Lists and Abstraction

CSCI 136 :: Williams College

Administrative Details

- Lab 3 Is due tonight
  - I have office hours 1-2pm in the lab
- Lab 4 is posted
  - Another partner lab. Please fill out the form if preferences

Last Class

- ???

This Class

- Abstraction
  - What & why
- List interface & class
  - We've used them, but we haven't dug into the details
- Singly Linked List implementation details

Abstraction is Beautiful

- Abstraction lets us solve complex problems elegantly by ignoring the "irrelevant" details
  - What does it mean to be irrelevant?
    - As a systems researcher, I spend a lot of time on the “irrelevant details”, but that is an even stronger argument in favor of abstraction…
  - What does it mean for a problem to be complex?
- We simply can't reason about complex systems without breaking the problem down into reasonably-sized, simplistic parts.

We Already Use Abstraction

- How have we seen abstraction so far in CS136?
  - We started using Vector objects before we looked at how they were implemented. How is that possible?
    - We learned the function behaviors (inputs + outputs) before we learned the data structure implementation (member variables, method code)
    - We use public/private/protected to help us to hide implementation details

We Already Use Abstraction

- We've also benefited from abstraction without explicitly saying so
- Vector extends and implements other Java classes/interfaces

structure5
Class Vector<E>

```
java.lang.Object
  └structure5.AbstractStructure<E>
      └structure5.AbstractList<E>
          └structure5.Vector<E>
```

All Implemented Interfaces:
java.lang.Cloneable, java.lang.Iterable<E>, List<E>, Structure<E>

Java gives us two very powerful tools for abstraction:
the Interface and the Abstract class

Abstraction helps us to be Lazy

- We often optimize algorithm performance by minimizing big-O
- But once I heard how much money engineers get paid, I started to appreciate other optimization targets: saving programmer's time

- Let's figure out how to save the programmer's time in two ways:
  - Code that *uses* data structures should be faster to write
  - Code that *implements* data structures should be faster to write

Saving programmer effort: Interfaces Define *Behavior*

- Consider the List interface:
  - How many programs that you've written use something that implements the List interface?
  - Do you care which class is used as long as it implements List?
    - MAYBE!
    - But you can write your code in a way that lets you pick a specific class later

An Interface defines a Contract

- If a class implements an interface, they must adhere to that contract
  - This means they must implement *all* methods in the interface
  - But as a result, I can swap any class that implements the interface into this sample code in place of SinglyLinkedList

```
public static void main(String[] arguments)
 {
     List argList = new SinglyLinkedList();
     for (int i = 0; i < arguments.length; i++){
         if (!argList.contains(arguments[i])){
             argList.add(arguments[i]);
         }
    }
    System.out.println(argList);
 }
```

Takeaway: an interface defines behaviors, and that is all a programmer needs to start writing code

Saving programmer effort: Inheritance allows reuse

- Are there List methods that I can write without knowing the low-level implementation details?
  - Let's look at the AbstractList class
    - Are there methods with real code?
      - Yes
    - Are all of the methods there?
      - No. Otherwise it wouldn't be *abstract*

Saving programmer effort: Inheritance allows reuse

- A programmer can *extend* an (abstract) class and complete its implementation
  - This makes the class ***concrete***.

- Lets look more closely at the code for the Vector class

Takeaway: an abstract defines behaviors, and it lets us define general code. We can overwrite that code as needed.

Review of Java Tools

- public/private/protected
- Interfaces
- Abstract Lists

- The structure5 hierarchy so far

Abstract Data Types (ADTs)

- Abstract Data Type (according to Wikipedia):
  - "In computer science, an abstract data type (ADT) is a mathematical model for data types, where a data type is defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations. This contrasts with data structures, which are concrete representations of data, and are the point of view of an implementer, not a user."

Singly Linked Lists

- Two Classes:
  - SinglyLinkedList<E> (SLL)
  - Node<E>

- Let's look at their relationship
  - SinglyLinkedList is made up of Node<E> objects
  - This relationship is completely hidden from the user
    - No SLL methods return Node<E> objects
    - No SLL methods accept Node<E> objects as input
  - Why create this separation?