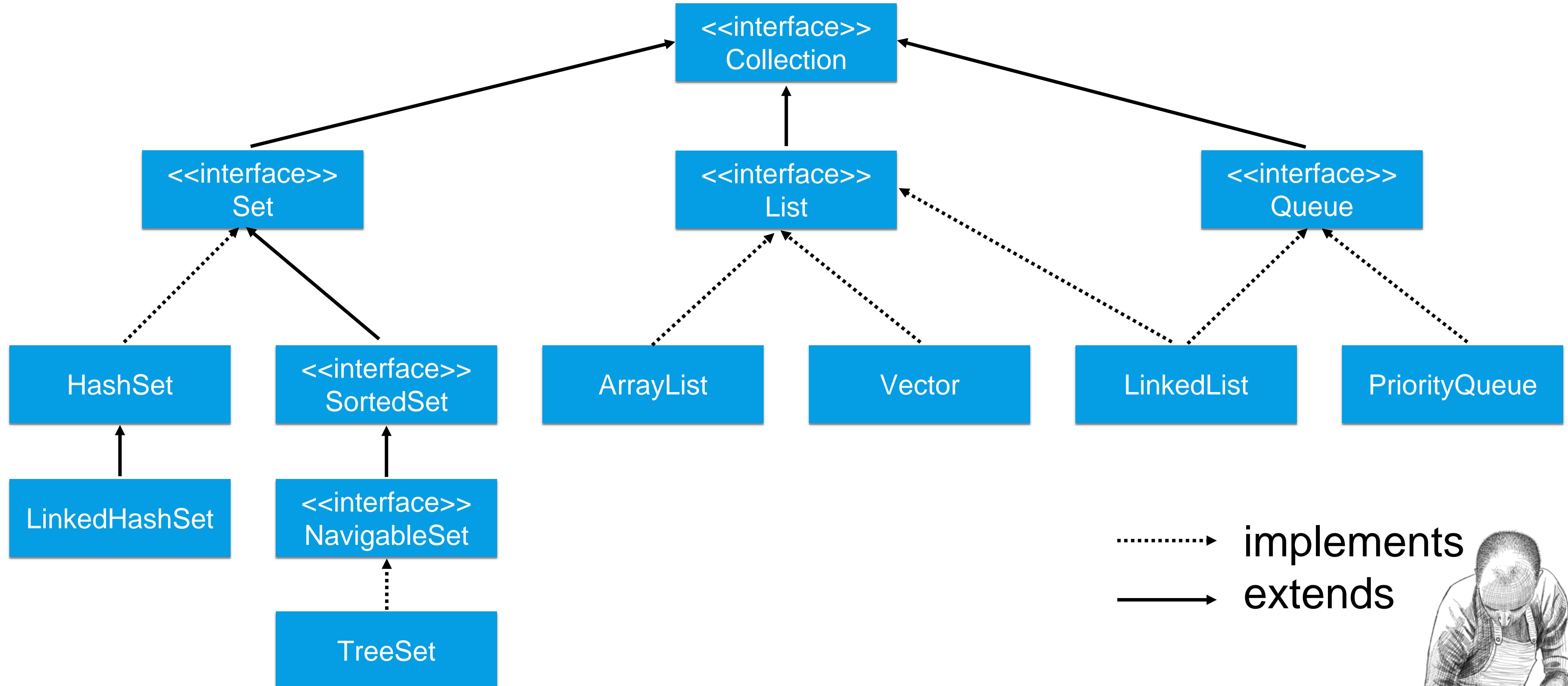
<<interface>>  
Map

A value does not have a specific position in this map,  
but can be retrieved with the key it is related to.

Relax if you don't get it now, we will look at it in more detail later on.



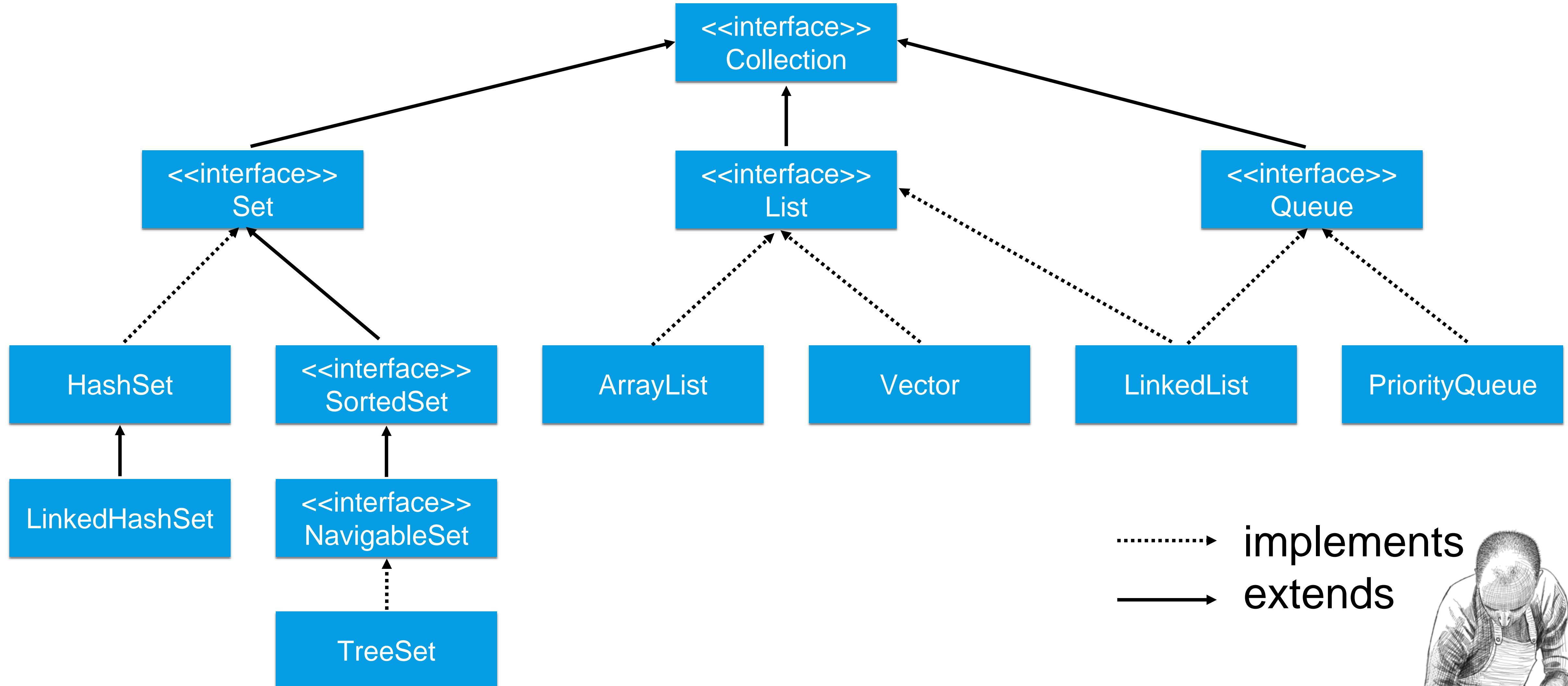
# Collection Interface



So here you see the hierarchy of classes and interfaces extending or implementing the collection interface.



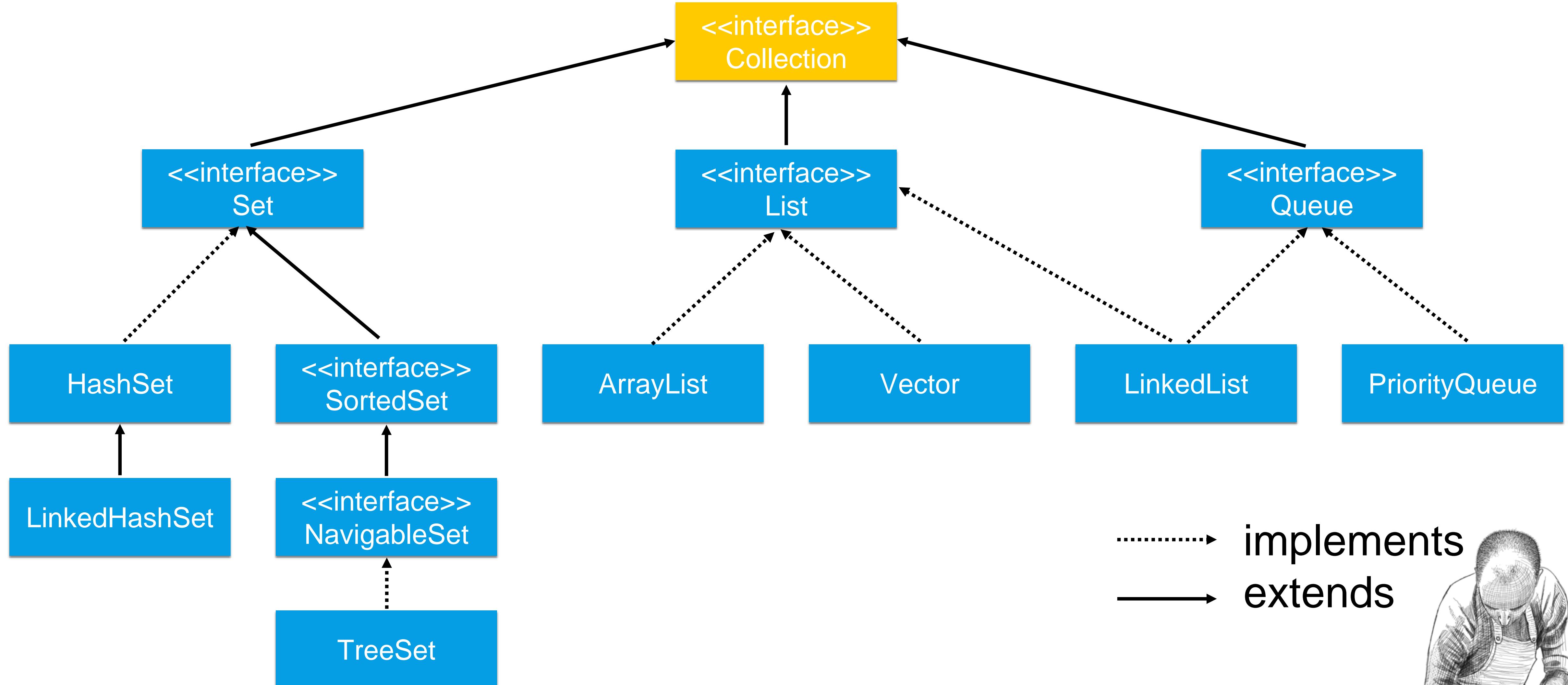
# Collection Interface



Just try to remember some of the names listed here.  
This is just an overview so far.



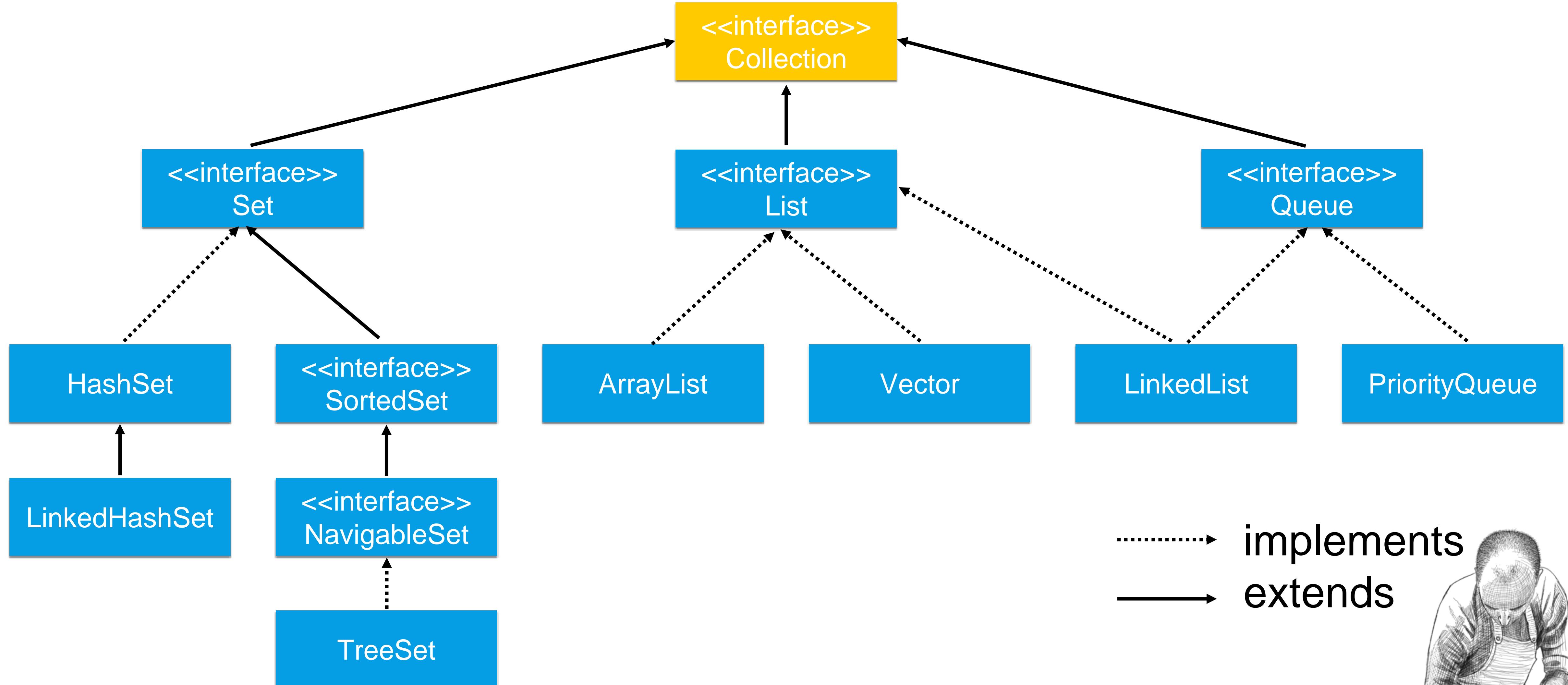
# Collection Interface



As you can see, the collection interface sits on top of a number of sub interfaces and implementing classes.



# Collection Interface

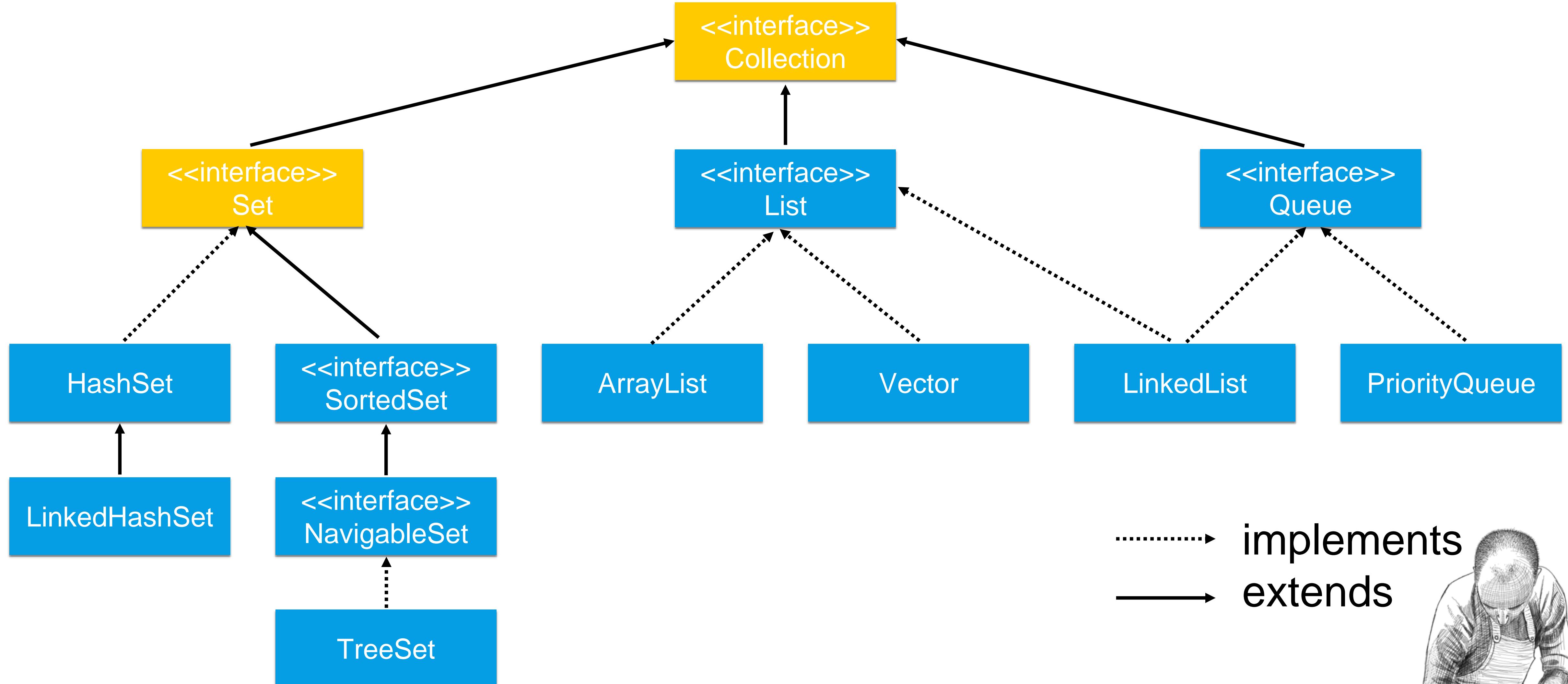


A collection can hold a group of objects.

The Collection interface is extended by the interfaces Set, List and Queue.



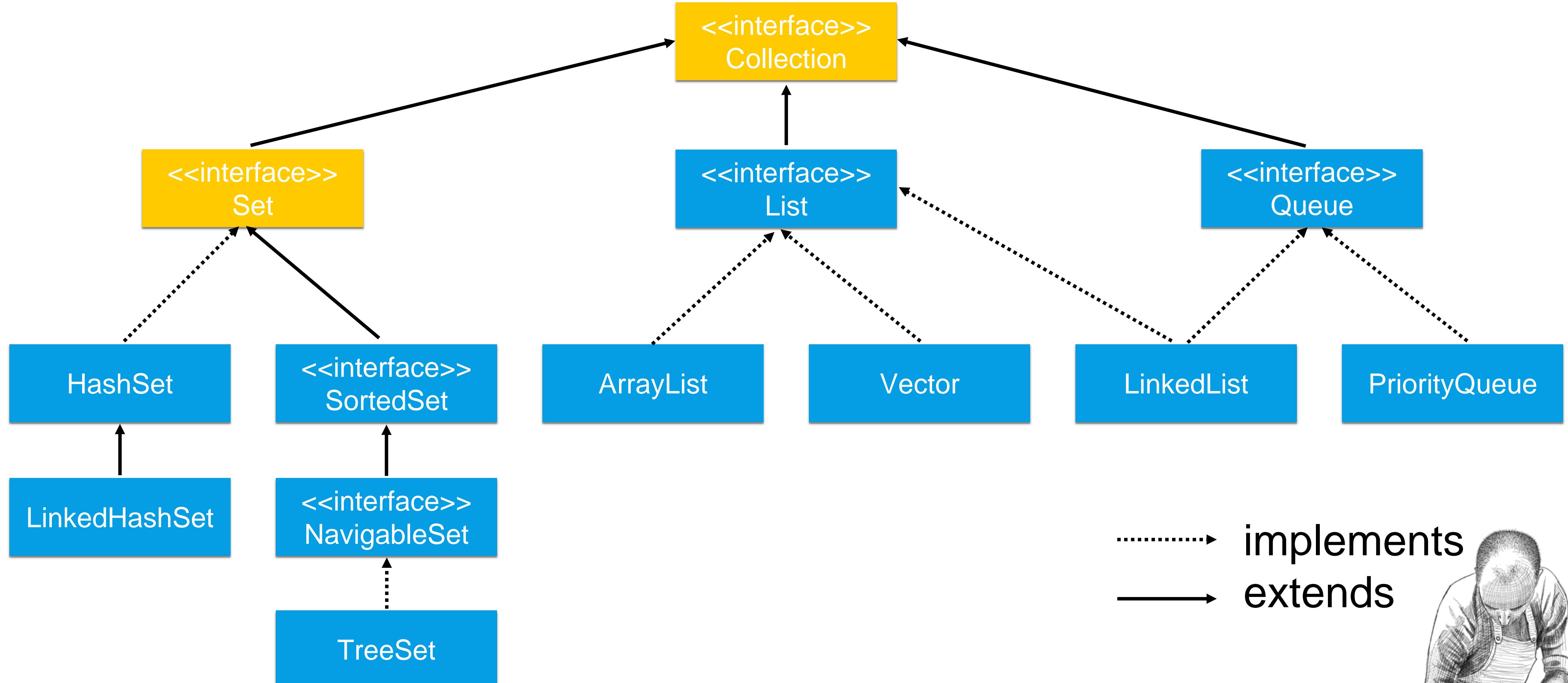
# Collection Interface



A Set is defined as a group of unique objects.



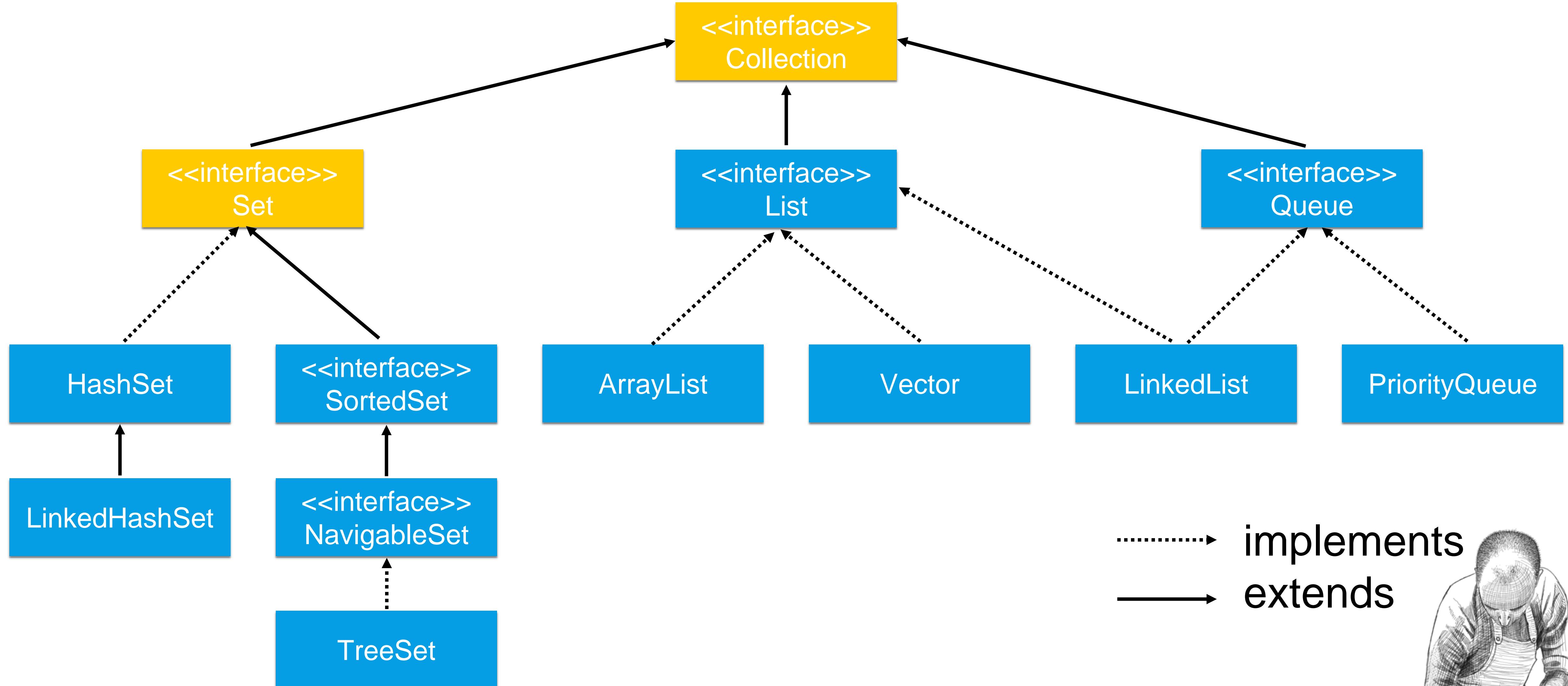
# Collection Interface



**What is considered as unique is defined by the equals method of the Object type the Set holds.**



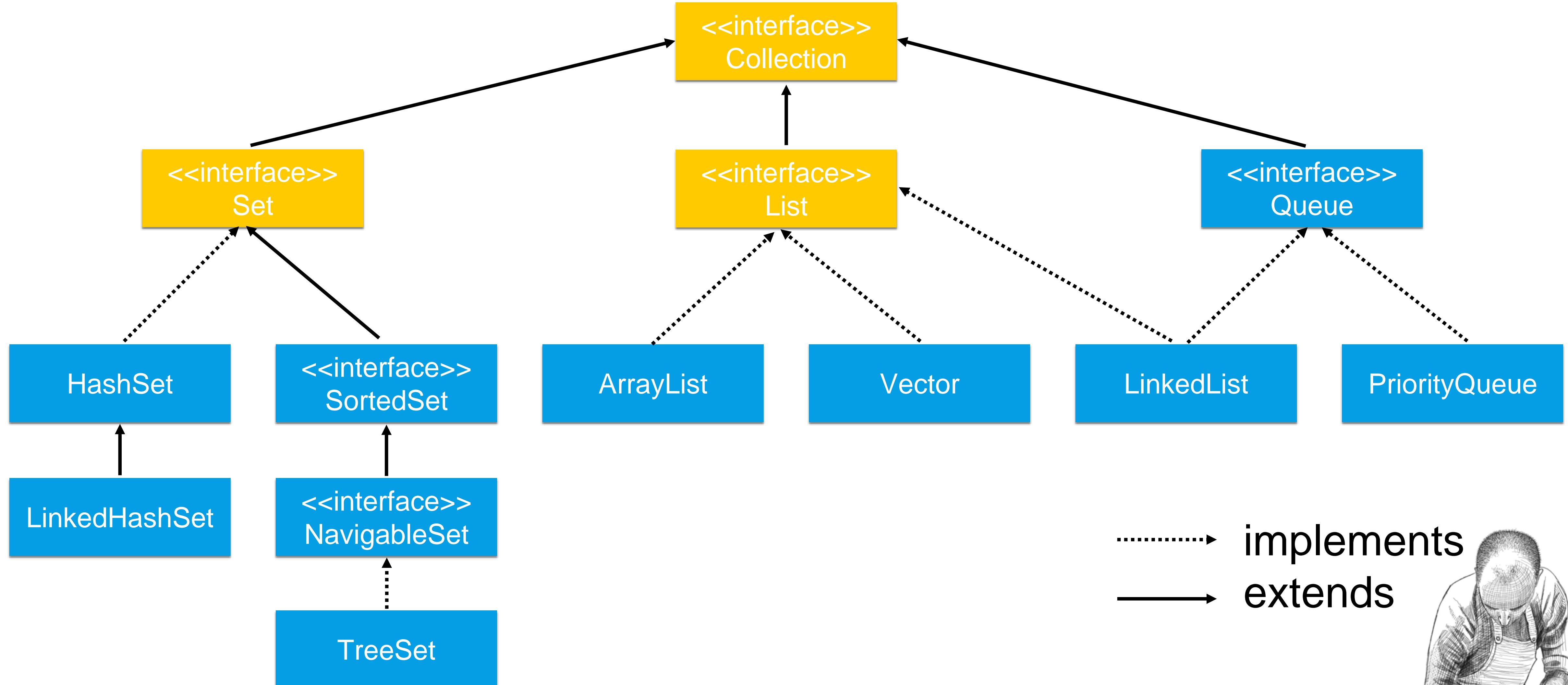
# Collection Interface



So in other words, a **Set** can not hold two equal objects.



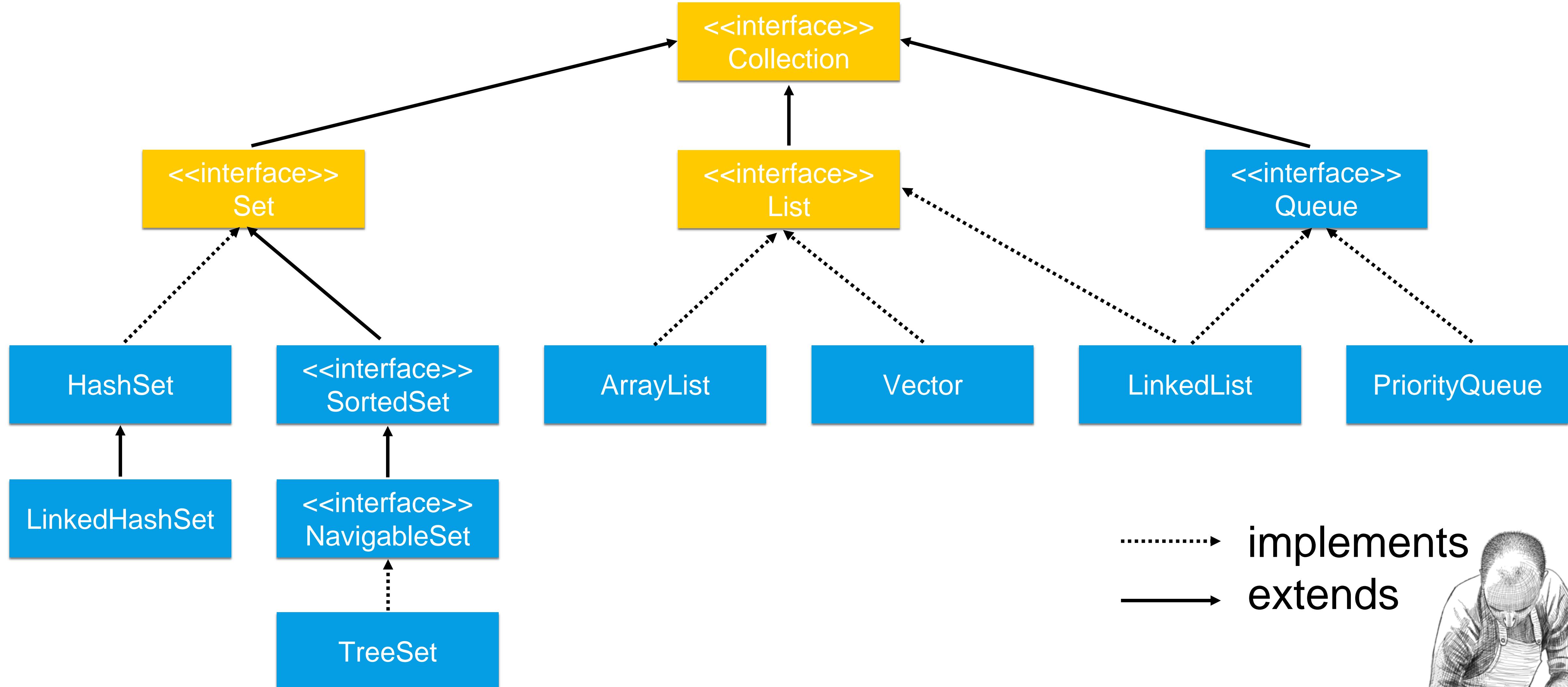
# Collection Interface



A List is defined as a sequence of objects.  
So unlike a Set, a List can contain duplicate entries.



