Inheritance and Stack Applications

Instructors: Sam McCauley and Dan Barowy

April 6, 2022

Admin

- Remember to fill out the partner form this morning (asap if haven’t done so far).

- Any questions?

Inheritance

Inheritance

- We’ve been building towards inheritance throughout the course
- Seen it (and used it) a couple times
- Let’s talk about it a bit more formally
- You’ll use inheritance in Lab 9 (we’ll revisit it briefly right before then)

Creating a Subclass

- Use the extends keyword
- The subclass that you write inherits the fields (instance variables, static variables, etc.) and methods of the parent class
- Cannot access the *private* members of the parent class.
  - They're still there, and can be accessed by parent class methods
  - *Can* access protected members
- In short: subclasses allow us to *add functionality* to a class without rewriting it
- Can also *refine* classes for specific scenerios

Inheritance and Constructors Intro

- The subclass automatically calls the default constructor of the parent class
- Or, can use the `super` keyword on the first line of the constructor to call a different parent class constructor
- We'll come back to this before Lab 9

Example

- I want to keep track of 136 students in a course
- 136 students have name, ID, grade just like any other student
- They also have ten *lab grades*
- How can we make a `StudentIn136` class?
  - One option: just write it out
  - Much easier: use inheritance! Let's take a look

Class Types and Inheritance

- Every object of a class is also an object of the parent class
- So: a `StudentIn136` is also a student
- We can, for example, sort `Studentin136`s using the code we wrote a few weeks ago. Let's see an example of that.

Example 2: MyVector

- You created a MyVector class: a Vector that can also sort
- Your MyVector class could access any public Vector methods. (Could also access protected methods.)
- The `data[]` array is private. What happens if we try to access it directly?
- The underlying array is a good example of when we want a variable to be `private` instead of `protected`: we really don't want anyone accessing it, even subclasses; any changes they want to make can be through the Vector interface

Inheritance

- Inheritance is perhaps the *main reason* people use object oriented programming

- So far: objects and class types help us create self-contained pieces of code that can help us store data about a single concept or accomplish a single task

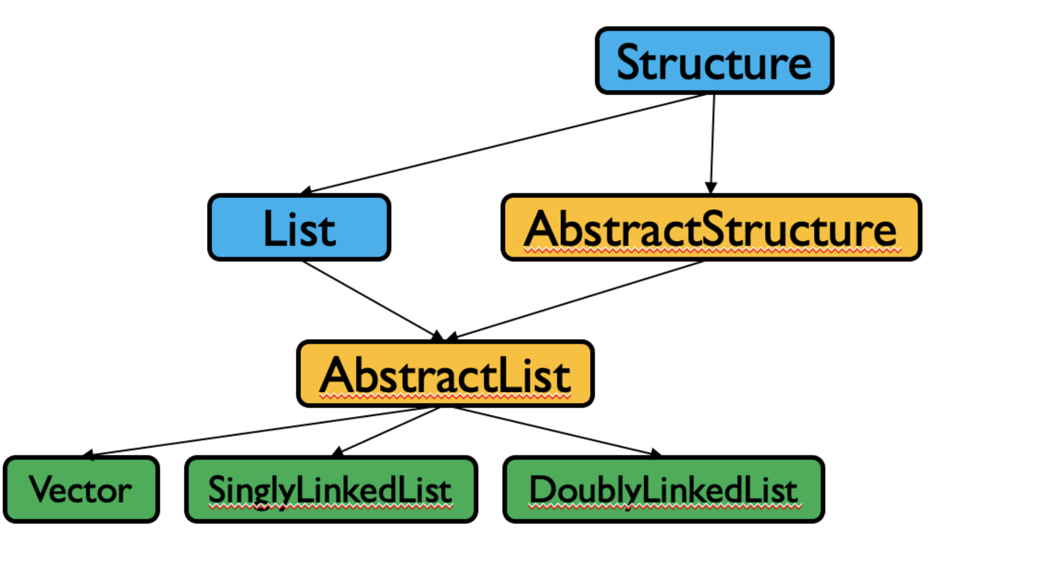
- With inheritance: we have an easy way to modify our code for a new task

- Saves us work!

