Concurrent queues

One of the most popular kinds of concurrent collections is a **concurrent queue**.

Concurrent queues are often used to organize communication between multiple threads within an application by exchanging data (messages, tasks, units of work, or something else).

To achieve this communication, several threads should have a reference to a common queue and invoke its methods.

You already know that a queue is a collection that works according to the **first-in-first-out principle** (FIFO): the first element added to the queue will be the first one to be removed.

Thread-safety of a ConcurrentLinkedQueue

The simplest type of concurrent queue is a ConcurrentLinkedQueue.

A ConcurrentLinkedQueue is very similar to a standard queue but is **thread-safe**.

It has two methods called add and offer to add an element to the tail of a queue.

The following example demonstrates the thread safety of this concurrent queue. The program adds new elements to this queue using two threads and then prints the total number of elements:

```
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;

public class ConcurrentQueueDemo {

    public static void main(String[] args) throws InterruptedException {
        // assign a thread-safe implementation
        Queue<Integer> numbers = new ConcurrentLinkedQueue<>();

        Thread writer = new Thread(() -> addNumbers(numbers));
        writer.start();

        addNumbers(numbers); // add numbers from the main thread

        writer.join(); // wait for the writer thread

        System.out.println(numbers.size()); // prints 200_000
```

```
    }

    private static void addNumbers(Queue<Integer> target) {
        for (int i = 0; i < 100_000; i++) {
            target.add(i);
        }
    }
}
```

It is not surprising that this program always prints 200000. As expected, no element is lost. You may start this program as many times as you need.
So, ConcurrentLinkedQueue is thread-safe. There is also no ConcurrentModificationException if we want to iterate over the elements of this queue.

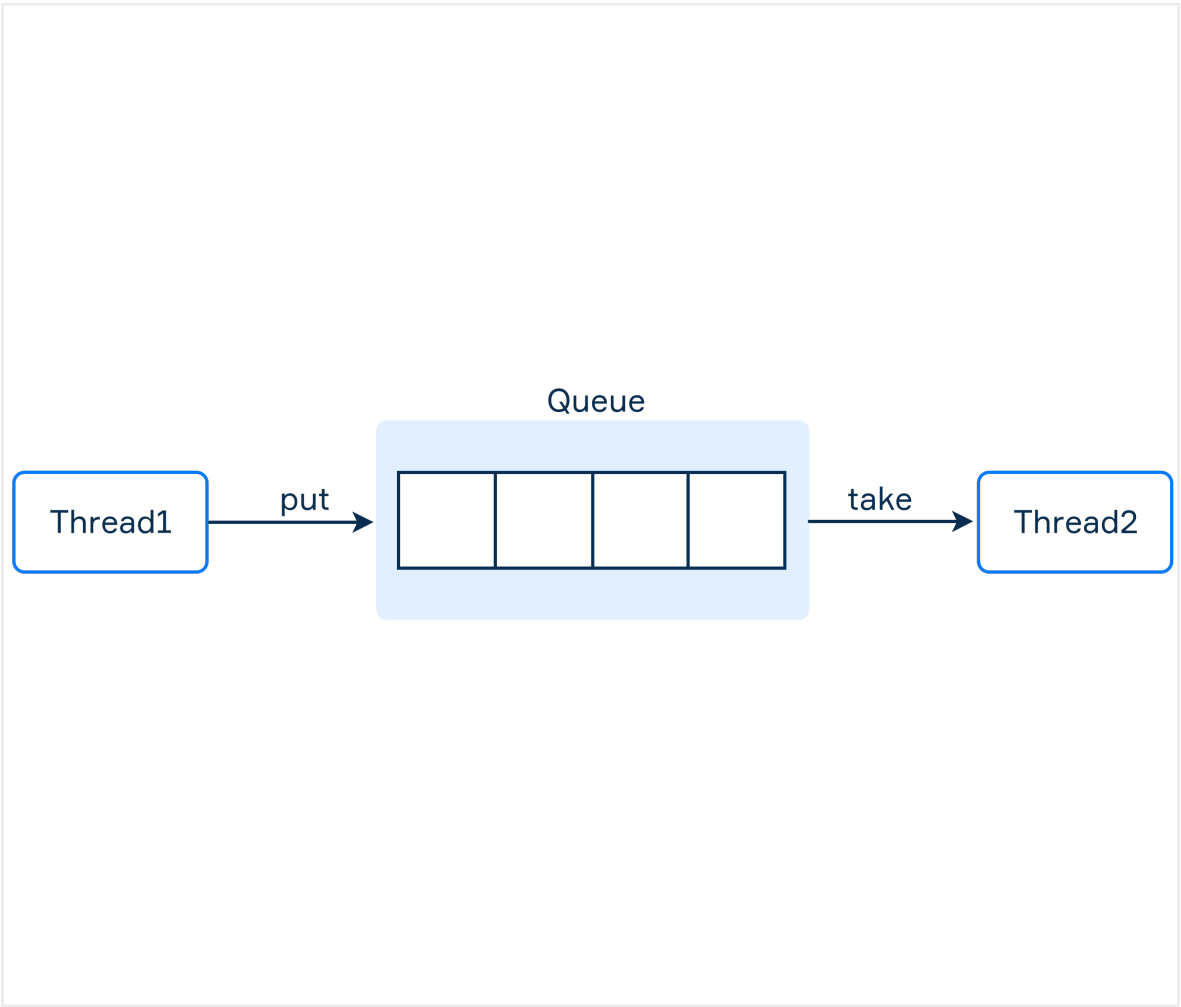
Note that any single operation provided by this queue is thread-safe.

