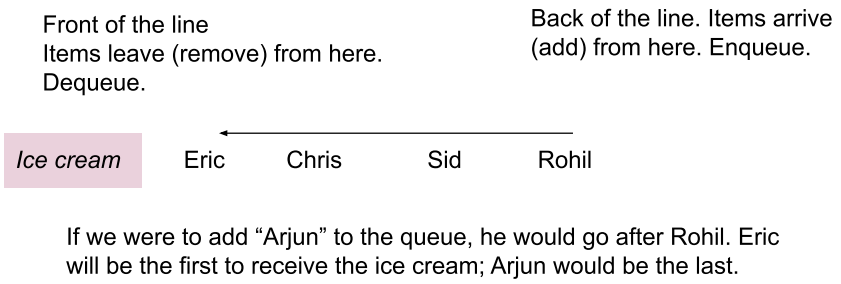
Queue, FIFO and LILO

A queue is an interface in the Collections framework.

The queue has two actions. You can **enqueue**, or add to the back of a queue, or **dequeue**, or remove from the head of the queue.



If we were to add "Arjun" to the queue, he would go after Rohil. Eric will be the first to receive the ice cream; Arjun would be the last.

Think of it like a line of people! The first person to enter the line will be the first out of the line (first to get on the ride). The last person to enter the line will be the last to exit the line (and sadly, the last to get on the ride). This is the "First In, First Out" and "Last In, Last Out" principle.

The front of the queue is like the start of a List or array (left). The back of the queue can be thought of as the rightmost element, also just like a List or array.

Implementing the Queue Interface

A queue is typically implemented via LinkedList.

Java provides a LinkedList class.

LinkedList is designed to add to the front of the list in O(1).

- Pseudo code: `x = new Node(new value, x);`

Java also keeps a reference to the last node of a Linked List. So, we can add to the back of the list in O(1) (as opposed to traversing the linked list until we hit a null pointer).

- Pseudo code: `x.lastNode.setNext(new Node(new value, null))`

This is good because the queue needs to be removed from the head (front of LinkedList), or dequeue, and added to the back (back of LinkedList), or enqueue, very fast.

The LinkedList class in java is also doubly linked, meaning it can be traversed both forwards and backwards.

Let's move away from PseudoCode into real Java. To make a queue of Integers:

```
Queue<Integer> q = new LinkedList<Integer>();
```

Methods of the Queue Interface

`boolean add(E item)` //adds item to the back of the queue. This is an *enqueue.*

`E remove()` //returns the object it just removed from the head of the queue.

- Do not confuse this with Stack's method for removal, `E pop()`

`E peek()` //reference to the object at the head. *Does not remove (aka does not dequeue).*

`boolean isEmpty()` //true if list is empty, false otherwise

`int size()` //returns # of objects in the queue

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

There are two versions of the methods to insert elements, remove elements, and examine the element at the head of the queue. The methods listed above are preferred.

Examples of Using the Methods for Queues

Example 1: Find the output of:

```
Queue<Double> dsa = new LinkedList<Double>();
for(double i = .5; i >= -3.5; i--) {
      dsa.add(i);
}
dsa.remove();
dsa.peek();
int count = 1;
while(!dsa.isEmpty()) {
      System.out.print(dsa.peek() + " ");
      if(count % 2 == 0) {
            dsa.remove();
      }
      count++;
}
```

The for-loop makes dsa look like: [.5, -.5, -1.5, -2.5, -3.5]

The remove takes off the .5, and the peek does not change the queue.

The while loop will print every element twice.

Output: `-0.5 -0.5 -1.5 -1.5 -2.5 -2.5 -3.5 -3.5`

Note: `dsa` is now empty.

Example 2: Find the output of:

```
Queue<String> usaco =  new LinkedList<String>();
usaco.add("bronze");
usaco.add("silver");
usaco.add("gold");
usaco.add(usaco.peek());
System.out.println(usaco);
```

The first three method calls to add make usaco look like: [bronze, silver, gold]
The fourth add will put "bronze" at the end of the list: [bronze, silver, gold, bronze]
Output: `[bronze, silver, gold, bronze]`

Example 3: Explain why the following would produce an error:

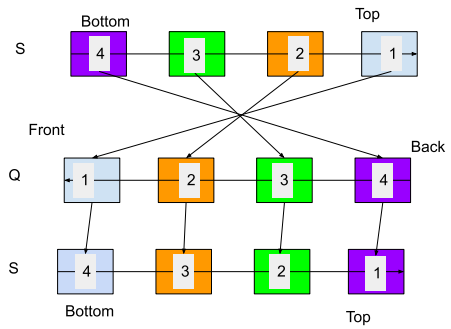
```
Queue<Long> xyz = new LinkedList<Long>();
xyz.add(1L);
xyz.set(0,14L);
xyz.add(124L);
xyz.add(56L);
xyz.remove();
```

The use of the `set` method will not work. Declaring the variable `xyz` with the `Queue` interface means it can only use `Queue Abstract Data Type` methods (ADT). If we wanted to use the `set` method, we would just declare the variable as a `LinkedList` type. Every other statement works fine.