# Collection Interface

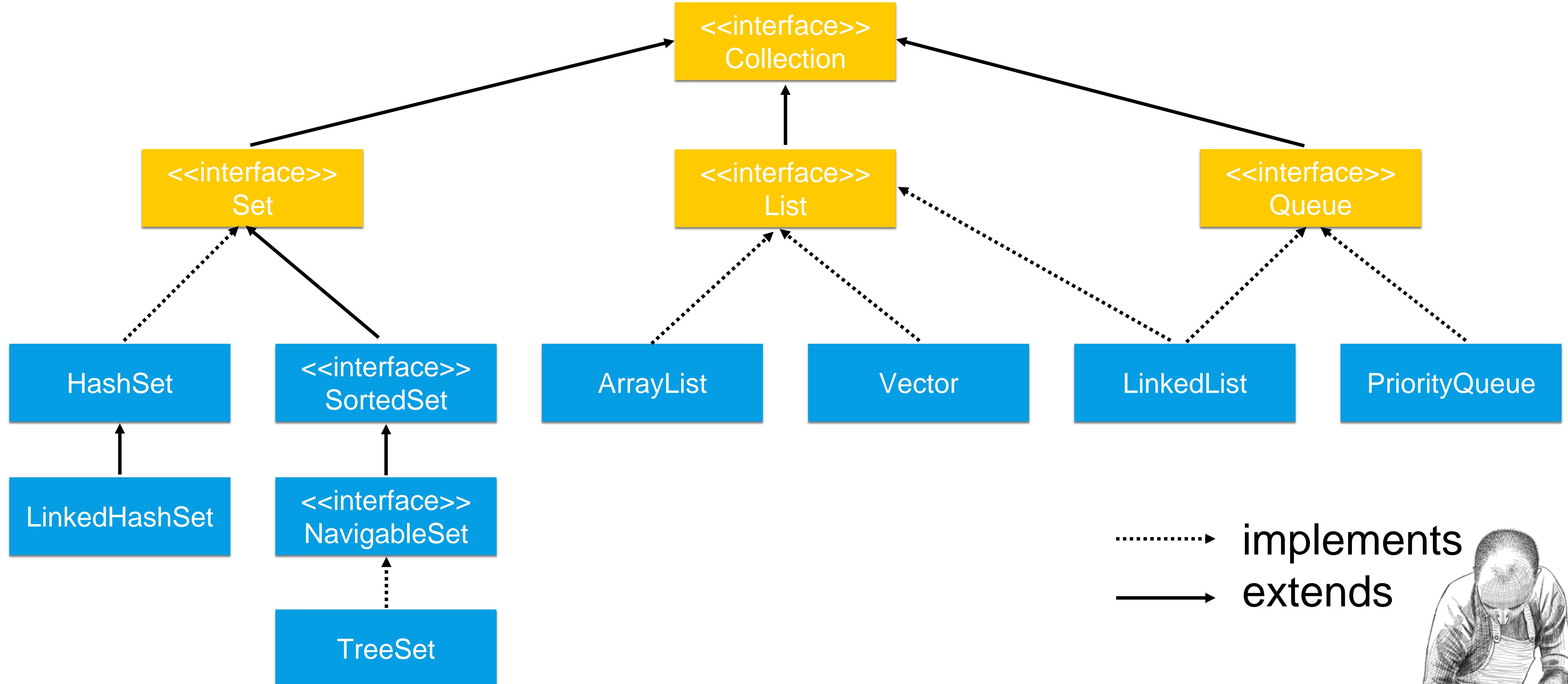


Besides,

a List keeps its elements in the order they were inserted into the list.



# Collection Interface

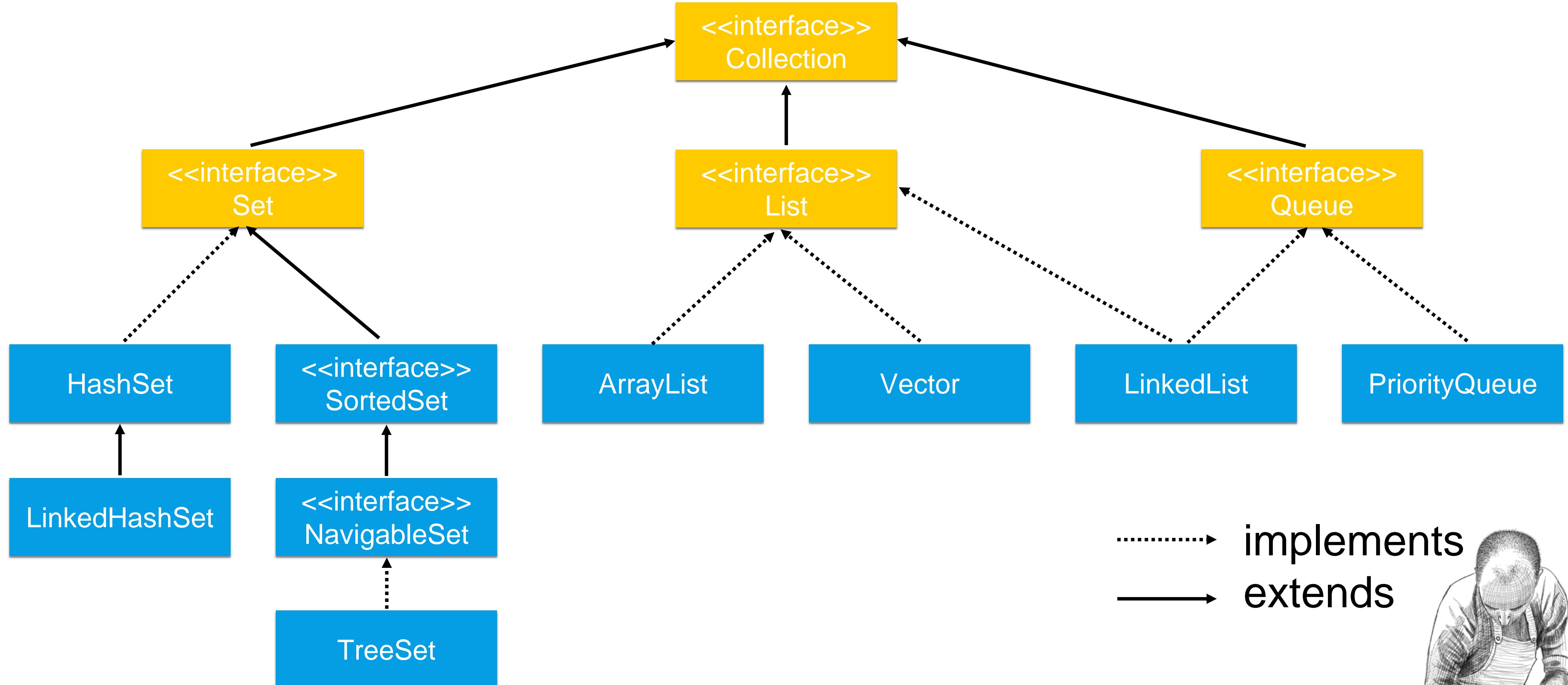


A queue has two sides.

Entries are added to the end and removed from the top of the queue.



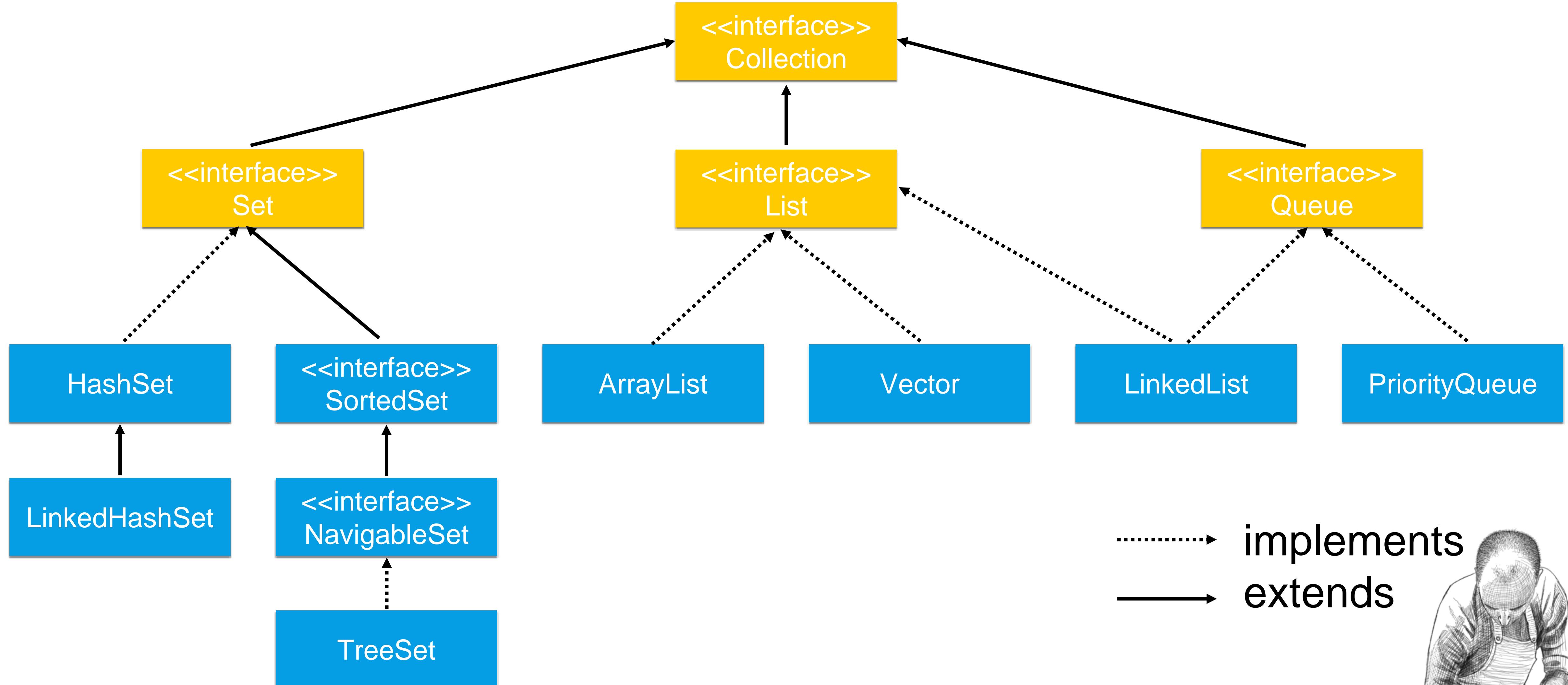
# Collection Interface



This is often described as “first in first out”,  
which is pretty much like a waiting line in real life works.



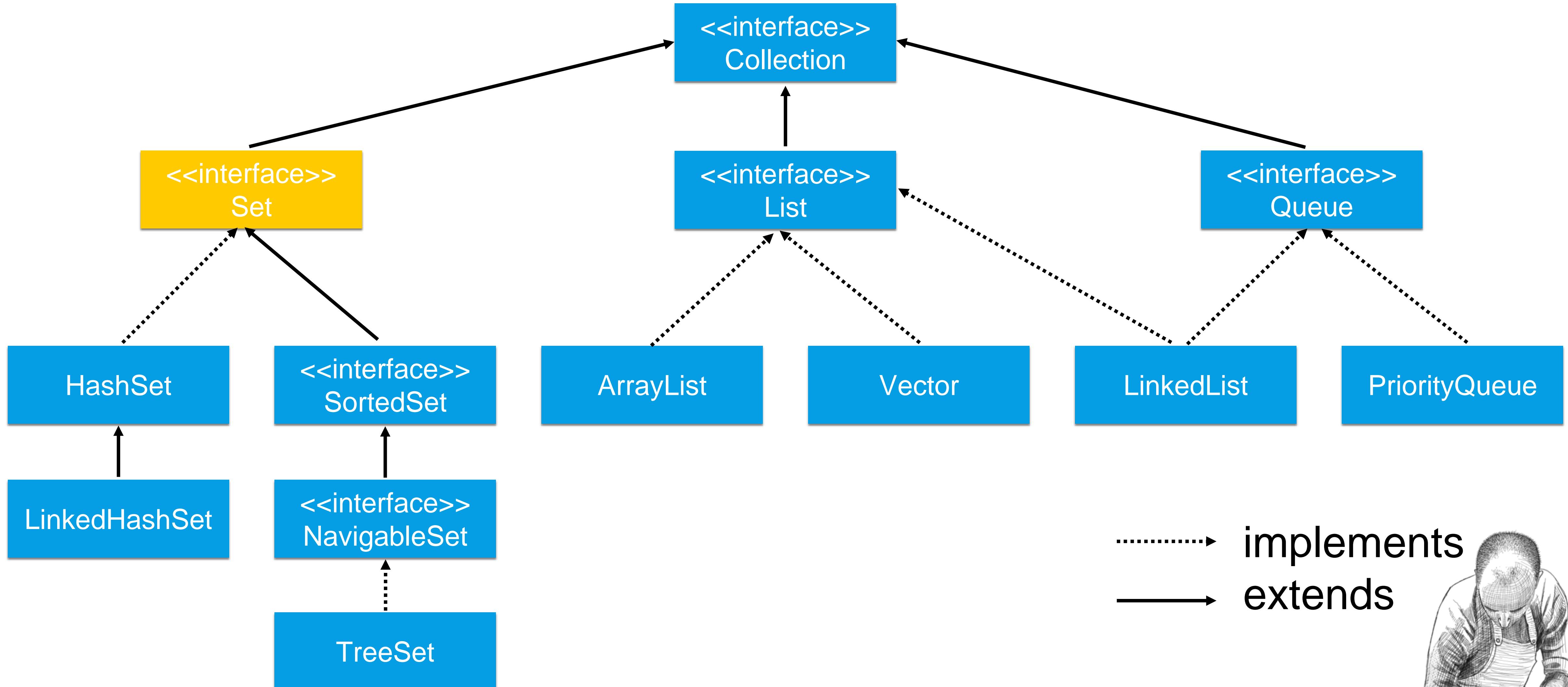
# Collection Interface



**The first person queuing up will also be  
the first person leaving the queue.**



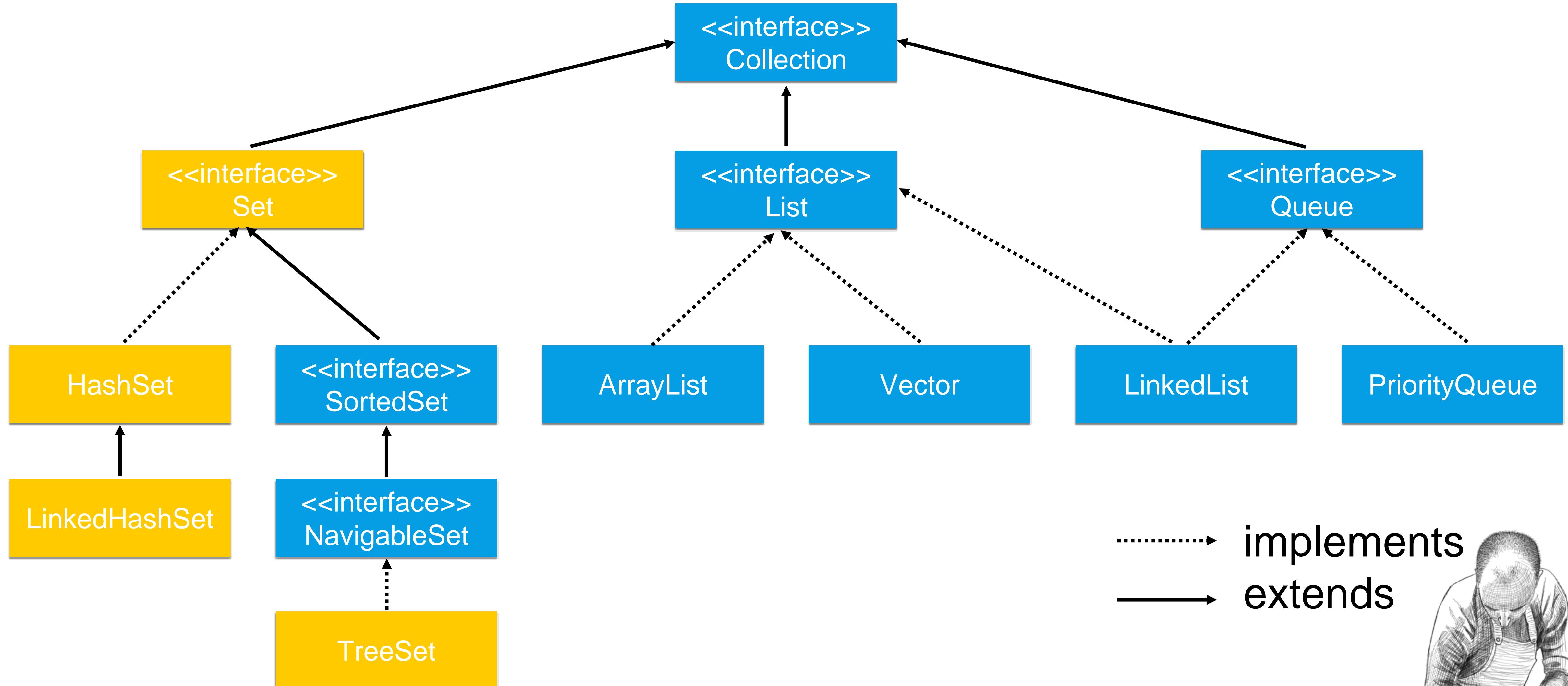
# Collection Interface



Now let's have a closer look at the interfaces and classes that extend or implement the **Set** interface.



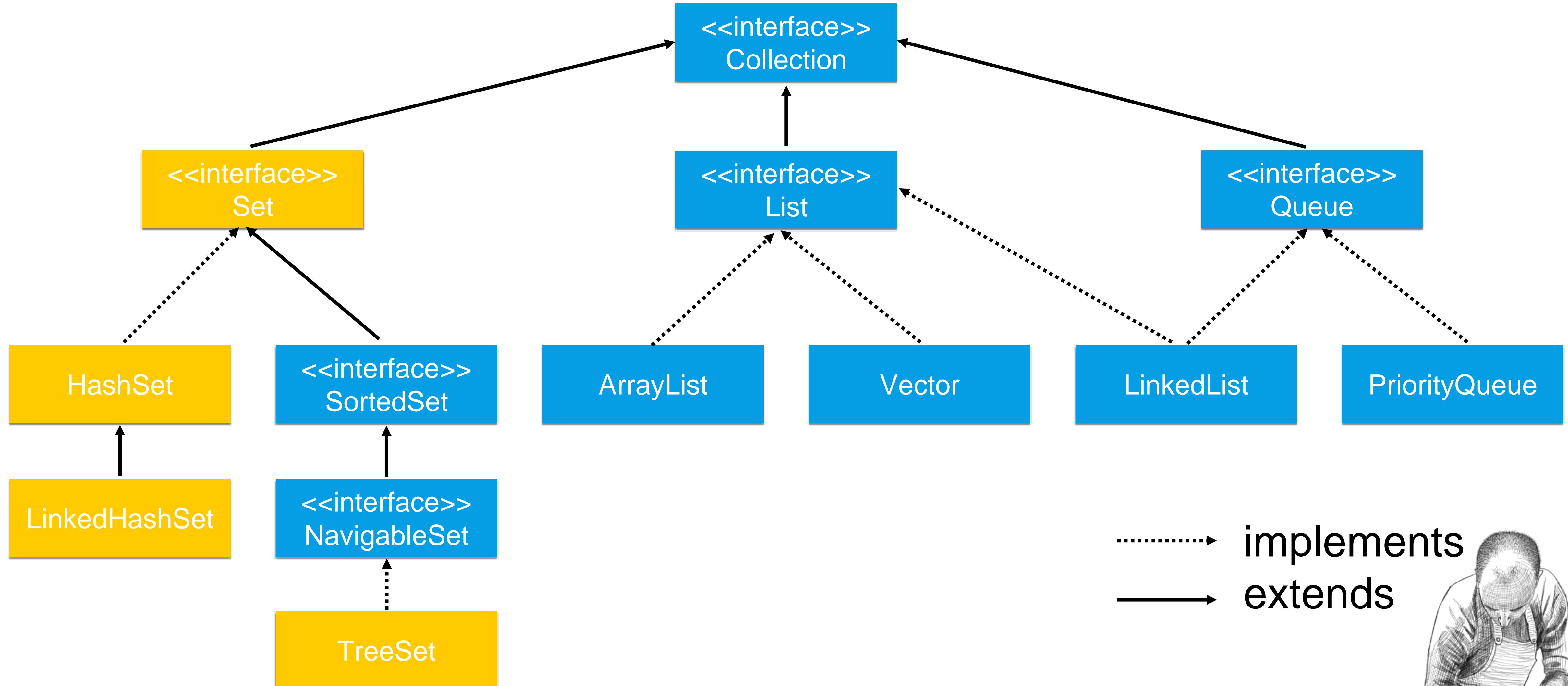
# Collection Interface



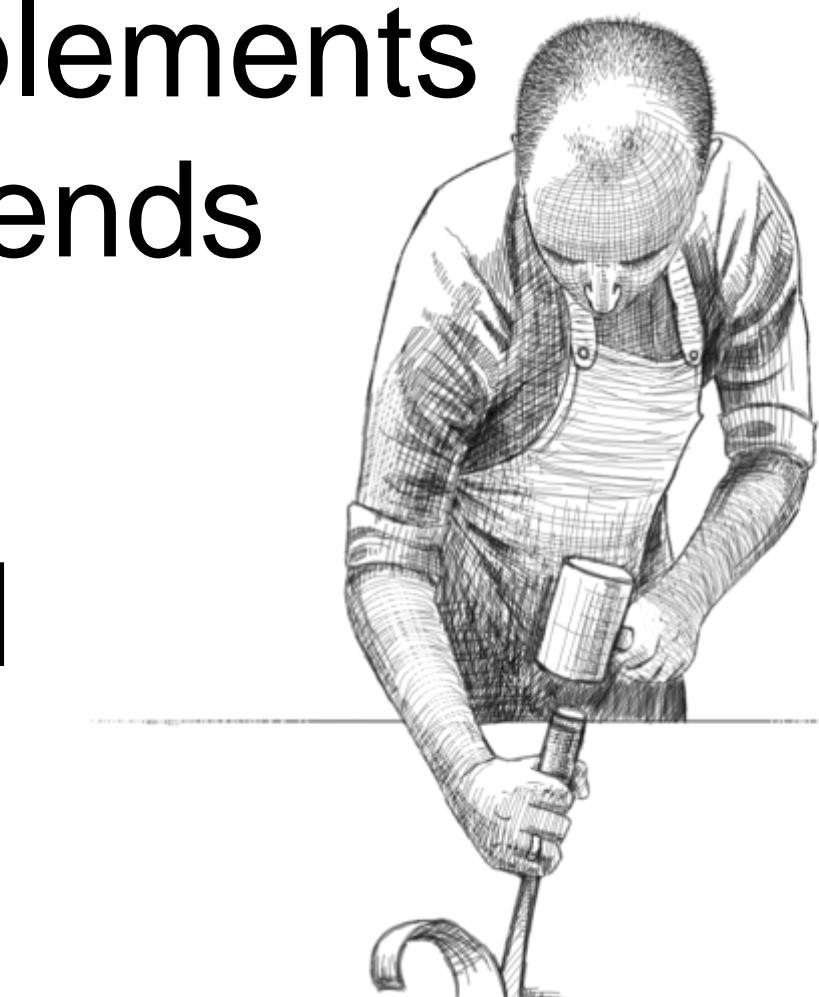
**HashSet, LinkedHashSet and TreeSet  
are all implementing the Set interface.**



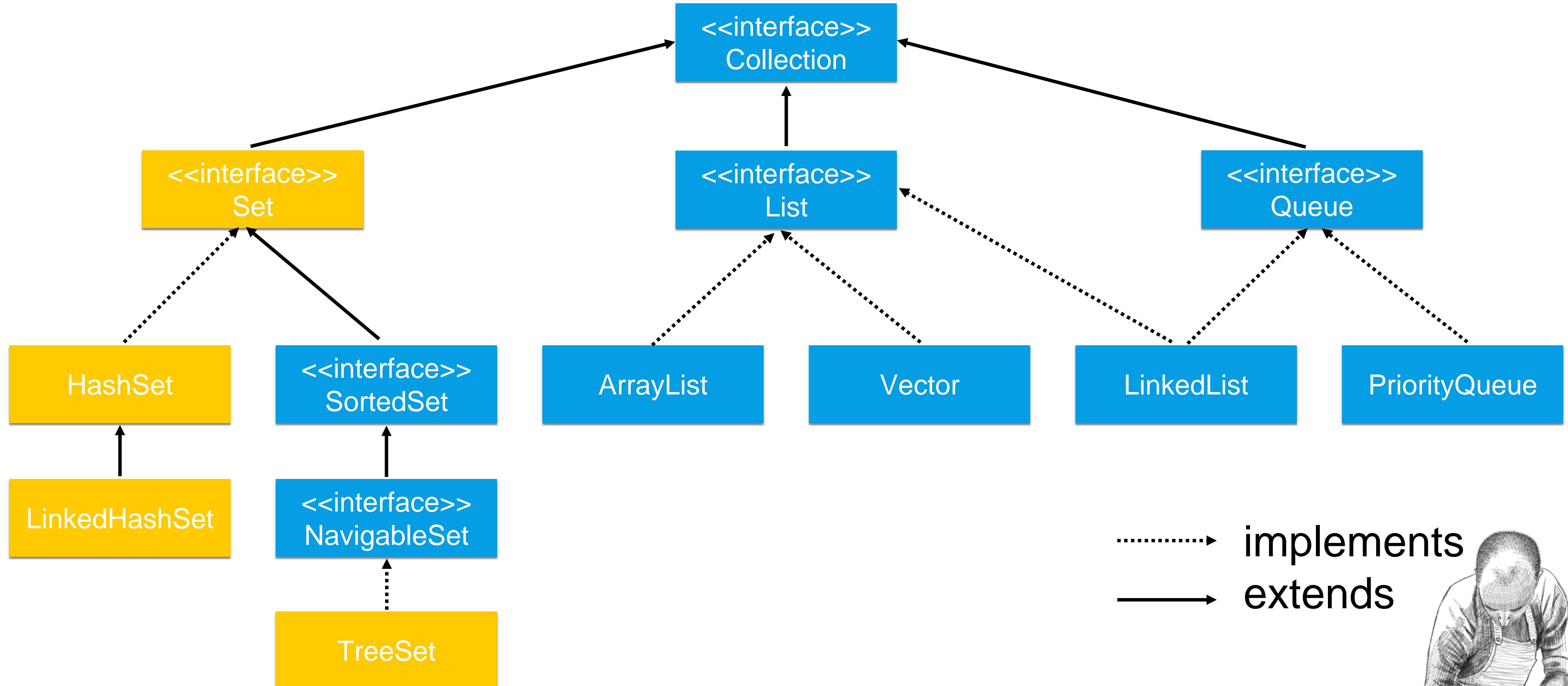
# Collection Interface



**HashSet is the default implementation that is used in the majority of cases.**



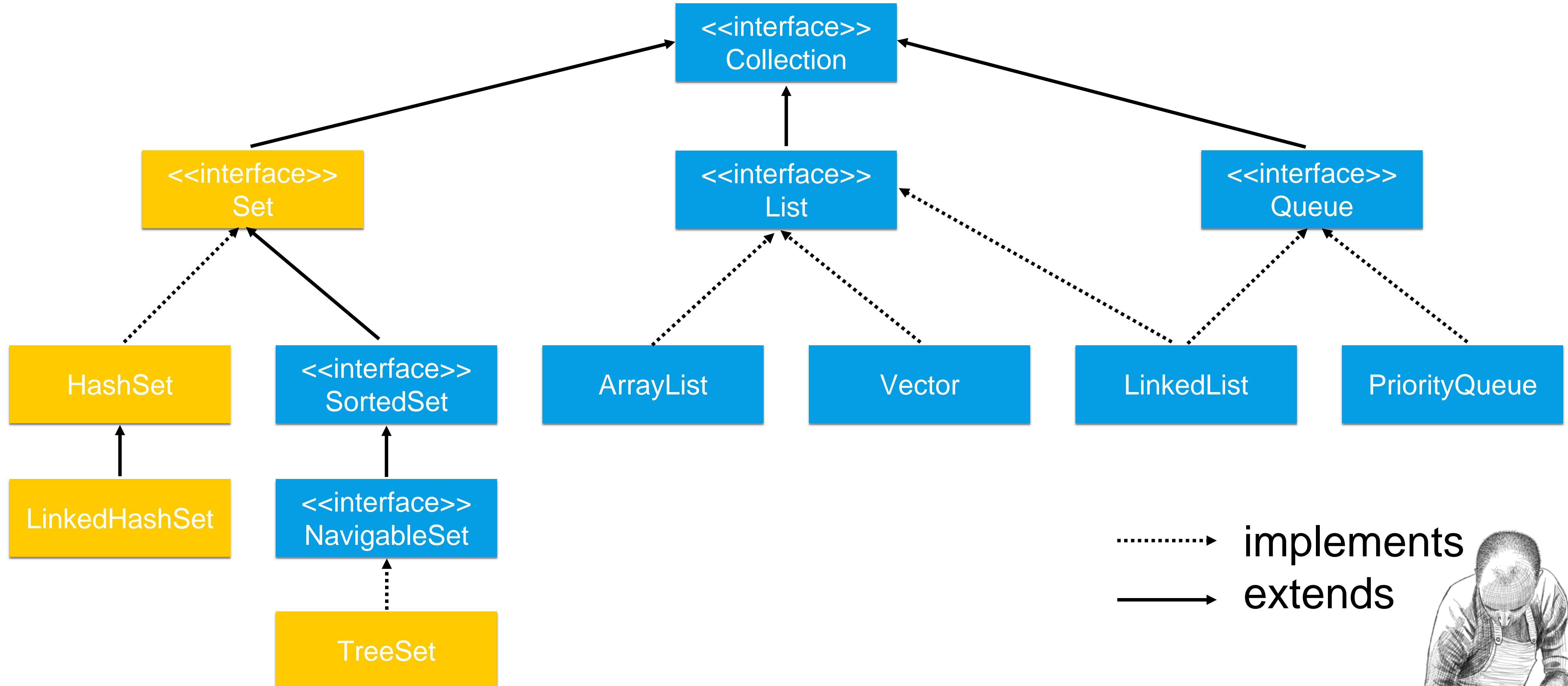
# Collection Interface



**LinkedHashSet** is like a mix of a **HashSet** and a **List**,  
as it does not allow duplicate entries like a **Set**,



