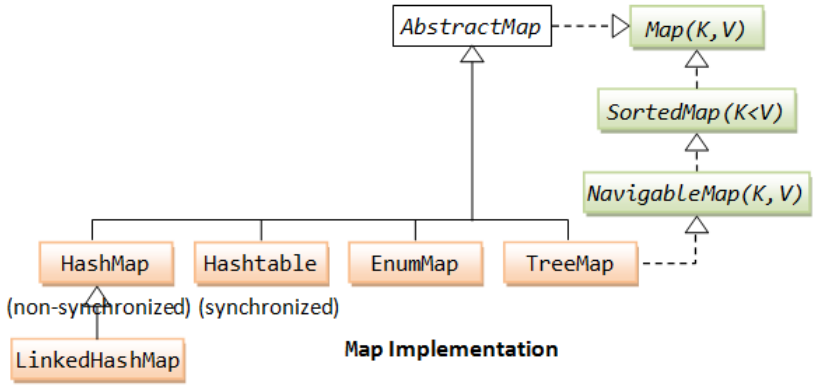
What is a Map?

A map is a collection of key-value pairs (e.g., name-address, name-phone, isbn-title, word-count). Each key maps to one and only value. Duplicate keys are not allowed, but duplicate values are allowed. Maps are similar to linear arrays, except that an array uses an integer key to index and access its elements; whereas a map uses any arbitrary key (such as `Strings` or any objects).

Note that a `Map` is not considered to be a true collection, as the `Map` interface does not extend the `Collection` interface. Instead, it starts an independent branch in the Java Collections Framework

The interface `Map<K,V>`, which takes two generic types `K` and `V` and read as `Map` of Key type `K` and Value type `V`, is used as a collection of "key-value pairs". No duplicate key is allowed. Frequently-used implementations include `HashMap`, `Hashtable` and `LinkedHashMap`. Its sub-interface `SortedMap<K, V>` models an ordered and sorted map, based on its key, implemented in `TreeMap`.

Take note that `Map<K,V>` is not a sub-interface of `Collection<E>`, as it involves a pair of objects for each element



Characteristics of a Map

Because a `Map` is not a true collection, its characteristics and behaviors are different than the other collections like `List` or `Set`.

A `Map` cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null value (`HashMap` and `LinkedHashMap`) but some does not (`TreeMap`).

The order of a map depends on specific implementations, e.g `TreeMap` and `LinkedHashMap` have predictable order, while `HashMap` does not.

Why and When Use Maps:

Maps are perfectly for key-value association mapping such as dictionaries. Use Maps when you want to retrieve and update elements by keys, or perform lookups by keys. A couple of examples:

- A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.
- A map of classes and students. Each class (key) is associated with a list of students (value).

Implementations of Map:

In the inheritance tree of the `Map` interface, there are several implementations but only 3 major, common, and general purpose implementations - they are `HashMap` and `LinkedHashMap` and `TreeMap`.

HashMap

This implementation uses a hash table as the underlying data structure. It implements all of the `Map` operations and allows null values and one null key. This class is roughly equivalent to `Hashtable` - a legacy data structure before Java Collections Framework, but it is not synchronized and permits nulls. `HashMap` does not guarantee the order of its key-value elements. Therefore, consider to use a `HashMap` when order does not matter and nulls are acceptable.

LinkedHashMap

This implementation uses a hash table and a linked list as the underlying data structures, thus the order of a `LinkedHashMap` is predictable, with insertion-order as the default order. This implementation also allows nulls like `HashMap`. So consider using a `LinkedHashMap` when you want a `Map` with its key-value pairs are sorted by their insertion order.

TreeMap

This implementation uses a red-black tree as the underlying data structure. A `TreeMap` is sorted according to the natural ordering of its keys, or by a `Comparator` provided at creation time. This implementation does not allow nulls. So consider using a `TreeMap` when you want a `Map` sorts its key-value pairs by the natural order of the keys (e.g. alphabetic order or numeric order), or by a custom order you specify.

The `Map<K,V>` interface declares the following `abstract` methods:

Method	Description
V get(Object key)	Returns the value of the specified key
V put(K key, V value)	Associate the specified value with the specified key
boolean containsKey(Object key)	Searches for the specified key
boolean containsValue(Object value)	Searches for the specified value
Set<K> keySet()	Returns a set view of the keys
Collection<V> values()	Returns a collection view of the values
Set entrySet()	Returns a set view of the key-value

1. Creating a new Map

Creating a HashMap:

Always use interface type (`Map`), generics and diamond operator to declare a new map. The following code creates a `HashMap`:

```
Map<Integer, String> mapHttpErrors = new HashMap<>();
mapHttpErrors.put(200, "OK");
mapHttpErrors.put(303, "See Other");
mapHttpErrors.put(404, "Not Found");
mapHttpErrors.put(500, "Internal Server Error");
```

```
System.out.println(mapHttpErrors);
```

This maps HTTP status codes to their descriptions. Output:

```
{404=Not Found, 500=Internal Server Error, 200=OK, 303=See Other}
```

As you can see in the output, a `HashMap` does not impose any order on its key-value elements.

You can create a new `Map` that copies elements from an existing map. For example:

```
Map<Integer, String> mapErrors = new HashMap<>(mapHttpErrors);
```

The map `mapErrors` is created with initial elements copied from the map `mapHttpErrors`.

Creating a LinkedHashMap:

The following code creates a `LinkedHashMap` that maps phone numbers with contact names:

```
Map<String, String> mapContacts = new LinkedHashMap<>();

mapContacts.put("0169238175", "Tom");

mapContacts.put("0904891321", "Peter");

mapContacts.put("0945678912", "Mary");

mapContacts.put("0981127421", "John");

System.out.println(mapContacts);
```

Output:

```
{0169238175=Tom, 0904891321=Peter, 0945678912=Mary, 0981127421=John}
```

As you can see, the `LinkedHashMap` maintains its elements by their insertion order.

Creating a TreeMap:

The following code creates a `TreeMap` that maps file extensions to programming languages:

```
Map<String, String> mapLang = new TreeMap<>();

mapLang.put(".c", "C");

mapLang.put(".java", "Java");

mapLang.put(".pl", "Perl");

mapLang.put(".cs", "C#");

mapLang.put(".php", "PHP");

mapLang.put(".cpp", "C++");

mapLang.put(".xml", "XML");

System.out.println(mapLang);
```

Output:

```
{.c=C, .cpp=C++, .cs=C#, .java=Java, .php=PHP, .pl=Perl, .xml=XML}
```

As you can see, the `TreeMap` sorts its keys by their natural ordering, which is the alphabetical order in this case.

2. Performing Basic Operations on a Map

The basic operations of a Map are association (`put`), lookup (`get`), checking (`containsKey` and `containsValue`), modification (`remove` and `replace`) and cardinality (`size` and `isEmpty`).

Associating a value with a key:

The `put(K, V)` method associates the specified value V with the specified key K. If the map already contains a mapping for the key, the old value is replaced by the specified value:

```
Map<Integer, String> mapHttpErrors = new HashMap<>();

mapHttpErrors.put(400, "Bad Request");

mapHttpErrors.put(304, "Not Modified");

mapHttpErrors.put(200, "OK");

mapHttpErrors.put(301, "Moved Permanently");

mapHttpErrors.put(500, "Internal Server Error");
```

Getting a value associated with a specified key:

The `get(Object key)` method returns the value associated with the specified key, or returns null if the map contains no mapping for the key. Given the map in the previous example:

```
String status301 = mapHttpErrors.get(301);

System.out.println("301: " + status301);
```

Output:

```
301: Moved Permanently
```

Checking if the map contains a specified key:

The method `containsKey(Object key)` returns true if the map contains a mapping for the specified key. For example:

```
if (mapHttpErrors.containsKey("200")) {

        System.out.println("Http status 200");

}
```

Output:

```
Found: Http status 200
```

Checking if the map contains a specified value:

The method `containsValue(Object value)` returns true if the map contains one or more keys associated with the specified value. For example:

```
if (mapHttpErrors.containsValue("OK")) {
	System.out.println("Found status OK");
}
```

Output:

```
Found status OK
```

Removing a mapping from the map:

The `remove(Object key)` method removes the mapping for a key from the map if it is present (we care about only the key, and the value does not matter). This method returns the value to which the map previously associated the key, or null if the map doesn't contain mapping for the key. Here's an example:

```
String removedValue = mapHttpErrors.remove(500);
if (removedValue != null) {
	System.out.println("Removed value: " + removedValue);
}
```

Output:

```
Removed value: Internal Server Error
```

Similarly, the `remove(Object key, Object value)` method removes the mapping of a specified key and specified value, and returns true if the value was removed. This method is useful in case we really care about the key and value to be removed.

