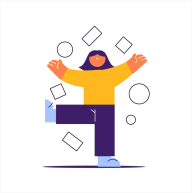
Lec 18: Stacks and Inheritance

Sam McCauley

April 6, 2026

Admin



- Welcome back!
- Lab Wednesday
  - Start early!
  - First step: fix any remaining issues in your CoinStrip implementation (I will give whatever help you want; goal is just to get it done.)
- Practice quiz available in TA hours/office hours from today
- Anything else?

Stacks

Stack

- Can only add or remove to the *top* of the stack
- (No adding to the middle or removing from the middle.)
- The item we remove is the *most recently added* item

Stack Operations

Stacks have their own vocabulary. (This is fairly universal; you do need to know it.)

- `push()`: Add a new item to the top of the stack
  - Think `addLast()`
- `pop()`: Remove (and return) the top item on the stack
  - Think `removeLast()`
- `peek()`: Return the top item on the stack without removing
  - Think something like `get(size() - 1)`

Stack Operations

There are a few ways to implement a stack: it could be that we're adding and removing the first element!

- `push()`: Add a new item to the top of the stack
  - *Alternative:* `addFirst()`
- `pop()`: Remove (and return) the top item on the stack
  - *Alternative:* `removeFirst()`
- `peek()`: Return the top item on the stack without removing
  - *Alternative:* `get(0)`

Stack

- Only three operations to worry about!
- How can we implement a stack using data structures we already have?
- What are the tradeoffs between some of the options we have?

Implementing a Stack with an Array

- On board: How would this work?
- Downside?
  - Need to specify maximum items in the stack up front.
- Running time for operations?
  - `push()`: $O(1)$
  - `pop()`: $O(1)$
  - `peek()`: $O(1)$

Implementing a Stack with an ArrayList

- On board: How would this work?
- Don't need `int top` anymore; can resize
- Running time for operations?
  - `push()`: $O(n)$ in the worst case. $O(1)$ "on average"!
  - `pop()`: $O(1)$
  - `peek()`: $O(1)$

Implementing a Stack with a Linked List

- In pairs: how can we implement a stack with a Singly Linked List?
- Idea: need to store it backwards! Most recent element is the *head*
- to `push()` we can `addFirst()` which is $O(1)$
- to `pop()` we can `removeFirst()` which is $O(1)$

When to Use Stacks?

- Classic example: “JVM call stack”!
  - JVM: the Java Virtual Machine is what actually runs our code
  - Keeps track of what methods we have called
  - Each time a new method is called, we push it on the top of the stack
  - When the method returns, pop it off the top of the stack
- Useful in implementing backtracking search
- Or any last-in-first-out usage

Implementing Our Own Stack

- We'll implement a stack using inheritance!

- Plan for rest of today: go over inheritance, then come back and implement a stack.

Inheritance

Example: 136 Student

- Let's say I wanted to keep the data for each student in CS 136
- A CS 136 student has all the properties of a normal student (name, grad year, list of courses), but also has a unix ID and a 136 lab section
- How do we write this class?
  - Option 1: Copy-paste `Student` class, and add variables for the unix ID and 136 lab section
  - Option 2: Inheritance! Allows us to write what we mean: "a 136 student is a `Student`, plus an ID and lab section"

Inheritance

- When a class “inherits” from another, it immediately gains everything from that class: the instance variables, methods, etc.
- Done using `extends` keyword
- Then, we can add additional variables, methods, ec. of our own
- Sometimes good to think of it using types: every car is a vehicle, so the `Car` class should perhaps inherit from the `Vehicle` class. Every rectangle is a shape, so the `Rectangle` class may inherit from the `Shape` class.

Using Inheritance

- When declaring the class, use the `extends` keyword; now have “copied” all existing methods/instance variables
- Let’s write the `Student136` class using inheritance
- Any `public` or `protected` methods and instance variables can be access directly from the subclass. `private` ones can only be accessed/changed from the superclass. Let’s test this

Overriding and Inheritance

- Now that we have new information, we may want to “rewrite” some of our old methods. This is called *overriding* a method
    - Cannot override instance variables
- To override, just write the new version of the method in the subclass
- Always write `@Override` on the line immediately before. This tells the reader that you’re overriding a method, and tells the compiler to check that you are overriding correctly (no spelling mistakes or anything)
- Let’s override the `toString()` method for `Student136`

super keyword

- The `super` keyword allows us to access methods of the parent class (in particular, we can use it to call overridden methods)
- Most common usage is in the constructor
- What do we want the constructor of `Student136` to do?
  - Everything we did in the `Student` constructor, plus
  - Assign the new variables
- Calling `super()` calls the superclass' constructor. Let's try it.
- Just for practice, let's call the `toString` method of `Student` using `super` as well

