JAVA 21 SEQUENCED COLLECTIONS

Index

- Intro

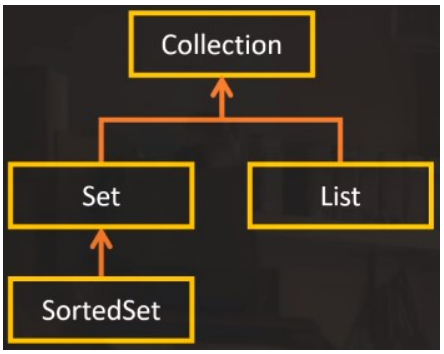
- The main goal of Sequenced collections is to model collections with a defined encounter order.

- Introducing the architecture of the Collections API

- The first version of the *Collection* API was super simple. Two hierarchies of interfaces: one for *collections* and another one for *maps*.

- The first hierarchy starts with the *Collection* interface, extended by *Set* and *List* on another branch.

- No duplicate in a *Set*, which is extended by Sorted*Set*, that keeps its elements sorted. On the other hand, List tells you that when you add elements in a certain order, let say A, B and C, then when you iterate over these elements, you will first meet A, then B, then C. You can't do that with *SortedSets*. You can add A, B and C, but when you iterate over these elements, the encounter order is imposed by the comparison logic.



- From this point, you already need to understand two notions: first what does "sorted" mean, and second what does "ordered" mean. And third, what's the difference between the two.

- What does it mean for a collection to be sorted

- Sorting elements means that you need to use a ***comparator***. This *comparator* can be embedded in your objects, in the case they implement ***Comparable***, or it can be an external comparator. And in that case, it will supersede the possible comparable nature of your objects.

- What can you do with a sorted collection? Let us examine three operations: getting an element from a given position, removing an element from a given position, and adding an element at a given position. If you have a *SortedSet*, you can certainly get the third element from that *Set*. And because you can access it, you can also remove it from the *Set*. But there is no way you can tell, "hey, I would like to insert this element in the third position". You can add an element, and this element will be inserted according to your comparison logic. It may land anywhere in your *Set*.

- Ordering elements in a collection with an index

- *Ordering elements* means that each element has an index, and this is what lists are doing.

- In a list, you have an internal, maybe implicit or virtual series of integers, starting from 0, all the way to the size of this list. This could be represented by an array for instance, even if it can be something else.

- If you add an element to a list, it will take the next index. Because elements have an indexes, the three operations we just saw all make sense. You can get the third element, and you can remove it. And you can add an element at a given position. Something you cannot do with a *SortedSet*.

- If you insert and element in a list, then the index all of the elements after this one is incremented. If you remove an element from this list, then the index of all the elements after the one you removed is decremented.

- Differences between Lists and Sorted*Set*s

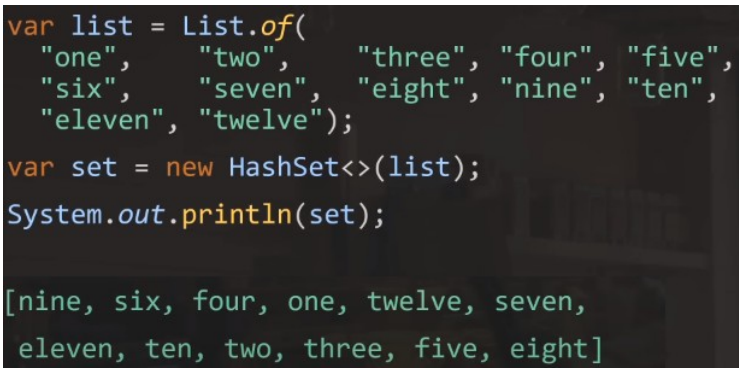
- First, adding an element to a list does not modify the index of the existing elements, because you add it at the end of the list. Adding an element to a *SortedSet* may change the order of this *Set* because you cannot control where your element will land in this sorted *Set*.

