GAME 3004

SpriteKit - Week 2

Lesson 2

Expectation

Understand the basic principles of Swift

Outcome

Understanding the more advanced principles of Swift

Key Concepts

Optionals Classes

Inheritance

Protocols

Enumerations

Structures

Dear Swift,

Why do I keep seeing a "?"

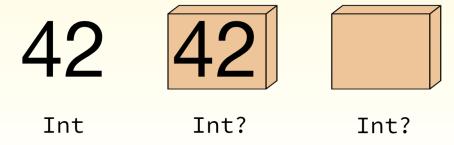
What's in this?!

-Programmer



Optionals are a special feature in Swift used to indicate that an instance may have a value, or be nil. This is Swift's solution to the problem of representing both a value and the absence of a value

Think of optionals like a box, it can contain a value or it might not.



There are times when we would like to access a value, but it might not yet be assigned or be nil.

Instead of seeing your program fail, we can use an Optional type.

This makes accessing values **safe**, **and** code will not crash with runtime error even if it's **nil**

You can declare an Optional type with "?"

var name: String

var date: String

var city: String

var country: String

var occupation: String?

var sinNumber: String?

An optional can be good when some fields in a program are not required to have a value - for example think of a **form**

In a form some values may be required, others may be optional. For example your name, date, city may be required while others may be not

var name: String = "Hi"

Declaring a regular String type guarantees a value at runtime. You will not be able to run this program unless it has been assigned



var name: String? = "Hi"

Declare an **Optional** type by placing a "?" after the type.

Note: This is no longer a String. It is an Optional String. Know the difference!



Optional Binding

Optional binding is a useful pattern to detect whether an optional contains a value.

If there is a value, then you can assign it to a temporary constant or variable and make it accessible inside a condition

You can check if an **optional is nil** and assign it to a **temporary value** using just **one line of code**

Optional Binding

Binding assigns a temporary variable to the value of the optional if it is not nil

If there is a value in the Optional value, assign to it a temporary constant, temporary Value

Even if the Optional is nil, your program will still not crash

Optional Binding

```
if let temporaryConstant = anOptional {
   //Do something with temporaryConstant
} else {
   //There was no value in anOptional; i.e., anOptional is nil
```

Optionals - Force Unwrapping

Force unwrapping can be used when an optional value is required to have value

Be careful! Once you force unwrap an optional variable, if it returns **nil**, the program will crash at runtime - Be sure it has a value!

Use "!" to force unwrap an optional variable

print(anOptionalValue!)

Enumerations

Enums allow you to create instances of a predefined list of cases

Use the **enum** keyword followed by the name of the enumeration

Every enum must contain at least one case

```
enum TextAlignment {
    case Left
    case Right
    case Center
}
```

Classes

A class is a **blueprint** or **template** for an instance of that class

Classes are named types with properties and methods inside.

Use the **keyword class** followed by the **name** of the class to create it.

Classes are reference types

Classes

```
class Person {
  var firstName: String
  var lastName: String
  var age: Int?
  init(firstName: String, lastName: String) {
     self.firstName = firstName
     self.lastName = lastName
```

Inheritance

Subclasses receive all **properties and methods** defined in their superclass, plus **any additional methods and properties** the subclass defines for itself.

Subclasses can override methods used within the parent class with the override keyword

Use the keyword **super** when ever you are **overriding a method** from the superclass

```
class Student: Person {
   var grades: [Grade] = []
   override init(firstName: String, lastName: String) {
      super.init(firstName: firstName, lastName: lastName)
   }
}
```

Structures

Structures are named types that groups a set of related chunks of data together in memory. You can use a structure when you would like to group data under a common type.

Use the **struct** keyword to create this type

```
struct DeliveryArea {
    let center: Location
    var radius: Double
}
```

Classes vs Structures

Structs do not support inheritance, and so they cannot be subclassed

If the behavior you would like to represent in a **type is relatively straightforward and encompasses a few simple values**, consider starting out with a struct.

Structs have faster memory allocation (stack); classes have slower memory allocation (heap)

Structs are used for implicit copying of values; while classes are useful for implicit sharing of objects

Structs are a value type, while classes are a reference type

Protocols

Protocols define an **interface** or **blueprint** that actual concrete types conform to. With a protocol you define a common set of properties and behaviors that concrete types go and implement.

A protocol can be adopted by a **class**, **struct** or **enum**

When a type adopts a protocol, it's required to implement the methods and properties defined in the protocol. Once all methods and properties are implemented the type is said to conform to the protocol.

Protocol

Use the **protocol** keyword to define a protocol

```
protocol Vehicle {
    func accelerate()
    func stop()
}
```

Protocol Conformance

Use ':' to allow a **class**, **enum or struct** access to this protocol. Implement all required methods or parameters to conform.

```
class Unicycle: Vehicle {
   var peddling = false
   func accelerate() {
       peddling = true
   func stop() {
       peddling = false
```

Properties

Properties are fancy names for **variables or constants** that belong to a **class**, **struct** or **enum**

Properties can be of two varieties: **stored** or **computed**

Stored Properties

Stored properties simply store the actual values for each instance

Stored properties can be given a default value

var name: String = "Mark"

Computed Properties

Computed properties provide a **getter and optional setter to retrieve or set** a property's value

Computed properties are calculated every time your code requests them and aren't stored as a value in memory

```
var townSize: Size {
  get {
    return self.population
}
set {
```

Computed Properties

Computed properties provide a **getter and optional setter to retrieve or set** a property's value

Computed properties are calculated every time your code requests them and aren't stored as a value in memory

```
var townSize: Size {
  get {
    return self.population
}
set {
```

Access Control

open - entities are open to entire program

public - entities can be seen by class framework and subclasses

internal - entities can be seen within a framework

file private - entities can be used within an entire file

private - entities are private to the enclosed declaration

Access Control

By default, entities have an internal access level

Classes can also have access level keywords

public ClassA {

}

This would define what access level the class would have when declared