Suzy's Swift Cheat Sheet

Arrays/Lists

`var numItems = ["Hello", "Wow"].count`	Return the number of items in a list. Note that this is a property, not a method.

Sets

Unlike lists, sets are unordered.

`var newSet: Set = ["Hello", "Wow"]`	Initialize a new set. You need to explicitly type it as a `Set`, or else it defaults to initializing a list.
`newSet.contains(Value)`	Check if the set `newSet` contains `Value`. Returns a `Bool` value
`newSet.insert(Value)`	Inserts `Value` into `newSet`.
`var newSet = setA.interesection(setB)`	Creates a new set that contains the elements in common with `setA` and `setB`.
`var newSet = setA.union(setB)` `var newSet2 = setA.union(setB).union(setC)`	Creates a new set that combines the elements in `setA` and `setB`. Note that you can chain union operations to combine more than 2 sets.
`var newSet = setA.subtracting(setB)`	Creates a new set that contains values only

	found in setA but not in setB.

Dictionaries

`var newDict: [KeyType: ValueType] = [:]` `var newDict2 = [KeyType: ValueType]()`	Initialize an empty dictionary. Although Swift's documentation uses initializer syntax (the second method), there's no practical difference between the two.
`var newDict = [2956123: "Suzy Lee", 3810473: "Joe Marley"]`	You can initialize a non-empty dictionary without specifying the type.
`currDict.updateValue("Suzy Lee", forKey: 2956123)` `currDict["Suzy Lee"] = 2956123`	Updating the value of an existing key in a dictionary.
`currDict[2956123] = nil` `currDict.removeValue(forKey: 2956123)`	Removes the specified key and value pair from `currDict`. Note: The keyword `nil` describes something as being empty.
`currDict.removeAll()`	Clears out all the contents of `currDict`.

`var currStudent = currDict[2956123]`
`// Optional("Suzy Lee")`
`// Two ways to extract value from an Optional:`
 	
`// #1: CONDITIONAL UNWRAPPING`
`// if-let statements check if a value exists inside an Optional.`
`if let currStudent = currDict[2956123] { print(currStudent)}`
`else { print("Cannot recognize student ID"). }`
 	
`// #2: FORCED UNWRAPPING`
`// if you KNOW that a key exists in the dictionary, just use a !`
`var currStudent = currDict[2956123]!`
	Accessing values from a Swift dictionary.
	
	Swift stores dictionary values as an `Optional` type. This is used as a precaution, in case we try to access the value of a nonexistent key. `Optional<String>` **==** `String?`
	
	OPTIONAL BINDING: You must unwrap the value from an Optional in order to use it. There are two ways to do this:
	(1) **Conditional Unwrapping:** If--let statements
	(2) **Forced Unwrapping:** ! unwrapping [Warning: This method results in errors if you try to unwrap a nonexistent key]

```
for (studID, studName) in currDict {
     print("\(studName) has an ID of
     \(studID)")
}
```

Iterate through the keys and values of a Swift dictionary.

```
currDict.keys // [3810473, 2956123]
currDict.values // ["Suzy Lee", "Joe
Marley"]

for studName in currDict.values {
     print(studName)
}
```

Access just the keys or just the values of a dictionary.

You can also use these properties to iterate through just the keys or just the values.

Functions

```
func greeting() -> Void {
     print("Hello there!")
}

func printCustomGreeting(name) {
     print("Hello there \(name)!")
}

func returnGreeting() -> String {
     Return "Hello there"
}

func favoriteBook() -> (name: String,
author: String, yearPublished: Int) {
  return ("Harry Potter and the
Philosopher's Stone", "J.K. Rowling",
1997)
}
```

Function definitions.

If the function returns nothing, either type `-> Void` or skip the return type specification entirely.

If you want to return a tuple, you have to specify the type of every single item in said tuple (look at the last code snippet).

```
func customGreeting(_ name: String,
age: Int) -> String{
     return(name, age)
}
```

Omitting argument labels.