- Defining the encounter order for *List*, *Set*, and *SortedSet*

- The *encounter order* is in fact the order in which an iterator will iterate through the elements of the collection, whether it is a *Set*, a Sorted*Set*, or a List.

- For List: no problem, the *encounter order* is always the same, that is the semantic of the List interface, whether your are using ArrayList, which is of course your best choice, or LinkedList, which you should not be using anymore.

- For *Set*s, things are a little tricky. Odds are that your are using *HashSet* as an implementation for your *Set*s. Adding elements to a *Set* does not guarantee the same *encounter order*, and in fact, adding more elements may change the existing encounter order of the previous elements. For instance, suppose that you have this *HashSet* with the following elements that have been added to it:



- If you add just one element 'thirteen', then this order becomes different. Internally, there is a *HashMap* in a *HashSet*. And in the *HashMap* there is an array. If this array becomes full, it will be copied to a bigger array. And during this process, all the key value pairs stored in this array will be **rehashed** in the new array. And this may change your *encounter order*. So your code should absolutely not rely on any kind of stability of this encounter order, in the case of *Sets*.

```
var list = List.of(
  "one",    "two",    "three", "four", "five",
  "six",    "seven",  "eight", "nine", "ten",
  "eleven", "twelve");
var set = new HashSet<>(list);
System.out.println(set);
set.add("thirteen");
System.out.println(set);


[nine, six, four, one, twelve, seven,
 eleven, ten, two, three, five, eight]
[nine, six, one, seven, two, three, thirteen,
 eight, four, twelve, eleven, ten, five]
```

- But, if you really need it, you can also use another implementation of *Set*, which is called ***LinkedHashSet***. Internally, Linked*HashSet* is built on a **Linked*HashMap***, that has a ***LinkedList*** to iterate over its elements. This ***LinkedList*** gives you the stability of the encounter order. So whatever elements you add to a Linked*HashSet*, you will always visit them in the same order.

→ **Why is a *HashSet* backed by a *HashMap*? (from GPT)**

```
public HashSet(Collection<? extends E> c) {
    this.map = HashMap.newHashMap(Math.max(c.size(), 12));
    this.addAll(c);
}
```

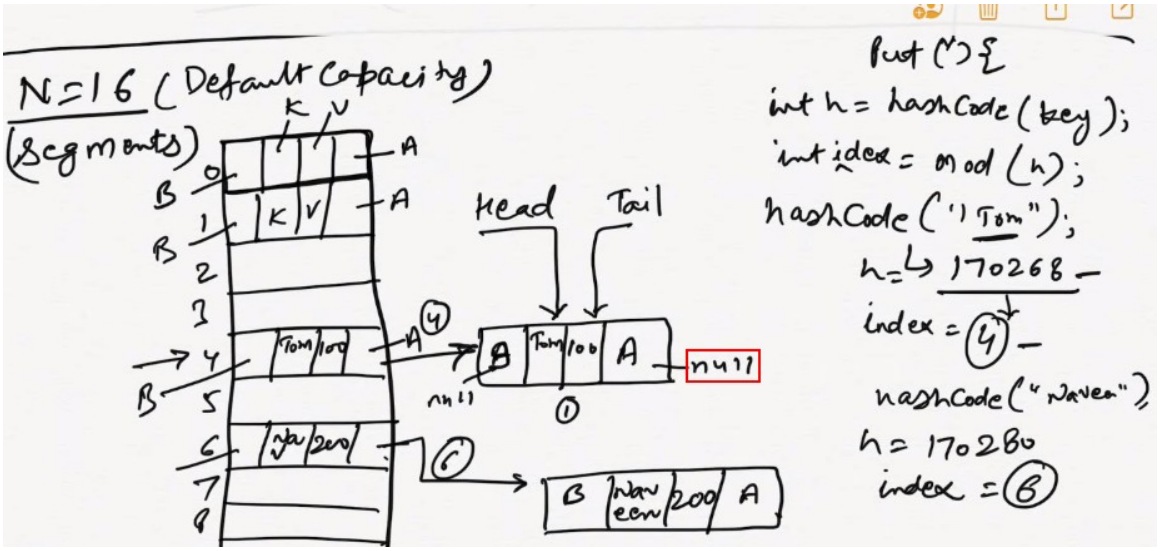
- Internally, when you create a *HashSet*, it actually creates an instance of a *HashMap* where the elements you add to the *HashSet* are used as keys in the *HashMap*, and the corresponding values are all *Set* to a single dummy object, often known as *PRESENT*, to represent that these keys are present in the *HashSet*.

