```
System.out.println("stack = " + s);
queueToStack(q, s);
System.out.println("after queueToStack:");
System.out.println("queue = " + q);
System.out.println("stack = " + s);
```

It produces output like the following:

```
queue = [75, 76, 53, 82, 88, 77, 63, 28, 86, 7]
stack = []
after queueToStack:
queue = []
stack = [75, 76, 53, 82, 88, 77, 63, 28, 86, 7]
```

We can write a similar method `stackToQueue`:

```
public static void stackToQueue(Stack<Integer> s, Queue<Integer> q) {
    while (!s.isEmpty()) {
        int n = s.pop();
        q.add(n);
    }
}
```

Sum of a Queue

How would we write a method to find the sum of the values in a queue? It is a cumulative sum task, which involves initializing a `sum` variable to `0` outside the loop and then adding each value to the sum as we progress through the loop. A first attempt might look like this:

```
public static int sum(Queue<Integer> q) {
    int sum = 0;
    while (!q.isEmpty()) {
        int n = q.remove();
        sum += n;
    }
    return sum;
}
```

If you call this version of the method and print the queue afterward, you'll find that the queue is empty. As a side effect of calculating the sum, we are destroying the contents of the queue. This is generally not acceptable behavior.

So how do we examine each of the queue elements while preserving the state of the structure? You could ask the queue for an iterator or use a for-each loop to get each value from the structure, but we are interested in understanding the queue in its most basic form, so we aren't going to use either iterators or for-each loops when working with stacks and queues.

The only other way to find out what is in a queue is to remove each of the values. We just need to find a way to do that while restoring the queue to its original form when we are done. One solution is to use a second queue as auxiliary storage:

```
// Improved version that uses auxiliary queue for storage
// so that it does not destroy the queue passed in.
public static int sum(Queue<Integer> q) {
    int sum = 0;
    Queue<Integer> temp = new LinkedList<Integer>();
    while (!q.isEmpty()) {
        int n = q.remove();
        sum += n;
        temp.add(n);
    }

    while (!temp.isEmpty()) {    // restore the queue
        q.add(temp.remove());
    }

    return sum;
}
```

This approach works, but there is an easier way. Why not use the original queue itself? As we remove values to be processed, we can add them back into the queue at the end. Think of it as cycling through values where the value at the front goes to the back.

Of course, then the queue never becomes empty. So instead of a `while` loop looking for an empty queue, we would write a `for` loop using the size of the queue:

```
// Best version that re-adds elements to the queue passed in
// so that it does not destroy the contents of the queue.
public static int sum(Queue<Integer> q) {
    int sum = 0;
    for (int i = 0; i < q.size(); i++) {
        int n = q.remove();
        sum += n;
        q.add(n);
    }
    return sum;
}
```

Sum of a Stack

Now let's write a `sum` method for stacks:

```
public static int sum(Stack<Integer> s) {
    ...
}
```

This method can also be called `sum` because the two methods have different signatures. Remember that a signature of a method is its name plus its parameters. These are both called `sum` and they both have just a single parameter, but the parameter types are different, so this is okay.

So how do we write the `sum` method for stacks? We can start out by trying to simply substitute stack operations for queue operations:

```
public static int sum(Stack<Integer> s) {
   int sum = 0;
   for (int i = 0; i < s.size(); i++) {
       int n = s.pop();
       sum += n;
       s.push(n);
   }
   return sum;
}
```

Unfortunately, this code doesn't work. We'd see output like this when we test it:

```
stack = [42, 19, 78, 87, 14, 41, 57, 25, 96, 85]
sum = 850
```

The sum of these numbers is not 850. We're getting that sum because the loop pops the value `85` off the stack 10 different times and then pushes it back onto the top of the stack. With a queue, values go in at one end and come out the other end. But with a stack, all the action is at one end of the structure (the top). So this approach isn't going to work.

In fact, you can't solve this in a simple way with just a stack. You'd need something extra like an auxiliary structure. Consider how we could solve it if we were allowed to use one queue available (and only one queue; no other auxiliary structures) as temporary storage. Then we can put things into the queue as we take them out of the stack and after we have computed the sum, we can transfer things from the queue back to the stack using our `queueToStack` method. Here is a second attempt:

```
public static int sum(Stack<Integer> s) {
    int sum = 0;
    Queue<Integer> q = new LinkedList<Integer>();
    for (int i = 0; i < s.size(); i++) {
        int n = s.pop();
        sum += n;
        q.add(n);
    }
    queueToStack(q, s);
    return sum;
}
```

This also doesn't work. Here is a sample execution:

```
initial stack = [32, 15, 54, 91, 47, 45, 88, 89, 13, 0]
sum = 235
after sum stack = [32, 15, 54, 91, 47, 0, 13, 89, 88, 45]
```

There are two problems here. Only half of the values were removed from the stack and those values now appear in reverse order. Why only half? We are using a `for` loop that compares a variable `i` against the size of the stack. The variable `i` is going up by 1 while the size is going down by 1 every time. The result is that halfway through the process, `i` is large enough relative to size to stop the loop. This is a case where we want a `while` loop instead of a `for` loop:

```
public static int sum(Stack<Integer> s) {
    int sum = 0;
    Queue<Integer> q = new LinkedList<Integer>();
    while (!s.isEmpty()) {
        int n = s.pop();
        sum += n;
        q.add(n);
    }
    queueToStack(q, s);
    return sum;
}
```

Even this is not correct. It finds the right sum, but it ends up reversing the values in the stack. If we could use a stack instead of a queue as auxiliary storage, then this problem would go away. In many of our sample problems, we purposely restrict you to a particular kind of structure so that you can practice working within constraints. It becomes almost a brain teaser to think of how to solve the problem with a given structure. A stack would be more convenient, but that doesn't mean that you have to use a stack to solve the problem.

The problem is that by transferring the data from the stack into the queue and then back into the stack, we have reversed the order. The fix is to do it again so that it goes back to the original order. So we add two extra calls at the end of the method that move values from the stack back into the queue and then from the queue back into the stack:

```
public static int sum(Stack<Integer> s) {
    int sum = 0;
    Queue<Integer> q = new LinkedList<Integer>();
    while (!s.isEmpty()) {
        int n = s.pop();
        sum += n;
        q.add(n);
    }
```

```
    queueToStack(q, s);
    stackToQueue(s, q);
    queueToStack(q, s);
    return sum;
}
```

This is not the most efficient way to solve the problem, but it demonstrates that it can be done with a queue.

14.3 Complex Stack/Queue Operations

In this section we will examine two harder stack/queue problems that allow us to explore some of the issues and common bugs that come up when manipulating these structures.

Removing Values from a Queue

Consider the task of removing all occurrences of a certain value from a queue of integers. For example, suppose that a variable called `q` stores a reference to the following queue:

```
[18, 4, 7, 42, 9, 33, -8, 0, 14, 42, 7, 42, 42, 19]
```

and we make the following call:

```
removeAll(q, 42);
```

We would want the queue to store the following values after the call:

```
[18, 4, 7, 9, 33, -8, 0, 14, 7, 19]
```

This seems like a fairly straightforward task, but it leads to a subtle bug if we aren't careful. This task would be easier to solve if we could use a second queue. Then we could copy the values we want to keep from the first queue to the second and then copy them back when we are done. But this problem can be solved without an auxiliary data structure.

We saw that we can cycle through the elements of a queue by repeatedly removing the front value and then adding it back at the end of the queue. For example, the following loop cycles through the values in the queue exactly once:

```
for (int i = 0; i < q.size(); i++) {
   int n = q.remove();
   q.add(n);
}
```