However, if we group such operations in a single method or a sequence of statements, the group of operations will not be thread-safe.

Moreover, bulk operations of ConcurrentLinkedQueue that add, remove, or examine multiple elements, such as addAll, removeIf, and forEach, are *not* guaranteed to be performed atomically.

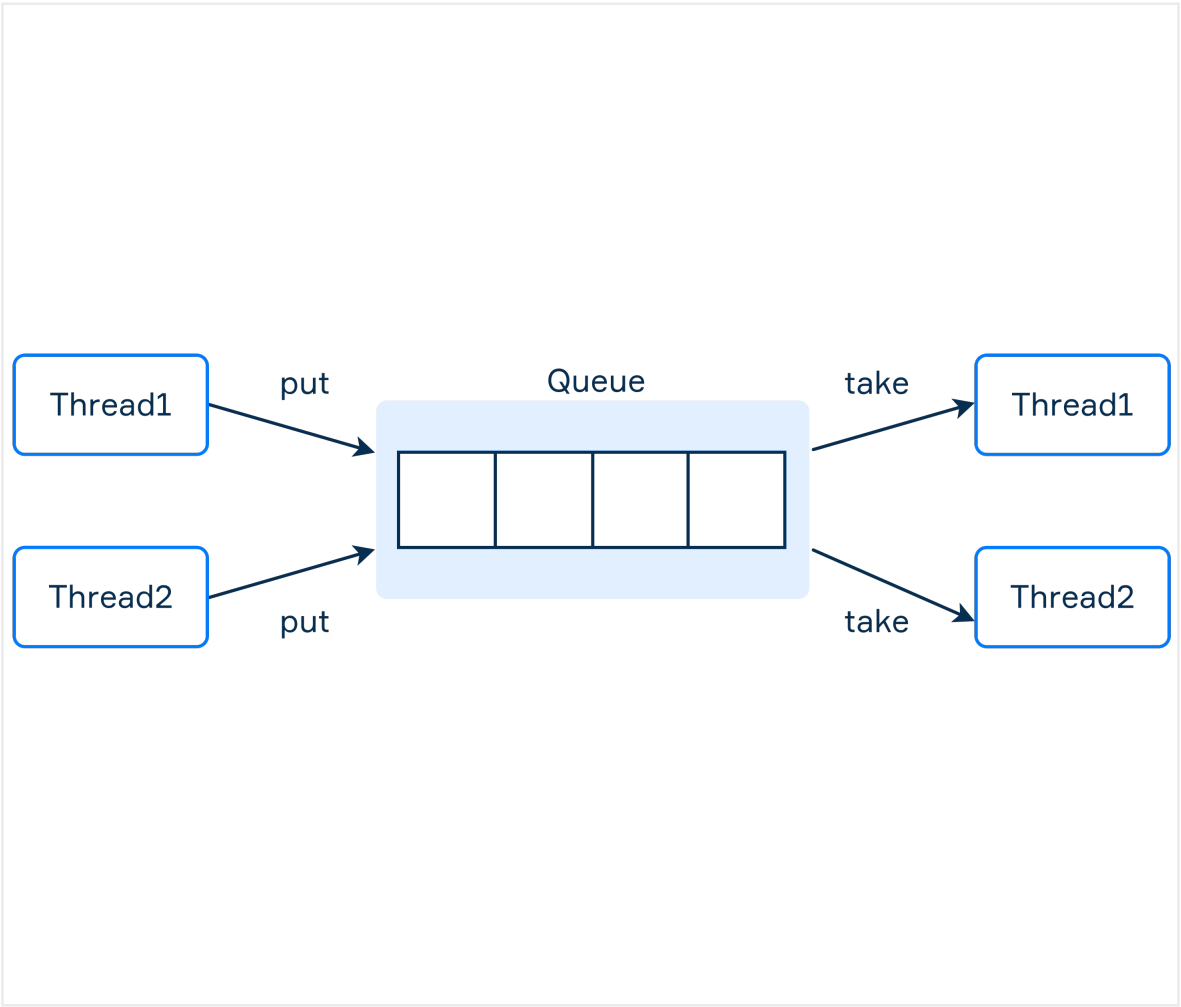
Communication between threads

The following picture demonstrates how to organize communication between threads using a queue. One thread adds elements to the tail of the queue, while another thread takes elements from its head.



Here, Queue has to be thread-safe. Otherwise, it will not work correctly.

Thread-safe communication is also possible when more than two threads interact through a queue.



The number of threads can be different than in this picture.

Suppose we want to exchange data between two threads using a concurrent queue. One thread will generate three numbers, while another thread will accept these numbers and print them.

The poll method can get the current first element of a concurrent queue. It returns this element or null if the queue is empty.

Here is a snippet of code with sleep invocations to make the output more predictable. The generator and poller threads interact using a concurrent queue, and no data is lost because the queue is fully thread-safe.

```
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.concurrent.TimeUnit;

public class GeneratorDemo {
```

```
    public static void main(String[] args) {
        Queue<Integer> queue = new ConcurrentLinkedQueue<>();

        Thread generator = new Thread(() -> {
            try {
                queue.add(10);
                TimeUnit.MILLISECONDS.sleep(10);
                queue.add(20);
                TimeUnit.MILLISECONDS.sleep(10);
                queue.add(30);
            } catch (Exception e) {
                e.printStackTrace();