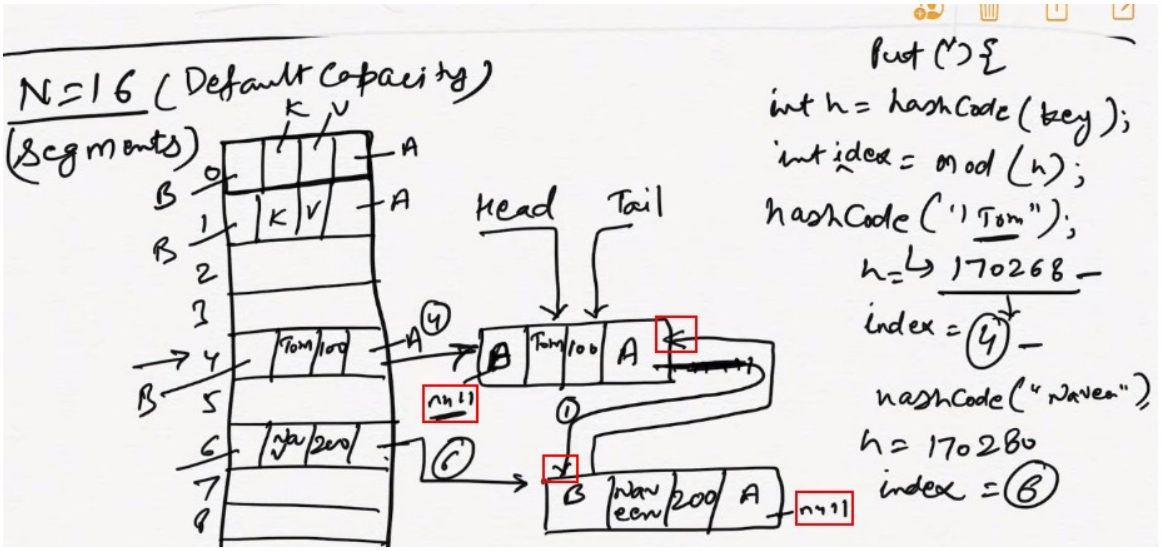
- The elements in a *HashSet* are effectively keys of a *HashMap*, and the values are not used for anything meaningful; they just serve as placeholders.

- The *HashMap* data structure provides **constant time performance** for basic operations like adding, removing, and checking for containment.

→ How *LinkedHashMap* works

→ Taken from: https://www.youtube.com/watch?v=dMqPqw6mulY





- That's why in the *LinkedHashMap* the encounter order is always preserved.
- (From GPT): If the preservation of insertion order is a crucial requirement for your application, then the slight overhead of using a LinkedHashMap is generally acceptable.

- Properties of *LinkedHashSet*

- This LinkedHashSet class is an implementation of Set, which makes sense because it does not allow doubles.

- It has a property of lists, which is: you always iterate over its elements in the same order. Does it have all the properties of List? Not quite. Can you get the third element for instance, that is an element from an index? The answer is yes. Could you remove and element at a given index? The answer is still yes. Could you insert an element at a given index? Well... not really. Because adding an element to a Set may fail, if this element is already present.

- LinkedHashSet has some properties that Lists have, but not all of them, which prevents it from implementing java.util.List.