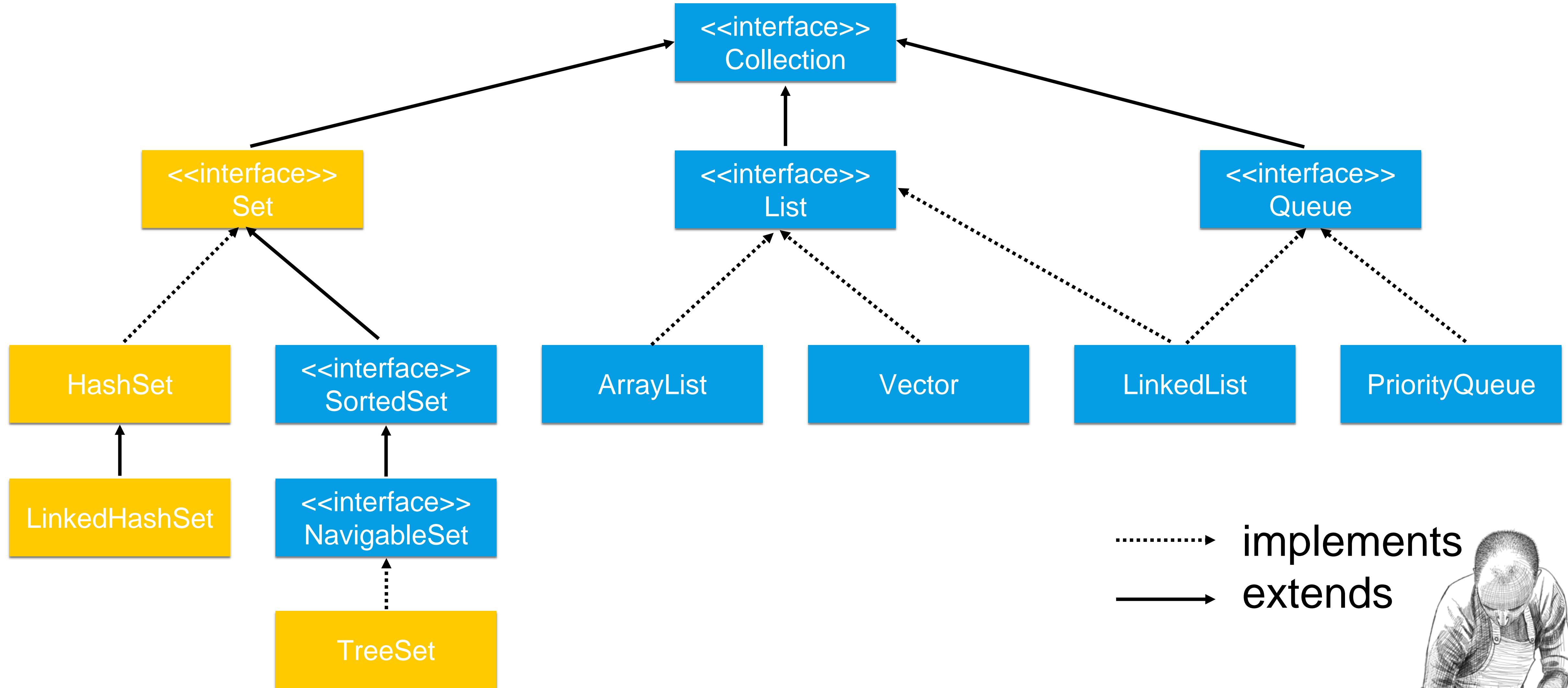
# Collection Interface



**but it returns its elements in the order  
in which they were inserted, like a List would do.**



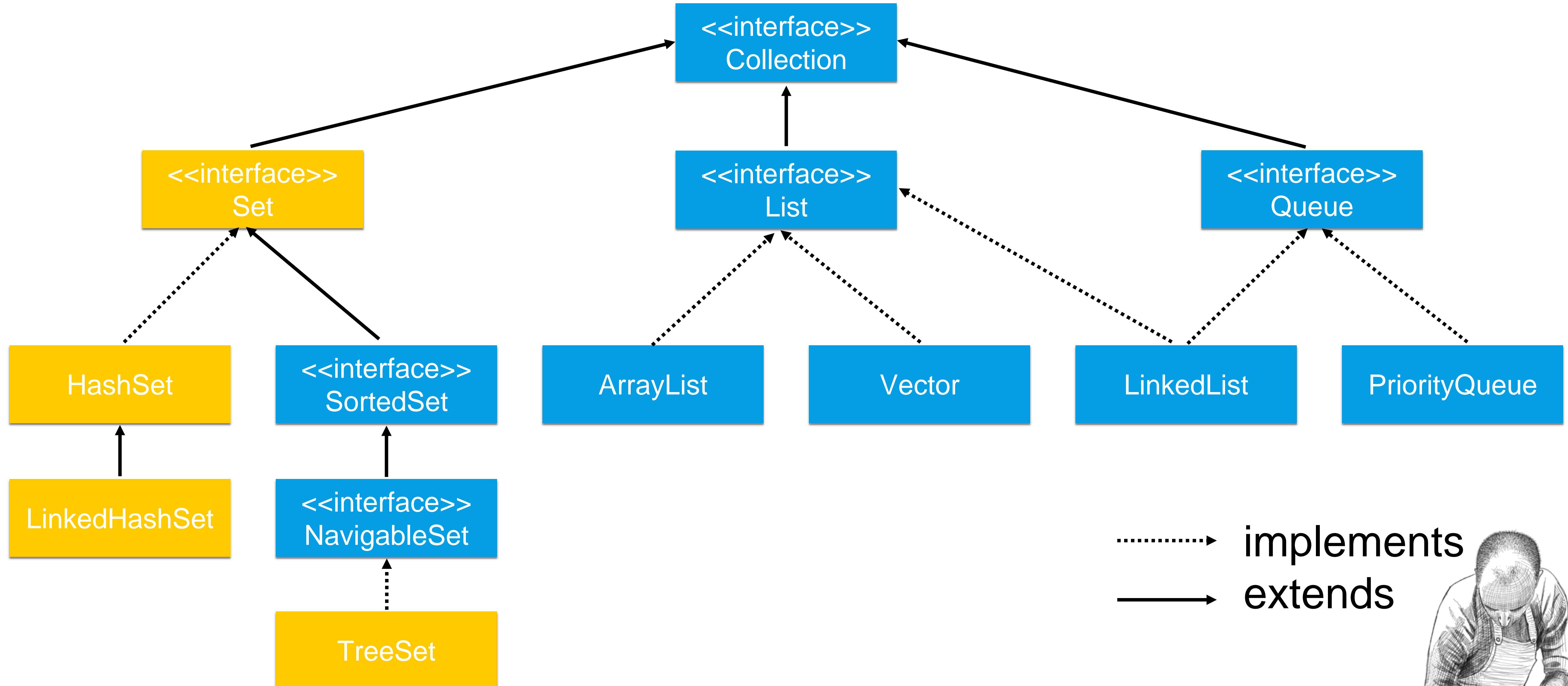
# Collection Interface



**TreeSet will constantly keep all its elements in sorted order.**



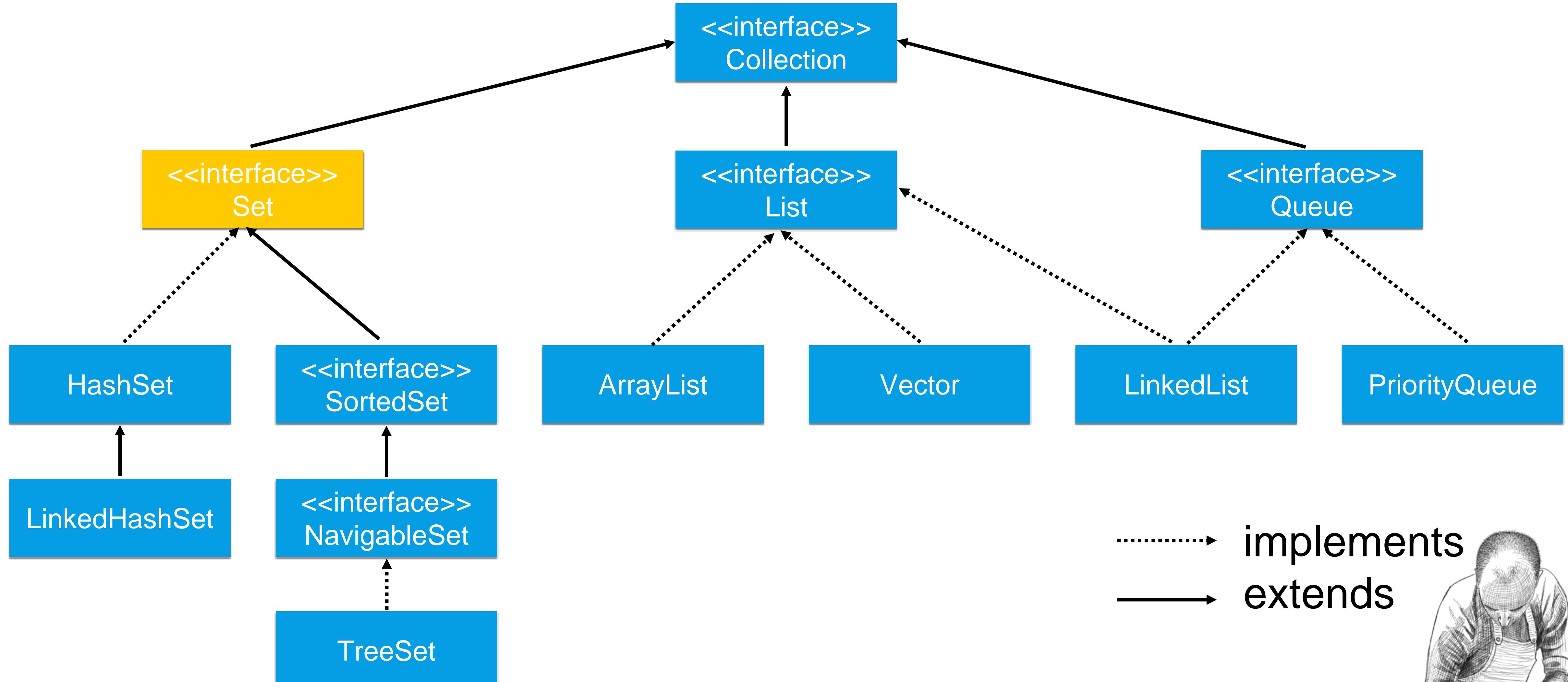
# Collection Interface



**But keep in mind “there is no free lunch”,  
every added feature comes at a certain cost.**



# Collection Interface

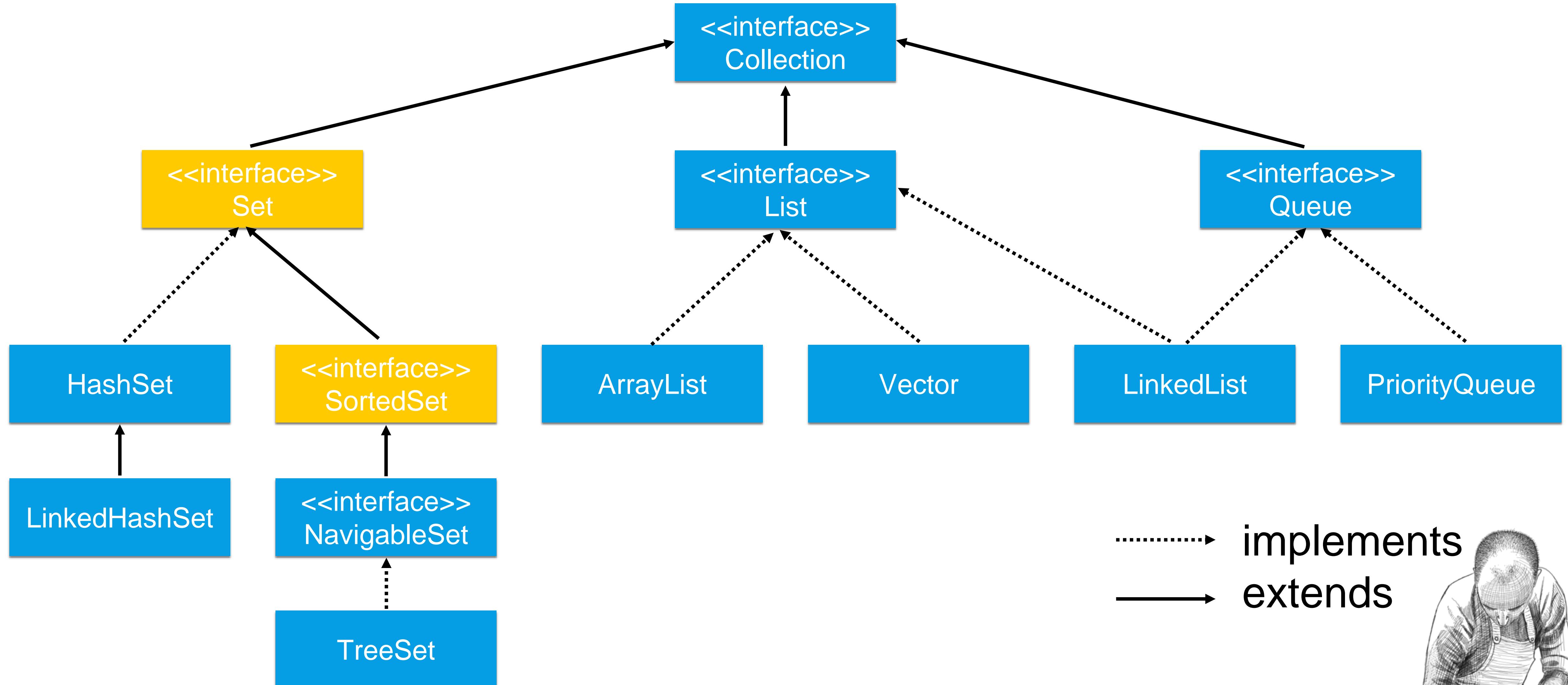


..... implements  
→ extends

After looking at the classes implementing the **Set** interface,  
let's also have a look at the two extending interfaces  
we haven't talked about yet.



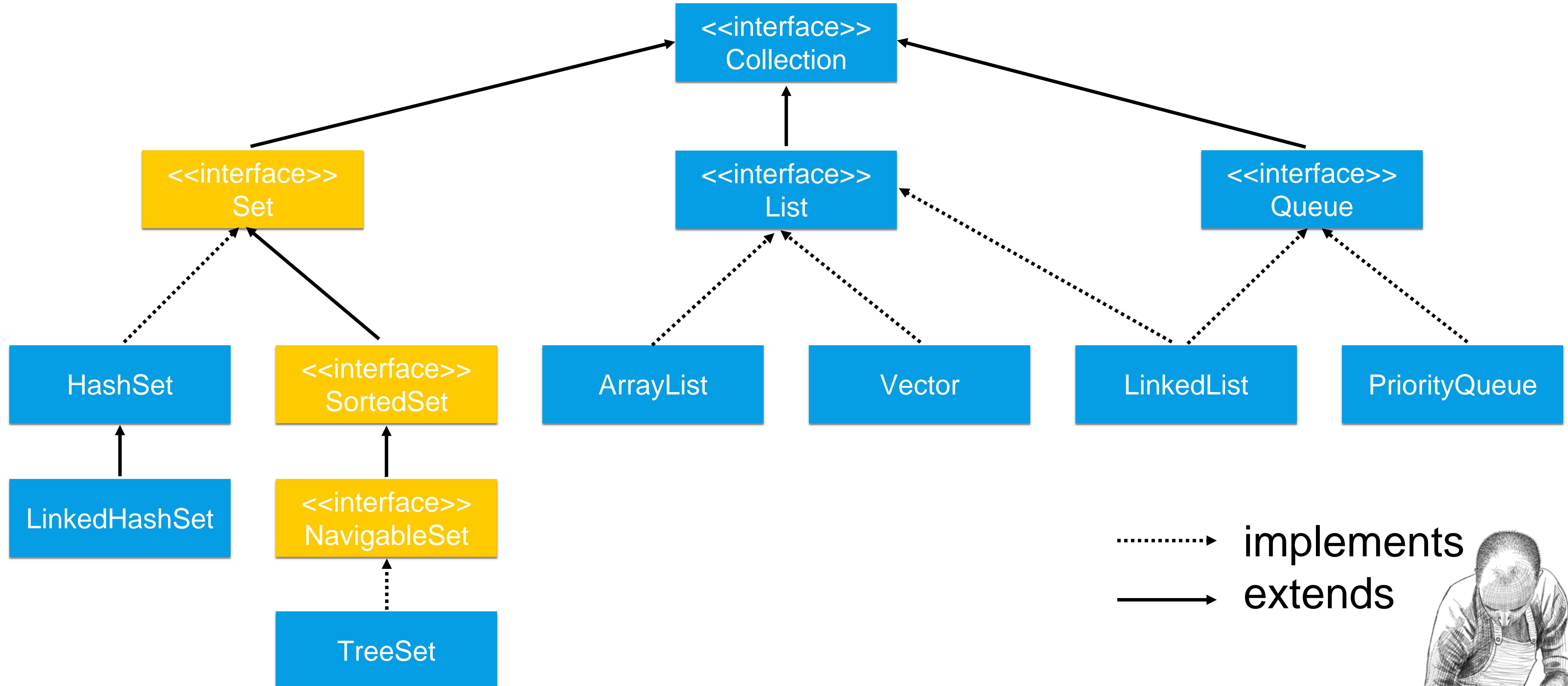
# Collection Interface



As the name implies,  
**SortedSet** is a **Set** that is constantly sorted.



# Collection Interface

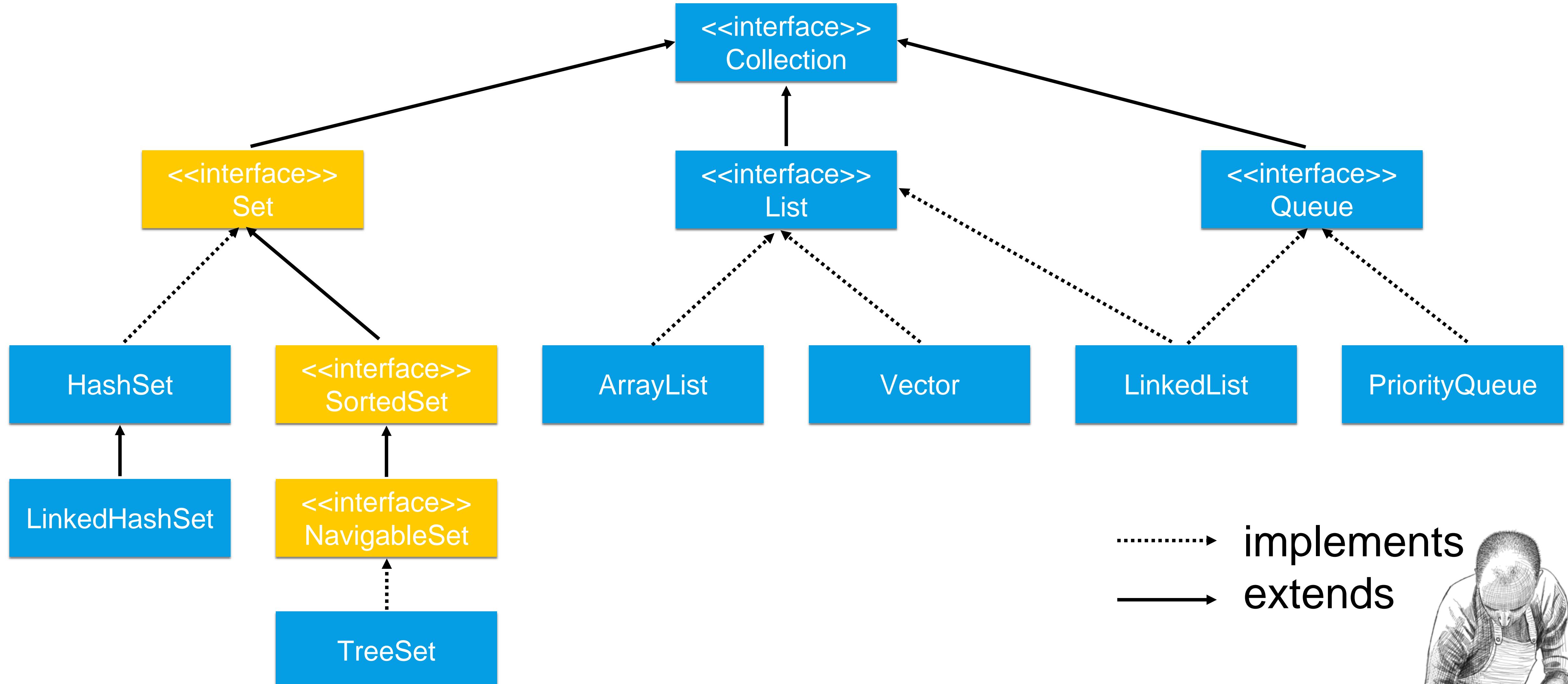


..... → implements  
→ extends



**The NavigableSet interface was added with Java 6.  
It allows to navigate through the sorted list**

# Collection Interface

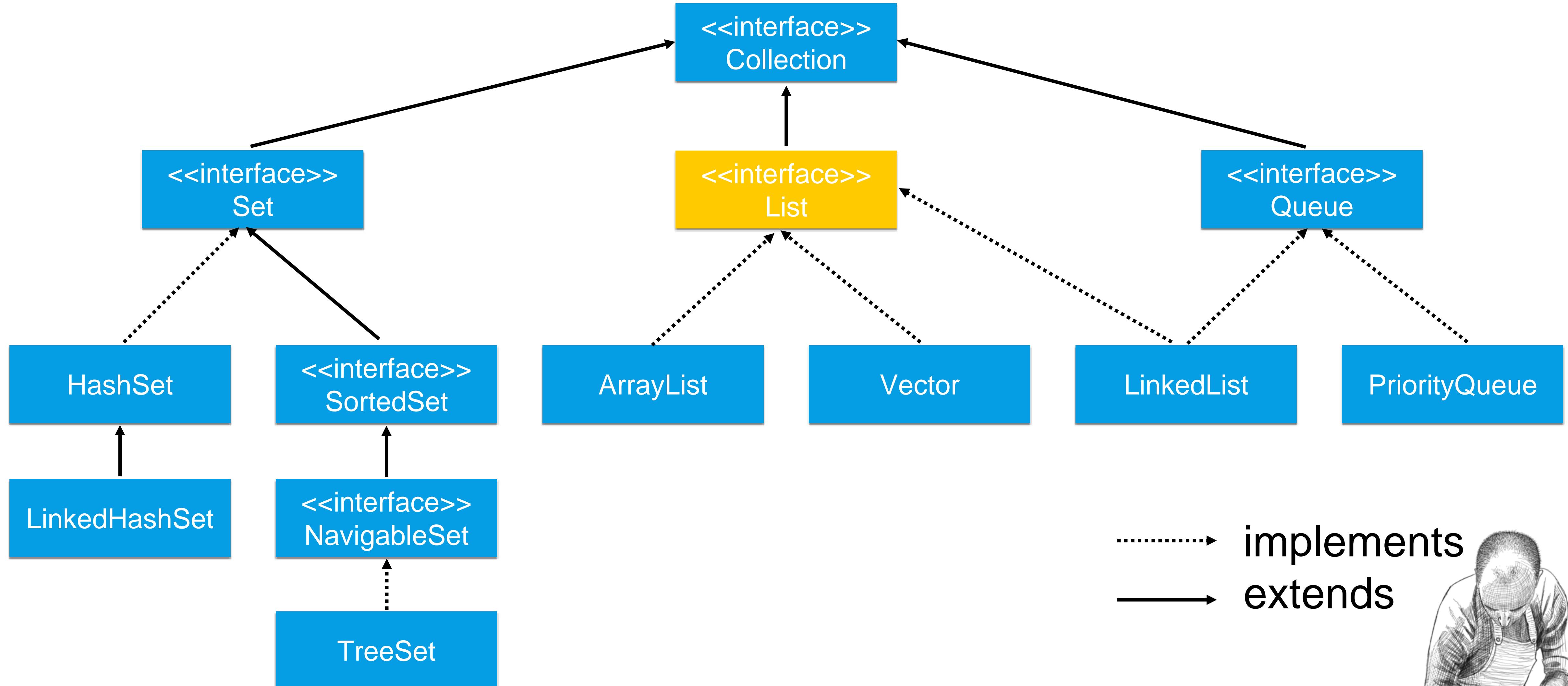


..... → implements  
→ extends



For example it provides methods to retrieve the next element greater or smaller then a given element of the Set.

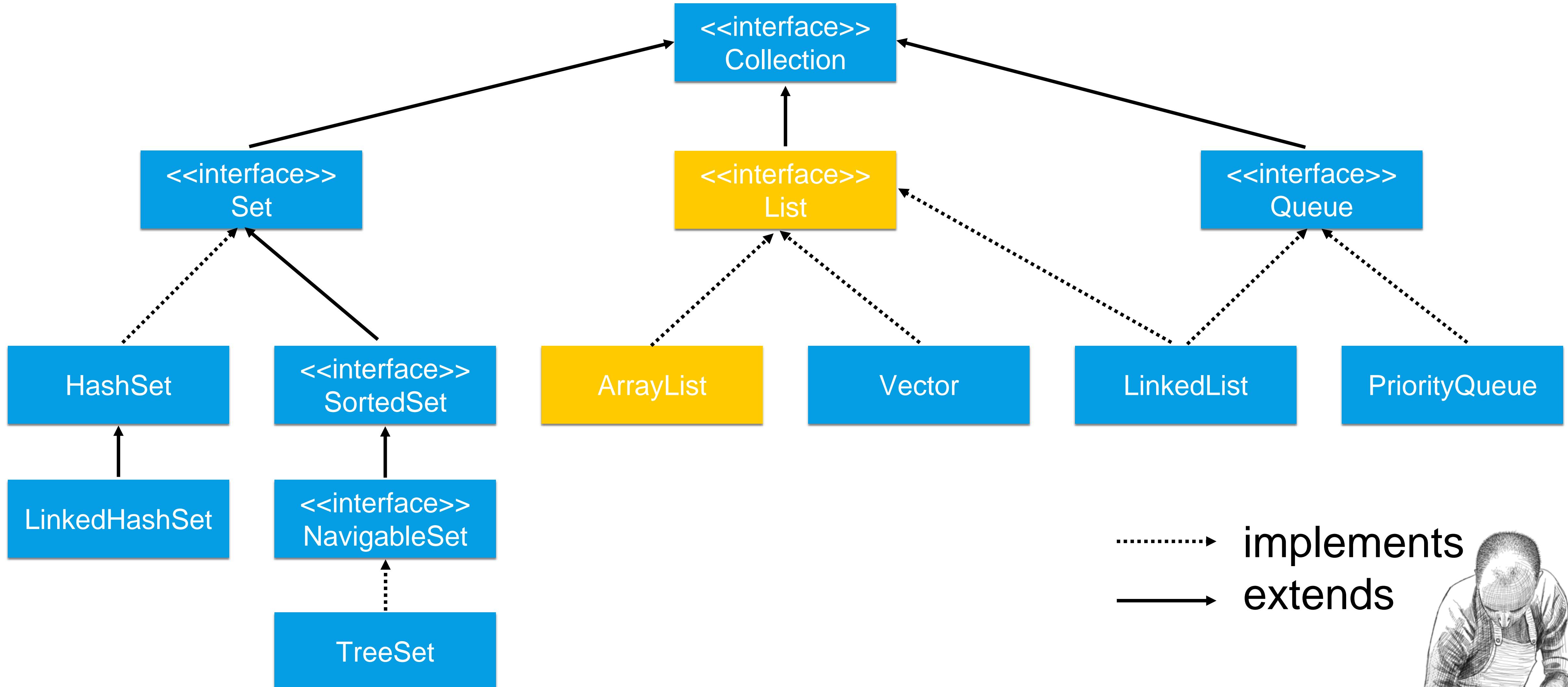
# Collection Interface



Next, let's have a closer look at the classes that implement the List interface.