Setting Up Java Class Hierarchy

- Every class has exactly one parent class
  - Cannot inherit from two different classes in Java!
  - If you do not state any parent class, then `Object` is the parent class
- So every class has a parent class, which has a parent class, which has a parent class, …, which has `Object` as a parent
  - This is why “everything is an object” in Java!
- This leads to a hierarchy, which is what we’ve been visualizing.

Structure 5 Hierarchy

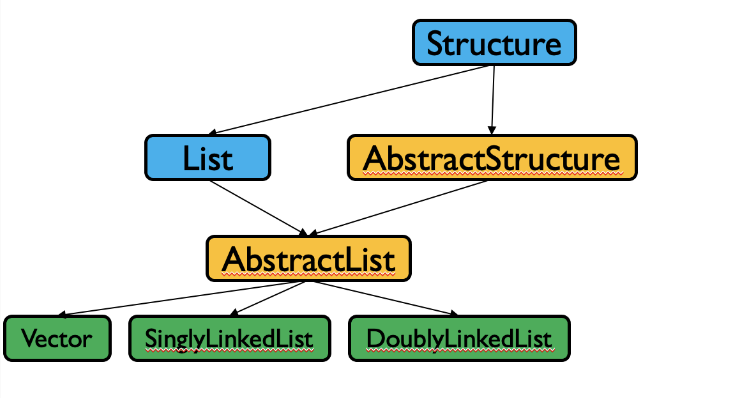


Idea: these lines represent that one class `extends` another. But, we still haven't

Inheritance with Interfaces

- First: a class must `extend` exactly one other class (`Object` if none is given)
- But, a class may `implement` any number of `Interfaces`
  - Makes some sense: an interface is just a contract. It’s possible that a class fits the requirements of many of these contracts.
- An `Interface` may `extend` another `Interface`
  - In fact, it can extend multiple interfaces…
- Same idea as classes: the interface “gets” the methods from its parent interface, and adds some more
- If a class implements this interface, it must implement all of the listed methods, plus all methods from its parent

Structure 5 Hierarchy



Blue: Interface

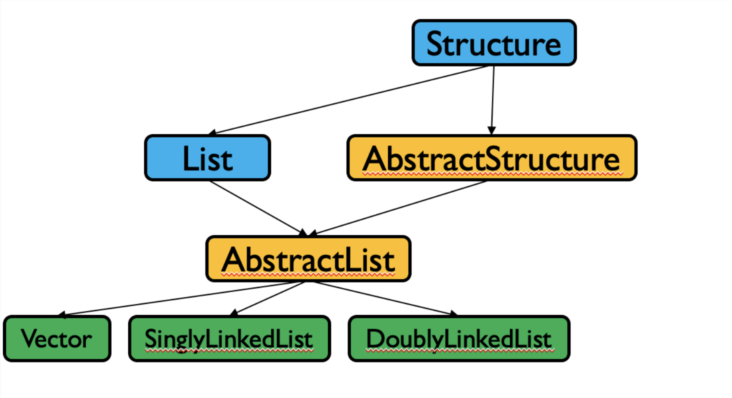
Yellow: Abstract Class

Green: Class

Idea: these lines represent that one class `extends` another; or that one interface `extends` another; or that a class `implements` an interface