- Common behavior between *List*, *SortedSet* and *LinkedHashSet*

- We have ***ordered collections*** called ***lists***, and on the other hand, we have ***sorted collections*** called ***SortedSet*** or ***NavigableSet***. And in the middle, we have ***LinkedHashSet***.

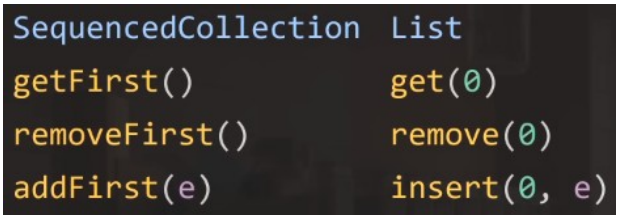
- All these are different, but they share some common behavior. And the fact is: there is nothing in the Collection API to model this shared behavior. And this is precisely the gap the JEP 431 called Sequenced collections aims to fill.

- ***SequencedCollections*** are modeling a common behavior, shared by *ordered collections* and *sorted collections*.

- Defining the behavior of sequenced collections

- The behavior that has been selected involves three operations: *getting*, *adding*, and *removing*. And from two places in the *SequencedCollection*: the start of the collection, and the end of the collection. That makes 6 methods: *getFirst()*; *getLast()*; *removeFirst()*; *removeLast()*; *addFirst(e)*; *addLast(e)*; and there is a bonus method: *reversed()*.

- How do these methods map to existing methods? Sometimes well, and sometimes not so well. If you consider the List interface for instance, you can call get(0), remove(0), or add(0, e) with an object. So operating on the first element works well.



- For the methods that operate on the last element, *list* is not doing that great. *add()* operates on the last element, so this one is ok. But to get the last element you need to call *get(list.size() - 1)*, and the same for *remove()*, you need also to pass size() - 1, which is not so great. So these two operations are not really supported directly.

SequencedCollection	List
getLast()	get(list.size() - 1)
removeLast()	remove(list.size() - 1)
addLast(e)	add(e)

- For the last one, *reversed()*, you can call *Collections.reverse()* and pass your list as an argument, which is not that great neither.

- For *SortedSet* or *NavigableSet*, the situation is better for getting the first element, which is actually the smallest, and the last element, which is the greatest. You actually have methods for that: *first()* and *last()*.

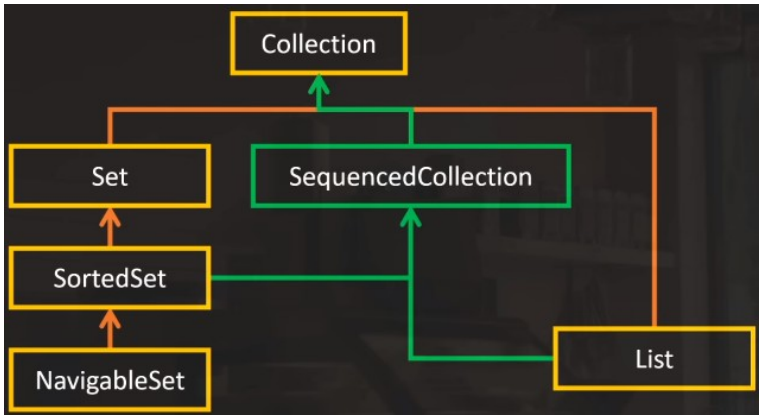
- When it comes to removing the first element without getting it first, then all you can do is use an *iterator* and call *remove()* on it, which is really not great.

SequencedCollection	NavigableSet
getFirst()	first()
removeFirst()	Ugly
addFirst(e)	Not supported

- For the removal of the last element, it's even worse, you don't want to see the code.

