Abstract Data Type

Array's have their benefits and drawbacks.
- This leads to the question: Can we build something that works differently but does the same thing?

- Let's divide up 2 different concepts to get organized:
--- **What your object does**: "Abstract Data Type" (ADT)
--- **How your object does it:** Implementation

Your ADT is implementation agnostic.

So - the Abstract Data Type for an array is a "List" or a "Sequence"
- **Common operation**s: Get Length, Find item, loop through items, insert, prepend, append, delete, reverse
- An alternative to an Array List is a **Linked List**
- The Linked List can do all the same things, but achieves it in a different way and with different trade-offs.

Recursive Data Types

- This is a Class/type that contains a reference to another object with that same Class/type.

Our example:

```
class Node {
  int data;
  Node next;

  public Node(int data, Node next) {
    this.data = data;
    this.next = next;
  }

  public Node(int data) {
    this.data = data;
  }
}
```

What is actually created when you call "new Node(.....)"?
- space for an int and a reference.

Ok - let's create a bunch of Nodes:

```
Node a = new Node(1, null);
Node b = new Node(2, a);
Node c = new Node(3, b);
Node d = new Node(4, c);
```

Draw this out.
- Notice that if I'm given node d, I can follow the links to get a list.

I can loop through the links to the "end" given a starting node:

```
// Given node d
for (Node current = d; current != null; current = current.next) {
  System.out.println(current.data)
}
```

We can use these nodes to build a list, but it turns out these Nodes can do more than this.

- This is a good thing if you want to use these “extra” capabilities
- But it's a bad thing if we want to assume that the node is part of a list (or that following the “next” links will eventually get you to a termination point).

1) Multiple branches are possible
Say: Node e = new Node (10, a)

2) Cycles are possible.
Say: a.next = c - what happens then?
- “Lollipop” Design (draw this out)

Linked List:

- Not a contiguous list.
- Elements are connected by nodes and references.

We can use these nodes to form a "linked list", and in some languages that's what you do:
- Lower-level languages, like in C you don't have classes/abstractions
- Higher languages: "final" all the way down, so you can't make a cycle, and you can reuse tails.

But in Java / OOP we use an abstraction:

- LinkedList vs ArrayList in java: both implement a List, but one uses the LinkedList approach, another uses the array approach.
- ListNode is internal!

Generics

I want to be able to create a List that contains data of a given type. I don't want to have to define a new list or a new node every time I change types. I can write it once in Java like this:

```
class Node<T> {
  T data;
  Node next;

  public Node(T data, Node next) {
    this.data = data;
    this.next = next;
  }

  public Node(T data) {
    this.data = data;
  }
}
```

Book Code

https://github.com/rysharprules/Data-Structures-and-Algorithms-in-Java-6th-Edition/blob/master/src/dsa6/chapter_03/SinglyLinkedList.java

Additional operations from the Book to look at:
- How the linked list keeps track of the list size
- Adding to the head or tail of a linked list
- Removing the head of a linked list (and returning the piece of data removed)
- Why isn't there an operation to remove the tail?
   - Because we'd have to loop through the entire list to find the "second to last" element to set its next to null, and to set the "tail" pointer to it.
   - It can be done (will show next time) but it's not an O(1) basic operation