Replacing a value associated with a specified key:

The `replace(K key, V value)` method replaces the entry for the specified key only if it is currently mapping to some value. This method returns the previous value associated with the specified key. Here's an example:

```
System.out.println("Map before: " + mapHttpErrors);
mapHttpErrors.replace(304, "No Changes");
System.out.println("Map after: " + mapHttpErrors);
```

Output:

```
Map before: {400=Bad Request, 304=Not Modified, 200=OK, 301=Moved Permanently}

Map after: {400=Bad Request, 304=No Changes, 200=OK, 301=Moved Permanently}
```

Similarly, the `replace(K key, V oldValue, V newValue)` method replaces the entry for the specified key only if it is currently mapping to the specified value. This method returns true if the value was replaced. Useful in case we want to replace exactly a key-value mapping.

Getting the size of the map:

The `size()` method returns the number of key-value mappings in this map. For example:

```
int size = mapHttpErrors.size();
```

Output:

```
Number of HTTP status code: 5
```

Checking if the map is empty:

The `isEmpty()` method returns true if the map contains no key-value mappings. For example:

```
if (mapHttpErrors.isEmpty()) {
	System.out.println("No Error");
} else {
	System.out.println("Have HTTP Errors");
}
```

Output:

```
Have HTTP Errors
```

3. Iterating Over a Map (using Collection views)

As a Map is not a true collection, there is no direct method for iterating over a map. Instead, we can iterate over a map using its collection views. Any Map's implementation has to provide the following three Collection view methods:

keyset()

`keySet()`: returns a Set view of the keys contained in the map. Hence we can iterate over the keys of the map as shown in the following example:

```
Map<String, String> mapCountryCodes = new HashMap<>();
mapCountryCodes.put("1", "USA");
mapCountryCodes.put("44", "United Kingdom");
mapCountryCodes.put("33", "France");
mapCountryCodes.put("81", "Japan");

Set<String> setCodes = mapCountryCodes.keySet();
Iterator<String> iterator = setCodes.iterator();

while (iterator.hasNext()) {
	String code = iterator.next();
	String country = mapCountryCodes.get(code);
	 System.out.println(code + " => " + country);
}
```

Output:

```
44 => United Kingdom
33 => France
1 => USA
81 > Japan
```

values()

`values()`: returns a collection of values contained in the map. Thus we can iterate over values of the map like this:

```
Collection<String> countries = mapCountryCodes.values();
for (String country : countries) {
	System.out.println(country);
}
```

Output:

```
United Kingdom
France
USA
Japan
```

entryset()

`entrySet()`: returns a `Set` view of the mappings contained in this map. Therefore we can iterate over mappings in the map like this:

```
Set<Map.Entry<String, String>> entries = mapCountryCodes.entrySet();
for (Map.Entry<String, String> entry : entries) {
	String code = entry.getKey();
	String country = entry.getValue();
	 System.out.println(code + " => " + country);
}
```

Output:

```
44 => United Kingdom
33 => France
1 => USA
81 => Japan
```

Since Java 8 with Lambda expressions and the **`forEach()`** statement, iterating over a `Map` is as easy as:

```
mapCountryCodes.forEach(
	(code, country) -> System.out.println(code + " => " + country));
Output:
44 => United Kingdom
33 => France
1 => USA
81 => Japan
```

4. Performing Bulk Operations with Maps

There are two bulk operations with maps: `clear()` and `putAll()`.
The `clear()` method removes all mappings from the map. The map will be empty after this method returns. For example:

```
mapHttpErrors.clear();
System.out.println("Is map empty? " + mapHttpErrors.isEmpty());
```

Output:

```
Is map empty? true
```

The `putAll(Map<K, V> m)` method copies all of the mappings from the specified map to this map. Here's an example:

```
Map<Integer, String> countryCodesEU = new HashMap<>();
countryCodesEU.put(44, "United Kingdom");
countryCodesEU.put(33, "France");
countryCodesEU.put(49, "Germany");

Map<Integer, String> countryCodesWorld = new HashMap<>();
countryCodesWorld.put(1, "United States");
countryCodesWorld.put(86, "China");
countryCodesWorld.put(82, "South Korea");

System.out.println("Before: " + countryCodesWorld);
countryCodesWorld.putAll(countryCodesEU);
System.out.println("After: " + countryCodesWorld);
```

Output:

```
Before: {1=United States, 82=South Korea, 86=China}
After: {1=United States, 33=France, 49=Germany, 82=South Korea, 86=China, 44=United
Kingdom}
```

5. Concurrent Maps

Unlike the legacy `Hashtable` which is synchronized, the `HashMap`, `TreeMap` and `LinkedHashMap` are not synchronized. If thread-safe is priority, consider using `ConcurrentHashMap` in place of `HashMap`. Or we can use the `Collections.synchronizedMap()` utility method that returns a synchronized (thread-safe) map backed by the specified map. For example:

```
Map<Integer, String> map = Collections.synchronizedMap(new HashMap<>());
```

And remember we have to manually synchronize the map when iterating over any of its collection views:

```
Set<Integer> keySet = map.keySet();
synchronized (map) {
	Iterator<Integer> iterator = keySet.iterator();
	while (iterator.hasNext()) {
		Integer key = iterator.next();
		String value = map.get(key);
	}
}
```

If you use a kind of `SortedMap`, e.g. `TreeMap`, consider using the more specific method `Collections.synchronizedSortedMap()`.

NOTE: If you use your own type for the key and value (e.g. `Student` or `Employee`), the key class and value class must implement the `equals()` and `hashCode()` methods properly so that the map can look up them correctly.

Summary of Map Implementation

Property	HashMap	LinkedHashMap	TreeMap	HashTable
Ordered	Unordered	Ordered by insertion	Sorted Order	Unordered
Null Values	Yes	Yes	Allowed	No
Duplicate	Keys =NO Value = yes	Keys =NO Value = yes	Keys =NO Value = yes	Keys =NO Value = yes
Synchronized	NO	NO	NO	YES
Initial Capacity	16	16	Not Applicable	11
Data Structure	HashTable	HashTable + Double Linked List	Red Black Tree	HashTable

Queue?

Queue means 'waiting line', which is very similar to queues in real life: a queue of people standing in an airport's check-in gate; a queue of cars waiting for green light in a road in the city; a queue of customers waiting to be served in a bank's counter, etc.

In programming, queue is a data structure that holds elements prior to processing, similar to queues in real-life scenarios. Let's consider a queue holds a list of waiting customers in a bank's counter. Each customer is served one after another, follow the order they appear or registered.

The first customer comes is served first, and after him is the 2nd, the 3rd, and so on. When serving a customer is done, he or she lefts the counter (removed from the queue), and the next customer is picked to be served next. Other customers come later are added to the end of the queue. This processing is called First In First Out or FIFO.

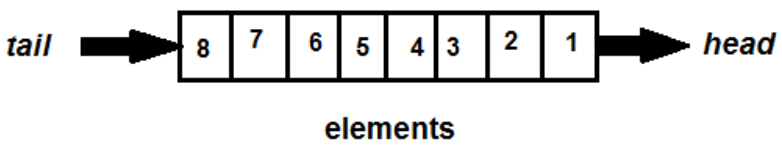
During the processing, the queue can be dynamically changed, i.e. processed elements are removed from the queue, and new elements are added to the queue.