Traverse an Entire Queue (and not Leave it Empty)

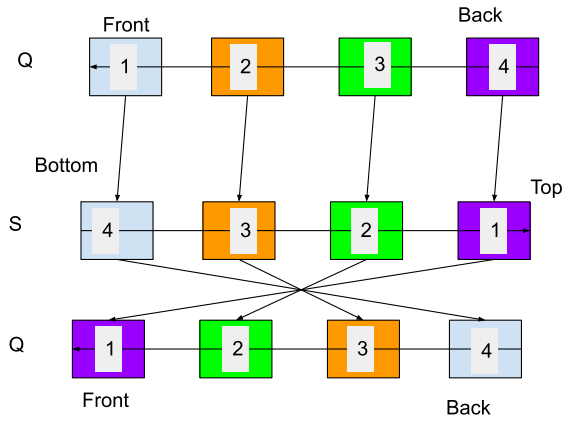
```
int len = q.size();
for(int i = 0; i < len; i++) {
      q.add(q.peek());
      //do something with the element at the front
      q.remove();
}
```

Reversing a Stack, Reversing a Queue



(Above Photo)Reverse a Stack: Empty the stack into the queue. Then, empty the queue into the stack.

```
while(!s.isEmpty()){
      q.add(s.pop());
}
while(!q.isEmpty()){
      s.add(q.remove());
}
```



Numbers show the order in which the elements are accessed.

(Above photo) Reverse a Queue: Empty the queue into the stack, and then empty the stack back into the queue. See picture.

```
while(!q.isEmpty()) {
      s.add(q.remove());
}
while(!s.isEmpty()) {
            q.add(s.pop());
}
```

Notice how in the picture, emptying out a queue preserves the color order, when viewed left-right, but because we went to a stack, the notion of "start" and "end" have flipped sides. So, the stack has flipped the original order.
Emptying out a stack will reverse the color order, when viewed left-right. However, this gets canceled by again changing the notion of "start/end". Reverse of a reverse = original.

So, the only reversal that matters is by the stack flipping the notion of "start/end".

This means:

- Stack → Queue. (Empty a stack into a queue)
    - We can read elements in the same order compared to before. Order stays the same.

- Queue→ Stack. (Empty a queue into a stack)
    - We now read elements in reverse order compared to before. Order reverses.

-

Using Both Stacks and Queues to Solve Problems

Examples:

1. **Print out everything from a queue of integers in order. Restore the queue to its original order. Use only a stack as auxiliary storage.**
    a. **The queue *`front [1,2,4,-5] back`* prints *`1 2 4 -5`***

Solution 1:
1.Use a while loop to print out the queue. To get to the next element, remove the element, and add the removed element to the stack.
2.At this point, emptying the stack will be the reverse of the original queue. Reverse the stack.
3. The stack will now empty in the original order of the queue. Empty the stack into the queue.

```
while(q.size() > 0){
        System.out.print(q.peek() + " ");
        s.add(q.remove());
    }
    //reverse stack
    while(s.size() > 0) {
        q.add(s.pop());
    }
    while(q.size() > 0) {
        s.add(q.remove());
    }
    //empty stack into queue
    while(s.size() > 0 ) {
        q.add(s.pop());
    }
```

Solution 2:
1.Traverse the queue and not leave it empty. No auxiliary necessary.

```
int len = q.size();
for(int i = 0; i < len; i++) {
     System.out.print(q.peek() + " ");
     q.add(q.remove());
}
```

2. **The length of a stack of integers will be even. Using one queue only as auxiliary storage, print out the sum of the pairs, from outermost to innermost. Return the stack to its original order.**
    a. **The stack *bottom [1, 2 ,3 ,6, 13, 12] top* will print out *13 15 9***

Solution:
1.Empty the stack into the queue.
2.Traverse the queue without destroying it and add a copy of everything into the stack. Going through the queue will be the original order, going through the stack is the reverse order.
3. Empty the stack. Print out the sum of the queue head and stack top for the first half of the stack length. Also make sure to traverse the queue without destroying it.
4.Emptying the queue (original order) into the stack. The stack is now in reverse order.
5.Reverse the stack.

```
while(s.size() > 0) {
          q.add(s.pop());
     }
int len = q.size();
for(int i = 0; i < len; i++) {
    s.add(q.peek());
    q.add(q.remove());
     }
len = s.size()/2;
while(!s.isEmpty()) {
    if(len > 0) {
      System.out.print((q.peek() + s.peek()) + " ");
       len--;
    }
    s.pop();
    q.add(q.remove());
    }
while(!q.isEmpty()) {
    s.add(q.remove());
    }
//reverse s
while(s.size() > 0 ){
    q.add(s.pop());
}
while(q.size() > 0) {
    s.add(q.remove());
```

```
}
```

Extra Practice:

https://www.cs.umd.edu/class/summer2018/cmsc132/javatest/queues/queues.html