Theory: Queue and Stack

1 hour

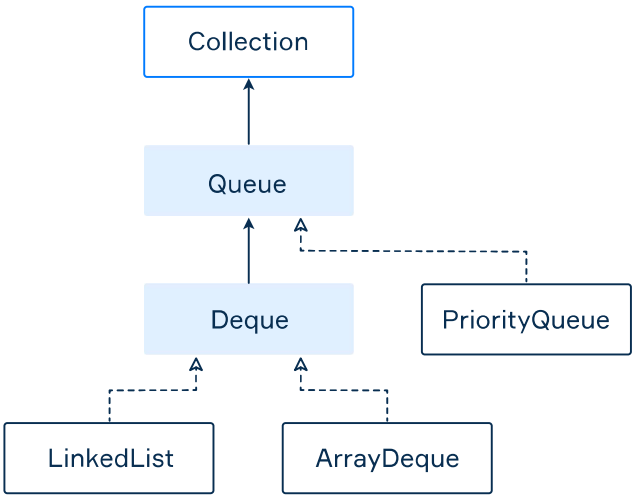
Verify to skip | Start practicing

A **queue** is a collection with limited access to elements: elements are inserted at the end of it and removed from the beginning. The collection follows the **first-in-first-out (FIFO)** principle. Queues are designed for holding elements prior to processing: tasks, events, or something else.

An excellent real-life example of a queue is a line of students in the food court. New additions to a line made to the back of the queue, while removal (or serving) happens in the front.

§1. Implementations of Queue

In Java, all queues are represented by the `Queue<E>` interface. However, the hierarchy of queues is more complex than it seems at first glance. The primary implementations of the `Queue<E>` are `LinkedList<E>` and `ArrayDeque<E>`. There is also a `PriorityQueue` which we will consider in a separate topic.



Both of the primary implementations inherit the `Deque<E>` interface which extends `Queue<E>` and represents a **double-ended queue** that supports **FIFO** and **LIFO** access principles. At the same time, the `LinkedList<E>` class also implements the `List<E>` interface, so it can be used as a queue and as a list depending on the task.

> If you are writing a program that processes a small number of elements, there is not much difference as to which implementation to use. But for processing a huge number of elements, `ArrayDeque` is more memory efficient than `LinkedList` since it does not need to create internal nodes for every element. Here you can find a more detailed discussion.

§2. The Queue interface

The `Queue<E>` interface extends `Collection<E>` and adds some new methods:

- `boolean offer(E e)` inserts the specified element into the queue if it is possible to do so immediately without violating capacity restrictions; it returns `true` / `false` depending on the result of this operation;
- `E remove()` retrieves and removes the head of this queue; if it's empty, the method throws `NoSuchElementException`;
- `E poll()` retrieves and removes the head of this queue, or returns `null` if this queue is empty;

2 required topics

✓ Deque — In project

✓ The Collections Framework overview — In project

4 dependent topics

Iterator and Iterable

Choosing the right collection

Memento

Concurrent queues

- `E element()` retrieves, but does not remove, the head of the queue; if it's empty, the method throws `NoSuchElementException`;
- `E peek()` retrieves, but does not remove, the head of this queue, or returns `null` if this queue is empty.

The differences between some of these methods and methods inherited from `Collection` may not be so obvious. Let's see:

- the `add(E e)` method does the same as `offer(E e)` but throws `IllegalStateException` if no space is currently available;
- `remove()` and `element()` throws `NoSuchElementException` where the queue is empty, but `poll()` and `peek()` just return `null` in this case.

In practice, you will use `offer`, `peek` and `poll` more often than others.