Let's Implement A Stack

- Idea: we'll inherit from `LinkedList.java`
- Add our own `push` `pop` and `peek` methods
- Let's do it!
- Diamond operator: you only need to state the generic type on the *left*

```
1          Stack<Integer> st = new Stack<>();  //OK!
2          Stack<> st2 = new Stack<>();  //not OK!
3          Stack<> st3 = new Stack<Integer>();  //not OK
              either!
```

Discussion

- The Java library actually implements a stack like this. (Actually, it inherits from, essentially, `ArrayList`.)
- What is an *upside* to implementing a stack using inheritance, as opposed to creating a new data structure?
  - It's easy!
- What is a *downside*?
  - A `Stack` is also a `LinkedList`. So you can call `get()`, `indexOf()`—all kinds of methods that stacks aren't supposed to support.

A test

- Let's say we override a method, and then a method we did not override (from the superclass) calls it. Which version is called?
- Example: let's override the `getNode()` method in `Stack`. It does the same thing, but outputs a message at the beginning so we know if the overridden version is called.
    - Note that for this to work, we need `getNode()` to be `protected`. You cannot override a `private` method.
- Answer: The overridden version is called!

Inheritance Example

Inheritance

- Inheritance is not just about extending functionality
- It can also be used for *organizing* code
- Let’s look at a project which could easily be written in a single class. We’ll write it using inheritance instead

Encryption

- Use a secret key to jumble a message. A random person looking at the jumbled message (hopefully) can't read it
- But your friend with the same key can read it!
- Let's look at a couple classic (meaning bad) encryption schemes
- We'll use only capital letters; no punctuation
- I won't test you on these encryption methods. But I want to go over them quickly so that we understand how inheritance helps implement them. (Plus, encryption is fun!)

Caesar Cypher

- Key is some number *k* from 1–25
- Encrypt: Move each letter “ahead” by *k* letters, wrapping around if you go past Z
- Decrypt: Move each letter “back” by *k* letters

Viginère Cypher

- Key is an arbitrary string, let’s say the string is "abd"
- Encrypt: move each letter “ahead” by the index of the corresponding letter in the key.
    - In our example, the 1st letter is moved ahead by 1 for a, the next by 2 for b, the next by 4 for d, then we loop around so the fourth letter is moved ahead by 1 for a again
- Decrypt: move each letter “back” by the index of the corresponding letter in the key.

Substitution Cypher

- Key consists of a list of 26 letters denoting how to map each letter to a new letter
- Encrypt: replace each letter with the new letter
- Decrypt: replace each letter with its original

Implementing All 3 Cyphers

- All three will have an encrypt, decrypt function
- All three need a key, though the type differs
- All three need a way to convert characters to integers and back
- The Viginère and Caesar cypher work pretty similarly. In fact, the Viginère cypher literally does the same thing, just with a rule to change the key on the fly

Design Plan

- We'll have a `Cipher` class with useful methods for all three ciphers
    - Its encrypt and decrypt methods won't do anything: the message will just map to itself
- `CaesarCipher` and `SubstitutionCipher` inherit from the `Cipher` class, overriding the encrypt and decrypt methods
- `VigenereCipher` will inherit from `CaesarCipher`
- Let's look at the code.

Abstract Classes

Motivation

- We did something a bit silly in this implementation: we implemented the `encrypt` and `decrypt` method in `Cipher` even though they didn't do anything

- We *had* to do this if we wanted to call `encrypt()` and `decrypt()` on variables of type `Cipher` in the main method

- Idea: why waste time writing this code? Abstract classes to the rescue!

Abstract Class

- An abstract class exists *only* to be extended by other classes
- It may be incomplete! It may have methods that *must* be overridden by its subclasses
    - For this reason, cannot *instantiate* an object of abstract class type. Some of its methods are missing!
- An abstract class is declared using the `abstract` keyword. Any “missing” method is written out, but the body is left out: a semicolon is used instead of the braces.
- Let’s turn `Cipher` into an abstract class. We’ll replace `encode()` and `decode()` with stubs.

Queues