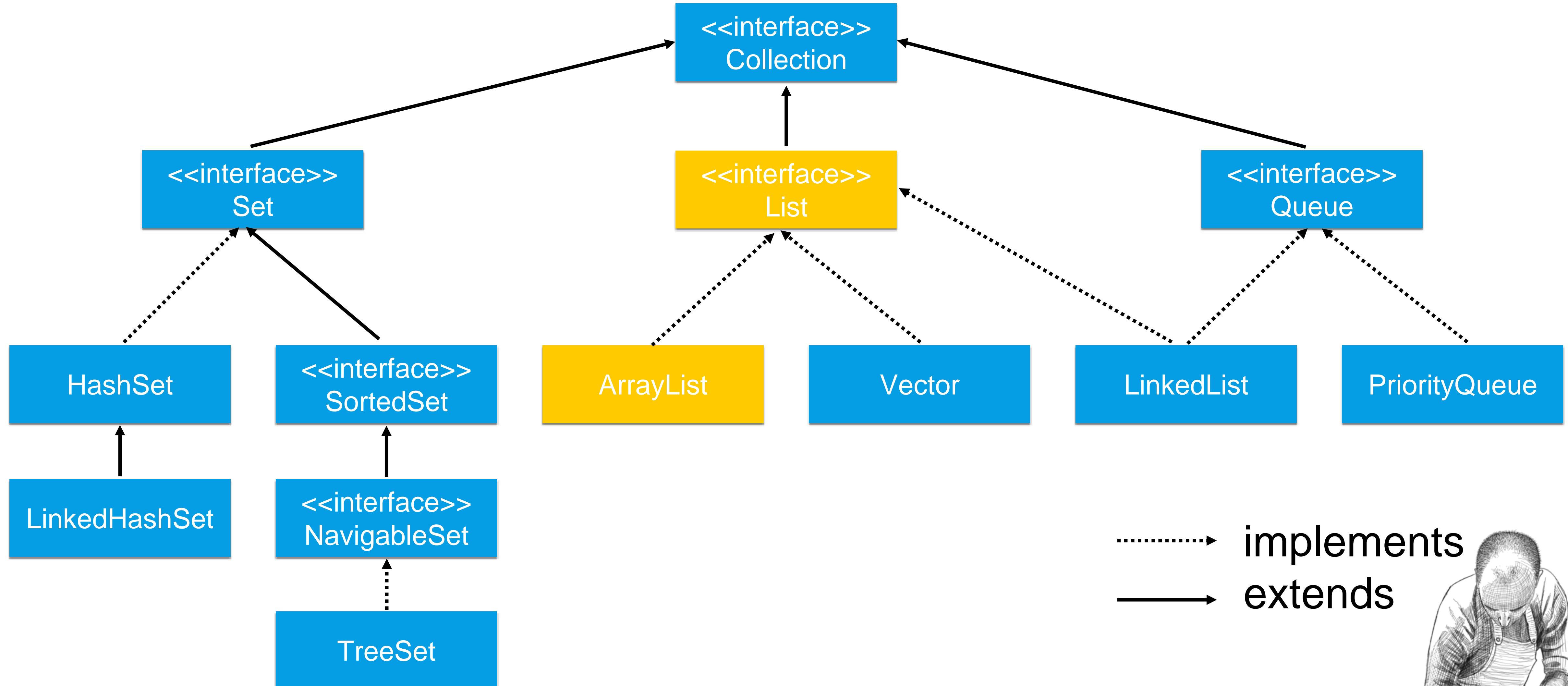
# Collection Interface



**ArrayList** is the default implementation of the **List** interface.  
Like any list implementation, it does allow duplicate elements,



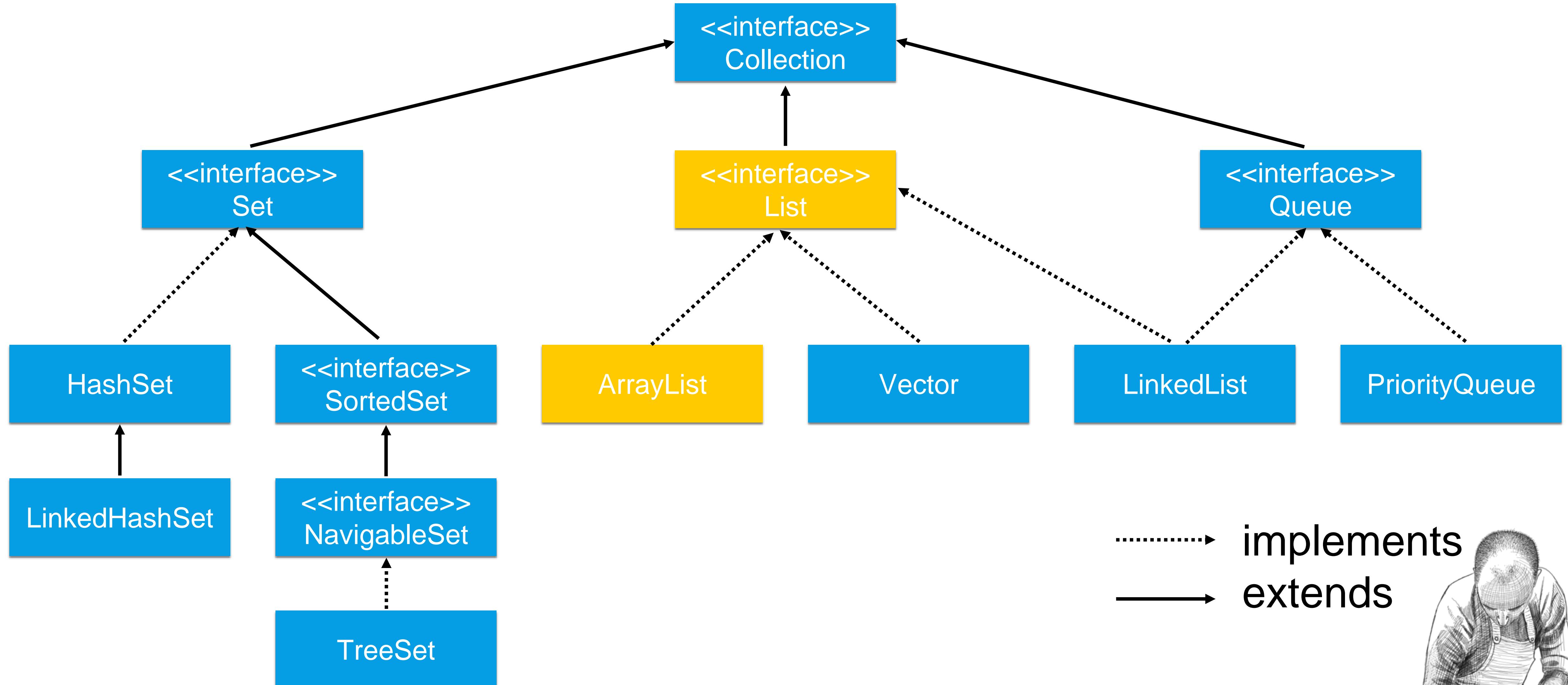
# Collection Interface



and it does allow to iterate the list in the order of insertion.



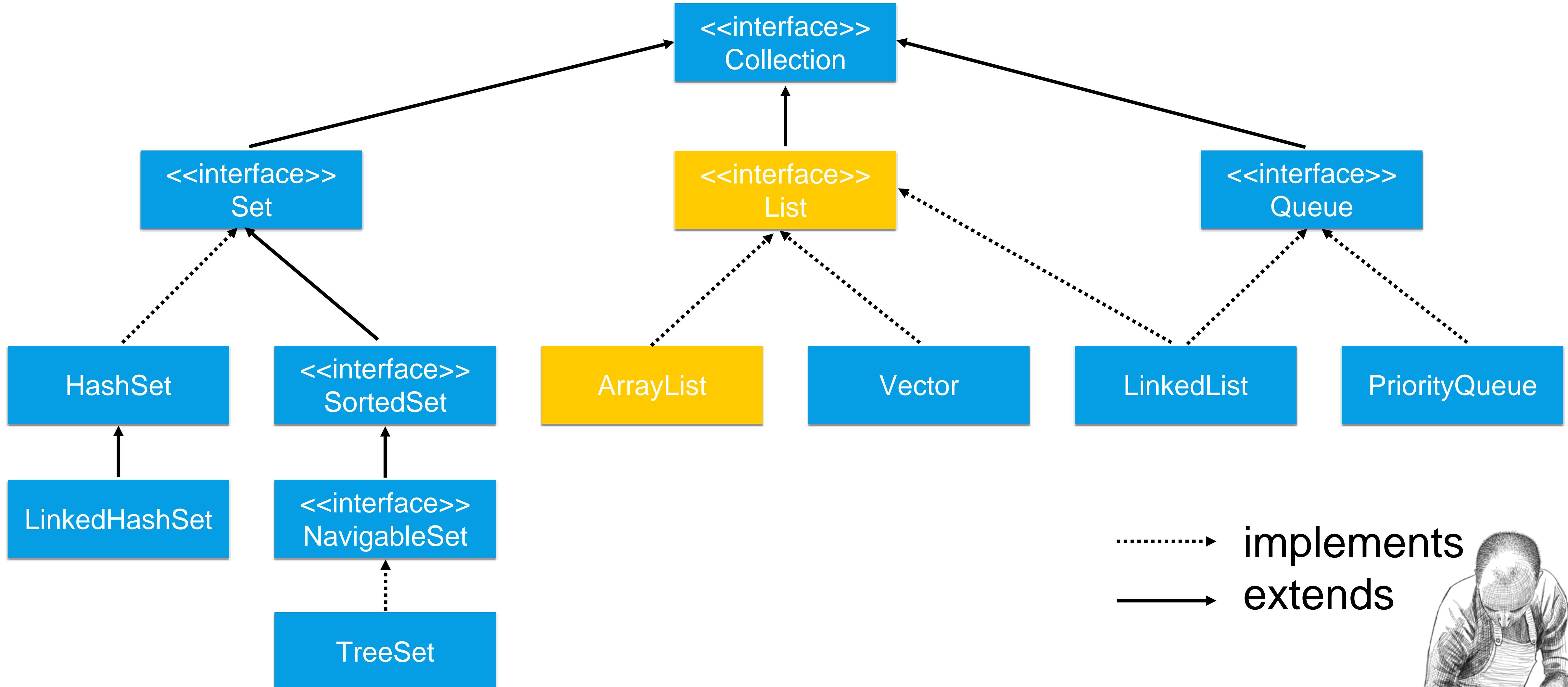
# Collection Interface



As it is based on arrays,  
it is very fast to iterate and read from an **ArrayList**,



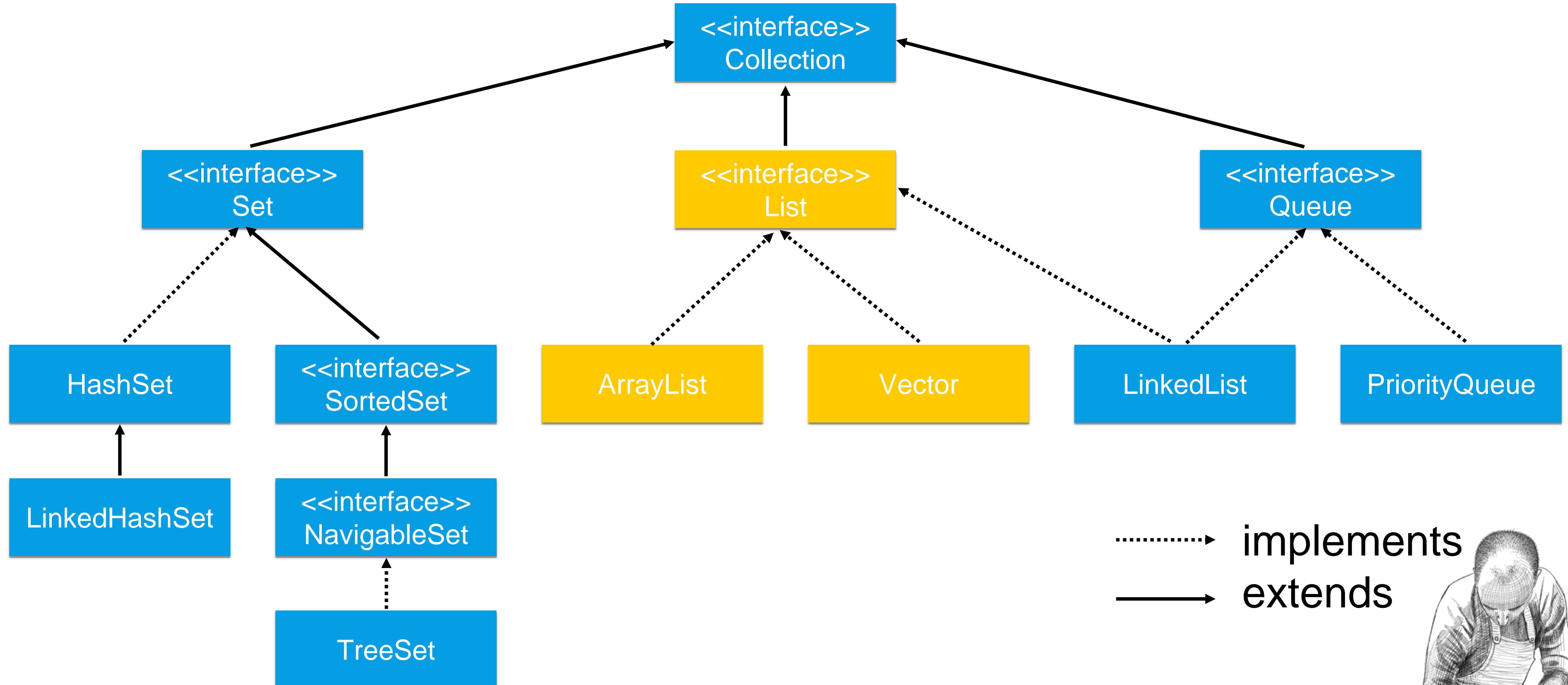
# Collection Interface



**but adding or removing an element at a random position is very slow,  
as this will require to rebuild the underlying array structure.**



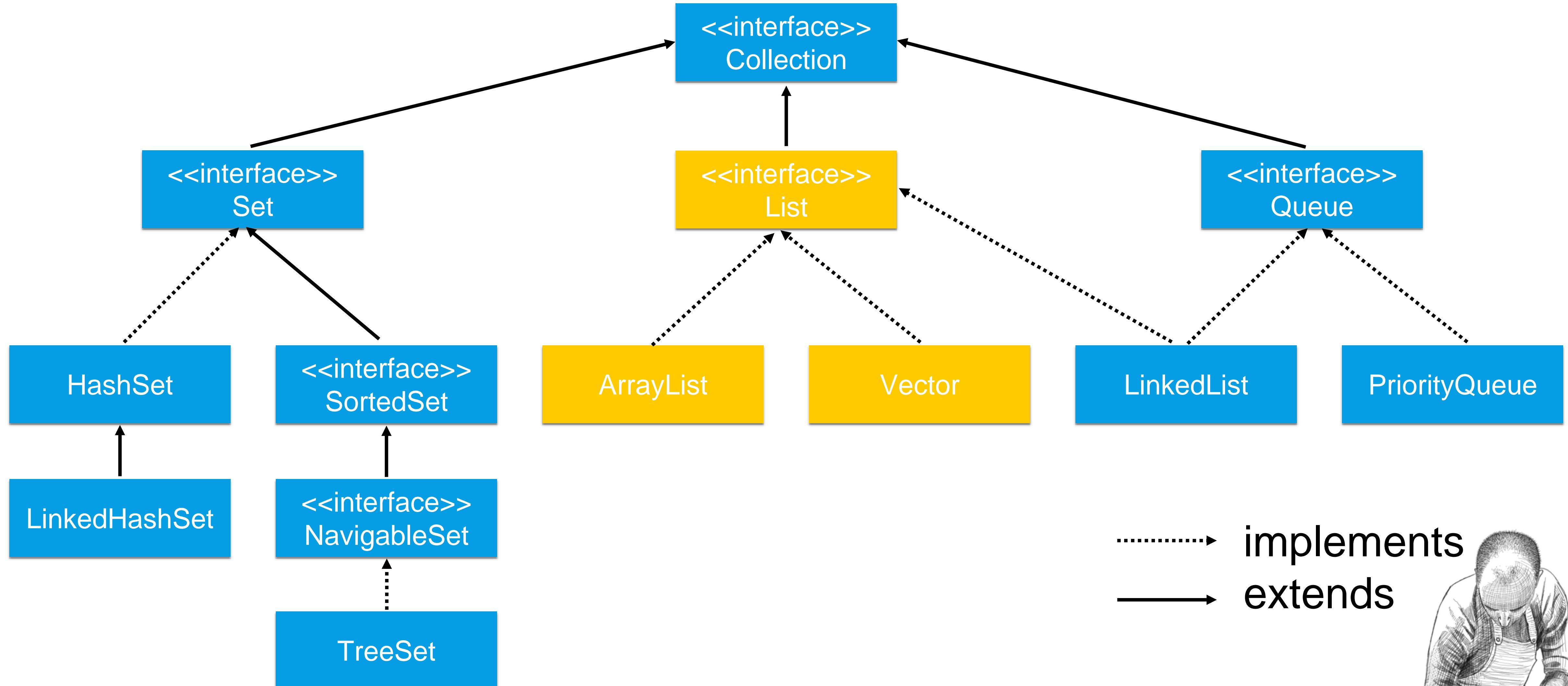
# Collection Interface



**Vector** is a class that exists since JDK 1,  
which is even before the Collections Framework was added with Java 2.



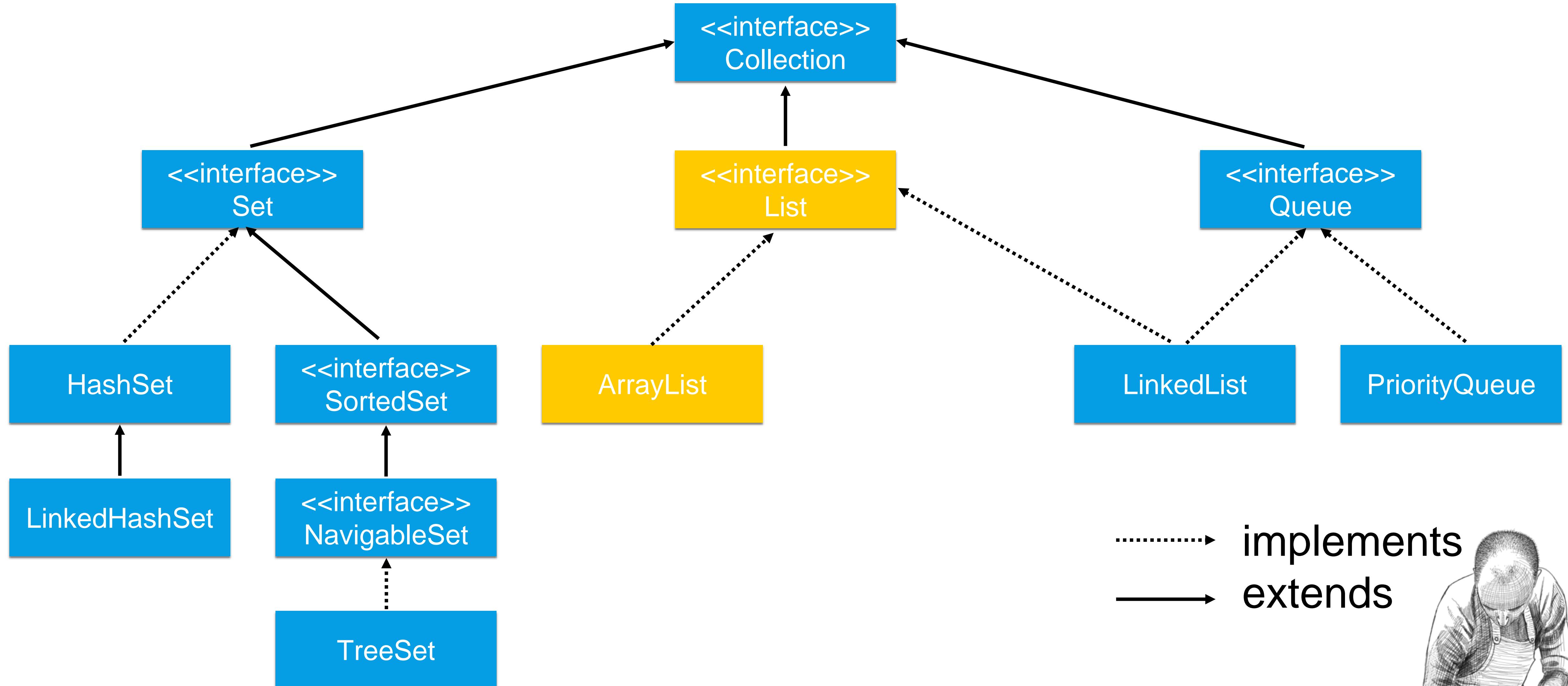
# Collection Interface



In short, its performance is suboptimal, so please never use it,  
use **ArrayList** or **LinkedList** instead. Let's directly remove it



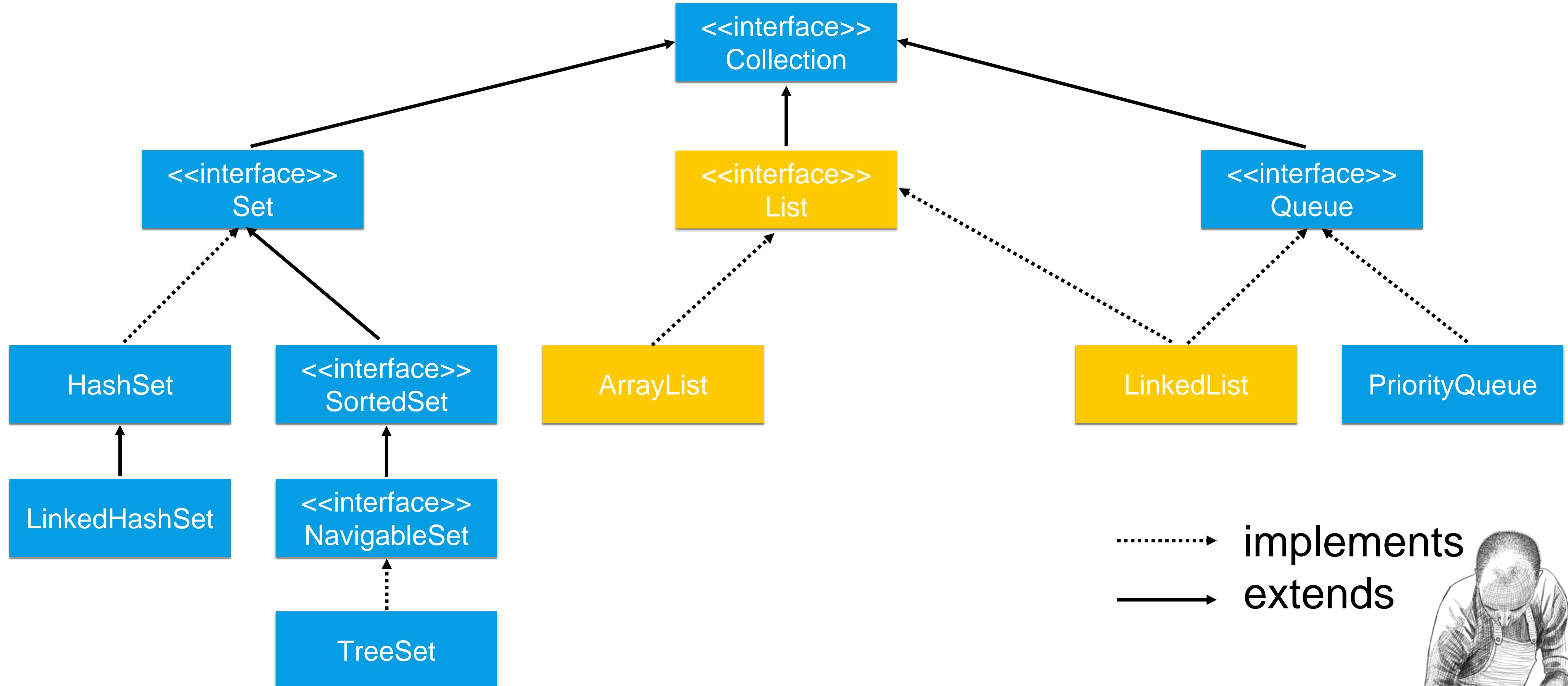
# Collection Interface



and forget about it.



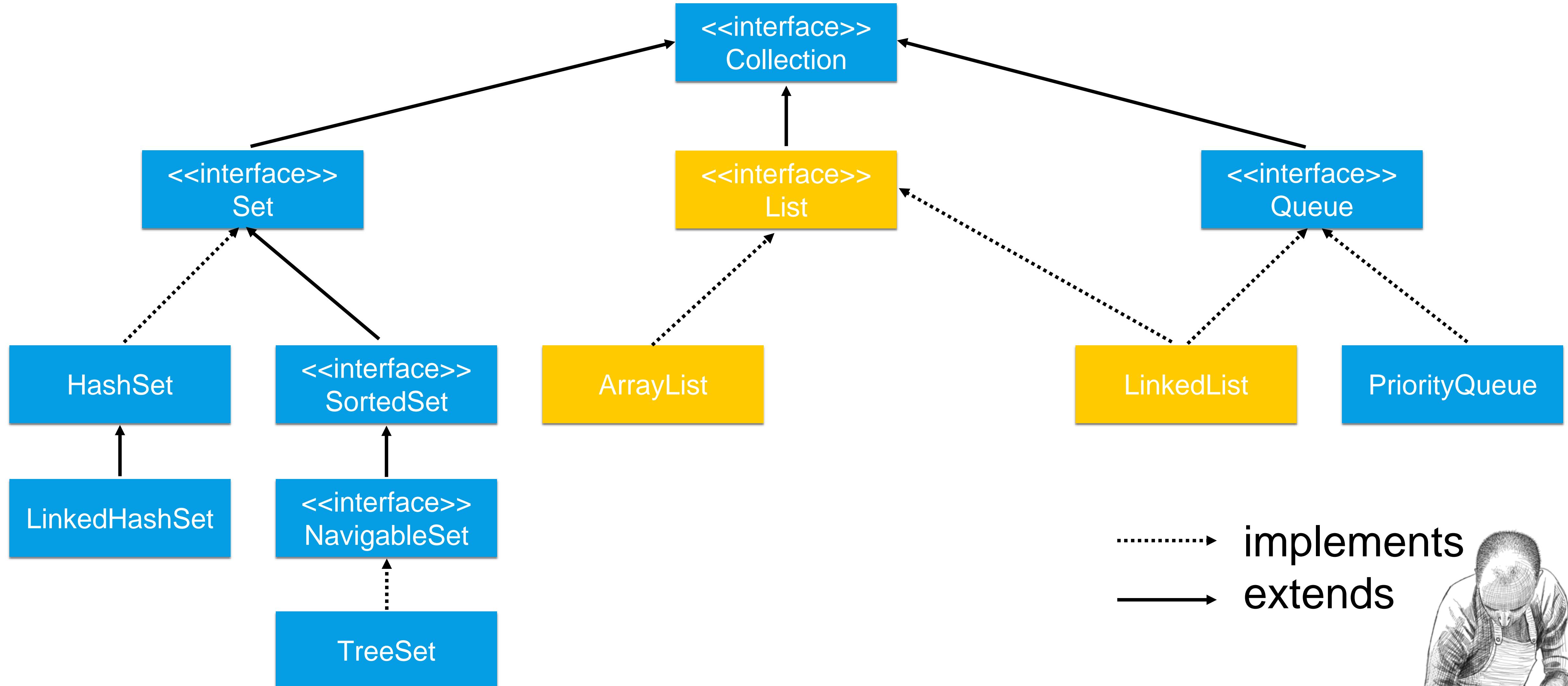
# Collection Interface



The next List implementation is **LinkedList**.  
As the name implies, its implementation is based on a **LinkedList**.



