Sprint 1 Day 1: Swift Fundamentals I

Competencies:

- **identify, use, and understand the difference between the basic data types:** `Int`, `Double`, `Bool`, `String`)
- **understand how to apply mutability to variables** (`let` vs. `var`)
- **use conditional logic to execute various code paths***
- **iterate over an array using a loop to accomplish some task***
- **group like instructions into a function and call that function to reuse/organize code***
- **understand how braces `{}` are used to group code into blocks and how scope affects the lifespan of a variable***

Overview

Any object written out in code has some sort of type. Think of it in the way of a Pokemon having a type. There's fire, water, psychic, fighting, etc, each with their own advantages, disadvantages, and uses. Written phrases or words are of type `String`, integers are of type `Int`, numbers with decimals are of type `Double`, and yes/no statements are of type `Bool`.

Let vs. Var

Constants are objects that will never change. A few examples of this would be things like your name, birthplace, eye color, birthdate, etc. These properties can be declared like this:
`let myBirthday: String = "September 29th, 1994"`

The reasoning for having constant properties over variable properties would be so that your functions and algorithms don't accidentally change your properties when you don't want them to be changed.

However, often times you will want to change properties. A few examples of occasions like this would be your age, height, location, etc. Properties that change can be declared like this:
`var myAge: Int = 24`
Once I turn 25, I can change this property by doing the following:

`myAge = 25`

Had I used a let, the complier would never let me modify myAge because it was rigid and *immutable*.

Conditional Logic

When get in a car, you can take a number of routes to accomplish a variety of errands, or just drive to nowhere at all. In programming, we might want to direct things to take a certain path. If statements help redirect the code to do what we want it to do if it meets certain criteria that we specify.

If statements in code are executed with the following syntax:

```
if (this) {
  (do this)
}
```

Let's say we have a variable called `today` and a constant called `myBirthday`. We can use a conditional statement for the code to run something special when they coincide. If they don't coincide, we can ask the code to do something else.

```
if today == myBirthday {
  print("Happy birthday!")
} else {
  print("It'll be your birthday soon!")
}
```

Switch Statements

Things can get pretty dicey if we have too many ifs and elses going. Switch statements can be used to swap the variable for several cases to produce different outcomes. This is the basic structure:

```
switch something {
case 1:
      (do this)
```

```
case 2:
      (do that)
default:
      (do nothing)
}
```

Switch statements always require a default case for situations when none of the cases parameters are met. A good way to get the function to end is to just default with a `break`

Loops

Sometimes we need to cycle through a certain code a set number of times until we get some sort of result. **For loops** are great for us to execute this. Here's the general format:

```
for newVariable in existingList {
  (do something)
}
```

If we want to do something like this in numbers, we can do the following:

```
for number in 1...50 {
    print(number)
}
```

This code defines a new variable of number and takes the set list of 1-50 to loop through. If we wanted it to go just up to 50 or just after 1, we can use the `..<` or `<..` syntax to accomplish that.
Another more complicated example would be something like this:

```
let groceries = ["apples", "cereal", "chicken broth", "milk", "oreos"]
var index = 1
for groceryItem in groceries {
    print("\(index). \(groceryItem)")
    index += 1
}
```

Here we give the loop an array of items and a value called index that gives a number. This print statement in the loop takes the index value and the groceryItem, and then adds 1 to the index to give a list like this:

1. apples
2. cereal
3. chicken broth
4. milk
5. oreos

While loops work a bit differently. They run certain code for as long as it takes for a certain specification to be met. The basic logic is as follows: while some condition is true, execute the block of code, then check the condition again. If it's still true, run the code block again. This continues until the condition evaluates to be false.
Let's say we have a timer going, and until that timer hits 0, we want it to run

```
while timer > 0 {
  runTimer()
}
```

Functions

A huge part of code is writing sections of code that accomplish a certain task, and where that task needs to be done more than once. So we don't have to repeat that code, we can write functions that do what we want, and take up minimal space and lines of code to get them to do it. Functions have a format something like this:

```
func functionName(thingYouPassIn: InputType) -> ReturnType {
  do this
  return result
}
```

Let's say we want to measure the circumference of a circle, but we have a bunch of circles to measure. We can easily take care of this by figuring out what needs to go into the function to get the result we want.
We know that pi * radius * 2 is the formula, and 2 and pi are constants.
We can declare pi, and use 2 in the multiplication. Radius is the only thing we have to pass

in.

```
var pi: Double = 3.14
func circleCircumference(radius: Double) -> Double {
  return 2*pi*radius
}
```

We can simply call the first part of the function, pass in the value, and it'll give us the result

```
circleCircumference(radius: 10)
circleCircumference(radius: 25)
circleCircumference(radius: 5.75)
```

Whenever you need to consolidate code, or perform a task more than once, functions are the way to go.

#Sprint_1