In the Java Collections Framework, `Queue` is the main interface, and there are four sub interfaces: `Deque`, `BlockingDeque`, `BlockingQueue`, and `TransferQueue`.

Except the `Deque` interface which is in the `java.util` package, all others are organized in the `java.util.concurrent` package, which is designed for multi-threading or concurrent programming.

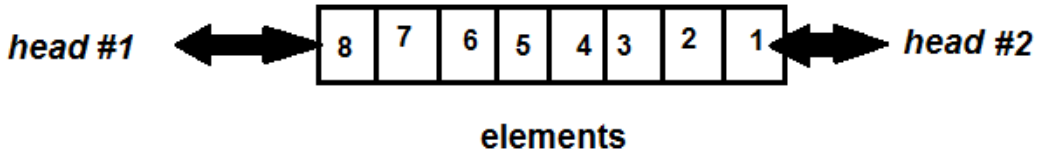
Characteristics of Queue:

Basically, a queue has a head and a tail. New elements are added to the tail, and to-be-processed elements are picked from the head. The following picture illustrates a typical queue:



Elements in the queue are maintained by their insertion order. The `Queue` interface abstracts this kind of queue.

Another kind of queue is double ended queue, or deque. A deque has two heads, allowing elements to be added or removed from both ends. The following picture illustrates this kind of queue:



The `Deque` interface abstracts this kind of queue, and it is a sub interface of the `Queue` interface. And the `LinkedList` class is a well-known implementation.

Some implementations accept null elements, some do not.

`Queue` does allow duplicate elements, because the primary characteristic of queue is maintaining elements by their insertion order. Duplicate elements in terms of equals contract are considered distinct in terms of queue, as there is no two elements having same ordering.

Additionally, the Java Collection Framework provides the `BlockingQueue` interface that abstracts queues which can be used in concurrent (multi-threading) context.

A blocking queue waits for the queue to become non-empty when retrieving an element, and waits for space become available in the queue when storing an element.

Similarly, the `BlockingDeque` interface is blocking queue for double ended queues.

Behaviors of Queue:

Due to Queue's nature, the key operations that differentiate `Queue` from other collections are extraction and inspection at the head of the queue.

For deques, the extraction and inspection can be processed on both ends.

And because the `Queue` interface extends the `Collection` interface, all Queue implementations provide core operations of a collection like `add()`, `contains()`,`remove()`, `clear()`, `isEmpty()`, etc.

And keep in mind that, with queues, operations on the head are fastest (e.g. `offer()` and `remove()`), whereas operations on middle elements are slow (e.g.`contains(obj)` and `remove(obj)`).

Queue's Interfaces:

Queue is the super interface of the queue branch in the Java Collection Framework. Under it, there are the following sub interfaces:

- **`Deque`**: abstracts a queue that has two heads. A deque allows adding or removing elements at both ends.
- **`BlockingQueue`**: abstracts a type of queues that waits for the queue to be non-empty when retrieving an element, and waits for space to become available in the queue when storing an element.
- **`BlockingDeque`**: is similar to `BlockingQueue`, but for double ended queues. It is sub interface of the `BlockingQueue`.

And since Java 7, the `BlockingQueue` interface has a new sub interface called `TransferQueue`, which is a specialized `BlockingQueue`, which waits for another thread to retrieve an element in the queue.

Major Queue's Implementations:

The Java Collection framework provides many implementations, mostly for the `BlockingQueue` interface. Below I name few which are used commonly.

Queue implementations are grouped into two groups: general-purpose and concurrent implementations.

-General-purpose Queue implementations:

- + **`LinkedList`**: this class implements both `List` and `Deque` interface, thus having hybrid characteristics and behaviors of list and queue. Consider using a `LinkedList` when you want fast adding and fast removing elements at both ends, plus accessing elements by index.

- + **`PriorityQueue`**: this queue orders elements according to their natural ordering, or by a `Comparator` provided at construction time. Consider using a `PriorityQueue` when you want to take advantages of natural ordering and fast adding elements to the tail and fast removing elements at the head of the queue.

- + **`ArrayDeque`**: a simple implementation of the `Deque` interface. Consider using an `ArrayDeque` when you want to utilize features of a double ended queue without list-based ones (simpler than a `LinkedList`).

Concurrent Queue implementations:

- + **`ArrayBlockingQueue`**: this is a blocking queue backed by an array. Consider using an ArrayBlockingQu0eue when you want to use a simple blocking queue that has limited capacity (bounded).

- + **`PriorityBlockingQueue`**: Use this class when you want to take advantages of both PriorityQueue and BlockingQueue.

- + **`DelayQueue`**: a time-based scheduling blocking queue. Elements added to this queue must implement the Delayed interface. That means an element can only be taken from the head of the queue when its delay has expired.

Mastering Queue Collection in Java

Queue API Structure

Before giving you code examples about using Queue collections, I think it's necessary to understand deeper about the API Structure of Queue collection, as queues have very different characteristics than other types of collection.

Because the Queue interface extends the Collection interface, al Queue implementations have basic operations of a collection:

- Single operations: add(e),contains(e), iterator(), clear(), isEmpty(), size() and toArray().
- Bulk operations: addAll(), containsAll(), removeAll() and retainAll().

Understanding Queue interface's API Structure:

Basically, `Queue` provides three primary types of operations which differentiate a queue from others:

1. **Insert**: adds an element to the tail of the queue.
2. **Remove**: removes the element at the head of the queue.
3. **Examine**: returns, but does not remove, the element at the head of the queue.

And for each type of operation, there are two versions:

- The first version throws an exception if the operation fails, e.g. could not add element when the queue is full.
- The second version returns a special value (either null or false, depending on the operation).

The following table summarizes the main operations of the `Queue` interface:

Type of operation	Throws exception	Returns special value
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

Understanding Deque interface's API Structure:

As you know, the `Deque` interface abstracts a double ended queue with two ends (first and last), so its API is structured around this characteristic.

A `Deque` implementation provides the `xxxFirst()` methods that operate on the first element, and the `xxxLast()` methods that operate on the last element.

The following table summarizes the API structure of `Deque`:

Type of operation	First element	Last element
Insert	addFirst(e) offerFirst(e)	addLast(e) offerLast(e)
Remove	removeFirst() pollFirst()	removeLast() pollLast()
Examine	getFirst() peekFirst()	getLast() peekLast()

Understanding BlockingQueue interface's API Structure:

A blocking queue is designed to wait for the queue to become non-empty when retrieving an element (the `put(e)` method), and wait for space to become available in the queue when storing an element (the `take()` method).

In addition, a blocking queue provides specialized operations that can wait up to a specified duration when inserting and removing an element.

The following table summarizes the API structure of `BlockingQueue` interface:

Type of operation	Throws exception	special value	blocks	times out
Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)

Examine	element()	peek()	N/A	N/A

Understanding BlockingDeque interface's API Structure:

Similarly, a `BlockingDeque` is a specialized `BlockingQueue` for double ended queue with two ends (head and tail). Its API is in scheme of `xxxFirst()` methods operating on the first element and `xxxLast()` methods operating on the last element.

The following table summarizes the API structure of `BlockingDeque`:

First Element (head)				
	Throws exception	special value	blocks	times out
Insert	addFirst(e)	offerFirst(e)	putFirst(e)	offerFirst(e, time, unit)
Remove	removeFirst()	pollFirst()	takeFirst()	pollFirst(time, unit)
Examine	getFirst()	peekFirst()	N/A	N/A

Last Element (tail)				
	Throws exception	special value	blocks	times out
Insert	addLast(e)	offerLast(e)	putLast(e)	offerLast(e, time, unit)
Remove	removeLast()	pollLast()	takeLast()	pollLast(time, unit)
Examine	getLast()	peekLast()	N/A	N/A

Performing Operations on Queue Collection

Let's go through various code examples to understand how to use `Queue` collections in daily coding. In the upcoming examples, I use different implementations like `LinkedList`, `ArrayDeque`, `PriorityQueue`,`ArrayBlockingQueue`, etc.