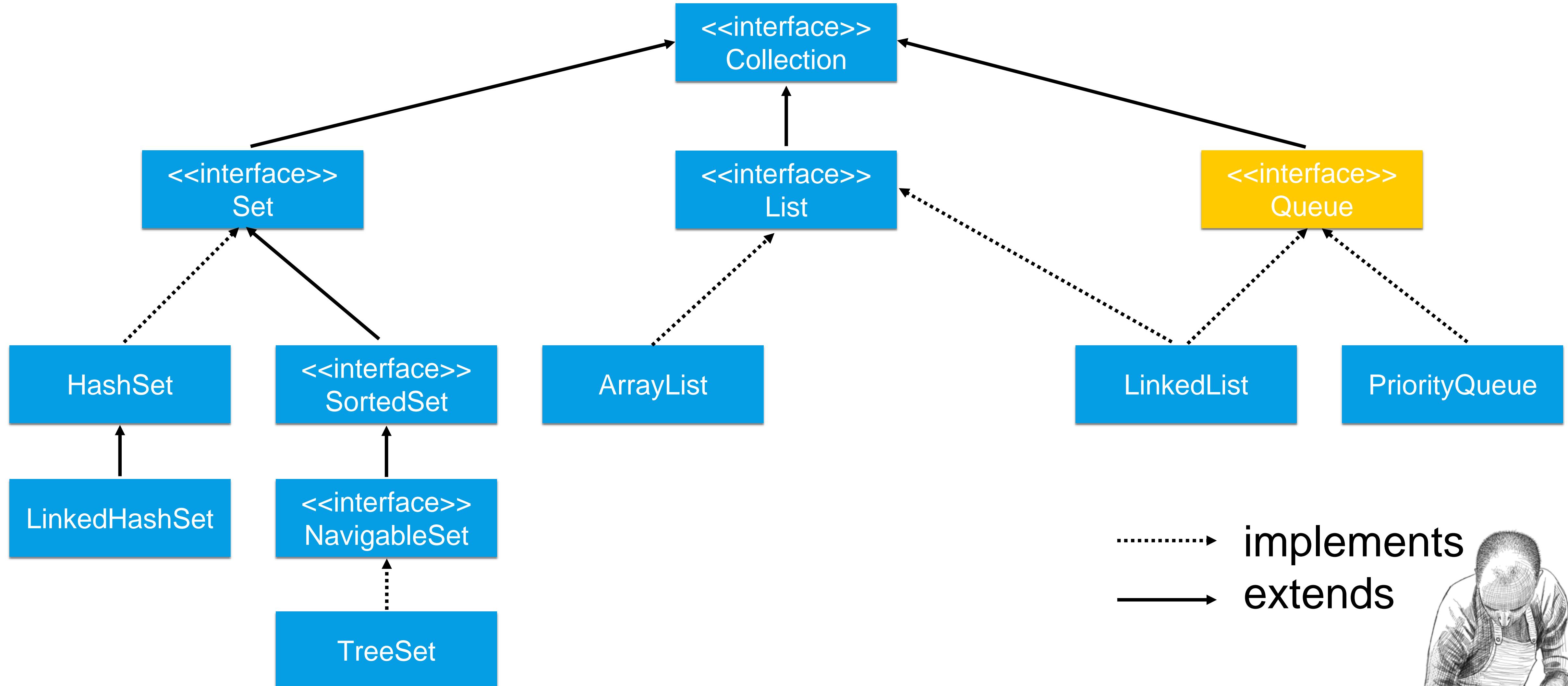
# Collection Interface



Which makes it easy to add or  
remove elements at any position in the list.



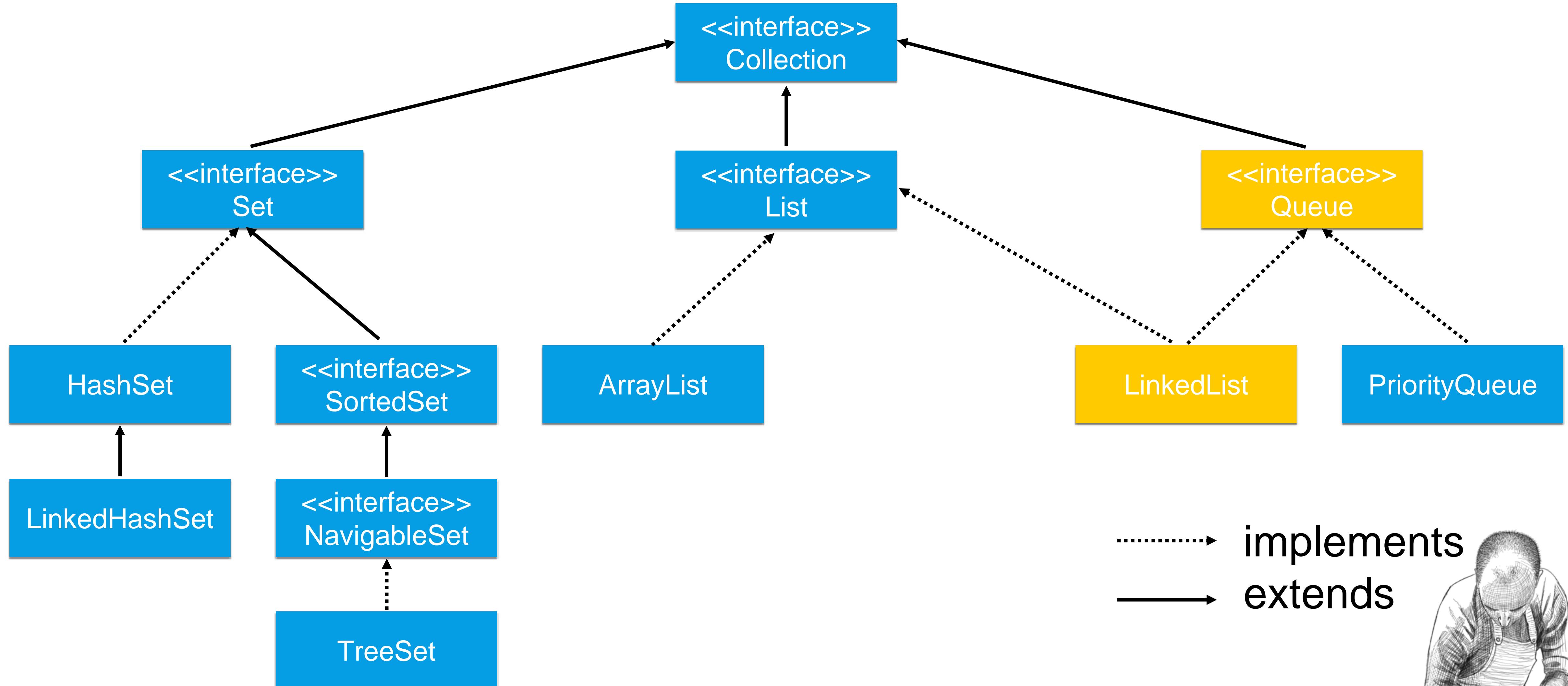
# Collection Interface



Last but not least, let's have a look at the classes implementing the Queue interface.



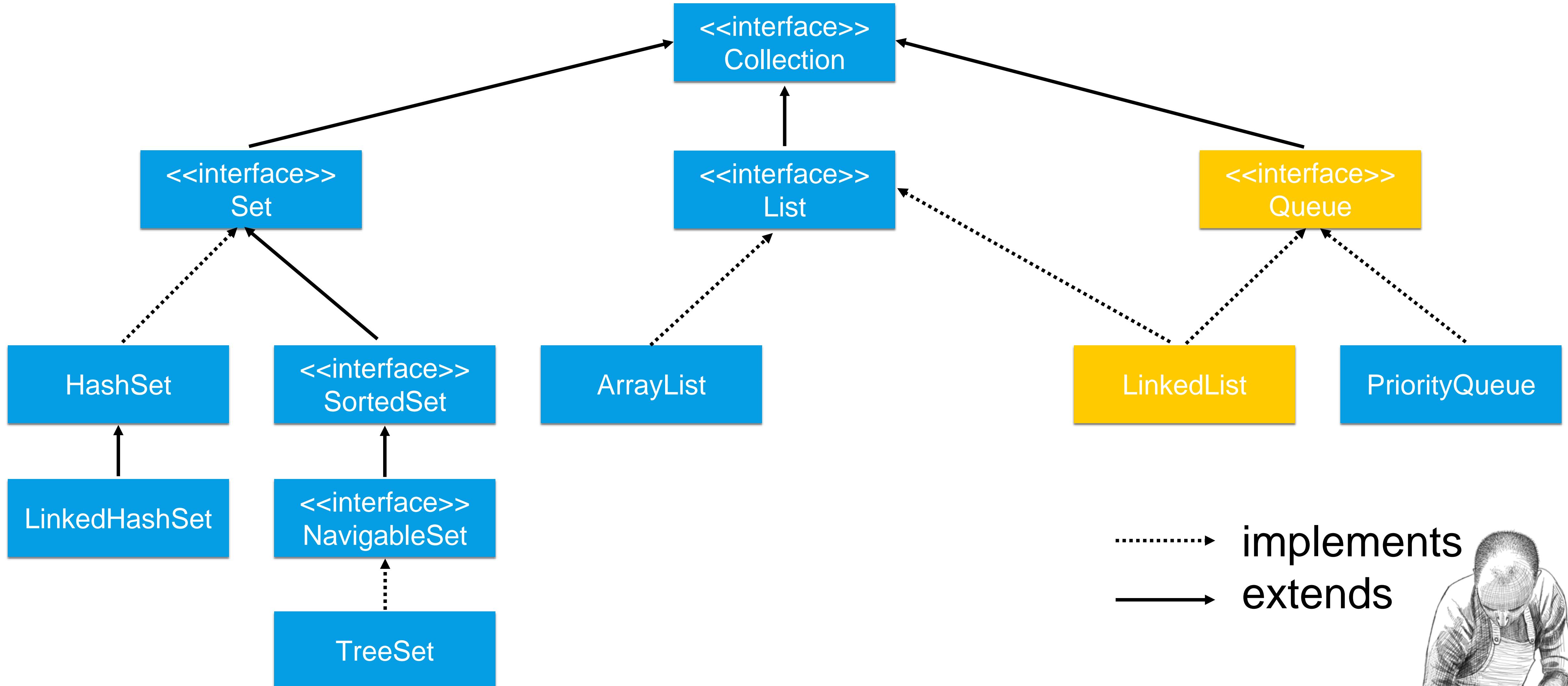
# Collection Interface



We already talked about **LinkedList**,  
as it also implements the **List** interface.



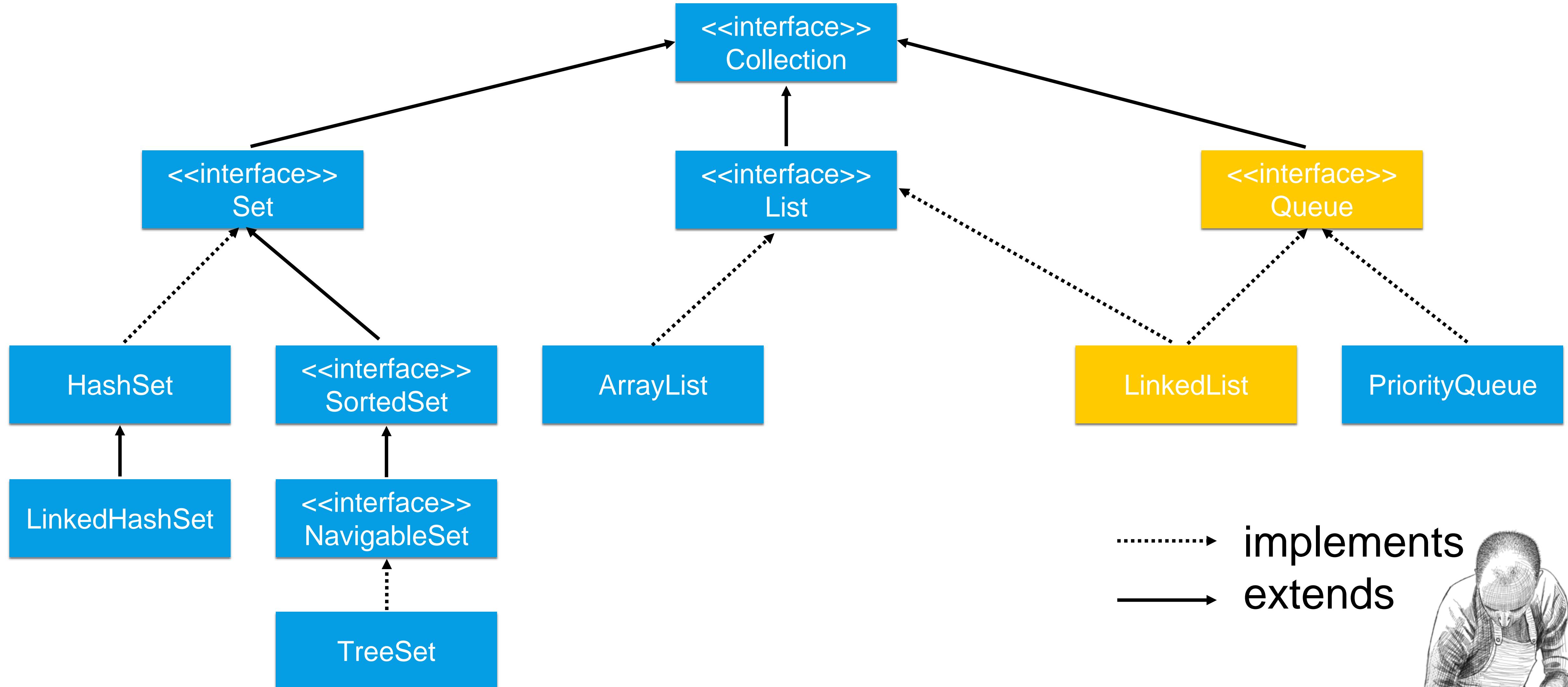
# Collection Interface



However, the fact that it is based on a DoubleLinkedList makes it quite easy to also implement the queue interface.



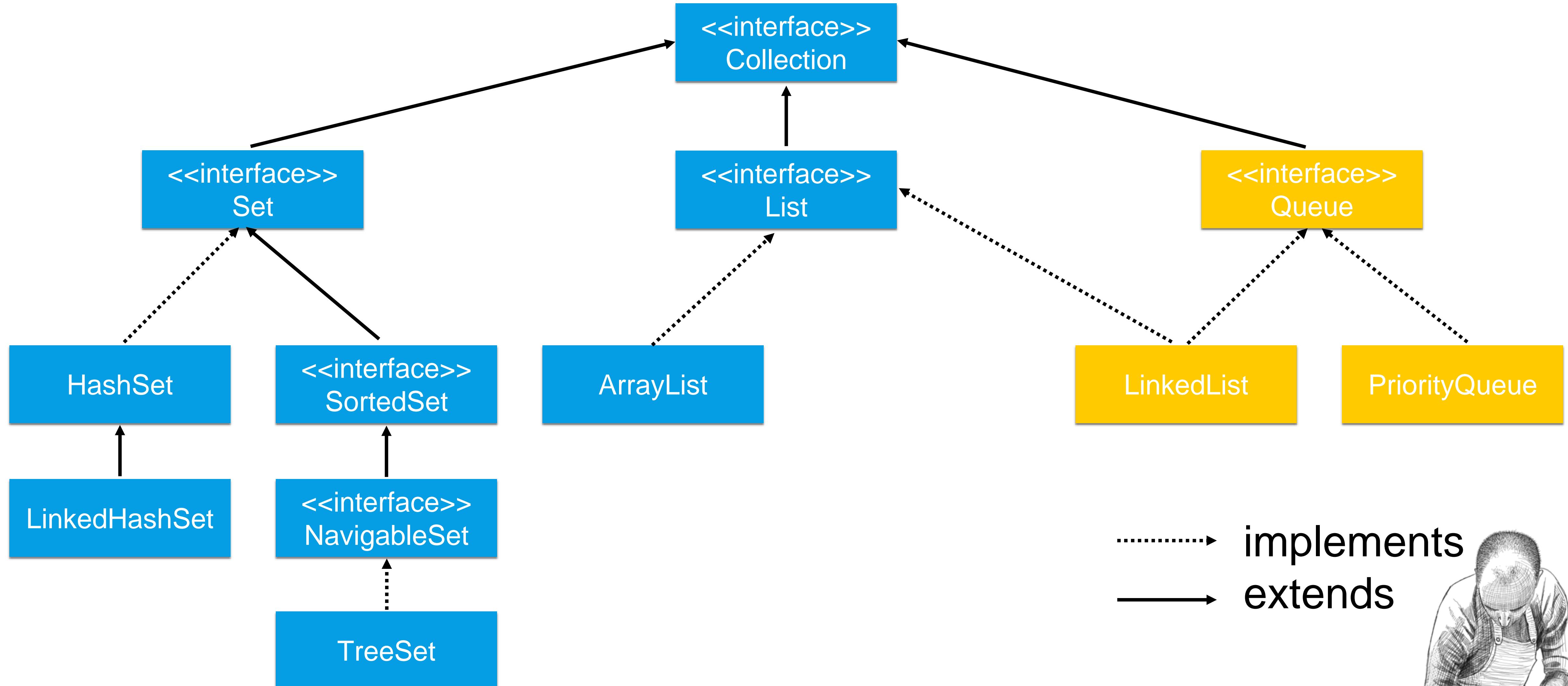
# Collection Interface



**LinkedList is the default Queue implementation.**



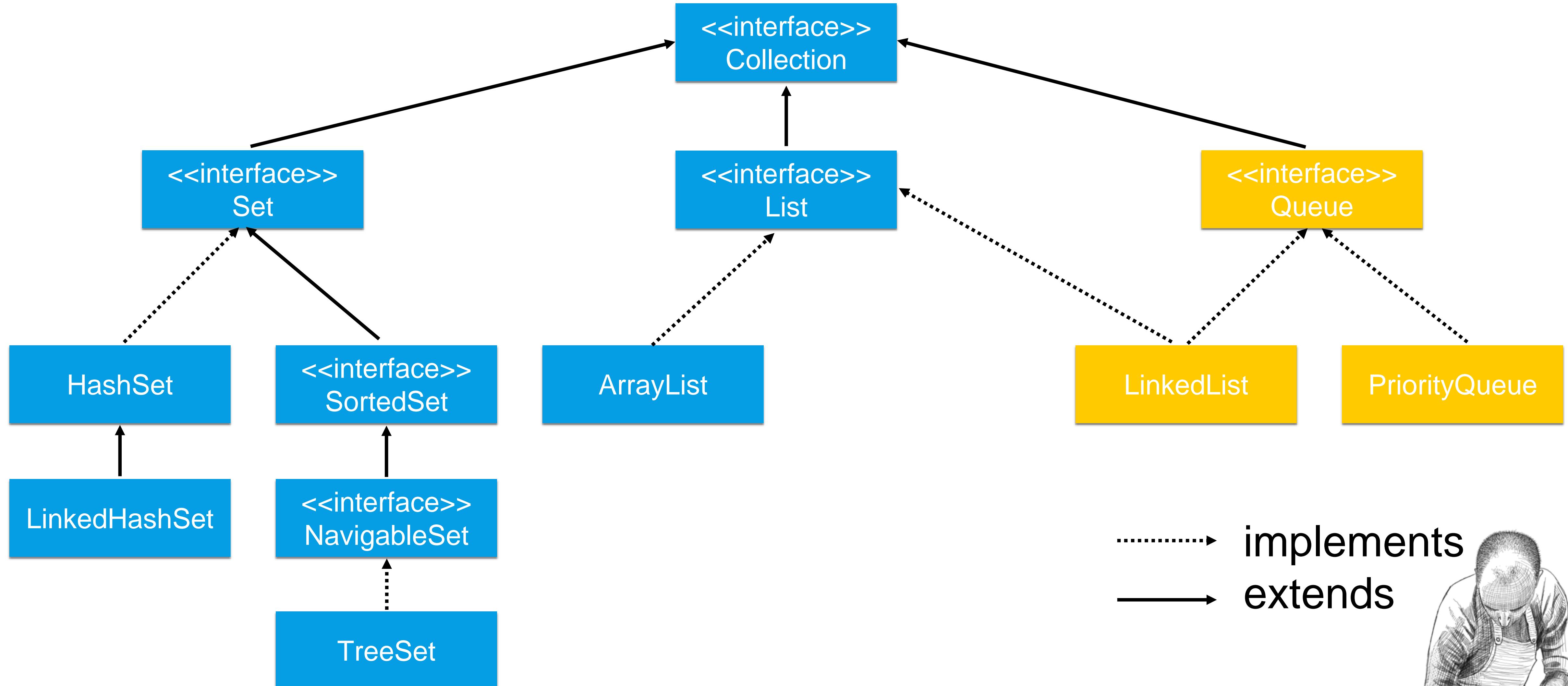
# Collection Interface



**PriorityQueue** is a **Queue** implementation that keeps its elements automatically ordered.



# Collection Interface

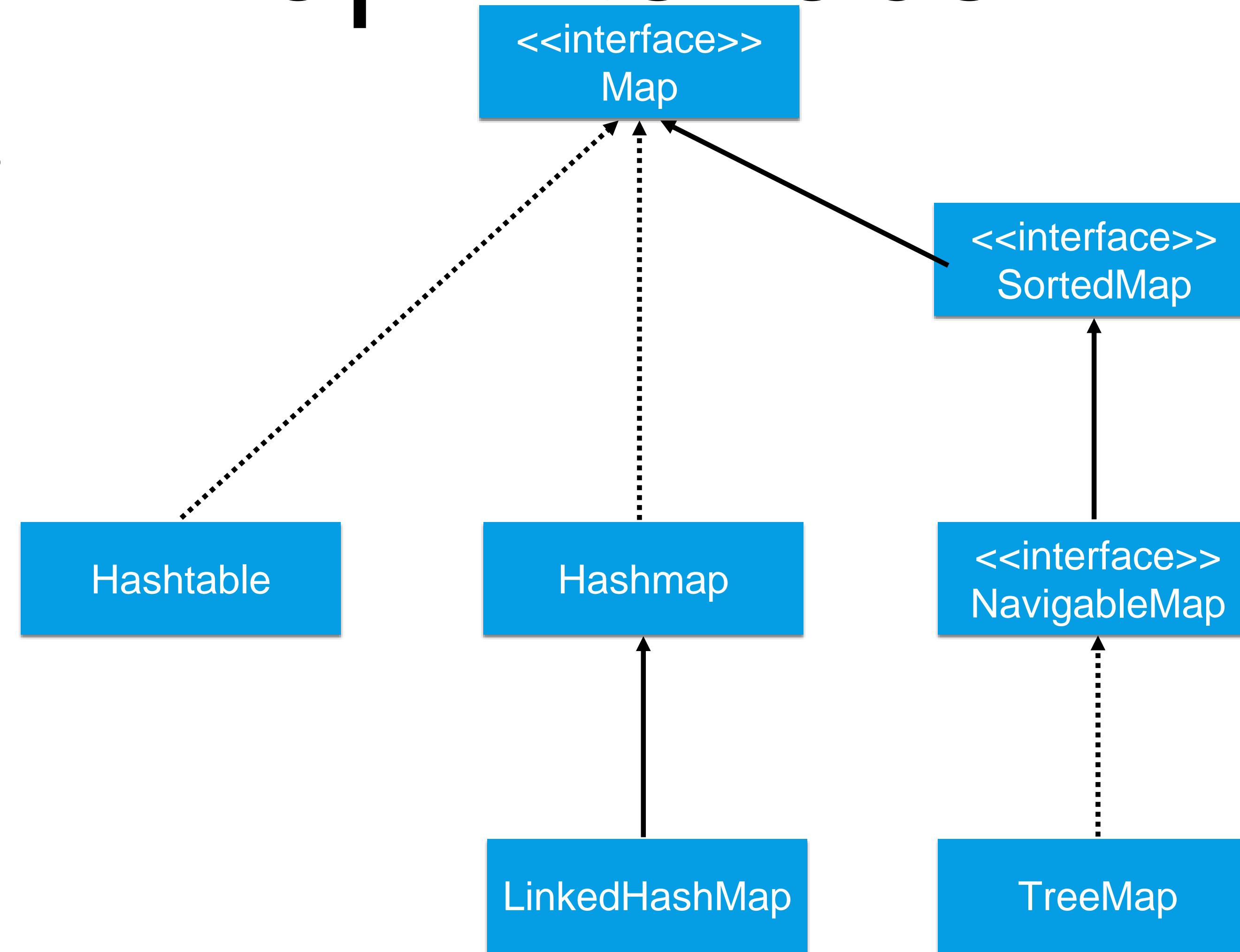


It has similar functionality like a **TreeSet**,  
but it does allow duplicate entries.



