Sprint 1 Day 2: Swift Fundamentals II

Competencies:

- **define custom types using class, struct, and enum**
- **show and explain the difference between value types vs. reference types**
- **create custom initializers**
- **understand and explain the difference between a method and a function**
- **understand and explain the purpose of optionals**
- **use an if let statement to unwrap an optional variable**
- **force unwrap an optional type and understand the potential dangers and benefits**

Overview

Swift is an object-oriented language. Meaning that it functions best when things are classified down to the bare bones properties they consist of. Swift uses enums, structs, and classes to define objects in code.

Enums

Enums are used to create a set of options almost like you would see in a drop down menu. If the object defined in code were a `Species`, the enum would have cases of different possible animal options. Enums are declared as follows:

```
enum Species {
    case dog
    case cat
    case fish
    case bird
    case turtle
}
```

This is a way to ensure that variables meet one of a set number of certain options. We can declare these enum cases for variables in two different ways:

1. `let myDog = Species.dog`
2. `let myDog: Species = .dog` this is called dot notation

Structs

Sometimes objects won't be as similar as pets. Structs are used to define larger overarching objects that may not fit so easily into enums. If we had a `Pet` object and wanted to define a number of things it consists of, we would use a struct or a class to do so.

```
struct Pet {
  let name: String
  let birthday: Date
  let species: Species
}
```

We definitely could give all of these pet properties an enum, but seeing as there are countless name and birthday possibilities, it really only makes sense to do it in a set list. People usually stick to the enum list we made when it comes to pet choices.
I can initialize a specific pet object by using the struct we made like this:

```
let myDog = Pet(name: "Sammy", birthday: 02/14/2019, species: .dog)
```

This way we have preset specifications for an object, and can fill them in as we know the information.

Classes

A class would actually be much better use in a situation where our object is a unique living thing. Structs are more simple and basic – good for storing information about sizes and shapes
Classes are known as reference types, meaning that the object can be referred back to as many times as necessary in the app.

Classes are unique in that they will complain if you don't give an initializer for the objects. If we had a customer we wanted to keep information on, we would follow a similar pattern to a struct, but with an initializer:

```
class Customer {
    var name: String
```

```
    var address: String
    var age: Int
    var purchases: [Purchase]

    init(name: String, address: String, age: Int) {
        self.name = name
        self.address = address
        self.age = age
        self.purchases = []
    }
}
```

Using that `self.property` syntax means that whatever property is passed into it will be set as the new property for that specific variable like name or age.

#Sprint_1