**ArrayList Class Day**

Main ideas:

* Introduce ADT
* Idea of interface/implementation
* Show them a List Interface in Java (literally code for an interface)
* And then we implement some methods together,
  + for an array that grows dynamically
* Side topics:
  + Java interfaces & implementing them [stub methods]
  + throwing exceptions
  + toString
  + equals? [opt]
  + iterators

Take-away from the Java review:

* One of the big concepts of OOP is that that correct OOP design lets us separate the ***internal representation*** *of a class (how the programmer knows the class really works)* from the ***interface*** *that we present to the outside world (how the user of the class perceives the class to work).* [sometimes calledencapsulation or information hiding]
* Example: Picture a car. Inside a car are a lot of devices to control the movement of the car, like the gas and brake pedals, the steering wheel, the levers for changing gears, and so on.
* All of these devices are part of the car's INTERFACE, in that they are the devices that people who drive the car use to control it. Now they may not understand how those devices work at a deep level, but they know that if they push the gas pedal, the car goes forward.
* Now, in a car, why is this separation important? For a few reasons:
  + 1. The user is less likely to break things in a car by simplifying the interface. It's very easy for people to remember "Push gas -> car goes forward." If the only way to make the car go forward was to open the hood and start messing around with the engine, then driving would be a lot harder.
  + 2. We can change things about a car's internal representation while keeping the interface the same. For example, think about electric or hybrid cars. When you push the gas pedal on one of those, the car still goes forward. Yet all the internal details about what happens when you push the pedal are completely different. Imagine how much harder driving would be if every kind of car had different pedals or a different interface to drive it. It would be a nightmare---you'd have to re-learn how to drive every time you drove a different car.
* The same ideas go for designing good classes in OOP. We separate the **internal representation** of the class (sometimes called the **implementation**) from the **outside interface**, then we can change the internal parts without the user knowing about it.
* I'll often use the terms "from the perspective of the programmer" and "from the perspective of the user."  
  - **Programmer** means the person writing the class. [often concerned with implementation]  
  - **User** means the person using the class. [often concerned with interface]