1. Creating a New Queue Instance

As a best practice, it's recommended to use generic type and interface as reference type when creating a new collection. For queues, depending on the need of a particular type (`queue, deque and blocking queue`), use the corresponding interface as the reference type.

For example, the following statements create 3 different types of queues:

```
Queue<String> namesQueue = new LinkedList<>();

Deque<Integer> numbersDeque = new ArrayDeque<>();

BlockingQueue<String> waitingCustomers = new ArrayBlockingQueue<>(100);
```

Most Queue implementations do not have restriction on capacity (unbounded queues), except the `ArrayBlockingQueue`, `LinkedBlockingQueue` and `LinkedBlockingDeque` classes. The following statement creates an `ArrayBlockingQueue` with fixed capacity of 200 elements:

```
BlockingQueue<String> waitingCustomers = new ArrayBlockingQueue<>(200);
```

Also remember that we can use the copy constructor to create a new Queue instance from another collection. For example:

```
List<String> listNames = Arrays.asList("Alice", "Bob", "Cole", "Dale", "Eric", "Frank");
Queue<String> queueNames = new LinkedList<>(listNames);
System.out.println(queueNames);
```

Output:

```
 [Alice, Bob, Cole, Dale, Eric, Frank]
```

2. Adding New Elements to the Queue

To insert an element to the tail of the queue, we can use either the `add()` or `offer()` method. The following code adds two elements to a linked list:

```
Queue<String> queueNames = new LinkedList<>();
queueNames.add("Mary");
queueNames.add("John");
```

When using an unbounded queue (no capacity restriction), the `add()` and `offer()` methods do not show the difference. However, when using a bounded queue, the `add()` method will throw an exception if the queue is full, while the `offer()` method returns false. The following example illustrates this difference:

```
Queue<Integer> queueNumbers = new ArrayBlockingQueue<>(3);
queueNumbers.add(1);
queueNumbers.add(2);
queueNumbers.add(3);
queueNumbers.add(4);// this line throws exception
```

The last line throws `java.lang.IllegalStateException: Queue full` because we declare the queue with capacity of 3 elements. Hence adding the 4th element results in an exception.

However, we are safe when using the `offer()` method, as shown in the following code snippet:

```
Queue<Integer> queueNumbers = new ArrayBlockingQueue<>(3);
System.out.println(queueNumbers.offer(1));
System.out.println(queueNumbers.offer(2));
System.out.println(queueNumbers.offer(3));
System.out.println(queueNumbers.offer(4));
```

Output:

```
true
true
true
false
```

The following code snippet adds an element to the head and an element to the tail of a double ended queue (notice the type of the interface is used):

```
Deque<String> queueNames = new ArrayDeque<>();
queueNames.offer("Katherine");
```

```
queueNames.offer("Bob");
queueNames.addFirst("Jim");
queueNames.addLast("Tom");
System.out.println(queueNames);
```

Output:

```
[Jim, Katherine, Bob, Tom]
```

For blocking queue, use the `put(e)` or `offer(e, time, unit)` in case you want the current thread to wait until space becomes available in the queue. For example:

```
BlockingQueue<Integer> queueNumbers = new ArrayBlockingQueue<>(100);
try {
	queueNumbers.put(2000);
} catch (InterruptedException ie) {
	ie.printStackTrace();
}
```

3. Removing the Head of the Queue

A Queue provides the `remove()` and `poll()` methods that allow us to pick the element at the head and remove it from the queue. And you should understand the difference between these two methods.

The `remove()` method returns the head element or throws an exception if the queue is empty. For example:

```
Queue<String> queueCustomers = new LinkedList<>();
queueCustomers.offer("Jack");
String next = queueCustomers.remove();
System.out.println("Next customer is: "+ next);
next = queueCustomers.remove();   // throws exception
```

Here, the queue has only one element, so the first call to `remove()` working fine. However the subsequent invocation results in `java.util.NoSuchElementException` because the queue becomes empty.

In contrast, the `poll()` method returns `null` if the queue is empty, as shown in the following example:

```
Queue<String> queueCustomers = new LinkedList<>();
queueCustomers.offer("Jack");
System.out.println("next: " + queueCustomers.poll());
System.out.println("next: " + queueCustomers.poll());  // returns null
```

Output:

```
next: Jack
next: null
```

The following example removes the head element and tail element from a deque:

```
Deque<String> queueCustomers = new ArrayDeque<>();
queueCustomers.offer("Bill");
queueCustomers.offer("Kim");
queueCustomers.offer("Lee");
queueCustomers.offer("Peter");
queueCustomers.offer("Sam");
```

```
System.out.println("Queue before: " + queueCustomers);
System.out.println("First comes: " + queueCustomers.pollFirst());
System.out.println("Last comes: " + queueCustomers.pollLast());
System.out.println("Queue after: " + queueCustomers);
```

Output:

```
Queue before: [Bill, Kim, Lee, Peter, Sam]
First comes: Bill
Last comes: Sam
Queue after: [Kim, Lee, Peter]
```

For a blocking queue, use the `take()`or `poll(time, unit)` methods in case you want the current thread to wait until an element becomes available. For example:

```
BlockingQueue<String> queueCustomers = new ArrayBlockingQueue<>(100);
queueCustomers.offer("Kathe");
try {
     String nextCustomer = queueCustomers.take();
} catch (InterruptedException ie) {
     ie.printStackTrace();
}
```

4. Examining the Head of the Queue

In contrast to the `remove()` method, the examine methods `element()` and `peek()` return (but do not remove) the head of the queue. So consider using these methods in case you just want to check what is currently in the head element without modifying the queue.

Also, you need to know the difference between the `element()` and `peek()` methods:

The `element()` method throws an exception in case the queue is empty, whereas the `peek()` method returns `null`. For example:

```
Queue<String> queueCustomers = new PriorityQueue<>();
queueCustomers.offer("Jack");
System.out.println("who's next: " + queueCustomers.poll());
// this returns null in case the queue is empty
System.out.println("who's next: " + queueCustomers.peek());
// this throws exception in case the queue is empty
System.out.println("who's next: " + queueCustomers.element());
```

For a deque, use the `getFirst()` or `peekFirst()` methods to examine the first element, and `getLast()` or `peekLast()` to examine the last element. Here's an example:

```
Deque<Integer> queueNumbers = new ArrayDeque<>();
queueNumbers.add(10);
queueNumbers.add(20);
queueNumbers.add(30);
queueNumbers.add(40);
System.out.println("first: " + queueNumbers.getFirst());
```

```
System.out.println("last: " + queueNumbers.peekLast());
```

There's no method for examining a blocking queue.

5. Iterating over Elements in the Queue

We can use the enhanced for loop, iterator and `forEach()` method to iterate over elements in the queue. The following code snippet illustrates how to iterate a linked list using the enhanced for loop:

```
Queue<String> queueNames = new LinkedList<>();
queueNames.add("Dale");
queueNames.add("Bob");
queueNames.add("Frank");
queueNames.add("Alice");
queueNames.add("Eric");
queueNames.add("Cole");
queueNames.add("John");
for (String name : queueNames) {
     System.out.println(name);
}
```

Output:

```
Dale
Bob
Frank
Alice
Eric
Cole
John
```

More simply, using Lambda expression with `forEach()` method in Java 8:

```
queueNames.forEach(name -> System.out.println(name));
```

The following example iterates over elements of a `PriorityQueue` which sorts elements by natural ordering:

```
Queue<String> queueNames = new PriorityQueue<>();
queueNames.add("Dale");
queueNames.add("Bob");
queueNames.add("Frank");
queueNames.add("Alice");
queueNames.add("Eric");
queueNames.add("Cole");
queueNames.add("John");
queueNames.forEach(name -> System.out.println(name));
```

Output:

```
Alice
```

```
Bob
Cole
Dale
Eric
Frank
John
```

As you can see in the output, the elements are sorted in the alphabetic order (natural ordering of Strings).