            }
        });

        Thread poller = new Thread(() -> {
            int countRead = 0;
            while (countRead != 3) {
                Integer next = queue.poll();
                if (next != null) {
                    countRead++;
                }
                System.out.println(next);
                try {
                    TimeUnit.MILLISECONDS.sleep(10);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });

        generator.start();
        poller.start();
    }
}
```

Here is an example of the possible output:
null
10
20
null
30
In practice, the output may be different, but all numbers should be printed.

Composite operations

Every standard method of a concurrent queue provides thread safety. However, if you want to compose several methods, there are no such guarantees. Suppose you want to add two elements to a concurrent queue so that they follow each other:

```
public static void addTwoElements(ConcurrentLinkedQueue<Integer> queue,
int e1, int e2) {
    queue.add(e1); // (1)
    queue.add(e2); // (2)
}
```

This method will only add two elements in the specified order if we have one writing thread. If there are more writing threads, one thread may perform (1), and another may intervene and do the same. Afterward, the first thread may perform (2). Thus, the order can be broken in some cases. This problem occurs because the method is not **atomic**.

As mentioned above, bulk methods such as addAll are not atomic and don't help to avoid this problem:

```
queue.addAll(List.of(e1, e2));
```

The problem can only be solved by external synchronization, for example:

```
public static synchronized void
addTwoElements(ConcurrentLinkedQueue<Integer> queue, int e1, int e2) {
    queue.add(e1); // (1)
    queue.add(e2); // (2)
}
```

In this case, you need to be sure that all operations which update the queue are synchronized, not only the method addTwoElements.