**Abstract Data Types**

* So the goal of this class is twofold:
  + One, we want to introduce the basic data structures and their associated algorithms, so you understand them enough to use them in practice.
  + Two, we want you to be able to develop your own data structures & algorithms when necessary, usually variations on the common ones that you'll see in this course.
* So we need a way to conceptualize the idea of a data structure, and we'll do that through something called an ADT.
* **ADT** = An ADT has 2 parts
  + A short description of what the data type represents (as an abstract concept)
  + A list of operations that the data type is capable of. The operations do not give low-level details about HOW the operation works, only WHAT it does from the perspective of the USER, not the programmer.
* So the ADT is the interface.
* An ADT must be combined with an implementation, and usually the implementation is a certain data structure. Example (List ADT, versus ArrayList/LinkedList)
* Mention: The distinction between an ADT and a data structure is important, but the names that we’ll use to talk about ADTs and data structures can sometimes refer to an ADT and sometimes to a data structure.
  + For instance, sometimes the term linked list refers to an abstract data type, because that doesn’t specify whether the list is 1x or 2x linked. But sometimes folks just automatically assume that “linked list” means a 1x linked list.
  + If it’s ever critically important to distinguish, I’ll say something like the “List ADT” or the “linked list data structure.”
  + My point is there’s no need to spend time memorizing which terms are ADTs and which are data structures, because if you ask different ppl you’ll get different answers.
* Create LIST ADT.
  + Let’s give a description of what a LIST represents.
  + Description: A LIST consists of a collection of positions, each of which contains a single element of the list. Each position has a unique index, which is an integer in the range 0 <= index < n, where n = # of elements in the list.
  + Note that this says nothing about how the List is actually stored in memory. All it says is there’s a correspondence between integers and the elements of the List. We are free to pick whatever implementation makes most sense in this context.
* **Ask: What operations should a List be able to do?**
  + Get an element at a certain index.
  + Set an element at a certain index.
  + (Add an element at the beginning.)
  + (Add an element at the end.)
  + (Add an element in the middle (at a certain index).)
  + Get the size.
  + Change the size(?)
    - Pros/cons to letting user have complete control over the size of the array vs letting it grow automatically as needed.
    - Pros: if we need to pre-grow the array, we can.
    - Cons: forces user to check on size and manually grow the array if we’re about to run out of room.
* Ask – what’s the obvious implementation for this List? What data structure?
  + An array is most obvious. Why?
  + Answer – b/c we want random access. The key characteristic of an array is that it permits us to access any element at any time, based on its index. So we want that to be fast.
  + We can do that with a linked list, but random access in a linked list is slower than in an array, b/c we have to traverse the list to find each element.
  + If we don’t need random access, and most accesses in the array are going to be in a sequence, either front to back or back to front, then the appeal of using an array rather than a linked list is lost.
* Java Interfaces
  + Java has a concept that allows you to separate the concept of an interface from the implementation of an interface.
  + Remember so far, we have looked at
    - defining a class
    - defining an abstract class – a class with one or more methods marked as "abstract" which means they don't have implementations. This is done b/c sometimes we want to represent a concept that is so abstract that we have to delegate its specific implementation to the subclasses.
    - Example was: Pet and Dog/Cat, with the speak() method. Speak for a Pet is such a general concept, there's no one way to write a speak method that will work for all pets, since each pet does it differently.
    - another example would be Musical Instruments. If there were a musical instrument class, the play() method would probably be abstract, because the way you play a musical instrument varies from instrument to instrument.
  + So with java, you can take this idea to the extreme, What if you define a class where ALL the methods are abstract? This is actually very common. This is called an INTERFACE in java.
  + Def'n : an interface is a group of methods where they are all empty! So it's a 100% abstract class. The other restriction is that an interface may not have any instance variables (fields). It only defines what something can DO (methods only) and the lack of any definition for the methods, and the lack of variables implies that it can't say anything about HOW it does it.
  + When you want to write a class that implements an interface, you use the implements keyword.
* Create the RList interface in Java. (called RList b/c there's already a List interface).
  + Size, get/set, append/prepend.
* Now we need to talk about details of the implementation.
* You all already said we should use an array.
* Now we could use an ArrayList in Java, but the whole point of this exercise is to actually implement our own version of an ArrayList, to see how it works.
* So what ArrayList actually gives us is an array that will automatically grow to accommodate anything we put into it. Python has this with its lists, C++ has this with its vectors, most programming languages have this concept.
* But the way this is implemented is kind of interesting. For various reasons, it's rather challenging to build-in an array that automatically grows directly into a programming language. Usually it's some kind of add-on.
* (Explain why. Usually arrays that are built-in to programming languages can't change sizes once they're created, and the reason is simply that in pretty much every programming language out there, when a variable is created, the computer must be able to figure out how much memory to reserve for it. This goes for any data type. For instance, in Java, an int is always 4 bytes. a long is always 8 bytes. This goes for all the primitive data types.
* So the way programming languages usually implement this concept (an array that can dynamically change its size), is that they start off by reserving a block of memory for the array, and if the array needs to exceed that size, a larger block is allocated, and everything is copied over. Usually some extra space is allocated, so that every time a new item is added, you don't have to re-allocate and copy everything over.
* Give visual example.
  + Start with capacity 3.
  + Add 3 elements, then a 4th. Resize, change pointers over.
* So here's our GOAL: (IMPORTANT!!) We want, from the user's perspective, this whole thing to be seamless. We want the **interface** of this object to appear to be an array that grows as needed, even though the **implementation** will be an array with a fixed size, that when needed, is copied to another array of a fixed size.
* This is the essence of creating good data structures. We can present a very simple usable interface, and hide all the messy details underneath.
* So this class will be called RArrayList. here's the way we're going to do this. Inside our class, we're going to keep track of TWO sizes. One is the size that we present to the outside world. In other words, from the users' perspective, an RArrayList will start off empty, and as we add elements, it will grow in size, one element at a time. This is exactly how (from the user's perspective), a python list works, and a regular Java ArrayList works.  
  + However, under the hood, we will have a second integer, that we will call the capacity. The capacity will be the current length of the fixed-size array that we are using to store our elements. The capacity must always be >= size.
  + If capacity > size, that means we have some extra slots at the end of our array that we can add new elements to.
  + If capacity == size, that means the array is full, and as soon as the next item arrives, we will have to grow the array.

* Start defining RArrayList class.
  + Start with private int[] data.
  + private int size.
  + Now you might think we need private int capacity, and in a language like C++ we would. In java, we can use data.length, and that will always give us the current capacity of the fixed-size array.
* Start with constructor.
  + size = 0
  + data = ...how many elements?
* Write size().
* Write append().
  + First version just throws an exception if out of space. IllegalStateException.
* Write toString().
* Fix up append() by writing expand().
* Write prepend().
* Write set/get if time, or move onto Iterator.
* Iterator.
  + Iterator is a Java interface that represents the abstract idea of being able to ITERATE through something. That is, it's used by almost all the Java data structures classes to allow programmers to loop over the contents of a data structure.
  + Only 2 methods. hasNext(), and next().
  + First build outside class.
  + public class RArrayListIterator implements Iterator<Integer>.