# Map Interface

..... → implements  
→ extends

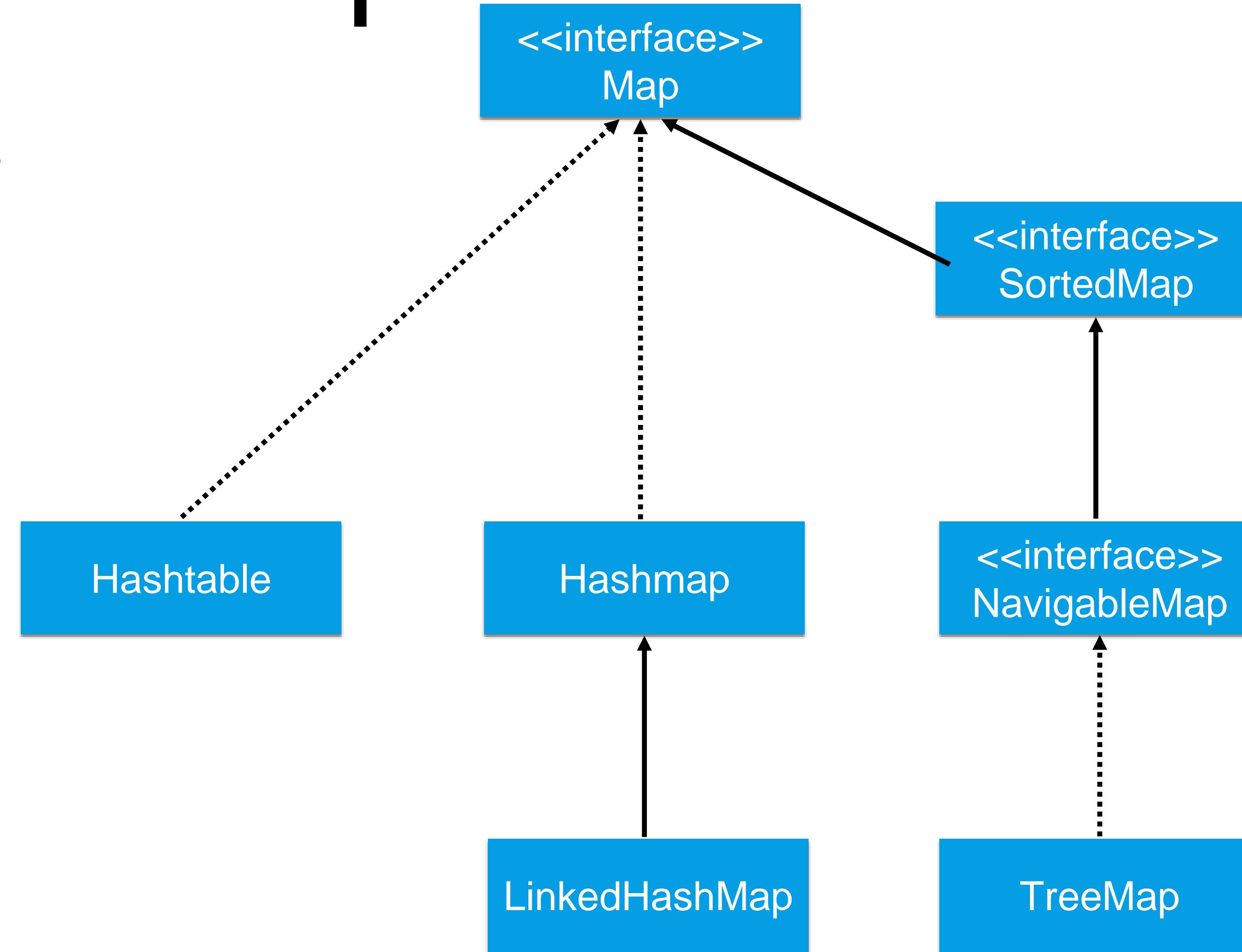


**Now let's look at the map interface.  
This interface has no relation to the Collection interface.**



# Map Interface

..... → implements  
→ extends

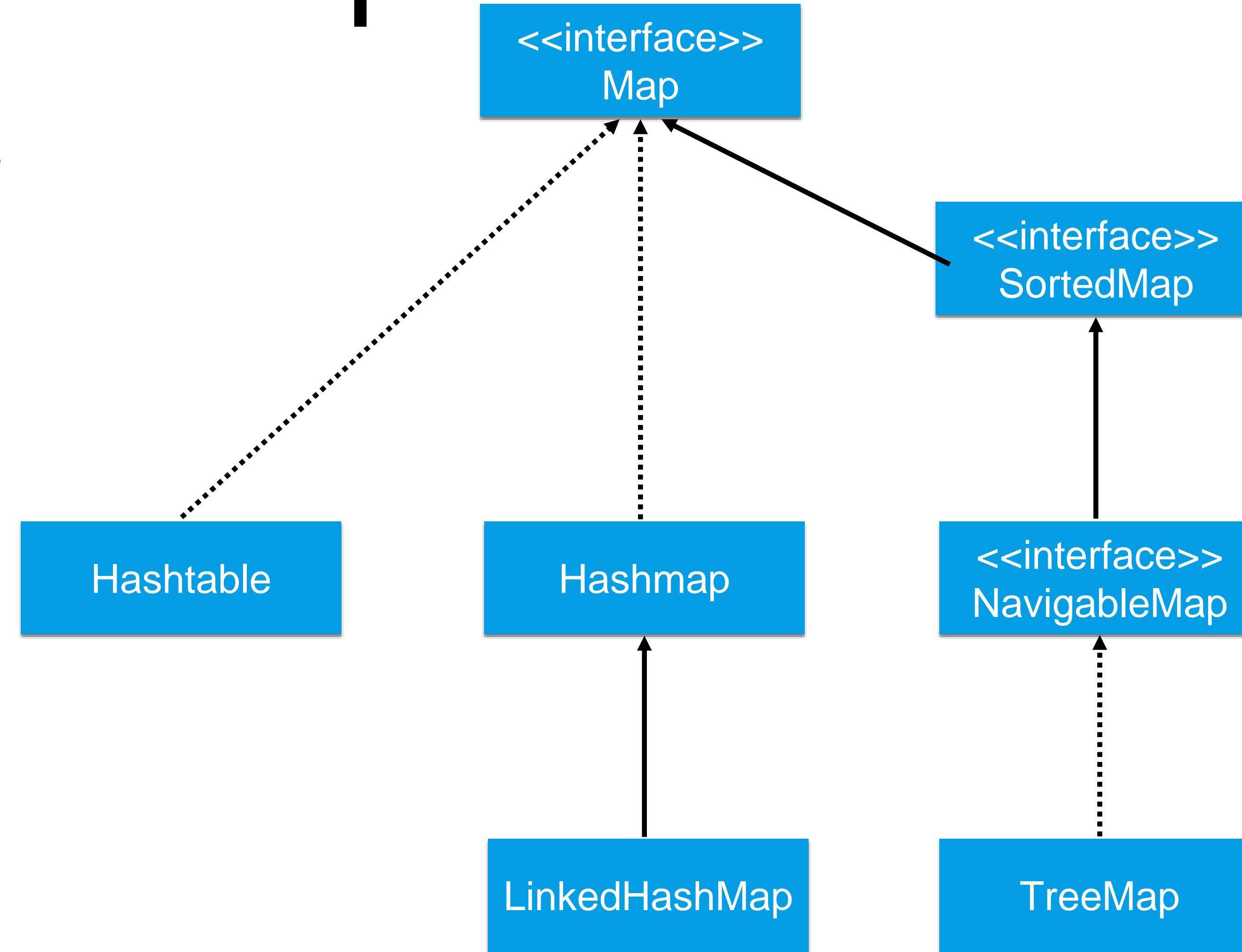


**A Collection operates on one entity,  
while a map operates on two entities -**



# Map Interface

..... → implements  
→ extends

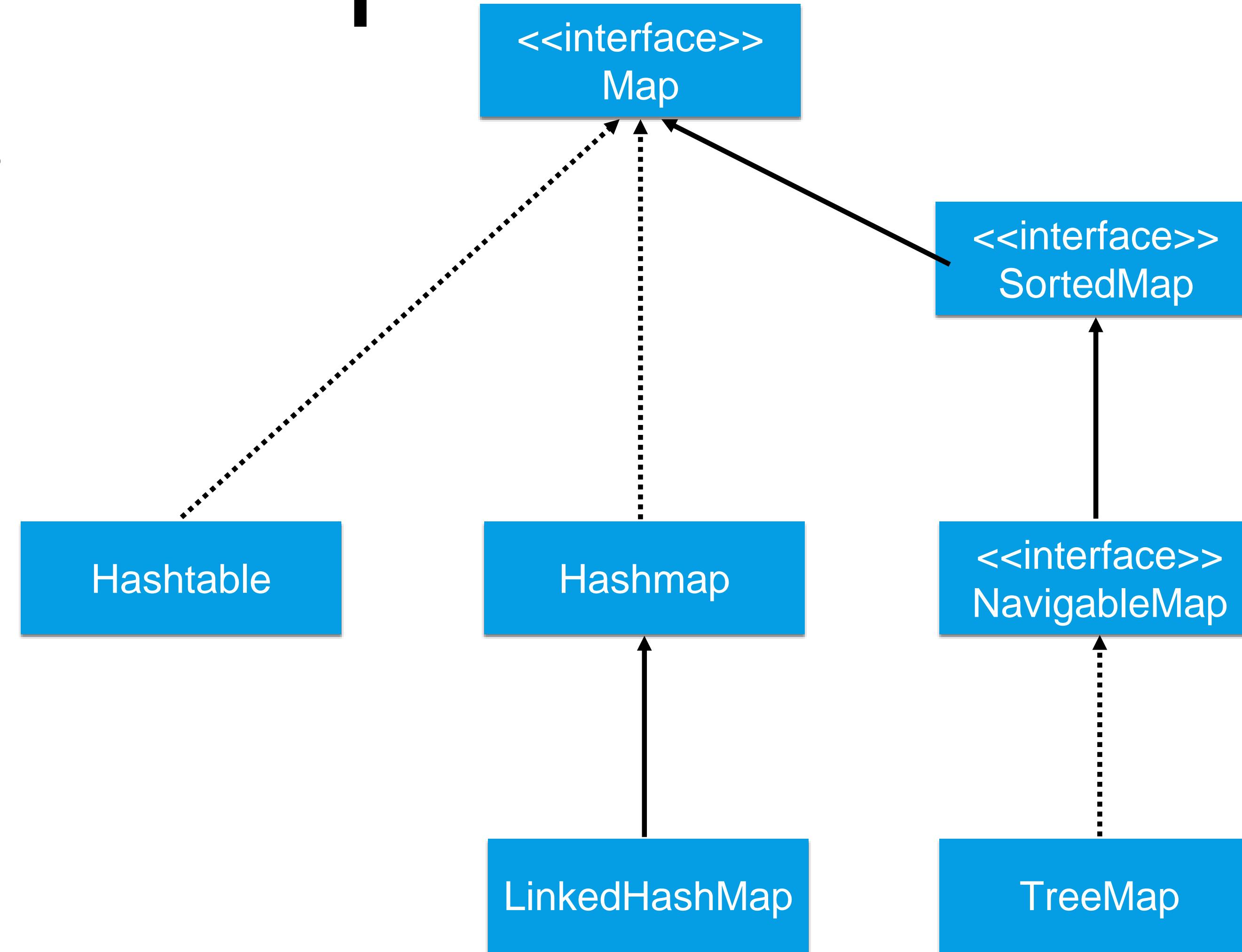


- A unique key, for example a Vehicle Identification Number,
- and an object that is related to this key, for example a car object.



# Map Interface

..... → implements  
→ extends

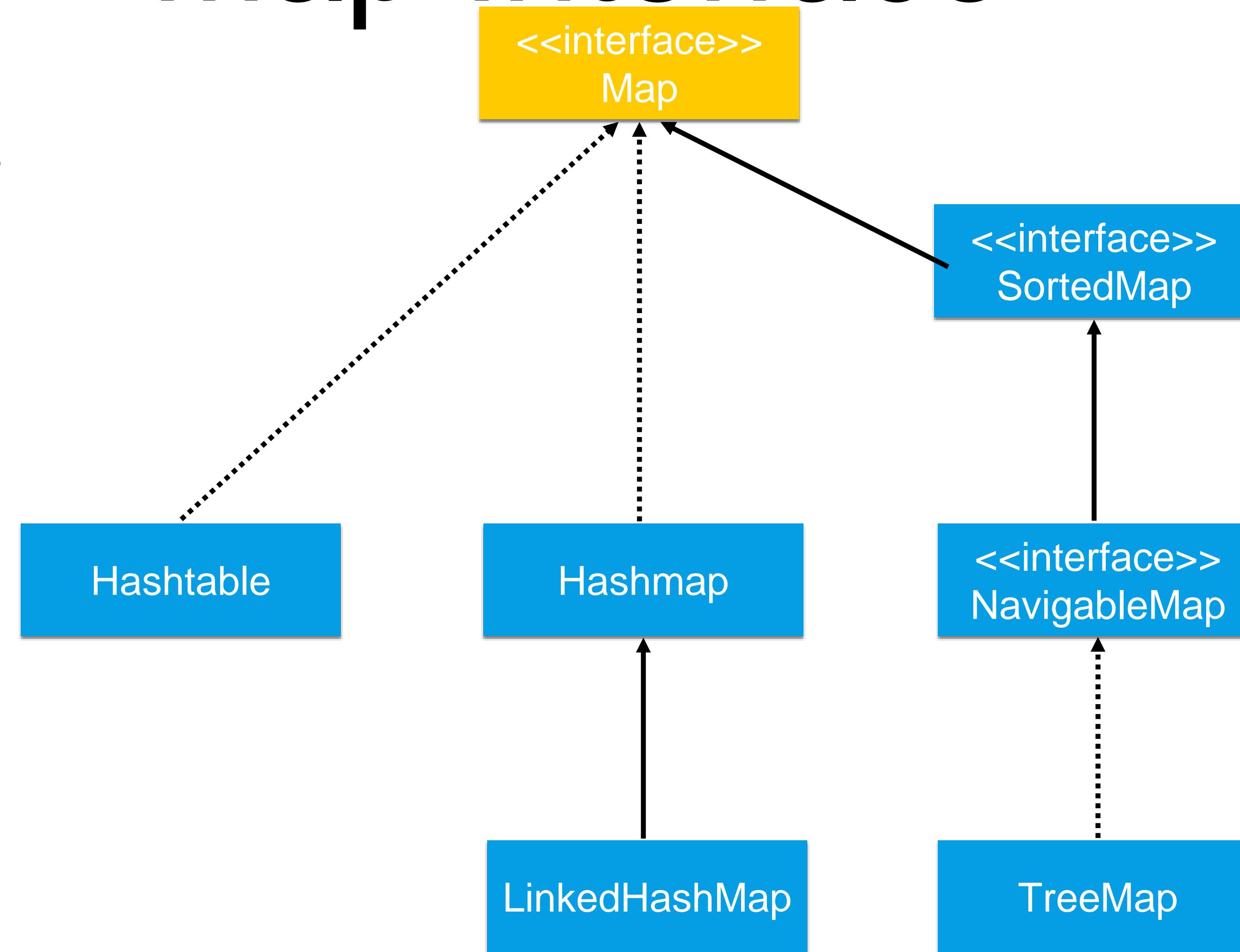


**With the help of the key you can retrieve the object it relates to.**

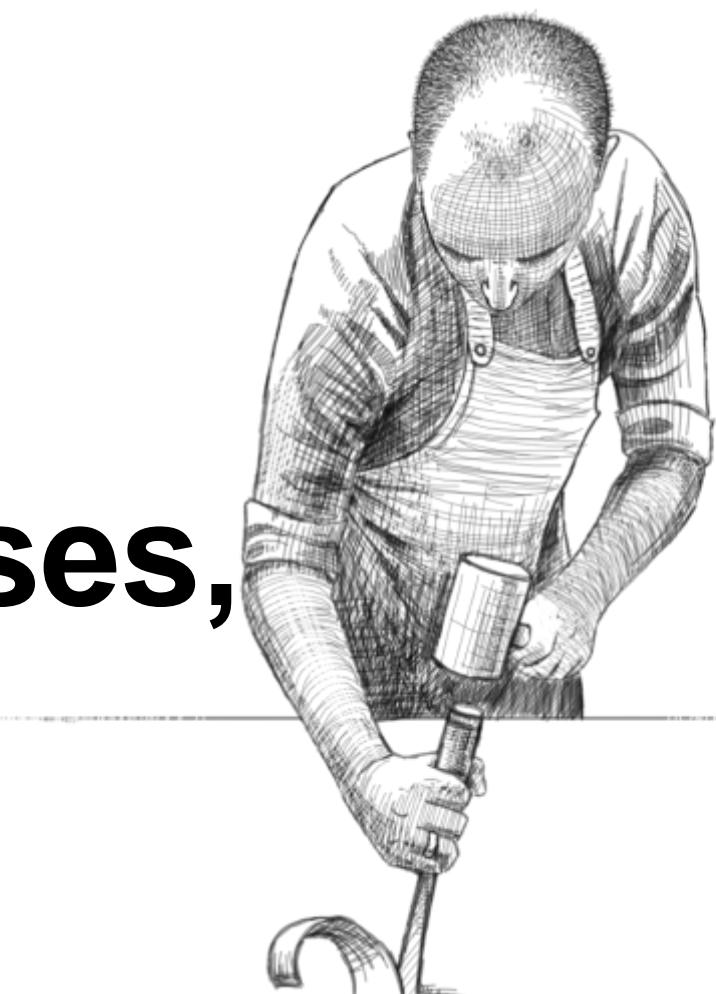


# Map Interface

..... → implements  
→ extends

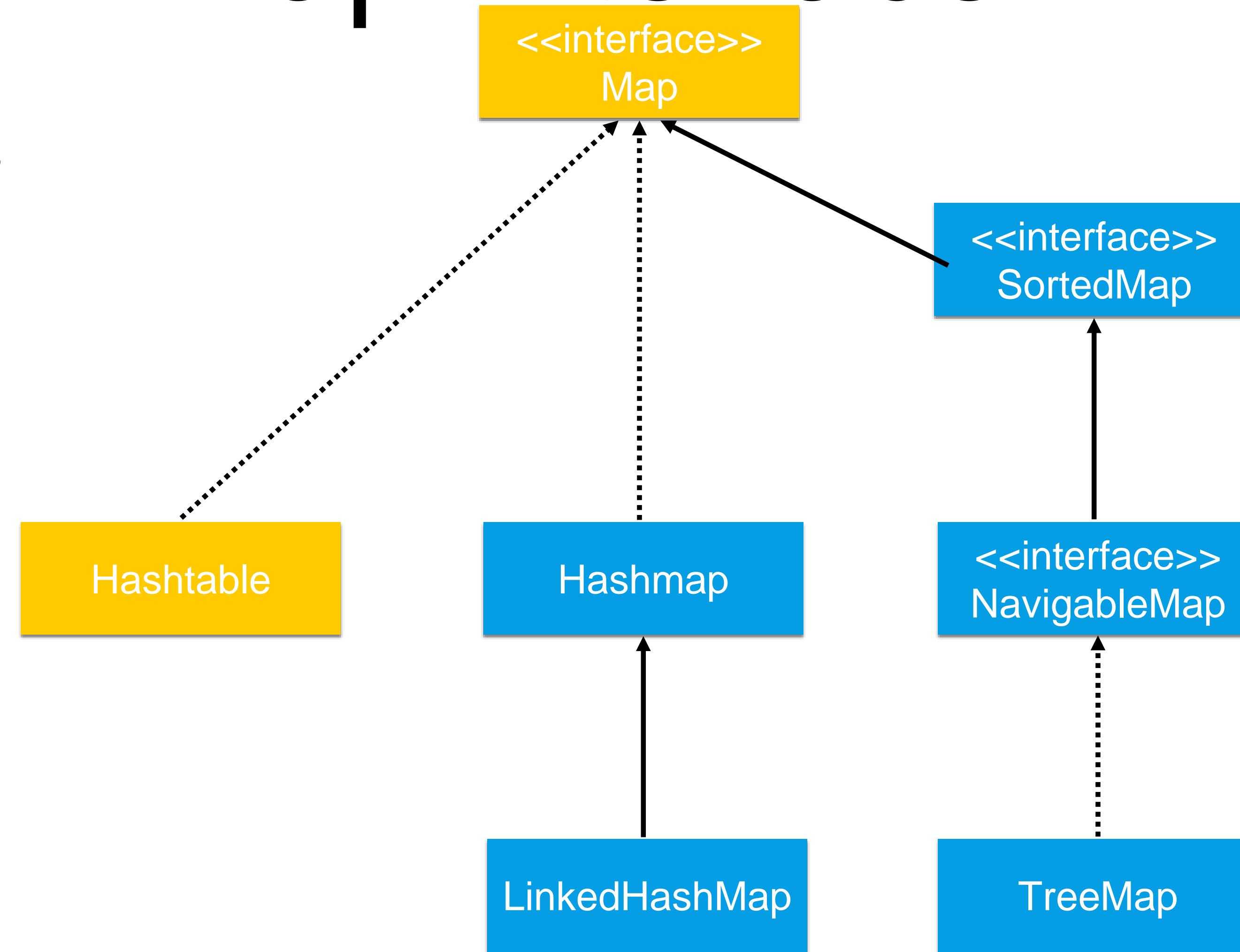


**The interface map is the root of a lot of interfaces and classes, which we will look at now.**



# Map Interface

..... → implements  
→ extends

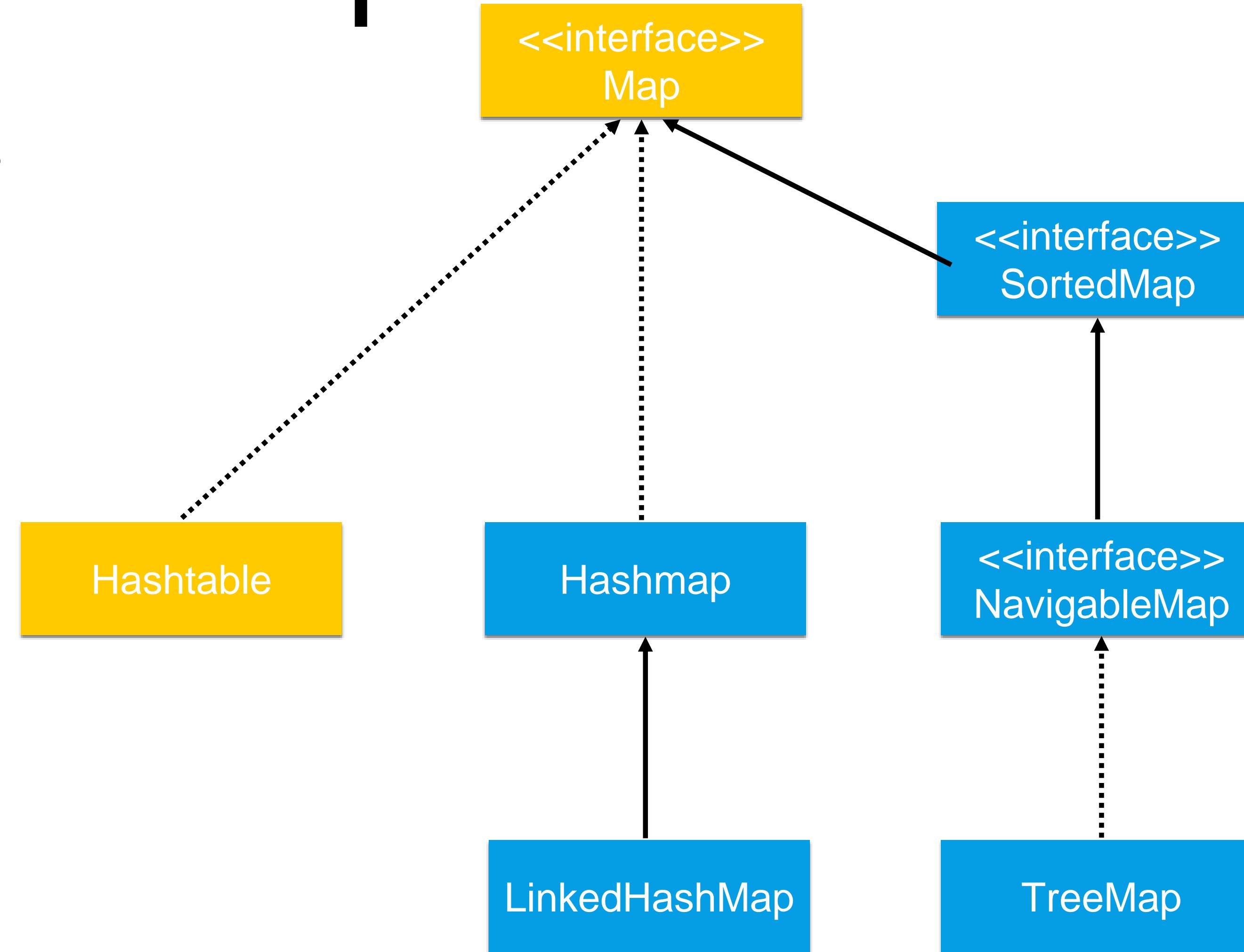


**The class `Hashtable` was the first collection in Java JDK1 that was based on the data structure hashtable,**



# Map Interface

..... → implements  
→ extends

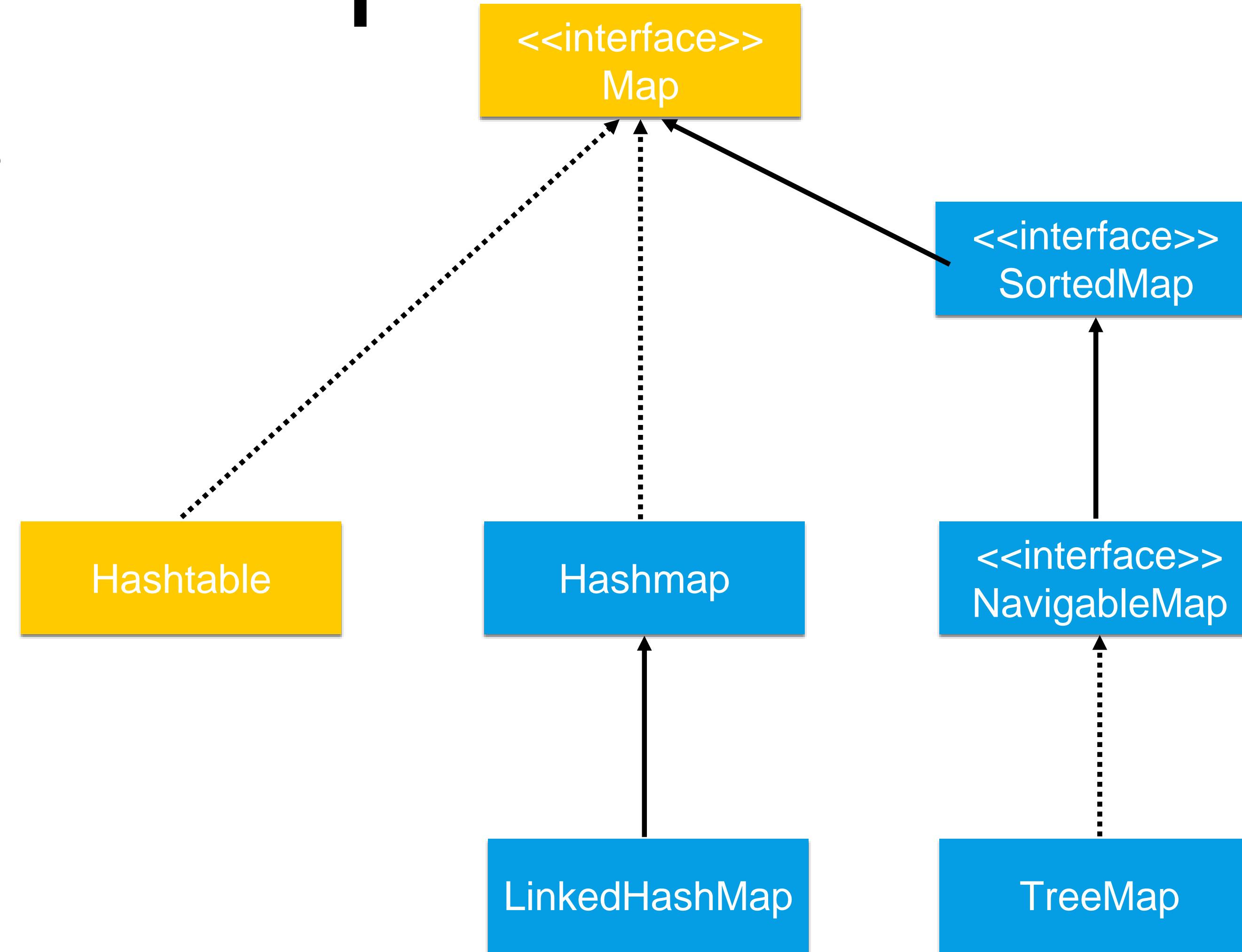


so the Java creators called it **Hashtable**.



