# Lecture 10

## Exceptions

The class web page has a link to the Streams cheat sheet, with helpful information for doing File I/O.

When doing I/O, we need to handle IOExceptions that may be thrown by the stream's methods. We do this by wrapping the calls in `try..catch` block.

Here's an example: http://www.exampledepot.com/egs/java.io/readlinesfromfile.html

## Linked Lists

One of the lessons of this class will be that there is no one data structure that is always best.

We looked at ArrayLists, and they have some nice features. Constant time access to element at specified index is possible due to contiguous arrangement of elements.

But trade-off is that we have to give a size. When size needs to increase, we need to copy the elements into a new array of larger size.

Linked lists are another kind of data structure. They compromise direct access for flexibility in size. That is, they grow when they need to grow, and allow constant time add method. Elements are *not* contiguous, so no direct access.

Linked lists are essentially a collection of `Node` objects.

A `Node` is an object that contains a value, and a reference to the next node. This next pointer is also of type `Node`.

Elements of LinkedList can be anywhere in memory, they are "linked" together by references to one another.

The `LinkedList` class maintains a reference of type `Node` called **`head`**. **`head`** points to the first item on the list. When the list is empty, it holds **`null`**.

Like all lists, we need a way to add, remove, and find elements on the list.

Our default `add` method adds a new value to the front of the list as follows:

1. Create a new node with the value to add to the list.
2. Set the next value of the new node to the value referenced by **`head`**.
3. Set the value of **`head`** to the new node.

Notice that the order of these steps is important. If we try to set the new value of **`head`** before we set the **`next`** reference for the new node, then we lose our reference to the rest of the list.

Notice, too, that adding to the front of the list requires constant work. All of the steps depend only on the value stored in **`head`**.

When dealing with linked lists, we always want to check our methods for special cases. When we add to the front of the list, our approach works even when the list is empty.

In order do things like look for values on the linked list, or print out the contents, we need a way to traverse the list.

The `toString` method in our sample code creates a local variable of type `Node` that is initialized to `head`. (We call this variable "`finger`" because we imagine a finger pointing to each node in the list.) Then, while the value of `finger` is not `null`, we concatenate the value to our return string.

We need this technique of walking the list to add a new value to our list at a specific index.

1. Find the element *before* the index to add to. This is because we need to link to the element before it in the list!
2. create a new node with the appropriate value.
3. set new node's next to `finger`'s next
4. set `finger`'s next to the new node.

Notice that we cannot reverse the order of 3 and 4 for the same reason as before. If we do, we will lose the reference to the remainder of the list.

Again, we need to be careful about special cases. There is no special case for adding to the end of the list, but there is for adding to index 0. The reason is that we cannot stop at the node before index 0. We can take care of this care by calling our default `add` method that adds a value to the front of the list.

Removing items works similarly:

1. walk list until `finger` references the node before the node that holds the value.
2. change `finger`'s next reference to `finger.next().next()`.

Here, we have two special cases.

If the item we are searching for is the first item on the list, then all we need is `head = head.next()`.

If the list is empty, then the value we are searching for is not on the list, and nothing needs to be done.