NOTE: Pay attention when using an iterator of a `PriorityQueue`, because it is not guaranteed to traverse the elements of the priority queue in any particular order.

6. Concurrent Queues

All implementations of `BlockingQueue` are thread-safe. The following implementations are not:

- `LinkedList`
- `ArrayDeque`
- `PriorityQueue`

When you want to use a synchronized linked list, use the following code:

```
List list = Collections.synchronizedList(new LinkedList<>());
```

And consider using the `PriorityBlockingQueue` instead of the `PriorityQueue` when you want to use a synchronized priority queue.

Producer - Consumer Example Using Queue

Typically, queue is used to implement producer-consumer scenarios. The following kinds of program will need to use queue:

- Chat applications: Messages are put into a queue. When you are sending a message, you are the producer; and your friend who reads the message, is the consumer. Messages need to be kept in queue because of network latency. Imagine network connection dropped when you are trying to send a message. In this case, the message is still in the queue, awaiting the receiver to consume upon the connection becomes available.

- Online help desk applications: Imagine a company has 5 persons working as customer support staffs. They chat with clients through a help desk application. They can talk with maximum 5 clients at a time, so other clients will be queued up. When a staff finishes serving a client, the next client in the queue is served next.

- Real-time processing applications such as screen recorder. The logic behind this kind of application is there are two threads working concurrently: The producer thread captures screenshots constantly and puts the images into a queue; the consumer thread takes the images from the queue to process the video.

- And much more.

Above I name only few types of application in which we need to use queue. Remember using queue when you need to implement producer-consumer processing.

In Java, using a `BlockingQueue` implementation is a good choice, as its `put(e)` method let the producer thread waits for space to become available in the queue, and its `take()` method let the consumer thread waits for an element become available in the queue.

The following is a pseudo-code of a typical producer-consumer application:

```
class Producer implements Runnable {
   private final BlockingQueue queue;
   Producer(BlockingQueue q) { queue = q; }
   public void run() {
       while (true) { queue.put(produce()); }
   }
   Object produce() { ... }
}
class Consumer implements Runnable {
   private final BlockingQueue queue;
   Consumer(BlockingQueue q) { queue = q; }
   public void run() {
       while (true) { consume(queue.take()); }
   }
   void consume(Object x) { ... }
}
class Program {
   void main() {
     BlockingQueue q = new SomeBlockingQueueImplementation();
     Producer p = new Producer(q);
     Consumer c1 = new Consumer(q);
     Consumer c2 = new Consumer(q);
```

```
        new Thread(p).start();
        new Thread(c1).start();
        new Thread(c2).start();
    }
}
```

Here, the Producer class is a thread which constantly produces objects and put them into the queue. In practice, we should specify condition to exit the loop, such as closing/shutdown the program or a maximum number of objects reached.

The Consumer class is another thread which constantly takes objects from the queue to process. In practice, we should specify condition to stop this thread by checking the queue for a special object (null, false or special value), for example:

```
while (true) {
            Integer number = queue.take();
            if (number == -1) {
             break;
            }
            consume(number);
}
```

And in this case, the producer is responsible to put this special object into the queue to indicate there's no more elements to process.

For you reference, here's a working example that gives you the idea. It's a program that creates one producer thread and one consumer thread:

Producer:

```
import java.util.*;
import java.util.concurrent.*;
public class Producer implements Runnable {
                    private BlockingQueue<Integer> queue;
                    public Producer (BlockingQueue<Integer> queue) {
                     this.queue = queue;
                    }
                    public void run() {
                     try {
                            for (int i = 0; i < 10; i++) {
                                    queue.put(produce());
                                    Thread.sleep(500);
                            }
                            queue.put(-1);       // indicates end of producing
                            System.out.println("Producer STOPPED.");
                     } catch (InterruptedException ie) {
                            ie.printStackTrace();
                     }
                    }
```

```
                        private Integer produce() {
                         Integer number = new Integer((int) (Math.random() * 100));
                         System.out.println("Producing number => " + number);
                         return number;
                        }
}
```

Consumer:

```
import java.util.*;
import java.util.concurrent.*;
public class Consumer implements Runnable {
              private BlockingQueue<Integer> queue;
              public Consumer(BlockingQueue<Integer> queue) {
               this.queue = queue;
              }
              public void run() {
               try {
                      while (true) {
                            Integer number = queue.take();
                            if (number == -1) {
                                   break;
                            }
                            consume(number);
                            Thread.sleep(1000);
                      }
                      System.out.println("Consumer STOPPED.");
               } catch (InterruptedException ie) {
                      ie.printStackTrace();
               }
              }
              private void consume(Integer number) {
               System.out.println("Consuming number <= " + number);
              }
}
```

Test Program:

```
import java.util.*;
import java.util.concurrent.*;
public class ProducerConsumerTest {
              public static void main(String[] args) {
```

```
                BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(20);
                Thread producer = new Thread(new Producer(queue));
                Thread consumer = new Thread(new Consumer(queue));
                producer.start();
                consumer.start();
               }
}
Compile and run this program would print the following output:
Producing number => 21
Consuming number <= 21
Producing number => 90
Producing number => 51
Consuming number <= 90
Producing number => 23
Producing number => 61
Consuming number <= 51
Producing number => 63
Producing number => 75
Consuming number <= 23
Producing number => 99
Consuming number <= 61
Producing number => 59
Producing number => 31
Producer STOPPED.
Consuming number <= 63
Consuming number <= 75
Consuming number <= 99
Consuming number <= 59
Consuming number <= 31
Consumer STOPPED.
```

Understanding SortedSet and TreeSet

You know, `TreeSet` does not only implement the Set interface, it also implements the `SortedSet` and `NavigableSet` interfaces. Therefore, besides inheriting behaviors of a typical `Set`, `TreeSet` also inherits behaviors of `SortedSet` and `NavigableSet`. The following picture illustrates the API hierarchy:

Understanding SortedSet:

The key characteristic of a `SortedSet` is that, it sorts elements according to their natural ordering or by a specified comparator. So considering using a `TreeSet` when you want a collection that satisfies the following conditions:

- Duplicate elements are not allowed.
- Elements are sorted by their natural ordering (default) or by a specified comparator.

Here's an example illustrates this characteristic of a `SortedSet`:

```
SortedSet<Integer> setNumbers = new TreeSet<>();
setNumbers.addAll(Arrays.asList(2, 1, 4, 3, 6, 5, 8, 7, 0, 9));
System.out.println("Sorted Set: " + setNumbers);
```

Output:

```
Sorted Set: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Here, we add elements of an array list to a `TreeSet`, and as you can see, the duplicate elements are removed and they are sorted by alphanumeric order (natural ordering of numbers).

In addition to basic collection operations and normal set operations, the `SortedSet` provides the following types of operations:

- **Range view**: extracts a portion of the set, i.e. a range.
- **Endpoints**: returns the first and the last element in the sorted set.
- **Comparator access**: returns the comparator, if an, used to sort the set.

Hence the following interface abstracts a `SortedSet`:

```
public interface SortedSet<E> extends Set<E> {
        // Range-view
        SortedSet subSet(E fromElement, E toElement);
        SortedSet headSet(E toElement);
        SortedSet tailSet(E fromElement);
        // Endpoints
        E first();
        E last();
        // Comparator access
        Comparator<? super E> comparator();
}
```

Let's look at each type of operation in details.

Range view operations:

- `+ SortedSet` **`subSet(E fromElement, E toElement)`**: returns a sorted set which is a portion of the set whose elements range from `fromElement`, inclusive, to `toElement`, exclusive.

- `+ SortedSet` **`headSet(E toElement)`**: returns a sorted set which is a portion of the set whose elements are strictly less than `toElement`.

- `+ SortedSet` **`tailSet(E fromElement)`**: returns a sorted set which is a portion of the set whose elements are greater than or equal to `fromElement`.

Endpoint operations:

- `+ E` **`first():`** returns the first (lowest) element currently in the set.
- `+ E` **`last()`**: returns the last (highest) element currently in the set.

Comparator access:

+ **`comparator()`**: returns the comparator used to order the elements in the set, or null if this set uses the natural ordering of its elements.

Code Examples:

The following code example demonstrates how these operations work with a **`TreeSet`** implementation:

```
SortedSet<Integer> setNumbers = new TreeSet<>();
setNumbers.addAll(Arrays.asList(2, 1, 4, 3, 6, 5, 8, 7, 0, 9));
System.out.println("Original Set: " + setNumbers);
Integer first = setNumbers.first();
System.out.println("First element: " + first);
Integer last = setNumbers.last();
System.out.println("Last element: " + last);
SortedSet<Integer> subSet = setNumbers.subSet(3, 7);
System.out.println("Sub Set: " + subSet);
SortedSet<Integer> headSet = setNumbers.headSet(5);
System.out.println("Head Set: " + headSet);
SortedSet<Integer> tailSet = setNumbers.tailSet(5);
System.out.println("Tail Set: " + tailSet);
Comparator comparator = setNumbers.comparator();
System.out.println("Sorted by natural ordering? " + (comparator == null));
```

Output:

```
Original Set: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
First element: 0
Last element: 9
Sub Set: [3, 4, 5, 6]
Head Set: [0, 1, 2, 3, 4]
Tail Set: [5, 6, 7, 8, 9]
Sorted by natural ordering? true
```

The following code snippet shows how to use a comparator:

```
class ZtoAComparator implements Comparator<String> {
            public int compare(String s1, String s2) {
             return s2.compareTo(s1);
            }
}
SortedSet<String> setStrings = new TreeSet<>(new ZtoAComparator());
setStrings.add("Banana");
setStrings.add("Apple");
setStrings.add("Grape");
setStrings.add("Lemon");
setStrings.add("Watermelon");
```

```
System.out.println(setStrings);
```

Output:

```
[Watermelon, Lemon, Grape, Banana, Apple]
```

As you see, the specified comparator sorts the elements into descending order.

If you use Java 8, use Lambda expression to simplify the comparator class like this:

```
SortedSet<String> setStrings = new TreeSet<>((s1, s2) -> s2.compareTo(s1));
```

Understanding NavigableSet and TreeSet

Understand the `NavigabeSet` interface in the Java Collections Framework with code examples using `TreeSet`. Besides `Set` and `SortedSet`, `TreeSet` also implements `NavigableSet`.

Understanding NavigableSet:

`NavigableSet` is a sub interface of the `SortedSet` interface, so it inherits all `SortedSet's` behaviors like range view, endpoints and comparator access. In addition, the `NavigableSet` interface provides navigation methods and descending iterator that allows the elements in the set can be traversed in descending order.

Let's look at each new method defined by this interface in details.

- `lower(E):` returns the greatest element which is less than the specified element E, or `null` if there is no such element.
- `floor(E):` returns the greatest element which is less than or equal to the given element E.
- `ceiling(E):` returns the least element which is greater than or equal to the given element E.
- `higher(E):` returns the least element which is strictly greater than the specified element E.
- `descendingSet():` returns a reverse order view of the elements contained in the set.
- `descendingIterator():` returns an iterator that allows traversing over elements in the set in descending order.
- `pollFirst():` retrieves and removes the first (lowest) element, or returns `null` if the set is empty.
- `pollLast():` retrieves and removes the last (highest) element, or returns `null` if the set is empty.

Furthermore, the `NavigableSet` interface overloads the methods `headSet(), subSet()` and `tailSet()` from the `SortedSet` interface, which accepts additional arguments describing whether lower or upper bounds are inclusive versus exclusive:

- `headSet(E toElement, boolean inclusive)`
- `subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)`
- `tailSet(E fromElement, boolean inclusive)`

Now, let's look at some code examples.

Code Examples:

The following example shows how to obtain a reverse order set from the original one:

```
NavigableSet<Integer> setNumbers1 = new TreeSet<>();
setNumbers1.addAll(Arrays.asList(4, 8, 3, 9, 1, 6, 4, 5, 3, 2, 7, 8, 0));
NavigableSet<Integer> setNumbers2 = setNumbers1.descendingSet();
System.out.println("Set Numbers 1: " + setNumbers1);
System.out.println("Set Numbers 2: " + setNumbers2);
```

Output:

```
Set Numbers 1: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
Set Numbers 2: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

The following example illustrates how to obtain the descending iterator from a navigable set:

```
NavigableSet<String> setFruits = new TreeSet<>();
setFruits.add("Banana");
setFruits.add("Apple");
setFruits.add("Orange");
setFruits.add("Grape");
setFruits.add("Mango");
System.out.println("Set Fruits: " + setFruits);
Iterator<String> descIterator = setFruits.descendingIterator();
System.out.print("Fruits by descending order: ");
while (descIterator.hasNext()) {
        System.out.print(descIterator.next() + ", ");
}
```

Output:

```
Set Fruits: [Apple, Banana, Grape, Mango, Orange]
Fruits by descending order: Orange, Mango, Grape, Banana, Apple,
```

The following example demonstrates the benefits of using navigation methods like `lower()`, `higher()`, `ceiling()` and `floor()`; and range view methods like `headSet()`, `subSet()` and `tailSet()`.

Given the following entity class:

```
class Employee {
        String name;
        int salary;
        public Employee(int salary) {
                this.salary = salary;
        }
        public Employee(String name, int salary) {
                 this.name = name;
                this.salary = salary;
        }
        public String toString() {
                return this.name + "-" + salary;
        }
        public String getName() {
                return this.name;
        }
        public Integer getSalary() {
                 return new Integer(this.salary);
        }
        public boolean equals(Object obj) {
                if (obj instanceof Employee) {
```

```
                        Employee another = (Employee) obj;
                        if (this.name.equals(another.name)
                              && this.salary == another.salary) {
                                    return true;
                              }
                        }
                        return false;
                  }
                  public int hashCode() {
                         return 31 * name.hashCode() + salary;
                  }
      }
```

Note that this class overrides the `equals()` and `hashCode()` methods based on employee's name and salary. The following comparator class compares two employees based on their salary:

```
public class EmployeeComparator implements Comparator<Employee> {

            public int compare(Employee emp1, Employee emp2) {

                  return emp1.getSalary().compareTo(emp2.getSalary());

            }

}
```

We add 8 employees into a navigable set like this:

```
NavigableSet<Employee> setEmployees = new TreeSet<>(new EmployeeComparator());
setEmployees.add(new Employee("Tom", 80000));
setEmployees.add(new Employee("Jack", 35000));
setEmployees.add(new Employee("Jim", 62500));
setEmployees.add(new Employee("Peter", 58200));
setEmployees.add(new Employee("Mary", 77000));
setEmployees.add(new Employee("Jane", 69500));
setEmployees.add(new Employee("David", 54000));
setEmployees.add(new Employee("Sam", 82000));
System.out.println("Employees: " + setEmployees);
```

Output:

```
Employees: [Jack-35000, David-54000, Peter-58200, Jim-62500, Jane-69500, Mary-77000, Tom-
80000, Sam-82000]
```

Here, an employee object is printed with name and salary.

*** Using the `higher()` method, we can know the employee whose salary is higher than the employee 'Tom':**

```
Employee Tom = new Employee("Tom", 80000);
Employee emp1 = setEmployees.higher(Tom);
if (emp1 != null) {
```

```
        System.out.println("The employee whose salary > Tom: " + emp1);
    }
```

Output:

The employee whose salary > Tom: Sam-82000

NOTE: to allow this kind of search possible, the entity class must correctly override the `equals()` and `hashCode()` method, as shown in the `Employee` class above.