Fitting Stacks Into Structure5

Putting the Classes Together



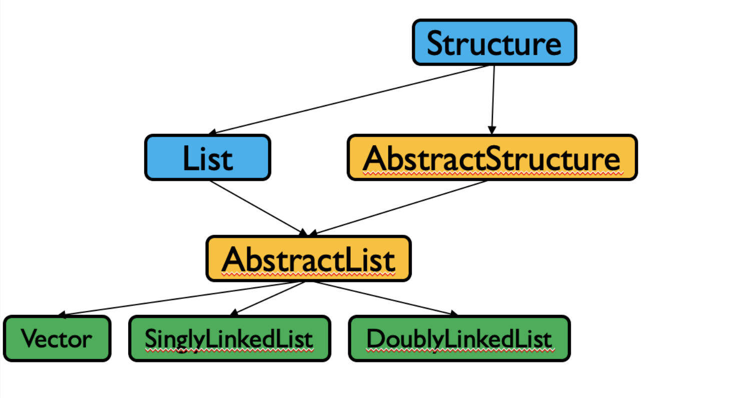
Blue: Interface

Yellow: Abstract Class

Green: Class

How can stacks and queues fit into `structure5`?

Putting the Classes Together



Blue: Interface

Yellow: Abstract Class

Green: Class

Where do stacks and queues go here? Are they a `List`? Are they a `Structure`?

Let’s look at both interfaces.

Stacks and Queues

- They are not a `List`: don’t have methods like `get(int i)` or `indexOf()`
- They probably could be a `Structure`: methods like `size()` and `clear()` make sense, as do `add()` and `remove()`
    - This is a judgement call to some extent!
    - In `structure5`, stacks and queues do implement `Structure`

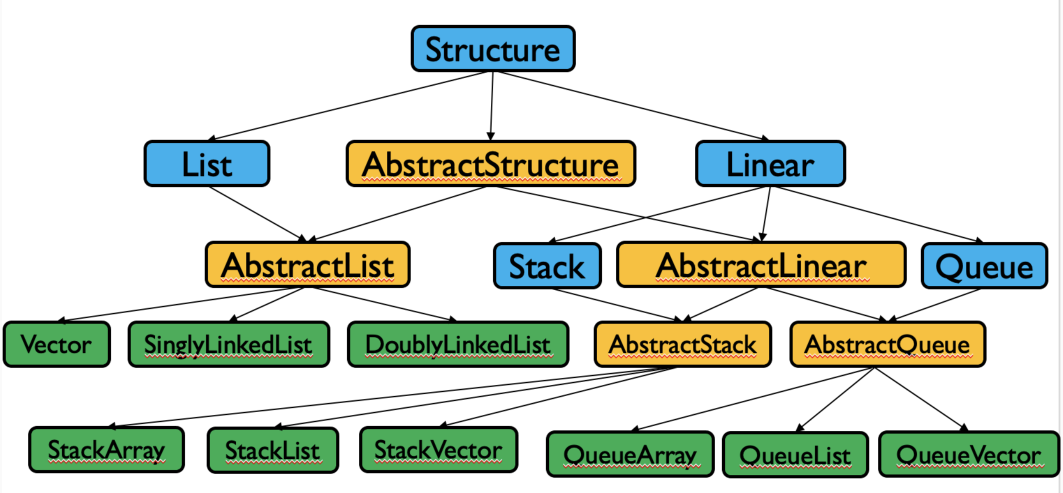
Filling out `structure5`

- First: a `Linear` interface common to both stacks and queues, and an `AbstractLinear` abstract class
    - What qualities does a `Linear` structure have?
    - Can add and remove items!
    - Let’s look quickly at the code
- Then, the `Stack` and `Queue` interface extend the `Linear` interface
- Have an `AbstractStack` and `AbstractQueue` abstract class
- Finally, each stack class `implements Stack` and `extends AbstractStack` (likewise for queues)

AbstractStack

- What methods are common to all stacks?
- Hint: abstract classes are very good for implementing methods that just call other methods
- Hint 2: the `Linear` interface promised some methods that don't quite line up with the stack terminology...
- Idea: we can implement `push()` by calling `add()` and `pop()` by calling `remove()`, and so on
- Same for `AbstractQueue`!
- Let's take a look at them

Current Structure5 Universe


Structure
List
AbstractStructure
Linear
AbstractList
Stack
AbstractLinear
Queue
Vector
SinglyLinkedList
DoublyLinkedList
AbstractStack
AbstractQueue
StackArray
StackList
StackVector
QueueArray
QueueList
QueueVector