§3. Using ArrayDeque as a queue

Let's consider an example of how to use `ArrayDeque` as a queue (**FIFO**).

```
1  Queue<String> q = new ArrayDeque<>();
2
3  q.offer("first");
4  q.offer("second");
5  q.offer("third");
6
7  System.out.println(q.peek()); // first
8  System.out.println(q.peek()); // first
9  System.out.println(q.poll()); // first,
10
11 System.out.println(q.peek()); // second
12 System.out.println(q.poll()); // second
13 System.out.println(q.poll()); // third
14
15 System.out.println(q.isEmpty()); // true
```

Just remember, `peek()` returns the current head element, but does not remove it from the queue, whereas `poll()` does it. The use of the `LinkedList` as a `Queue` is similar.

§4. Deque

As we mentioned before, `Deque<E>` extends `Queue<E>` and represents a queue where you can insert and remove elements from both ends. It combines access rules provided by queue (FIFO) and stack (LIFO) together.

The **Deque** interface provides methods for working with the first and the last element of a queue. Some of the methods throw an exception, while others just return a special value (`null`). Check out the table:

	First Element (Head)		Last Element (Tail)	
	Throws exception	Special value	Throws exception	Special value
Insert	addFirst (e)	offerFirst (e)	addLast (e)	offerLast (e)
Remove	removeFirst ( )	pollFirst ( )	removeLast ( )	pollLast ( )
Examine	getFirst ( )	peekFirst ( )	getLast ( )	peekLast ( )

Since `ArrayDeque` and `LinkedList` implement this interface, they both can work as a queue (FIFO), a stack (LIFO), or a deque.

We will consider an example of how to use it further.

§5. Deque as a stack

As you probably remember, **Stack** is an abstract data type where elements are inserted and removed according to the **last-in-first-out (LIFO)** principle. The simplest real-life example is a stack of books. Only a book placed at the top can be removed at a time, but a new book is always added on the top of the stack.

> ***The Standard Class Library*** provides the `Stack` class, but, according to JavaDoc, a more complete and consistent set of **LIFO** stack operations is provided by the `Deque` interface and its implementations, which should be used in preference to this class. So, it is recommended to use `Deque` for stacks.

In the following example, we illustrate some operations of `Deque` using it as a stack.

```
Deque<String> stack = new ArrayDeque<>();

stack.offerLast("first");
stack.offerLast("second");
stack.offerLast("third");

System.out.println(stack); // [first, second, third]

System.out.println(stack.pollLast()); // third
System.out.println(stack.pollLast()); // second
System.out.println(stack.pollLast()); // first

System.out.println(stack.pollLast()); // null
```

As you can see, it really works. In addition, the `ArrayDeque` implementation is quite efficient for representing large stacks.

§6. The old Stack class

Sometimes, the old `Stack<E>` class with a more minimalistic API can be found in legacy source code. It doesn't implement `Deque` or `Queue` interface. Here is a simple example.

```
Stack<String> stack = new Stack<>();

stack.push("first");
stack.push("second");
stack.push("third");

System.out.println(stack); // [first, second, third]

System.out.println(stack.pop()); // "third"
System.out.println(stack.pop()); // "second"
System.out.println(stack.pop()); // "first"

System.out.println(stack.pop()); // throws EmptyStackException
```

The method `pop()` always throws an exception if the stack is empty.

Do not forget, according to the Java Doc, it's preferable to use implementations of the `Deque` interface as stacks.

§7. Conclusion

We've considered the `Queue` interface, its subtype `Deque` interfaces as well as two of its implementations.

If you need to work with a queue **(**FIFO**)**, try to use `ArrayDeque` via the standard `Queue` interface. This implementation is quite efficient, and the interface provides all the required operations. If you need to work with a stack (LIFO) or deque (FIFO+LIFO), try to use `ArrayDeque` via the `Deque` interface which provides

Table of contents:

operations for both ends of a queue. The `LinkedList` can also be used as an implementation, but it is considered less memory efficient when working with a large number of elements.

Sometimes you can find an old `Stack` class in the source code, but it is recommended to avoid it in the new code.

Report a typo

230 users liked this piece of theory. 5 didn't like it. **What about you?**

Start practicing

Verify to skip

Comments (16) Hints (0) Useful links (0) Show discussion