OOPS! Didn't see you there! Today was mostly an opportunity to practice everything that we've covered up to this point. Tomorrow we'll get our first taste of recursion.

**Announcements**

* You should receive a practice assessment for the Foundations 2 Assessment in your cohort's slack channel after class today. We recommend you run through the practice assessment at least once every night until the day of the exam. Time yourself to see how fast you can complete the exam. The study guide for Foundations 2 is located in W2D5 on App Academy Open and tells you everything you need to know for Monday's assessment.
* Just a reminder, you are NOT allowed to look at solutions for any project before you have completed the project or until after 6 pm on the day of the project. Watching the video walkthroughs also count as reviewing solutions. If you are feeling stuck on a project, ask a question on Progress Tracker. We are here for you.

**Key Points**

**OOP (Object Oriented Programming)**

OOP is a common design pattern used by developers. Its goal is to create programs by creating objects and using those objects to interact with each other. As we've seen already, we use classes to create objects. We've learned about *two* of the four major principles of OOP:

1. **Abstraction**
2. **Encapsulation**
3. Polymorphism (you'll learn about this in a couple of weeks)
4. Inheritance (you'll also learn about this in a couple of weeks)

**Abstraction**

Abstraction refers to the idea of being able to use an object's interface (available methods), without needing to know what is going on behind the scenes. It makes your code easy to read and reason about.

* A classic example is a car. The basic interface consists of a steering wheel, a gas pedal, and a brake pedal. You don't need to know about anything that's going on when you hit the gas pedal, other than pressing the gas pedal → car accelerates.
* We encounter abstraction all the time in Ruby! Whenever you use a method like String#upcase or Array#include?, you don't need to know what it's doing under the hood. You only care about its expected behavior and return value.

**Encapsulation**

Encapsulation refers to the combining of state and behavior (instance variables and methods). This means hiding data (state or attributes) of an object, and defining how the attributes can be accessed (interface, or public methods).

* When we give attributes to an object via instance variables, we can't access those values by default from outside of the class (the data is hidden).
* We (the creators of the class) get to define if and how we can access these attributes, via getters and/or setters.

**Attribute Methods**

Remember how last week we learned about getters and setters? Meaning we could write methods like my\_instance\_variable\_name and my\_instance\_variable\_name= to access and update the instance variables? Because this is so common, Ruby gives us a handy way to define these methods using *attribute methods*.

* attr\_reader :instance\_variable\_name → defines a *getter* method for @instance\_variable\_name
* attr\_writer :instance\_variable\_name → defines a *setter* method for @instance\_variable\_name
* attr\_accessor :instance\_variable\_name → defines both a *getter* and *setter* for @instance\_variable\_name
* Note all of these methods can take a list of multiple instance\_variables as symbols, like attr\_reader :name, :age

Don't add these for everything willy-nilly! *You* get to choose *if* and *how* the attributes can be accessed, so only use them if you want to expose the data (*encapsulation*).

**Syntactic Sugar**

Syntactic Sugar refers to the many syntactical shortcuts that Ruby gives us!

* attr\_reader, attr\_writer, and attr\_accessor are examples you should be familiar with; they write the getters and setters for us

**Operator Methods**

In Ruby, you can define your own +, ==, >, etc on your classes as methods!

It's common to define a #== instance method to compare one instance with another

* When you call it, you could write my\_instance.==(other\_instance)
* **OR** you can sweeten it up with my\_instance == other\_instance

**Bracket Methods**

Writing bracket methods (#[] as the getter, and #[]= as the setter) is very common in Ruby as a way to access some sort of attribute in a class, usually an array (including 2D arrays) or a hash. (Keep in mind that [] and []= are the *actual names of these instance methods*).

Suppose we have an @grid = Array.new(8, { Array.new(8) } as an instance variable for a Board class. We could have an attr\_reader :grid, but then indexing into this 2D array means doing grid[rowIdx][colIdx] *every time* we want to access an element. Let's clean it up with bracket methods!

**#[] (Getter)**

We can define a #[] getter that takes in an array of two integers, and will return the element at that position (@grid[position.first][position.last])

* To get the element at [0, 0], we can write: board\_instance.[]([0, 0]). This is a little messy. (Colored the arguments for clarity).
* Syntactic sugar to the rescue! We can instead write  board\_instance[[0, 0]]

**#[]= (Setter)**

Similarly, we can define a #[]= setter that takes in an array of two integers and a value that we can set at the position in the 2D array (@grid[position.first][position.last] = new\_value)

* To set the element at [3, 3] to "banana", we can write: board\_instance.[]([3, 3], "banana")
* Instead, the preferred way to write it is: board\_instance[[3, 3]] = "banana"