*** Using the `lower()` method, we can know the employee whose salary is less than the employee Tom:**

```
Employee emp2 = setEmployees.lower(Tom);
if (emp2 != null) {
            System.out.println("The employee whose salary < Tom: " + emp2);
}
```

Output:

```
 The employee whose salary < Tom: Mary-77000
```

*** Using the `ceiling()` method, we can know the employee whose salary is greater than 60,000 USD/year like this:**

```
Employee emp3 = setEmployees.ceiling(new Employee(60000));
if (emp3 != null) {
            System.out.println("The employee whose >= 60000: " + emp3);
}
```

Output:

```
The employee whose >= 60000: Jim-62500
```

*** Using the `floor()` method, we can know the employee whose salary is less than 40,000 USD like this:**

```
Employee emp4 = setEmployees.floor(new Employee(40000));
if (emp4 != null) {
            System.out.println("The employee whose <= 40000: " + emp4);
}
```

Output:

```
The employee whose <= 40000: Jack-35000
```

*** With the `tailSet()` method, we can know the employees who are high paid (salary > 70,000 USD):**

```
SortedSet<Employee> highPaidEmployees = setEmployees.tailSet(new Employee(70000));
System.out.println("High paid employees: " + highPaidEmployees);
```

Output:

```
High paid employees: [Mary-77000, Tom-80000, Sam-82000]
```

*** With the `headSet()` method, we can know the employees who are low paid (under 40,000USD/year):**

```
SortedSet<Employee> lowPaidEmployees = setEmployees.headSet(new Employee(60000));
System.out.println("Low paid employees: " + lowPaidEmployees);
```

Output:

```
Low paid employees: [Jack-35000, David-54000, Peter-58200]
```

*** With the `subSet()` method, we can know the employees who are good paid (salary is between 60,000 and 70,000):**

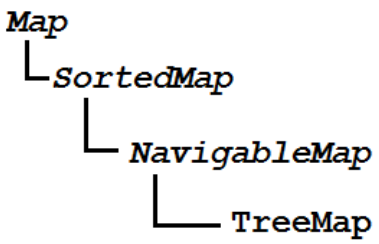
```
SortedSet<Employee> goodPaidEmployees = setEmployees.subSet(new Employee(60000), new
Employee(70000));

System.out.println("Good paid employees: " + goodPaidEmployees);
```

Output:

```
Good paid employees: [Jim-62500, Jane-69500]
```

Understanding SortedMap and TreeMap

`TreeMap` doesn't only implement the `Map` interface, it also implements the `SortedMap` and `NavigableMap` interfaces. Therefore, besides the behaviors inherited from the `Map`, `TreeMap` also inherits the behaviors defined by `SortedMap` and `NavigableMap`. The following picture depicts the API hierarchy of `TreeMap`:



Understanding SortedMap:

The main characteristic of a `SortedMap` is that, it orders the keys by their natural ordering, or by a specified comparator. So consider using a `TreeMap` when you want a map that satisfies the following criteria:

- null key or null value are not permitted.
- The keys are sorted either by natural ordering or by a specified comparator.

The following example realizes the concept of a `SortedMap`:

```
SortedMap mapDomains = new TreeMap<>();
mapDomains.put(".com", "International");
mapDomains.put(".us", "United States");
mapDomains.put(".uk", "United Kingdom");
mapDomains.put(".jp", "Japan");
mapDomains.put(".au", "Australia");
System.out.println(mapDomains);
```

Output:

```
{.au=Australia, .com=International, .jp=Japan, .uk=United Kingdom, .us=United States}
```

Here, this map contains mappings of domain=country, and as we see in the output, the domains (keys) are sorted by alphabetic order (natural ordering of Strings).

Besides the operations inherited from the `Map` interface, the `SortedMap` also defines the following operations:

- **Range view**: returns a sub sorted map whose keys fall within a range of keys in the original map.
- **Endpoints**: returns the first or last key in the sorted map.
- **Comparator access**: returns the comparator (implements the `Comparator` interface), if any, used to sort the map.

Hence the following code is the definition of a `SortedMap`:

```
public interface SortedMap extends Map{
            Comparator comparator();
            SortedMap subMap(K fromKey, K toKey);
            SortedMap headMap(K toKey);
            SortedMap tailMap(K fromKey);
            K firstKey();
```

```
        K lastKey();
}
```

Let's look at each type of operation in details.

Range View Operations:

- **+ subMap(K fromKey, K toKey):** returns a sorted map whose keys range from `fromKey`, inclusive, to `toKey`, exclusive.
- **+ headMap(K toKey):** returns a sorted map whose keys are strictly less than `toKey`.
- **+ tailMap(K fromKey):** returns a sorted map whose keys are greater than or equal to `fromKey`.

Endpoint operations:

- **+ firstKey():** returns the first (lowest) key currently in the map.
- **+ lastKey():** returns the last (highest) key currently in the map.

Comparator access:

- **+ comparator():** returns the comparator used to order the keys in the map, or returns `null` if this map uses the natural ordering of its keys.

The following code example demonstrates how to work with these operations on a `TreeMap`:

```
SortedMap mapHttpStatus = new TreeMap<>();
mapHttpStatus.put(100, "Continue");
mapHttpStatus.put(200, "OK");
mapHttpStatus.put(300, "Multiple Choices");
mapHttpStatus.put(400, "Bad Request");
mapHttpStatus.put(401, "Unauthorized");
mapHttpStatus.put(402, "Payment Required");
mapHttpStatus.put(403, "Forbidden");
mapHttpStatus.put(404, "Not Found");
mapHttpStatus.put(500, "Internal Server Error");
mapHttpStatus.put(501, "Not Implemented");
mapHttpStatus.put(502, "Bad Gateway");

        System.out.println("All key-value pairs: ");
        for (Integer code : mapHttpStatus.keySet()) {
                    System.out.println(code + " -> " + mapHttpStatus.get(code));
        }

System.out.println();
Integer firstKey = mapHttpStatus.firstKey();
String firstValue = mapHttpStatus.get(firstKey);
System.out.println("First status: " + firstKey + " -> " + firstValue);
```

```
        System.out.println();
        Integer lastKey = mapHttpStatus.lastKey();
        String lastValue = mapHttpStatus.get(lastKey);
        System.out.println("Last status: " + lastKey + " -> " + lastValue);

        System.out.println();
        SortedMap map4xxStatus = mapHttpStatus.subMap(400, 500);
        System.out.println("4xx Statuses: ");
        for (Integer code : map4xxStatus.keySet()) {
                System.out.println(code + " -> " + map4xxStatus.get(code));
        }

        System.out.println();
        SortedMap mapUnder300Status = mapHttpStatus.headMap(300);
        System.out.println("Statuses < 300: ");
        for (Integer code : mapUnder300Status.keySet()) {
                System.out.println(code + " -> " + mapUnder300Status.get(code));
        }

        System.out.println();
        SortedMap mapAbove500Status = mapHttpStatus.tailMap(500);
        System.out.println("Statuses > 500: ");
        for (Integer code : mapAbove500Status.keySet()) {
                System.out.println(code + " -> " + mapAbove500Status.get(code));
        }

        Comparator comparator = mapHttpStatus.comparator();
        System.out.println("Sorted by natural ordering? " + (comparator == null));
```

Output:

```
        All key-value pairs:
        100 -> Continue
        200 -> OK
        300 -> Multiple Choices
        400 -> Bad Request
        401 -> Unauthorized
        402 -> Payment Required
        403 -> Forbidden
        404 -> Not Found
        500 -> Internal Server Error
        501 -> Not Implemented
        502 -> Bad Gateway
```

```
First status: 100 -> Continue

Last status: 502 -> Bad Gateway
4xx Statuses:
400 -> Bad Request
401 -> Unauthorized
402 -> Payment Required
403 -> Forbidden
404 -> Not Found

Statuses < 300:
100 -> Continue
200 -> OK

Statuses > 500:
500 -> Internal Server Error
501 -> Not Implemented
502 -> Bad Gateway
Sorted by natural ordering? true
```