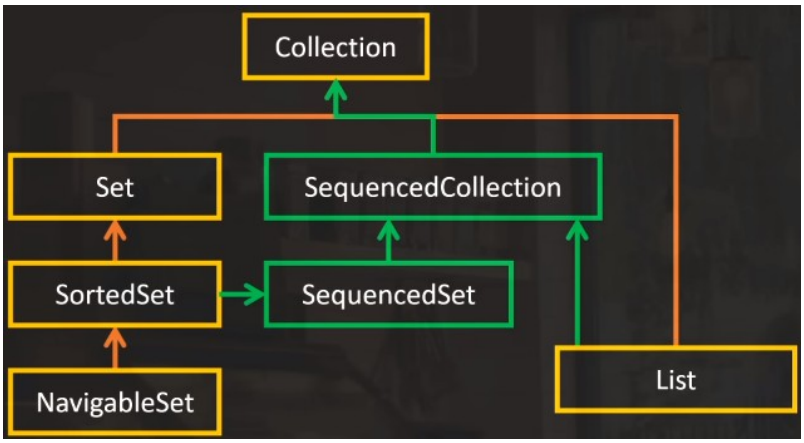
- Adding *SequencedCollection* and *SequencedSet* to the Collections API

- *SequencedCollection* extends the *Collection* interface. And the two interfaces *List* and *SortedSet* extend the *SequencedCollection* interface.

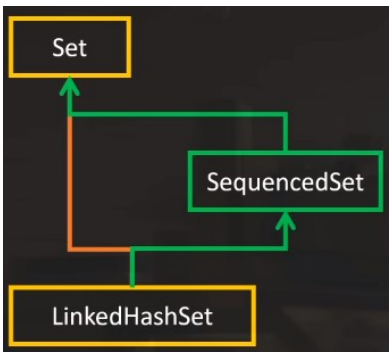


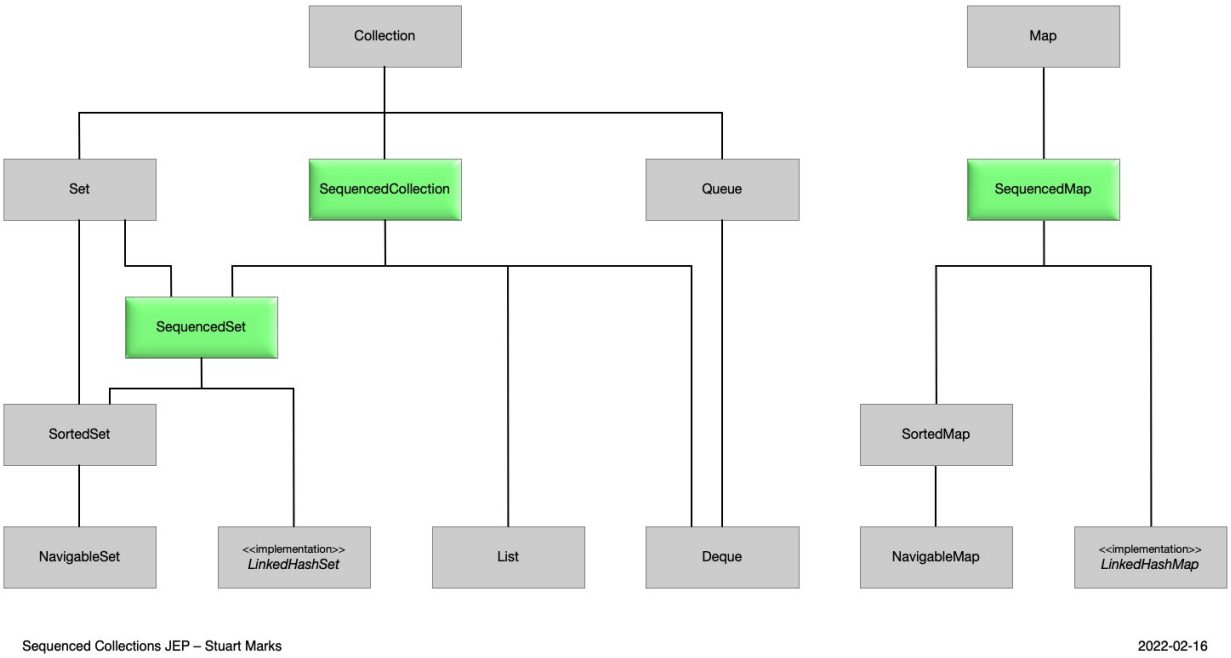
- *SequencedCollection* has a *reversed()* method that returns a *SequencedCollection*. In the case of *SortedSet*, this return type should also be a *SortedSet*. To implement that, another interface has been added, called *SequencedSet*.