# Map Interface

..... → implements  
→ extends

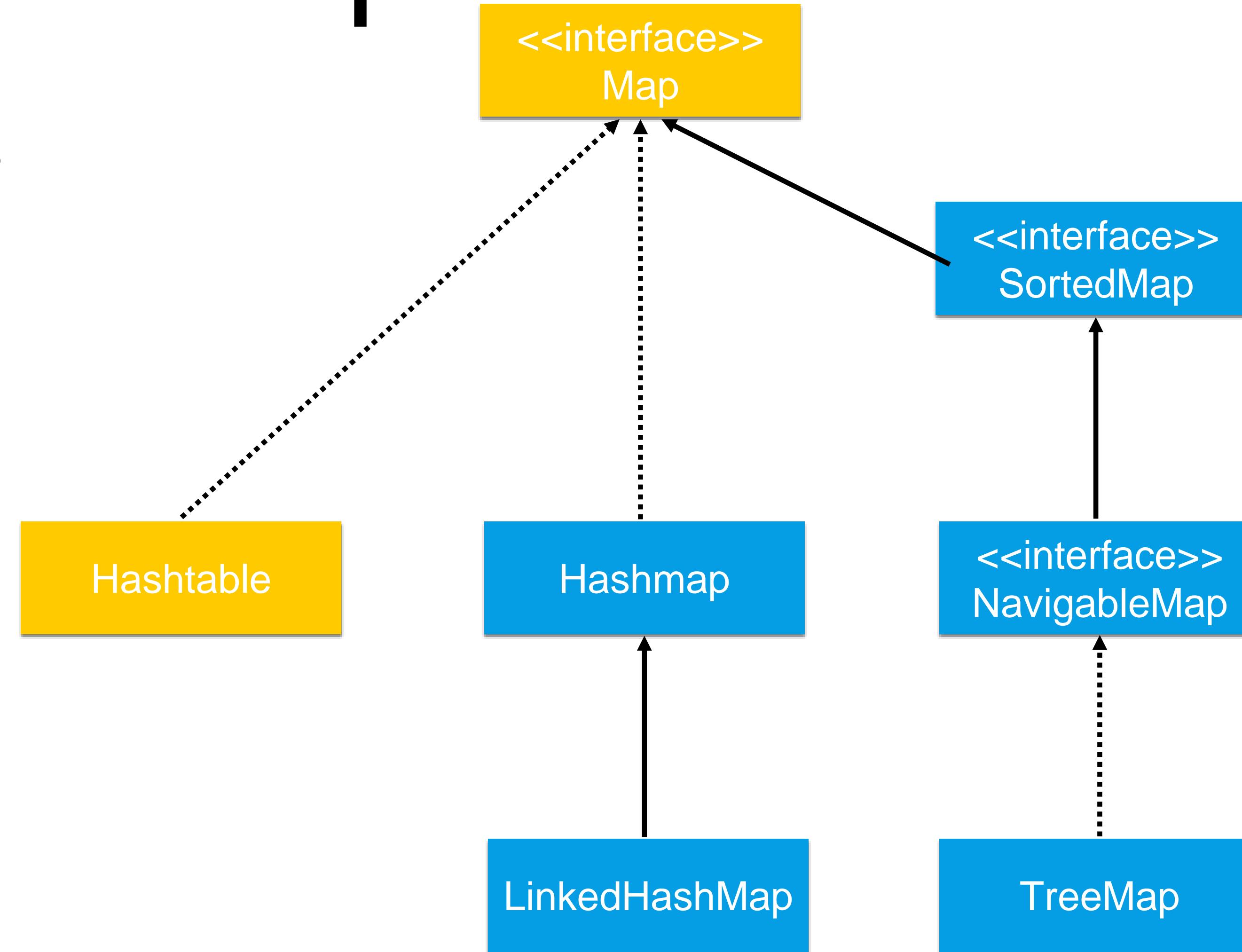


**Unfortunately, this makes it hard to differentiate between the two.**



# Map Interface

..... → implements  
→ extends

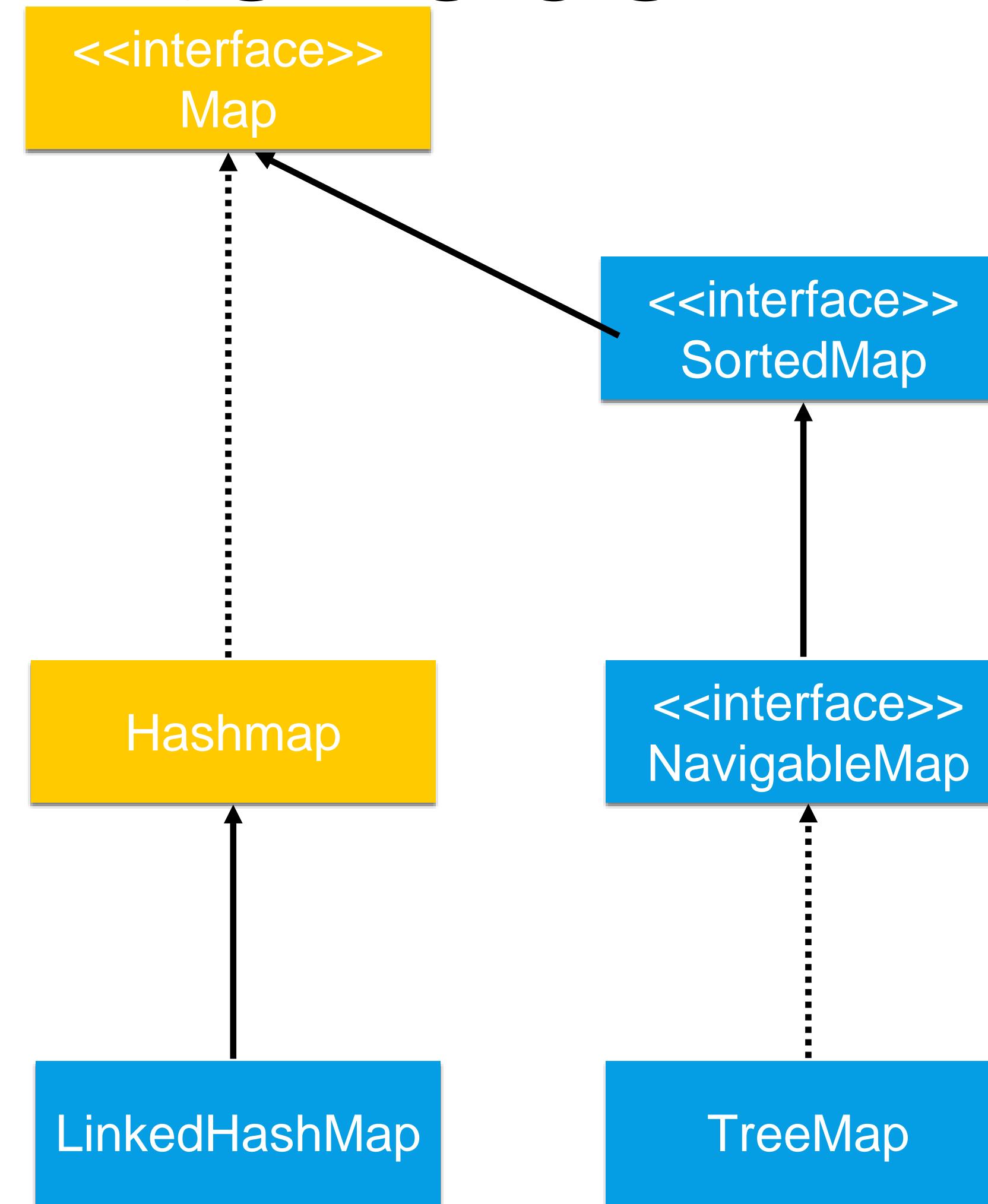


**Like Vector, the class is deprecated because of its suboptimal performance**



# Map Interface

..... → implements  
→ extends

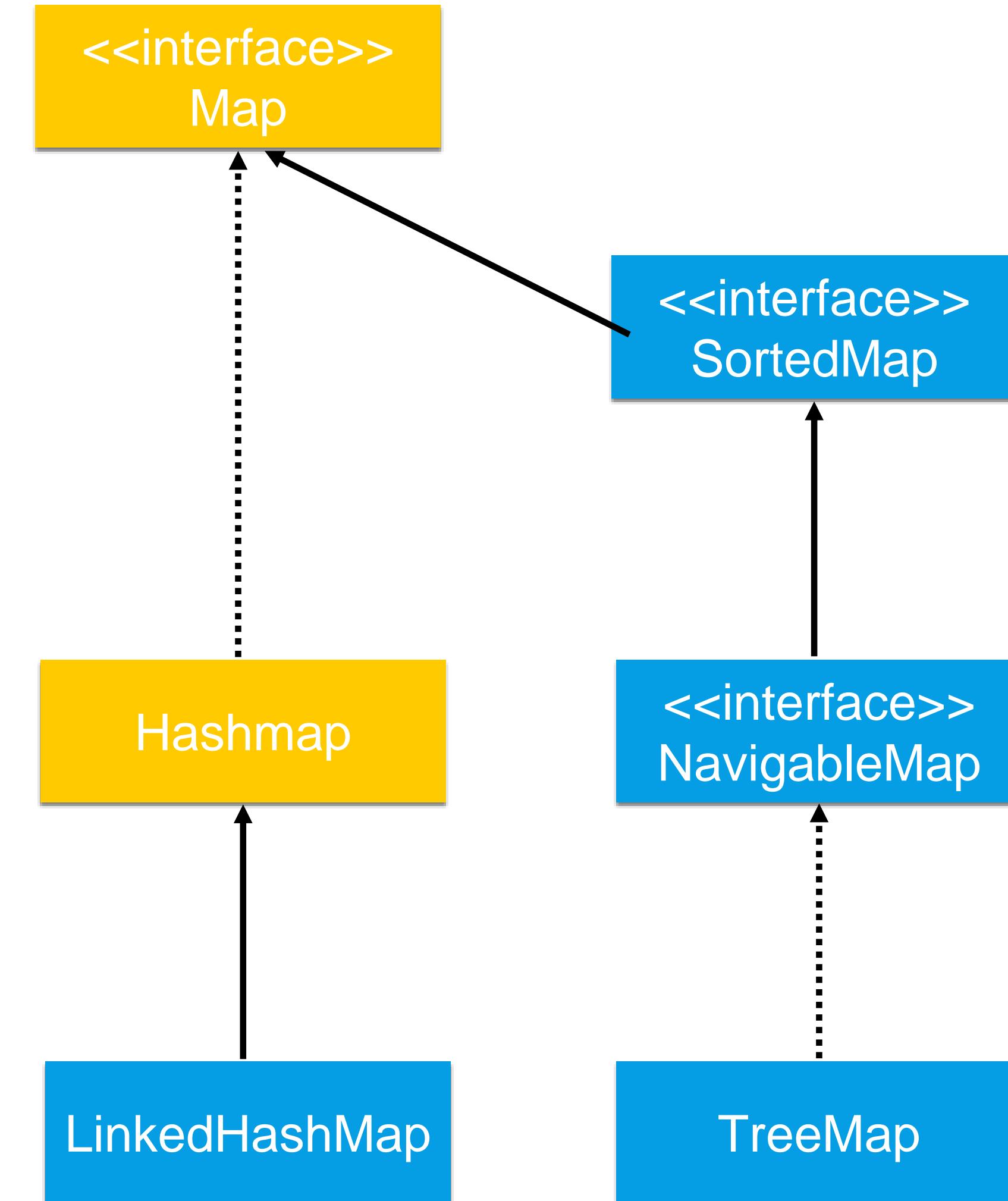


**so let's remove and forget about it, too.  
Instead, use one of the other classes that implement  
the map interface.**



# Map Interface

.....→ implements  
→ extends

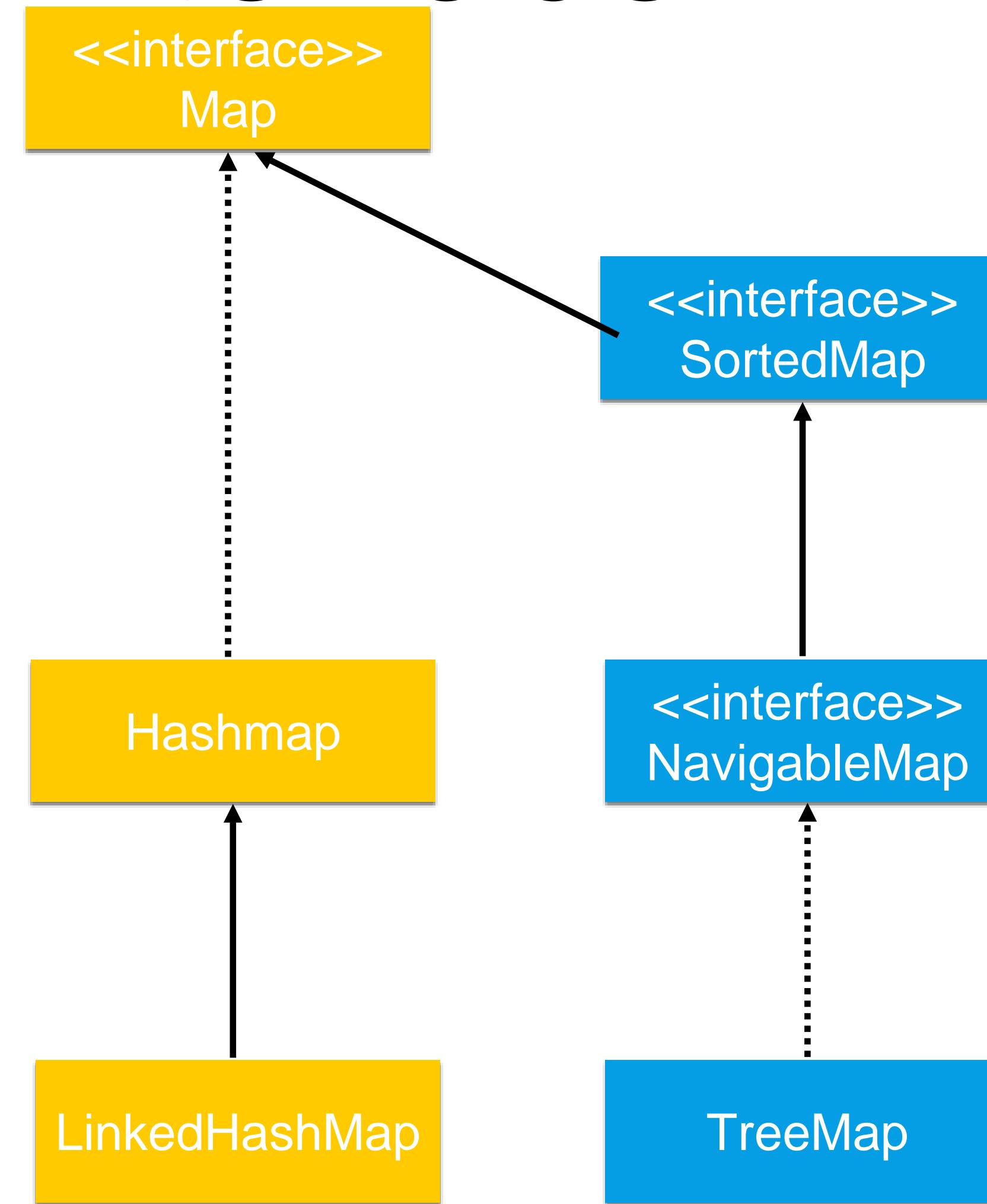


**HashMap is the default implementation that you should use in the majority of cases.**



# Map Interface

.....→ implements  
→ extends

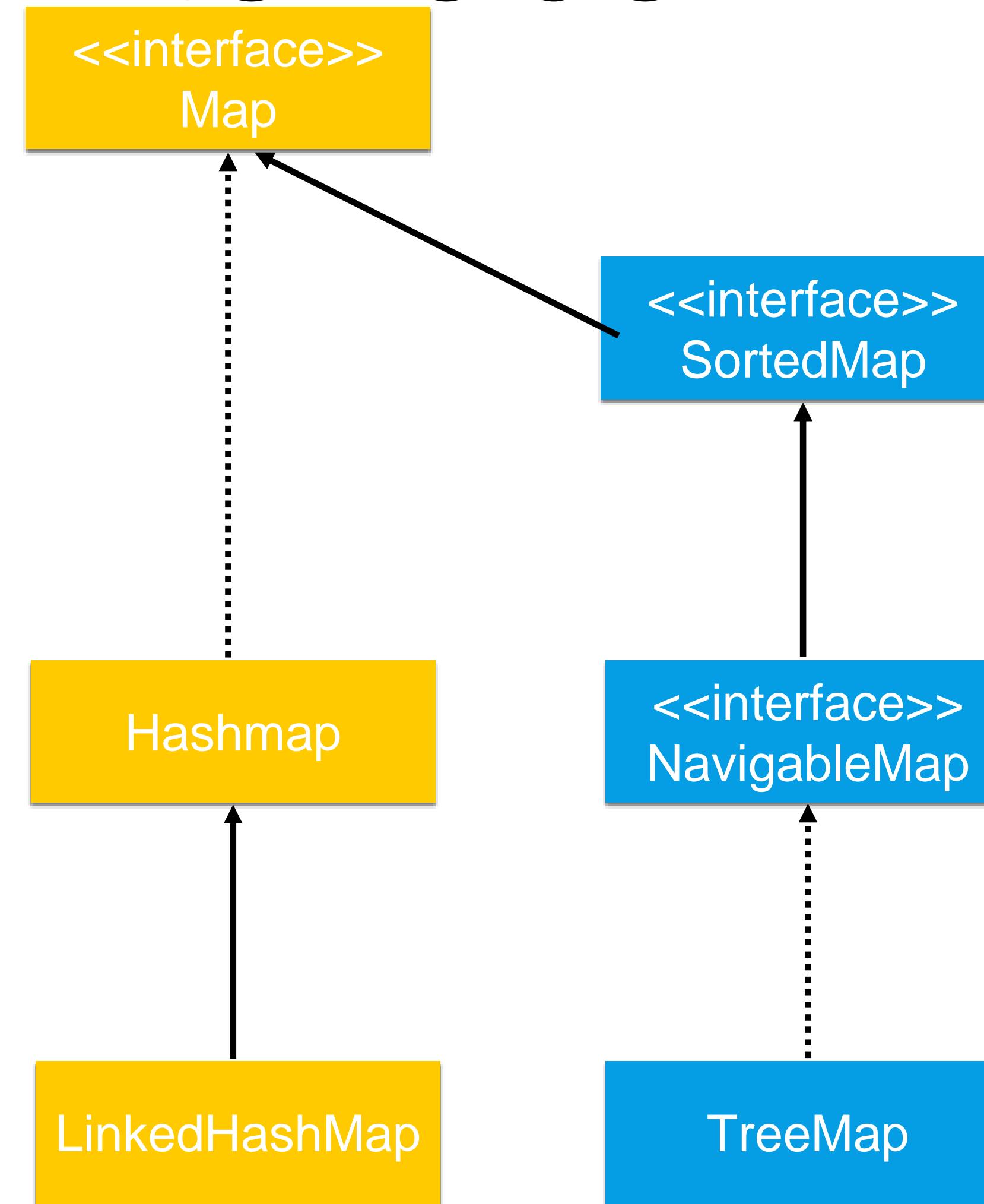


**A Map usually does not make any guarantees on how it internally stores its elements.**



# Map Interface

..... → implements  
→ extends

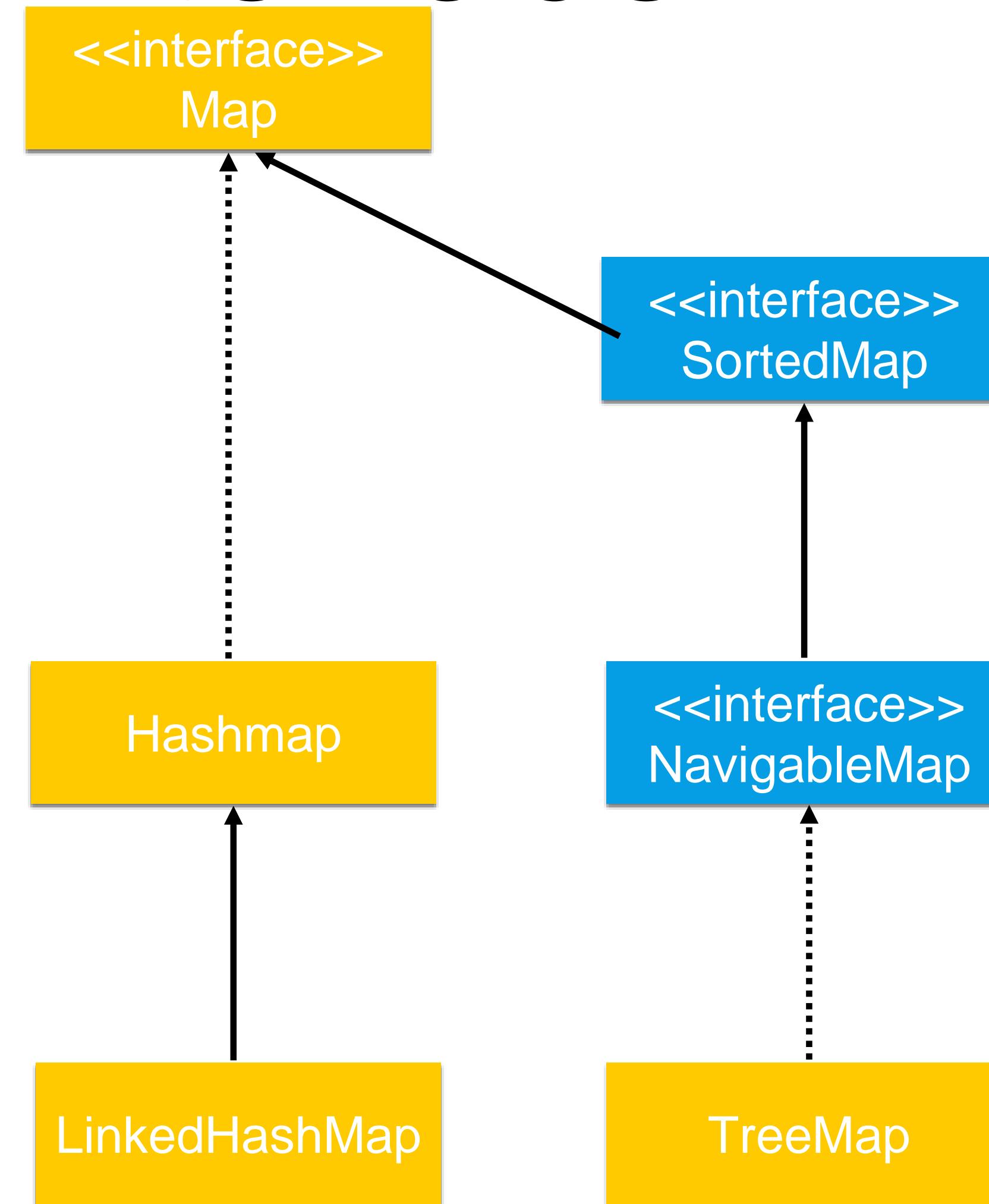


An exception to this rule is `LinkedHashMap`,  
which allows to iterate the map in the order of insertion.



# Map Interface

.....→ implements  
→ extends



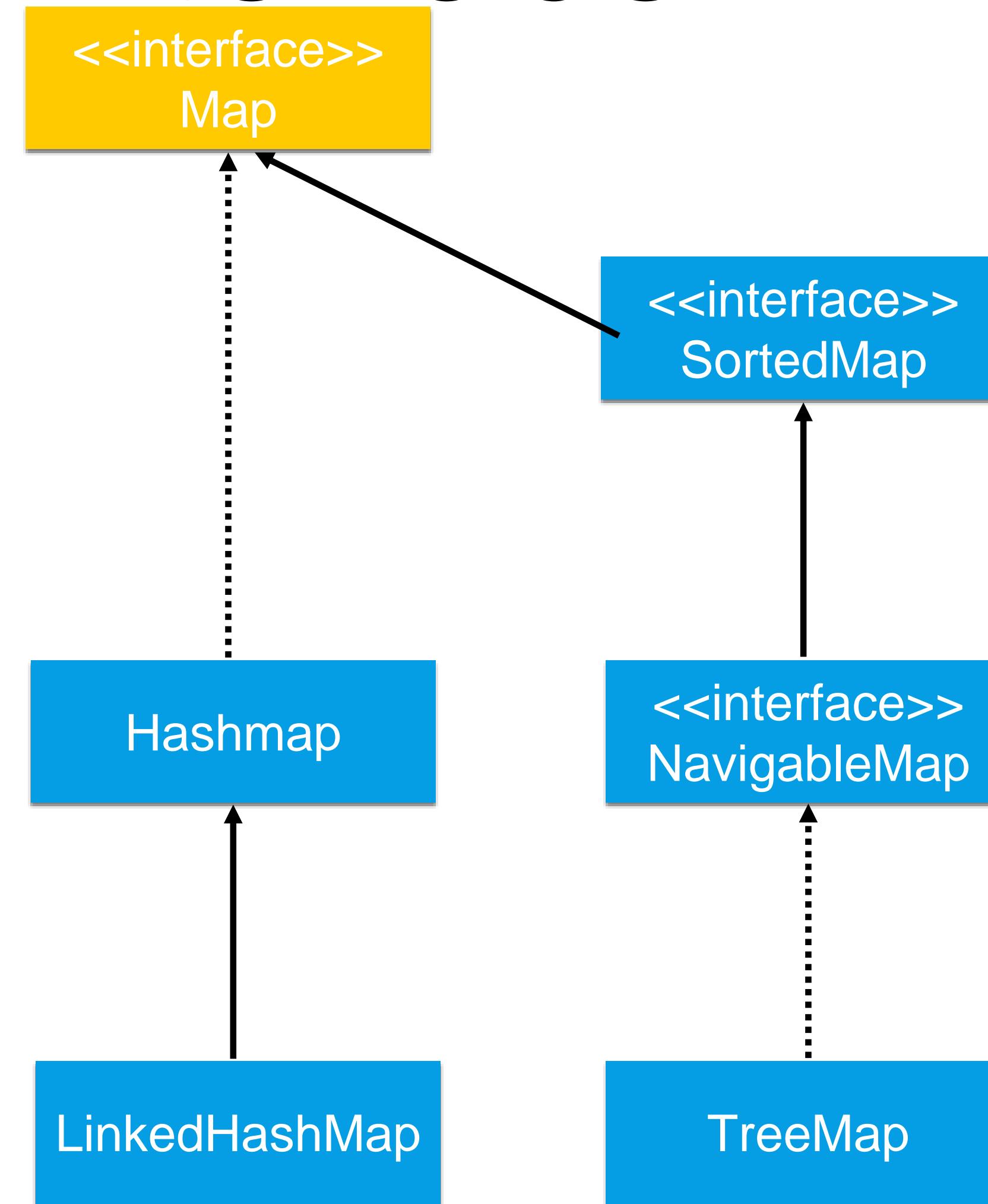
**Last but not least, TreeMap is a constantly sorted map.**



Property	HashMap	TreeMap	LinkedHashMap
Iteration Order	no guarantee order will remain constant over time	sorted according to the natural ordering	insertion-order
Get/put remove containsKey	O(1)	O(log(n))	O(1)
Interfaces	Map	NavigableMap Map SortedMap	Map
Null values/keys	allowed	only values	allowed
Fail-fast behavior	Fail-fast behavior of an iterator cannot be guaranteed impossible to make any hard guarantees in the presence of unsynchronized concurrent modification		
Implementation	buckets	Red-Black Tree	double-linked buckets
Is synchronized	implementation is not synchronized		

# Map Interface

..... → implements  
→ extends

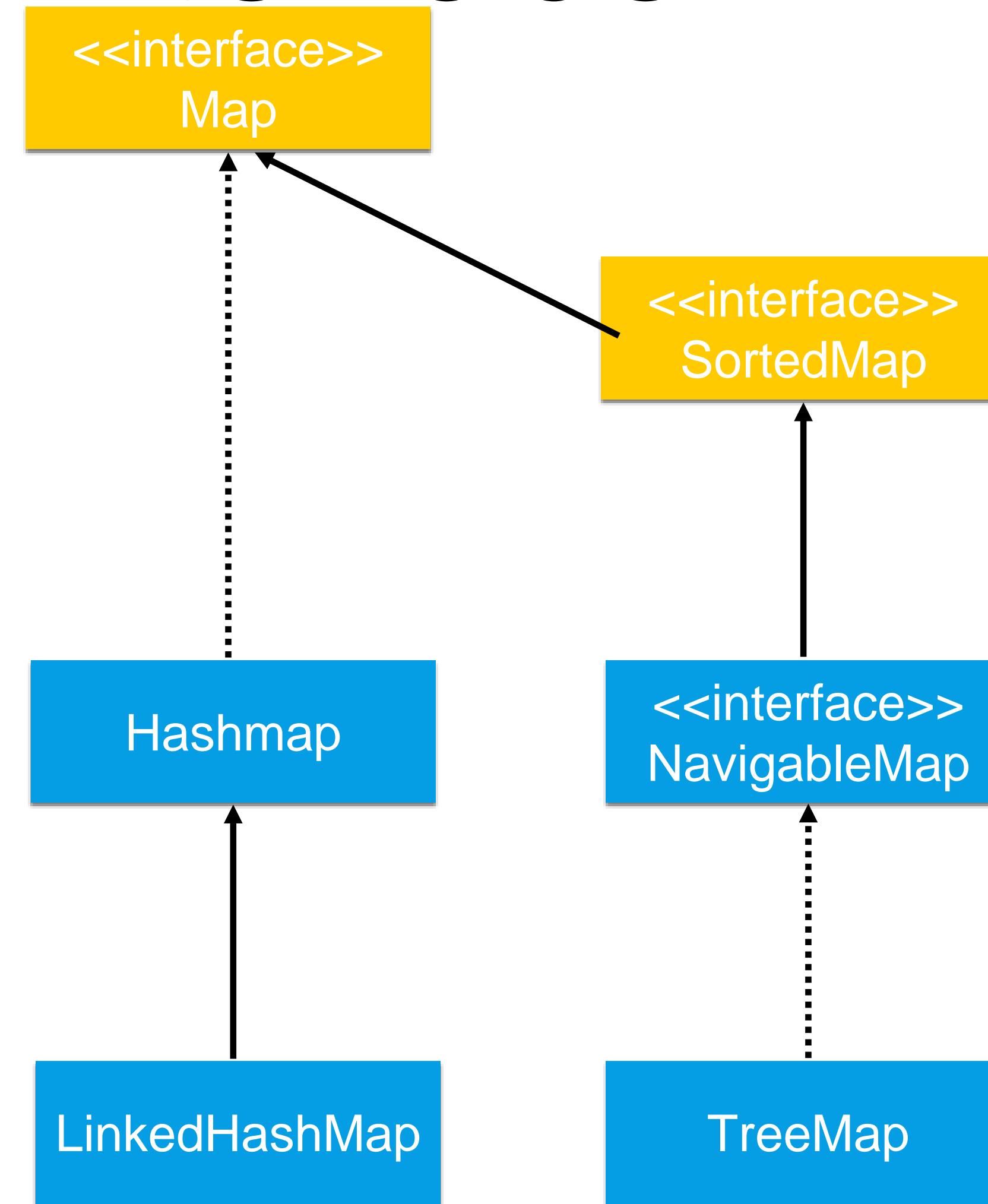


**Now let's look at the interfaces that extend the map interface.**



# Map Interface

..... → implements  
→ extends

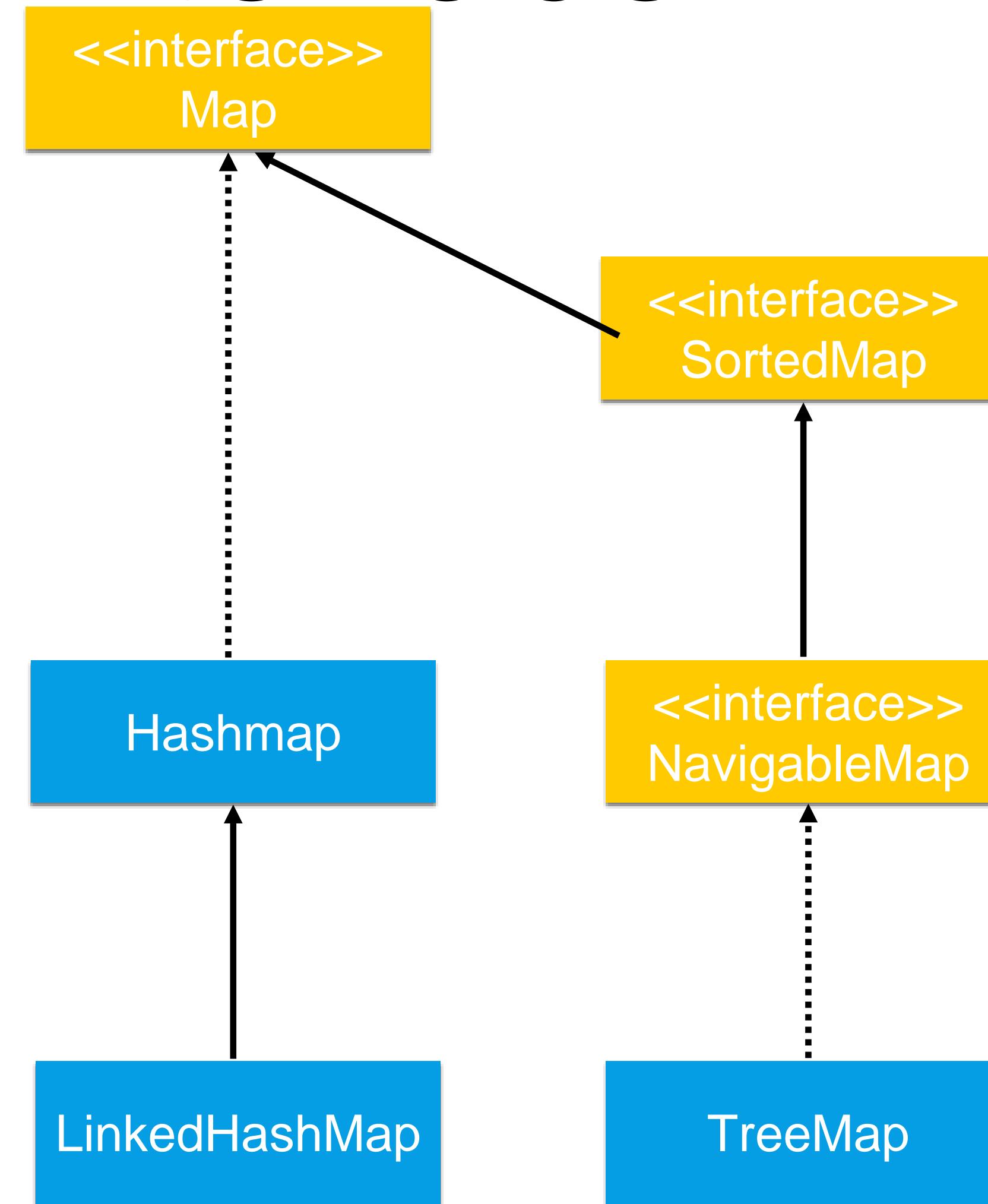


**As the name implies,  
the interface `SortedMap` extends the map interface and  
defines the contract of a constantly sorted map.**



