Lecture 19: Interfaces, Inner Classes

- These last 3 weeks, we're going to discuss object-oriented programming techniques
  - Rather than structures, good programming practices
  - A bunch of miscellaneous topics today - ask lots of questions
- Back in Week 2, we introduced the idea of an *abstract data type* (ADT)
  - Specifies WHAT the collection does, not HOW it does it
  - So we had a List<E> abstraction
    - ArrayList<E> and LinkedList<E> implementations
    - But have the same methods
  - We also saw Queue<E>, Set<E>, Map<E>
  - The way we specify an ADT in Java is with something called an ***interface***
  - Today, we'll discuss how to create your own abstractions/interfaces
- We have some client code that creates an ArrayIntList and LinkedIntList, adds some stuff, removes, prints
  - Why did we have two implementations in the first place?
    - ArrayIntList: fast random access, slow add at the beginning
    - LinkedIntList: slow random access, fast add at the beginning
  - There's redundancy here - same code twice!
- We could create a helper method

```
public class ListClient {
    public static void main(String[] args) {
        ArrayIntList list1 = new ArrayIntList();
        processList(list1);

        LinkedIntList list2 = new LinkedIntList();
        processList(list2);
    }

    public static void processList(ArrayIntList list) {
        list.add(18);
        list.add(27);
        list.add(93);
        System.out.println(list);
        list.remove(1);
        System.out.println(list);
    }
}
```

  - But this doesn't compile - you can't pass in a LinkedIntList as an ArrayIntList
  - But we want to think of these two structures as the same type of object - they have all the same methods, so we SHOULD be able to do this
  - We need to come up with an *abstraction* to capture what is similar between the two
    - They both have "integer list" functionality - they can add, remove, get integers
    - However, they are implemented very differently

- Java has support for this idea of functionality abstraction, called an *interface*
    - An interface is a declaration of behavior, without any functionality
    - Method headers without method bodies
    - So copy/paste the ArrayIntList code and delete all method bodies/comments
    - To make it an interface...
        - We change the name: `ArrayIntList --> IntList`
        - We change the type of the file: `class --> interface`
        - Replace curly braces of methods with a semicolon
            - Semicolon says “I haven’t given the implementation”
    - This is like a hollow radio - it has the right buttons and knobs, but no innards
    - Or like a CPA certificate - the certificate is a guarantee, but it’s no good unless you have SOMEONE (an actual person) who possesses the certificate
- So now we can go back to the client code and change it:

```
public static void processList(IntList list) {
```

    - (run the code)
    - But this still doesn’t work! It says that we can’t apply a method that takes an IntList to an ArrayIntList
        - But both LinkedIntList and ArrayIntList have the required methods...
    - So what’s the problem?
        - Java requires that classes explicitly say what interfaces they implement
        - Like putting “MD” after your name on a resume to affirm that you are a medical doctor, or CPA
    - So we change both classes headers to look like this:

```
public class ArrayIntList implements IntList {
```

    - Now everything works properly
- Another experiment: try creating a new IntList

```
IntList list = new IntList();
```

    - You can’t do this - there is no implementation, it is incomplete
- Think about the types of our variables in main
    - Why does our code work, considering that one variable has type ArrayIntList, one has type LinkedIntList, and we’re passing both into a method that takes an IntList?
        - Objects have more than one type
    - An ArrayIntList is both an ArrayIntList AND an IntList - just like I could be both myself AND a medical doctor
    - Interfaces let us define new types, but objects will always keep their old types
    - However, it’s good to define variables with the most general type that you can
    - Better:

```
public static void main(String[] args) {
    IntList list1 = new ArrayIntList();
    processList(list1);

    IntList list2 = new LinkedIntList();
    processList(list2);
}
```