If you add an underscore (_) exactly one space ahead of an argument name, you

```swift
let currTuple = customGreeting("Suzy",
21)
print("\(currTuple.name) is
\(currTuple.1) years old")
// Prints: Suzy is 21 years old.
```

can omit the parameter name in the function call.

```swift
func calculateAverage(numbers:
Double...) -> Double {
     var total: Double = 0
     for num in numbers {
          Total += num
     }
     return total/(numbers.count)
}
```

Variadic Parameters. A parameter that accepts 0 or more values of a certain type.

Note: A function can only have one variadic parameter.

```swift
var currColor = "green"

func changeBatteryCol(_ percentage:
Int, batteryColor: inout String) {
     if percentage < 5 {
          batteryColor = "red"
     }
     else if percentage < 20 {
          batteryColor = "yellow"
     }
     else {
          batteryColor = "green"
     }
}

changeBatteryCol(15, batteryColor:
&currColor)
```

In-out Parameters

By default, all parameters passed into Swift functions are constants, so their values cannot be changed.

If you want to alter the value of a variable, you must pass it in as an `inout` parameter. Also, when you call the function you must pass in the `inout` argument with an ampersand in front of it.

Structures

```
struct Book {
 var title: String
 var pages: Int

 init (title: String, pages: Int) {
    self.title = title
    self.pages = pages
 }
}
```

Basic definition of a structure and declaring an instance. You can add default values for a property.

Initialization method has virtually the same syntax as a python `init` method. Make sure to declare the properties before the `init`.

```
struct Dog {
  var name = "Dots"
  var age = 15

  func makeNoise() -> String {
    return "Bark bark!"
  }
}
```

Instance Methods

Normal functions, except that they're attached to an instance of this struct. Just like a normal Swift function, you must specify the return type.

```
struct Human {
  var name = "Suzy Lee"
  var age = 21

  mutating func isBirthday() -> Int {
    self.age += 1
    return self.age
  }
}
```

Mutating Methods

Any instance method that changes the value of an instance's property. You must always mark an instance method with the `mutating` keyword in order to make `self` mutable. If a normal method tries to change the value of `self`, it will throw an error.

```
var youngDog = Dog(age: 5, name: "Jo")

// Here we only store the VALUES of
// youngDog, not a reference to it
var oldDog = youngDog
oldDog.age = 10

print(oldDog.age) // prints 10
print(youngDog.age) // prints 5
```

Structures are VALUE types (not reference).

Thus, every time an instance is created or copied, each instance has its OWN set of unique values.

Classes

Unlike structures, classes have inheritance and are **reference** types.

```
class Animal {
  var name = ""
  func makeSound() -> String {
    return "Rawr"
  }
}

class Dog: Animal {
  override func makeSound() -> String
{
    return "bark"
  }
}
```

Overriding Methods

For a subclass to provide its own implementation of a method is inherited from a superclass, it must redeclare said method with the `override` keyword.

```
var krustyKrab = Restaurant(rating:
7.8)
var krustyKrab2 = krustyKrab
krustyKrab2.rating = 4.1

print(krustyKrab.rating)
// Prints: 4.1
print(krustyKrab2.rating)
// Prints: 4.1
```

Structures are REFERENCE types (not values). So be careful when you change the property values of an instance of a class.

If you need the instances of a class to be completely separate from each other, avoid initializing them by copying references to previous instances.

```
class GigantamaxPokemon: Pokemon {
 var location = ""
 init(num: Int, name: String, type:
[String], ability: [String], location:
String) {
   super.init(num: num, name: name,
type: type, ability: ability)
   self.location = location
 }
}
```

Super keyword

When declaring the `init` method for a subclass, use the `super` keyword to inherit the superclass's initialization.

If you don't declare a custom `init` for a subclass, you'll get an error!

You don't need to add the `override` keyword for init. For other normal class methods, you do.

Structs v.s. Classes

When do I use a struct and when do I use a class?

Rule of thumb: Start off by declaring a `struct`, and then later convert it to a `class` if you need to use inheritance.

Structs	Classes
When a data type is a `struct`, you can be certain that no other part of your code holds a reference to it. Thus, you can grasp data changes in your codebase a lot easier.	There may be a line of code that changes the value of a property in your class instance! Not very protected.
All instances of structs are just copies of the values.	Classes come with a sense of identity, so you can use the equality operator (==) **to check if two variables are referring to the same class instance**.