And the following example shows how to use a comparator:

```
SortedMap mapHttpStatus = new TreeMap<>(new ReverseComparator());
mapHttpStatus.put(100, "Continue");
mapHttpStatus.put(200, "OK");
mapHttpStatus.put(300, "Multiple Choices");
mapHttpStatus.put(400, "Bad Request");
mapHttpStatus.put(401, "Unauthorized");
mapHttpStatus.put(402, "Payment Required");
mapHttpStatus.put(403, "Forbidden");
mapHttpStatus.put(404, "Not Found");
mapHttpStatus.put(500, "Internal Server Error");
mapHttpStatus.put(501, "Not Implemented");
mapHttpStatus.put(502, "Bad Gateway");

for (Integer code : mapHttpStatus.keySet()) {
        System.out.println(code + " -> " + mapHttpStatus.get(code));
}
```

Here's the code of the comparator class:

```
class ReverseComparator implements Comparator {
        public int compare(Integer num1, Integer num2) {
                return num2.compareTo(num1);
```

```
                }
        }
```

Output:

```
502 -> Bad Gateway
501 -> Not Implemented
500 -> Internal Server Error
404 -> Not Found
403 -> Forbidden
402 -> Payment Required
401 -> Unauthorized
400 -> Bad Request
300 -> Multiple Choices
200 -> OK
100 -> Continue
```

As you can see, this comparator sorts the map by the descending order of its keys.

In case you are working on Java 8, use Lambda expressions to shorten the comparator code like this:

```
SortedMap mapHttpStatus = new TreeMap<>((i1, i2) -> i2.compareTo(i1));
```

Understanding NavigableMap and TreeMap

Understanding NavigableMap:

`NavigableMap` is sub interface of `SortedMap` interface, so it inherits all behaviors of a sorted map like range view, endpoints and comparator access operations. In addition, the `NavigableMap` interface provides navigation methods and descending views like `NavigableSet`. And due to the nature of a map which stores key-value mappings, the additional APIs are designed for both keys and key-value entries in the map.

Let's look at these methods in details.

Operations on key-value mappings (entries):

- **`lowerEntry(K key):`** returns a key-value mapping associated with the greatest key strictly less than the given key.
- **`floorEntry(K key):`** returns a key-value mapping entry which is associated with the greatest key less than or equal to the given key.
- **`ceilingEntry(K key):`** returns an entry associated with the lest key greater than or equal to the given key.
- **`higherEntry(K key):`** returns an entry associated with the least key strictly greater than the given key.
- Note that all these methods return null if there is no such key.
- **`firstEntry():`** returns a key-value mapping associated with the least key in the map, or null if the map is empty.
- **`lastEntry():`** returns a key-value mapping associated with the greatest key in the map, or null if the map is empty.
- **`descendingMap():`** returns a reverse order view of the mappings contained in the map.
- **`pollFirstEntry():`** removes and returns a key-value mapping associated with the least key in the map, or null if the map is empty.
- **`pollLastEntry():`** removes and returns a key-value mapping associated with the greatest key in the map, or null if the map is empty.

Operations on keys only:

- **`lowerKey(K key):`** returns the greatest key strictly less than the given key.
- **`floorKey(K key):`** returns the greatest key less than or equal to the given key.
- **`ceilingKey(K key):`** returns the least key greater than or equal to the given key.
- **`higherKey(K key):`** returns the least key strictly greater than the given key.
- **`descendingKeySet():`** returns a `NavigableSet` containing the keys in reverse order.

Note that all these methods return null if there is no such key.

Furthermore, the `NavigableMap` interface overloads the `headMap(), subMap()` and `tailMap()` methods of the `SortedMap` interface, which accept additional arguments describing whether lower or upper bounds are inclusive versus exclusive:

```
- headMap(K toKey, boolean inclusive)
- subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)
- tailMap(K fromKey, boolean inclusive)
```

Now, let's look at some code examples.

NavigableMap Examples with TreeMap:

The following example shows us how to obtain the reverse order view of the keys in a map:

```
NavigableMap mapHttpStatus = new TreeMap<>();


mapHttpStatus.put(100, "Continue");
mapHttpStatus.put(200, "OK");
mapHttpStatus.put(400, "Bad Request");
mapHttpStatus.put(401, "Unauthorized");
mapHttpStatus.put(500, "Internal Server Error");
mapHttpStatus.put(501, "Not Implemented");



Set ascendingKeys = mapHttpStatus.keySet();
System.out.println("Ascending Keys: " + ascendingKeys);
Set descendingKeys = mapHttpStatus.descendingKeySet();
System.out.println("Descending Keys: " + descendingKeys);
```

Output:

```
Ascending Keys: [100, 200, 400, 401, 500, 501]
Descending Keys: [501, 500, 401, 400, 200, 100]
```

Given the above map, the following code snippet gets a reverse order view of the map:

```
NavigableMap descendingMap = mapHttpStatus.descendingMap();
for (Integer key : descendingMap.keySet()) {
            System.out.println(key + " => " + descendingMap.get(key));
}
```

Output:

```
501 => Not Implemented
```

```
500 => Internal Server Error
401 => Unauthorized
400 => Bad Request
200 => OK
100 => Continue
```

Using operations on keys is explained in the following example:

```
Integer lowerKey = mapHttpStatus.lowerKey(401);
System.out.println("Lower key: " + lowerKey);
Integer floorKey = mapHttpStatus.floorKey(401);
System.out.println("Floor key: " + floorKey);
Integer higherKey = mapHttpStatus.higherKey(500);
System.out.println("Higher key: " + higherKey);
Integer ceilingKey = mapHttpStatus.ceilingKey(500);
System.out.println("Ceiling key: " + ceilingKey);
```

Output:

```
Lower key: 400
Floor key: 401
Higher key: 501
Ceiling key: 500
```

And the following example demonstrates how to work with operations on key-value mapping entries:

```
Map.Entry firstEntry = mapHttpStatus.firstEntry();
System.out.println("First entry: " + firstEntry.getKey() + " => " + firstEntry.getValue());
Map.Entry lastEntry = mapHttpStatus.lastEntry();
System.out.println("Last entry: " + lastEntry.getKey() + " => " + lastEntry.getValue());
Map.Entry lowerEntry = mapHttpStatus.lowerEntry(401);
System.out.println("Lower entry: " + lowerEntry.getKey() + " => " + lowerEntry.getValue());
Map.Entry floorEntry = mapHttpStatus.floorEntry(401);
System.out.println("Floor entry: " + floorEntry.getKey() + " => " + floorEntry.getValue());
Map.Entry higherEntry = mapHttpStatus.higherEntry(500);
System.out.println("Higher entry: " + higherEntry.getKey() + " => " +
higherEntry.getValue());
Map.Entry ceilingEntry = mapHttpStatus.ceilingEntry(500);
System.out.println("Ceiling entry: " + ceilingEntry.getKey() + " => " +
ceilingEntry.getValue());

mapHttpStatus.pollFirstEntry();
mapHttpStatus.pollLastEntry();
System.out.println("\nMap after first and last entries were polled:");
for (Integer key : mapHttpStatus.keySet()) {
            System.out.println(key + " => " + mapHttpStatus.get(key));
}
```

Output:

```
First entry: 100 => Continue
Last entry: 501 => Not Implemented
Lower entry: 400 => Bad Request
Floor entry: 401 => Unauthorized
Higher entry: 501 => Not Implemented
Ceiling entry: 500 => Internal Server Error
Map after first and last entries were polled:
200 => OK
400 => Bad Request
401 => Unauthorized
500 => Internal Server Error
```