    - This keeps your code very flexible - can switch out the type easily
- Now we've seen an analogy to the interfaces we've been using with the Java collections
    - Another note: a class can implement more than one interface
        - Just as you can be both a certified doctor and a certified teacher
        - You can have as many "certifications" as you want
    - To implement more than one interface, separate the interface names with commas
- We'll talk about another important interface on Wednesday: the ***Comparable<E>*** interface
- Making our code work with a for-each loop
    - We've seen that for-each loops work on arrays, lists, and sets
    - (try to write a for-each loop over our ArrayIntList)
    - It doesn't work - we haven't told Java how to make a for-each work with our ArrayIntList
    - Does anyone remember how I said that for-each loops work?
        - They have an iterator underneath the hood
    - But back in Week 1, we implemented an ArrayIntList iterator, and added an iterator() method to the class
    - So why doesn't it work?
    - Well, because we haven't TOLD Java that our class has that method
        - Same reason it didn't work when we created an IntList interface but hadn't added "implements" to the ArrayIntList and LinkedIntList classes
    - Java has a way to do this using an interface
        - The Iterable<E> interface
        - So let's add another implements statement

```
public class ArrayIntList implements IntList, Iterable<Integer> {
```

    - But actually, we want something stronger than this - we want both of our lists to be Iterable
        - We could add the implements Iterable to both files, but we have a better option
        - We can say that the interface itself implements the Iterable

```
public interface IntList extends Iterable<Integer> {
```

        - The syntax means "this interface is a certificate, and it also promises everything in this other interface"
    - (compile)
    - But this doesn't compile - doesn't override the right method? Doesn't return the right type?
    - Well, Java expects a certain type of object back from the call on iterator()
    - It expects an iterator - an object with next(), hasNext(), and remove() methods
    - So we need to tell Java that our object - our ArrayIntListIterator - has those methods
    - The ArrayIntListIterator must implement another interface, Iterator<E>
    - We also have to change the return type of the iterator() method to return an Iterator<Integer>
        - Remember we can do this because an ArrayIntListIterator now has 2 types - itself and Iterator<Integer>
- Another new topic: inner classes
    - It doesn't really make sense to have two separate files for the ArrayIntList and the ArrayIntListIterator
        - We'll never use them separately

        - In fact, it's impossible to use them separately
    - Solve this by including the iterator INSIDE the list class
    - We call this an "inner class", and it is defined just like a method, except an entire class
    - (copy/paste the iterator code into the ArrayIntList)
    - We also define this class as "private" instead of public like we usually do, since the client doesn't need to know anything.
        - All the client sees is that it's an Iterator<Integer> - doesn't care about its implementation
    - In fact, the iterator now has access to all the private fields and methods of the ArrayIntList class
        - We no longer need a reference to the list
        - (remove the reference from the constructor)
        - (list.get() --> get(), list.size() --> size(), list.remove() --> remove())
    - One issue, though: Java gets confused when two methods have the same name
        - It can't distinguish between remove in the iterator and remove in the list
        - So we have to tell it "I want the list's remove, not the iterator's remove"
        - We say this with the following bizarre syntax

```
ArrayIntList.this.remove(position - 1);
```

    - Read this as "I don't want the reference to ME (this), I want the reference to my enclosing class (ArrayIntList.this)"
- We can also do this with the ListNode
    - Remember how we defined our list node class with public fields? We said it was ok because the client would never get access to our nodes
        - But having two separate files is really annoying - they should be one logical unit
        - You won't use the node class other than in the LinkedIntList class
    - We can solve this by including the ListNode class INSIDE the LinkedIntList class
    - (copy/paste the list node code into the LinkedIntList)
    - Also private
    - In addition, we add the modifier "**static**"
        - What does static mean in the context of a method, or a field/constant?
        - It means "one for the entire class", not "one per object"
        - In this case, what static means is that the ListNode does not depend on the LinkedIntList object that it is a part of
        - If it was not static, then the ListNode class could actually refer to the fields of its outer class, but we don't need that
        - Our nodes are independent of the list that they are a part of, so we make the class static