**Object Oriented Programming**

Object Oriented Programming is a design pattern developers use when building applications. You've been using this design pattern through the last few sections, but now let's truly adopt an object oriented mindset.

The goal of OOP is to create programs using objects that interact with each other. We implement classes to create those objects. Let's take a look at 2 pillars of OOP: **Abstraction** and **Encapsulation**.

**Abstraction**

In OOP, **abstraction** is the process of exposing essential features of an object while hiding inner details that are not necessary to *using* the feature. Take this analogy: Many drivers do not understand the mechanical details of **how** a steering wheel works, but they can still use the steering wheel to drive a car. In a similar way, our classes should have methods that are simple to use because they hide complex logic inside of them. Let's explore some examples.

Below is a class that does not abstract how to enroll a student into a course:

class Course

def initialize(name, teachers, max\_students)

@name = name

@teachers = teachers

@max\_students = max\_students

@students = []

end

def max\_students

@max\_students

end

def students

@students

end

end

course = Course.new("Object Oriented Programming 101", ["Ada Lovelace", "Brian Kernighan"], 3)

# Let's to enroll a student

if course.students.length < course.max\_students

course.students << "Alice"

end

Below is the class with a nicely abstracted #enroll method:

class Course

def initialize(name, teachers, max\_students)

@name = name

@teachers = teachers

@max\_students = max\_students

@students = []

end

def enroll(student)

@students << student if @students.length < @max\_students

end

end

course = Course.new("Object Oriented Programming 101", ["Ada Lovelace", "Brian Kernighan"], 3)

# Let's to enroll a student

course.enroll("Alice")

Abstraction results in code that is readable and easy to use. There could be many steps that need to take place when a student is enrolled, and a #enroll method can take care of all of them behind the scenes, unknown to the user of the class.

We have been using this concept of abstraction all the time! Array is a class and Array#include? is a method that we feel comfortable using although we do not know the exact implementation details within the method.

**Encapsulation**

Similar to abstraction, **encapsulation** closely relates methods and data attributes together with the hope of preventing misuse. For us, the goal of encapsulation is to give users access to the things that are *safe* for them to use. Some data we may choose to keep private or purposefully hide from outside users for the sake of security. One common way to encapsulate data attributes is by making them only accessible through methods that we explicitly design as programmers!

Here's some food for thought: Ruby classes have some level of encapsulation by default. How so? Well, if we create a class with attributes, those attributes are inaccessible from the outside unless we define *getter* or *setter* methods too!

Design your code in a way that safeguards against misuse! Let's take a look at an example. Say we wanted a class to track the order of people waiting in a line. We call this concept a Queue. The rules of a Queue are simple: if you are the first one in, then you are also the first one out. No cutting the line! More programmatically, we remove from the front of the line, but we add to the back of the line.

We'll use the index 0 of an array to represent what is at the front of the line.

Let's take a look at a properly encapsulated Queue class.

class Queue

def initialize

@line = []

end

def add(ele)

@line << ele # add ele to back of line

nil

end

def remove

@line.shift # remove front ele of line

end

end

grocery\_checkout = Queue.new

grocery\_checkout.add("bob")

grocery\_checkout.add("alice")

grocery\_checkout.remove # => "bob"

grocery\_checkout.add("noam")

grocery\_checkout.add("donald")

grocery\_checkout.remove # => "alice"

grocery\_checkout.remove # => "noam"

grocery\_checkout.remove # => "donald"

# people who are added first to the line will leave first!

You'll notice that above we decided to not include a getter for @line, that is because we *shouldn't include a full getter*. If we give the user full access to the @line then we can't enforce any of our rules and the result is disastrous:

# this class fails to encapsulate

class Queue

def initialize

@line = []

end

def line

@line

end

def add(ele)

@line << ele # add ele to back of line

nil

end

def remove

@line.shift # remove front ele of line

end

end

grocery\_checkout = Queue.new

grocery\_checkout.add("bob")

grocery\_checkout.add("alice")

grocery\_checkout.line.unshift("noam") # noam cut the line!

grocery\_checkout.remove # => "noam"

**Wrapping Up**

Abstraction and Encapsulation are just two pillars of OOP. As you progress in your programming career, you'll learn how to support other pillars of object oriented programming! So stay tuned.