# Map Interface

..... → implements  
→ extends

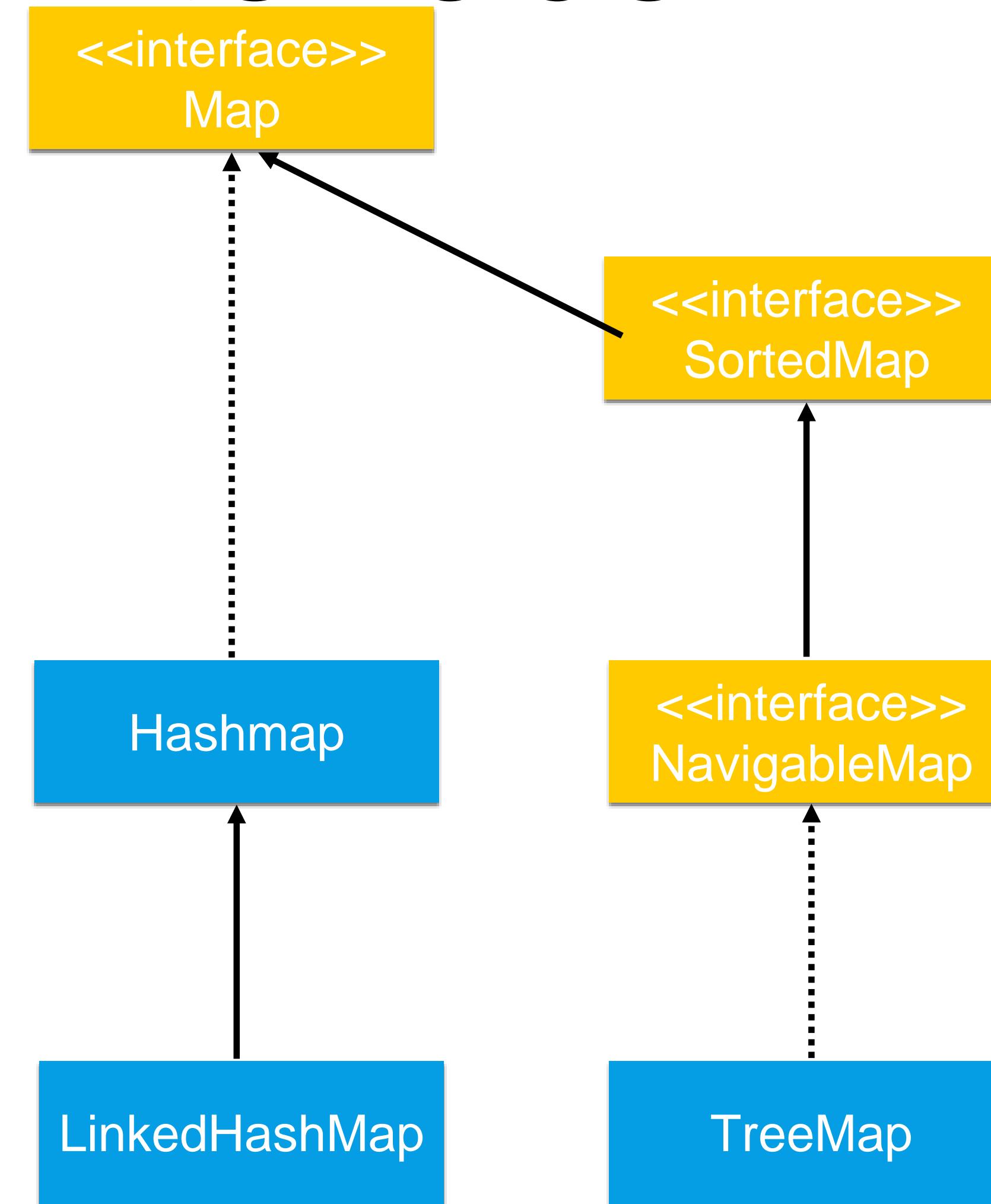


**NavigableMap again extends the SortedMap interface and adds methods to navigate through the map.**



# Map Interface

.....→ implements  
→ extends

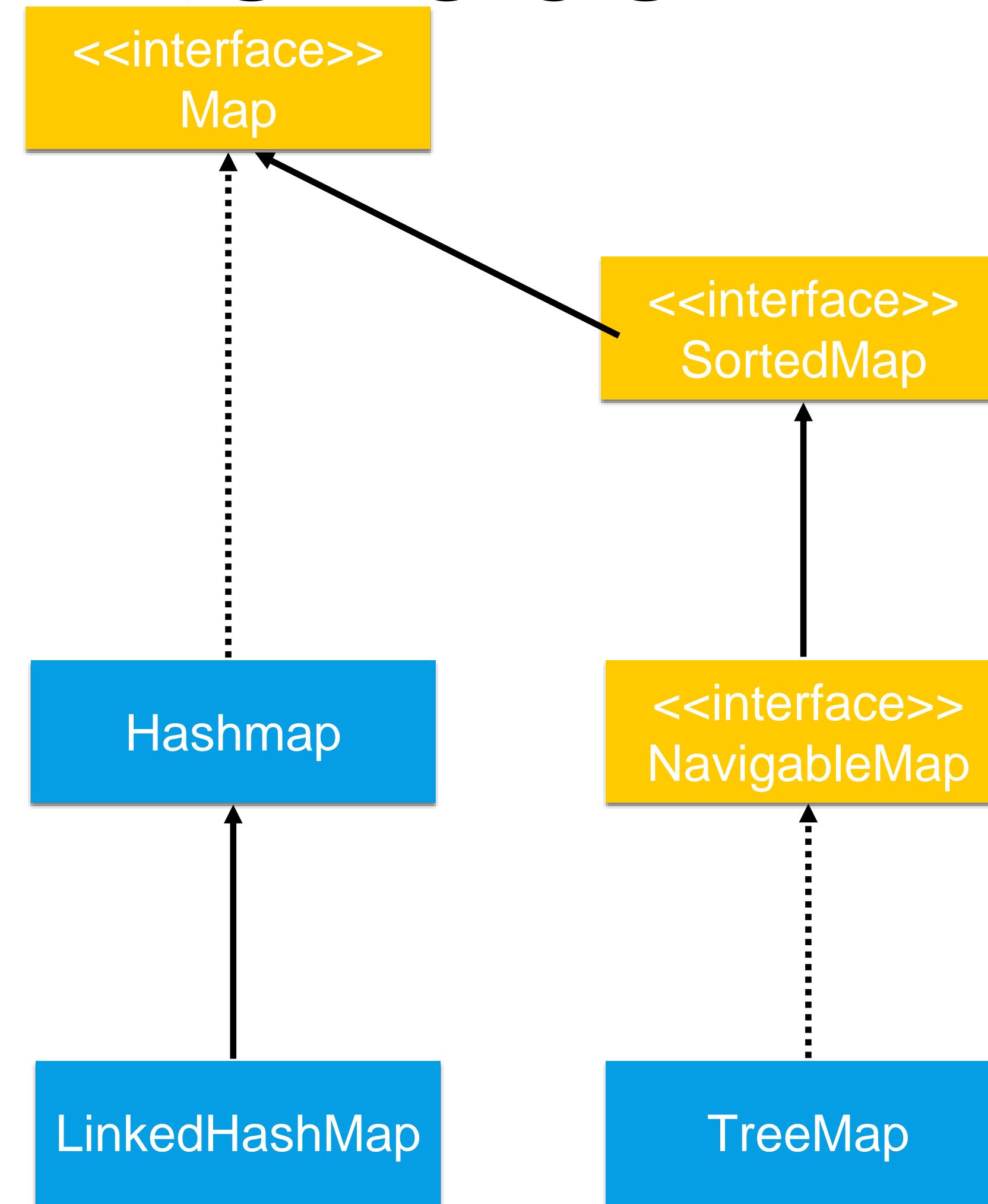


This allows you to retrieve all entries smaller or bigger then a given entry, for example.



# Map Interface

.....→ implements  
→ extends

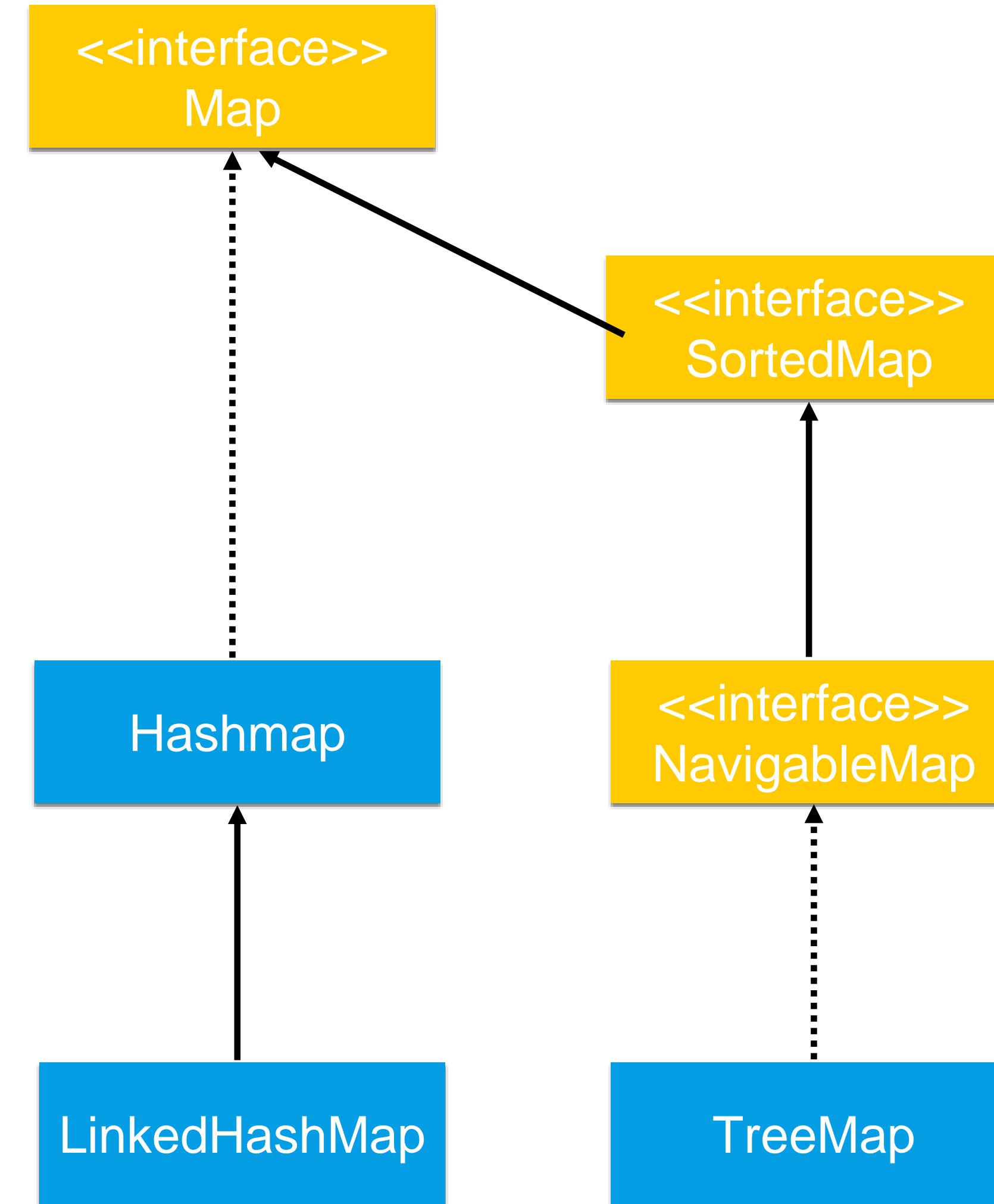


**Actually, there are many similarities between the Map and the Set hierarchy.**



# Map Interface

..... → implements  
→ extends

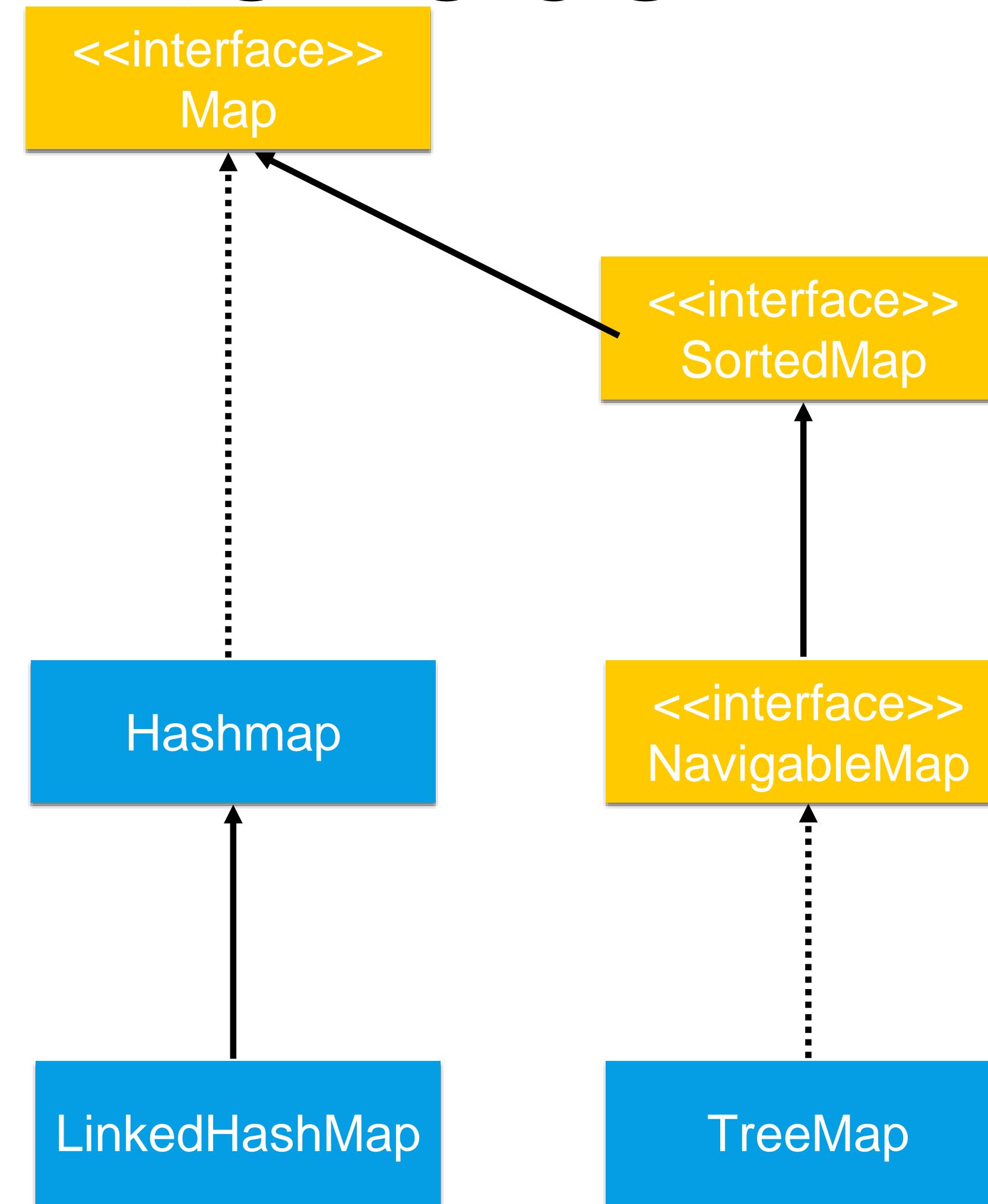


**The reason is that the Set implementations are actually internally backed by a Map implementation.**



# Map Interface

.....→ implements  
→ extends

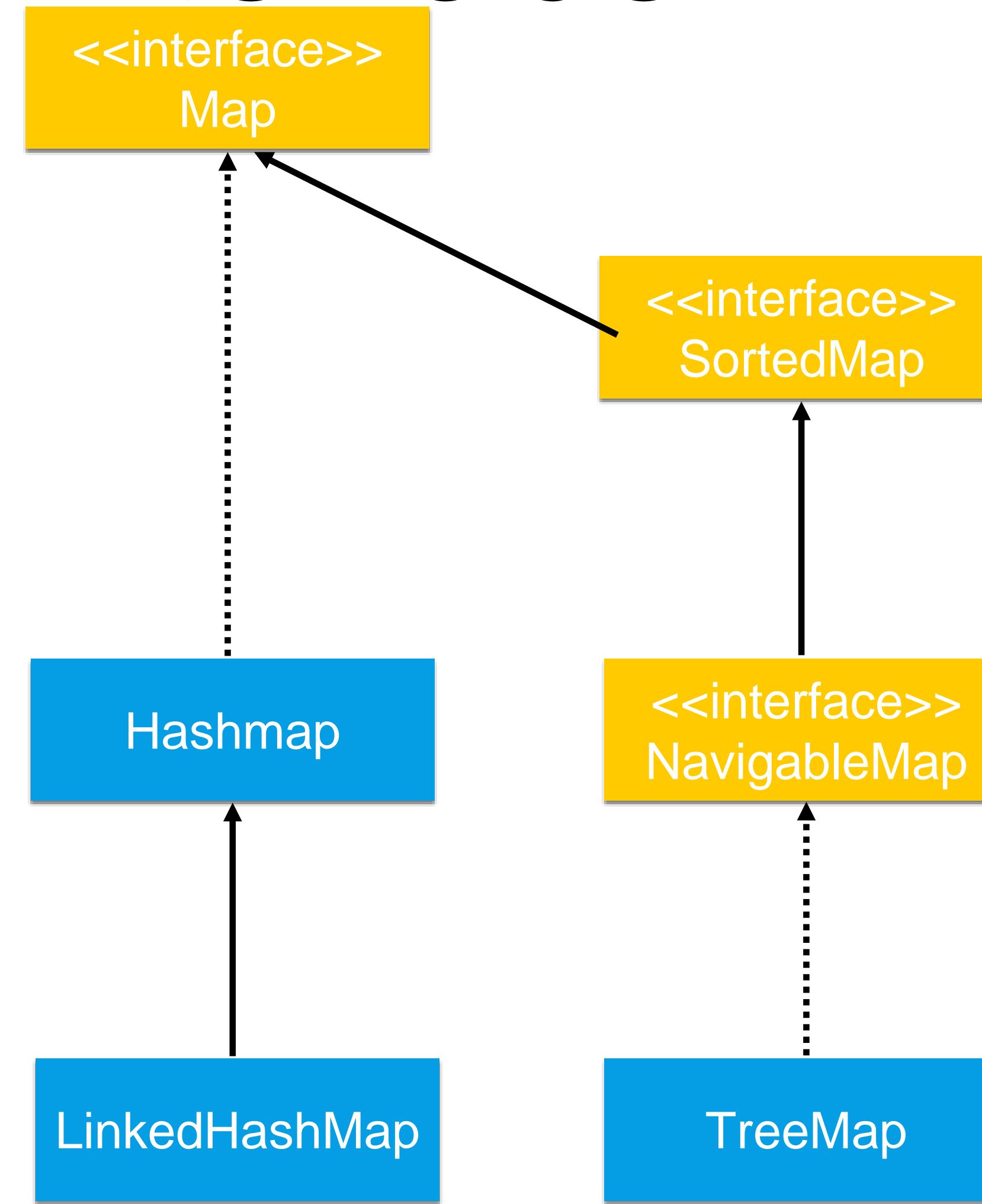


**Last but not least, you might have noticed,  
the Java Collection classes often contain the data  
structure they are based on in their name.**



# Map Interface

.....→ implements  
→ extends

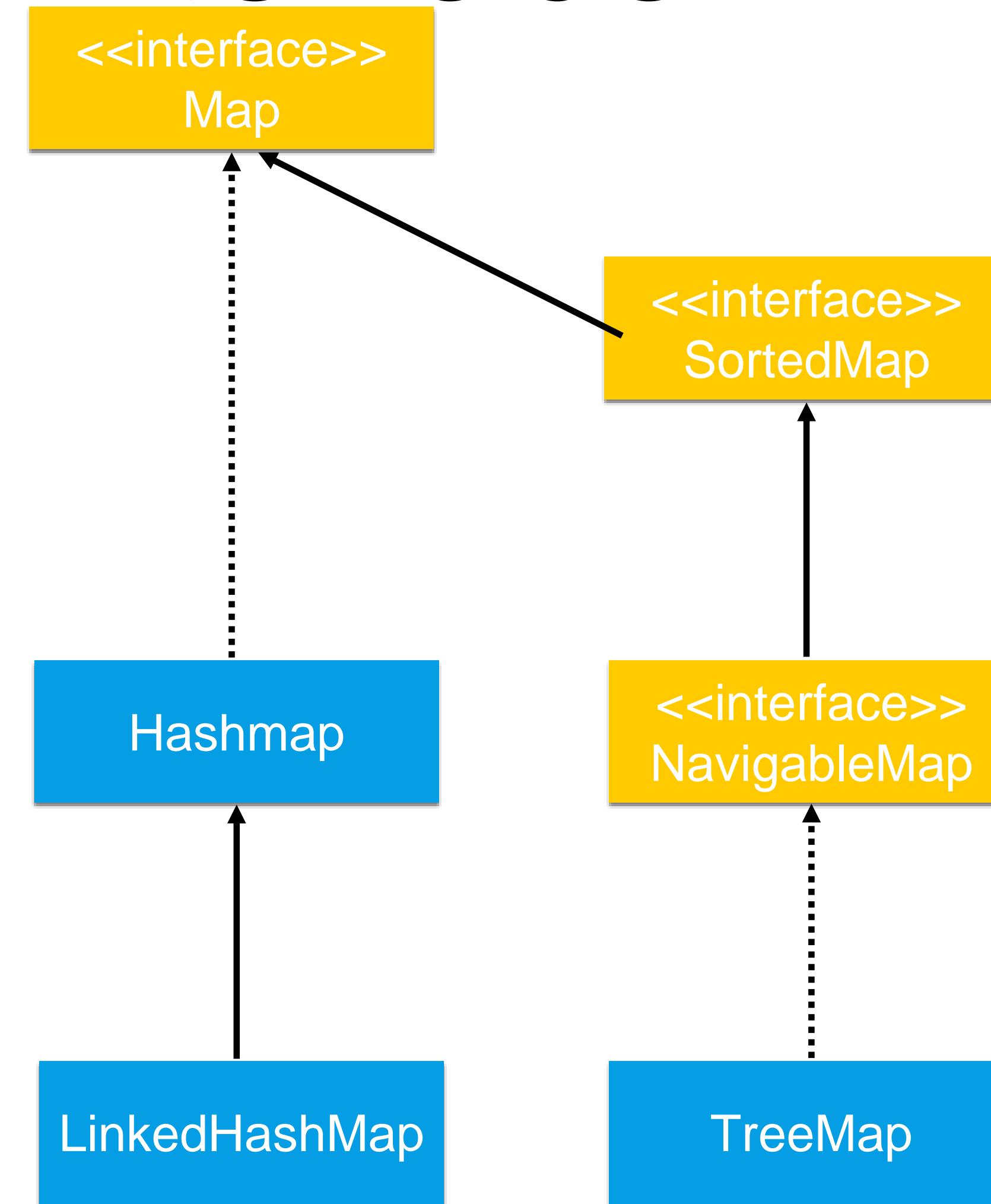


To choose the best collection for a given situation you have to compare the specific characteristics of data structures like **Array**, **LinkedList**, **Hashtable** or **Tree** first.



# Map Interface

..... → implements  
→ extends



In short, there is no single best option,  
each one has its very own advantages and disadvantages.



# Utility Interfaces

- **java.util.Iterator**
- **java.lang.Iterable**
- **java.lang.Comparable**
- **java.util.Comparator**

Okay now we are ready to look at some additional Utility interfaces of the Java Collections Framework.



# Utility Interfaces

- **java.util.Iterator**
- **java.lang.Iterable**
- **java.lang.Comparable**
- **java.util.Comparator**

They are implemented by classes of the Collections Framework or the JDK in general, but they can also be implemented by your own classes, making use of the power of the Collections Framework.



# Utility Interfaces

- **java.util.Iterator**
- **java.lang.Iterable**
- **java.lang.Comparable**
- **java.util.Comparator**

**Well - strictly speaking, the interface `java.lang.Iterable` is not part of the framework, but more precisely on top of it.**



# Utility Interfaces

- **java.util.Iterator**
- **java.lang.Iterable**
- **java.lang.Comparable**
- **java.util.Comparator**

It is the super interface of `java.util.Collection`,  
so every class that implements `java.util.Collection`  
will also implement the `java.lang.Iterable` interface.  
Okay, anyway, let's look at each interface in detail.



# java.util.Iterator

- **boolean hasNext();**
- **E next();**
- **void remove();**

An Iterator is an object that acts like  
a remote control to iterate through a collection.  
Let's look at its methods:



# java.util.Iterator

- **boolean hasNext();**
- **E next();**
- **void remove();**

**boolean hasNext(); -**

**Returns true if the collection has more elements.**



# java.util.Iterator

- **boolean hasNext();**
- **E next();**
- **void remove();**

**E next(); -**

**Returns the next element in the iteration.**



# java.util.Iterator

- **boolean hasNext();**
- **E next();**
- **void remove();**

**void remove(); -**

**Removes the last element returned by  
this iterator from the underlying collection.**



# java.lang.Iterable

- **Iterator<T> iterator()**

**This interfaces provides only one method  
which will return an Iterator.**

**Every collection that implements this interface  
can be used in the for each loop,**



# java.lang.Iterable

- **Iterator<T> iterator()**

**which greatly simplifies the usage of your home made collection.**



# java.lang.Iterable

- **Iterator<T> iterator()**

**In order to plug in your collection into the for each loop,  
you will have to execute two simple steps:**



# java.lang.Iterable

- **Iterator<T> iterator()**

**First create an Iterator that is able to iterate over your collection, with methods like hasNext and next(), as we saw on the last slide.**



# java.lang.Iterable

- **Iterator<T> iterator()**

**Second, you need to implement the Iterable interface by adding an iterator() method that will return an instance of this Iterator.**



# **java.lang.Comparable**

**int compareTo(T o)**

**Implementing the interface `java.lang.Comparable`  
defines a sort order for your entities.**



# java.lang.Comparable

**int compareTo(T o)**

**The interface contains only one method you need to implement,  
which is “int compareTo”.**



# java.lang.Comparable

**int compareTo(T o)**

**If you want to define a natural sort order for an entity class,  
make it implement this interface.**



# java.lang.Comparable

**int compareTo(T o)**

**Return a negative integer if the object is less than the given method argument, zero if the object is equal to the given method argument and a positive integer if the object is greater than the given method argument.**



# java.lang.Comparable

**Return a negative integer if the object is less than the given method argument, zero if the object is equal to the given method argument and a positive integer if the object is greater than the given method argument.**



# `java.lang.Comparable`

**Return a negative integer if the object is less than the given method argument, zero if the object is equal to the given method argument and a positive integer if the object is greater than the given method argument.**

**What means “smaller” or “greater” is for you to define.  
For numbers that would probably mean that  
1 is smaller then 5 for example.**

**But for colors?**

**This all depends on how you want to sort your entities.**



# java.lang.Comparable

**Return a negative integer if the object is less than the given method argument, zero if the object is equal to the given method argument and a positive integer if the object is greater than the given method argument.**

**When you put objects of an entity that implements the Comparable interface into a TreeSet or TreeMap, it will use your compareTo method to automatically sort all elements you put into the collection.**



# java.lang.Comparable

**Return a negative integer if the object is less than the given method argument, zero if the object is equal to the given method argument and a positive integer if the object is greater than the given method argument.**

**As you can see, the Java Collections framework has been greatly designed for extension, it offers a lot of possibilities to plug in your own classes.**



# java.util.Comparator

**int compare(T o1, T o2)**

**This interface is very similar to the Comparable interface.  
It allows you to define additional sorting orders,  
like a Reverse Ordering.**



# java.util.Comparator

**int compare(T o1, T o2)**

**So the sorting logic is not directly implemented in your entity,  
but in an external sorting strategy class that can optionally be added  
to a Collection or a sorting method to define an alternative  
sorting order for your collection of entities.**



# `java.util.Comparator`

`int compare(T o1, T o2)`

**The rules of the interface contract are pretty much  
the same as for Comparable:**



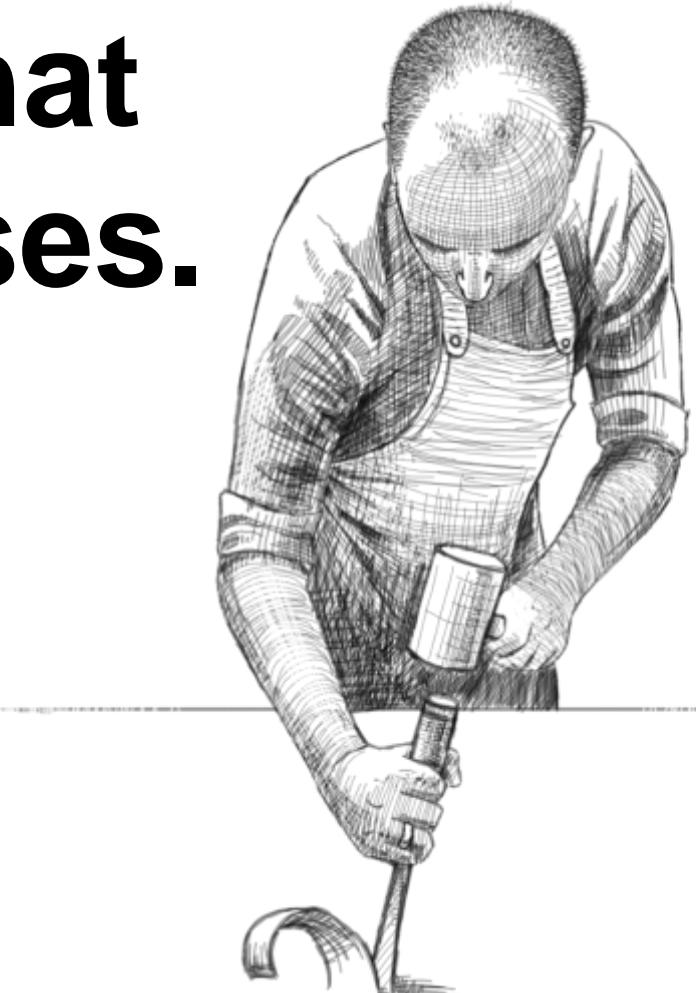
# java.util.Comparator

**Return a negative integer if the first argument is less than the second argument, zero if both arguments are equal and a positive integer if the first argument is greater than the second**



# Utility classes

**Last but not least, let's look at the two utility classes Collections and Arrays.  
Like a Swiss army knife they provide static helper methods that  
greatly enhance the general usefulness of the collection classes.**



# Utility classes

- **java.util.Collections**

**java.util.Collections –**  
**Offers methods like sort, shuffle, reverse, search, min or max.**



# Utility classes

- **java.util.Collections**
- **java.util.Arrays**

**java.util.Arrays -**

**Operates on Arrays and not on collections actually.**  
**Similar to the Collections class, it allows to sort arrays or**  
**to search through arrays, for example.**



**Copyright © 2016  
Marcus Biel  
All rights reserved**