- *SequencedSet* extends *SequencedCollection*, and overrides the *reversed()* method, that returns a *SequencedSet* in that case.



- We talked about *LinkedHashSet* earlier, and its specific behavior that was previously not captured by any interface of the Collection API. Well, this is not the case anymore, because now *LinkedHashSet* implements *SequencedSet*, which is great.





→ Taken from: https://belief-driven-design.com/looking-at-java-21-sequenced-collections-46c96/

- Making *Deque* implement *SequencedCollection*

- Before we jump into the *Map* interface, and we see how the *SortedMap* and *NavigableMap* can benefit from *SequencedCollection*, let us take a look at the *Queue* hierarchy.

- *Queue* interface was added to the JDK 5, back in 2004, as an extension of the Collection interface. And in the JDK 6, the *Deque* interface was added as an extension of *Queue*.

- You can ***push*** an element to a queue, you can ***pop*** an element from that queue, and check for the next available element with ***peek***. These three operations *push*, *pop* and *peek* are the same in any language.

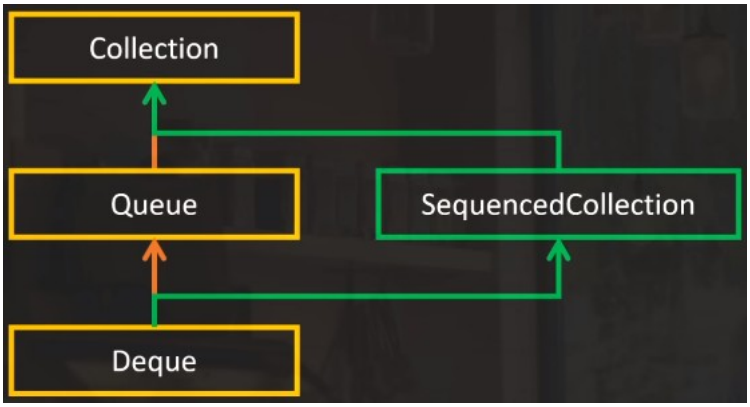
- Things have been made a little more complex than that in Java, because there are different ways to handle the fact that the queue may be full if you want to push an element to it, or empty, if you want to pop or peek an element from it.

- The choice was made to model two behaviors:
  - For *push*: return false or throw an exception
  - For *pop* and *peek*: return null or throw an exception.

- The *Deque* interface was added in the JDK 6. The idea of the Deque interface is to support both FIFO queues and LIFO stacks, with different behaviors when the *Deque* is empty or full.

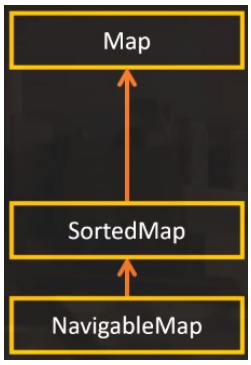
- *Deque* ends up with more than 20 methods, which makes it a little hard to understand at first. Because you can *add*, *remove* and *get* elements from the head or from the tail of a *Deque*, it then makes sense to make it extend *SequencedCollection*, and this is what has been done.

- A *Deque* is also a *SequencedCollection*, which may be useful to make your code simpler.



- Iterating over the elements of a Map

- At the top you have the Map interface, extended by SortedMap and NavigableMap.
- NavigableMap is an addition of the JDK 6 and there is really no reason to use SortedMap instead of NavigableMap nowadays.



- You can iterate on several elements of a map:
  - First, the entries of the map, that is, the key value pairs, modeled by the *Map.Entry* interface. You can get the set of the entries with the ***entrySet()*** method. It is a Set because you cannot have doubles among these entries.
  - Second, you can iterate over the *keys* of a map, that you can get with the ***keySet()*** method, that also gives you a set, for the same reason, you cannot have doubles among the keys neither.
  - Third, you can iterate over the *values* of the map, that you can get with the ***values()*** method. Now you can have doubles among the values, so what you get is a plain collection, not a set.

```
Set<Map.Entry<K, V>> entries = map.entrySet();
Set<K> keys                  = map.keySet();
Collection<V> values         = map.values();
```

- Note that what you get are actually views on the content of the map. And you can use these views to remove elements from a map. You cannot add anything, that wouldn't make sense, but you can remove stuff.

- Iterating over the elements of a *SortedMap* or *NavigableMap*

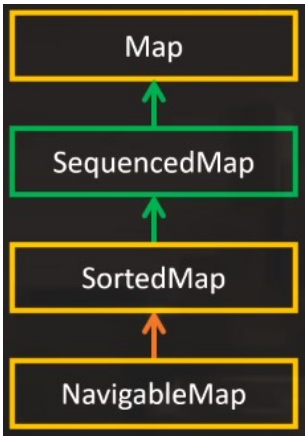
- Iterating over the elements of a map follows the same rules as for regular sets or collections: they are not ordered and not sorted. But if what you have is a SortedMap, then iterating over the set of the keys, or the set of the entries follows the ascending order of the keys or entries.

- In the case of NavigableMap, you have two more methods: navigableKeySet() and descendingKeySet(), that both give you a NavigableSet, which are views on the keys of this map, in the ascending or descending order. You do not have any equivalent for entries, unfortunately.

```
NavigableMap
navigableKeySet()
descendingKeySet()
```

- Introducing *SequencedMap*

- The set of the keys and the set of the entries are now *SequencedSet*.



- The methods are called ***sequencedKeySet()*** and ***sequencedEntrySet()***, bytheway, they are still views on the content of your map.

- You have a *reversed()* method, that gives you a reversed view of your map.

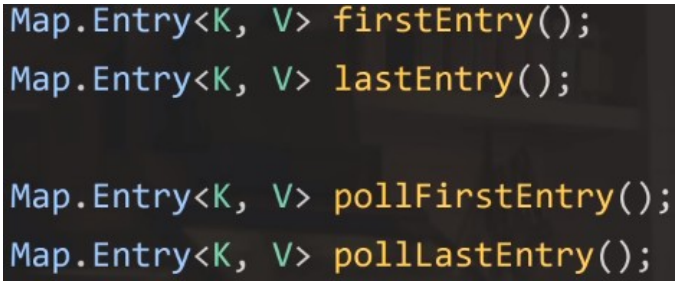
```
SequencedSet<K>                 sequencedKeySet();
SequencedSet<Map.Entry<K, V>    sequencedEntrySet();
SequencedCollection<V>          sequencedValues();

SequencedMap<K, V>              reversed();
```

- You have a *putLast()* and *putFirst()* methods, which are supported by *LinkedHashMap*, and not *SortedMap*, for the same reasons as *SortedSet*.

```
V putFirst(K k, V v);
V putLast(K k, V v);
```

- As a bonus, you get fourth set of methods already present on *NavigableMap*: *firstEntry()* and *lastEntry()* to read the first and last entry, and *pollFirstEntry()* and *pollLastEntry()* to remove the first and last entry of your map.



- Creating unmodifiable views with the Collections factory class

- You know that you have a family of factory methods to create unmodifiable views on collections, like *unmodifiableCollection()*, *unmodifiableList()* or *unmodifiableSet()*.

- Three factory methods have been added to this family: *unmodifiableSequencedCollection()*, *unmodifiableSequencedSet()*, and *unmodifiableSequencedMap()*. All these three methods create unmodifiable wrappers on the sequenced elements you pass as an argument.