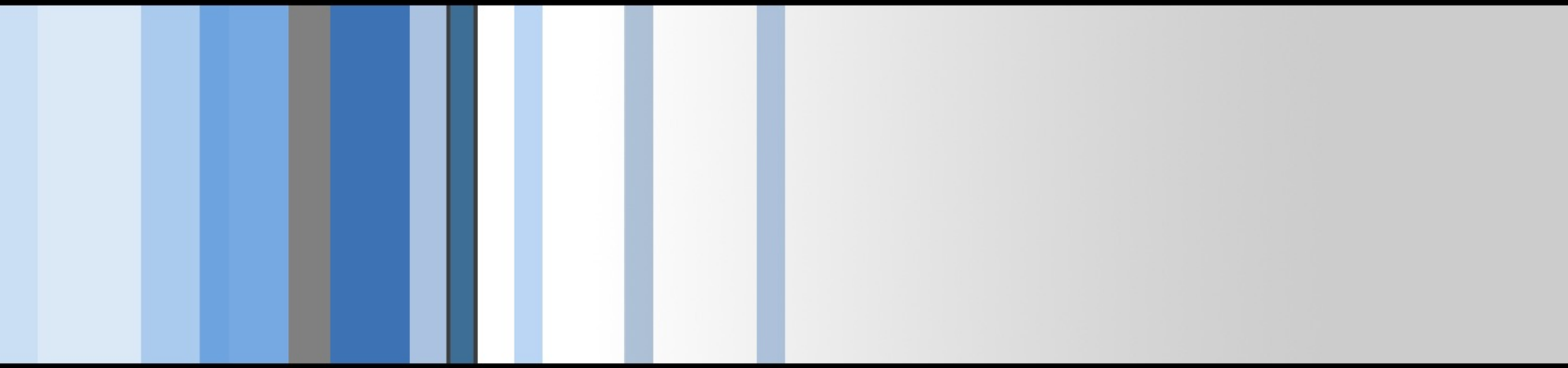
Class 5

CSC 496: iOS App Development

Swift Fundamentals: Functions, Structs, Computed PropertySi

Chen (schen@wcupa.edu)



Functions

- Functions are a building block of almost all programming languages.
- Allowing functionality to be defined and reused.
- The advantage of this is that code can be reused, and your overall codebase becomes much easier to understand.

Function Declaration

Declaring a function in Swift is accomplished using the **func keyword**. Let's take a look at a basic function declaration:

```
func greet() {
    print("Hello, world!")
}
```

In this example, greet is the name of the function. It doesn't take any parameters and doesn't return a value. Whenever this function is called, it will print out "Hello, world!" to the console.

Functions

Calling Functions

- After declaring a function, you can **call it** whenever you want to execute its defined task. You do this by using the function's name followed by parentheses. Let's see this in action relatively to our previous example:

```
greet()   //It prints "Hello, world!"
```

Return Types

- Some functions don't just perform a task, they also **return a value**. This is indicated by using the **->** symbol along with the **type of data** the function is going to return. Check out this example:

```
func addTwoNumbers(num1: Int, num2: Int) -> Int {
    return num1 + num2
}

let sum = addTwoNumbers(num1: 3, num2: 5)  // sum now holds the value 8
```

Functions

Calling Functions

- After declaring a function, you can **call it** whenever you want to execute its defined task. You do this by using the function's name followed by parentheses. Let's see this in action relatively to our previous example:

```
greet()   //It prints "Hello, world!"
```

Return Types

- Some functions don't just perform a task, they also **return a value**. This is indicated by using the **->** symbol along with the **type of data** the function is going to return. Check out this example:

```
func addTwoNumbers(num1: Int, num2: Int) -> Int {
    return num1 + num2
}

let sum = addTwoNumbers(num1: 3, num2: 5)  // sum now holds the value 8
```

Functions

```
func addTwoNumbers(num1: Int, num2: Int) -> Int {
    return num1 + num2
}

let sum = addTwoNumbers(num1: 3, num2: 5)  // sum now holds the value 8
```

In this example, **addTwoNumbers** is a function that takes in **two parameters (num1, num2)**, both of the type **Int**.

It performs the addition of these two numbers and then returns the result, which is also of the type **Int**.

Functions

Parameter Names

- In Swift, every parameter has both an **external name and an internal name**. The **internal** name is used within the function body, while the **external** name is used when calling the function. Let's break down into an example:

```
func greet(person: String, from hometown: String) -> String {
    return "Hello \(person)! Glad you could visit from \(hometown)."
}
```

In this function, **person** is the **internal name** of the first parameter and **hometown** is the **internal name** of the second parameter. But when we call this function we use the **external names person** and **from**:

```
print(greet(person: "Bill", from: "Cupertino"))
```

Functions

Default Parameters

- A default parameter value is a value that is assigned to a function parameter if no argument is provided when calling the function. In Swift, you can specify a default parameter value by assigning a value to the parameter when you define the function. Here's an example on how it's done:

```
func makeCoffee(type: String = "Espresso") {
    print("Your \(type) is ready!")
}
```

- You can call this function in two ways:

```
makeCoffee() //Output: 'Your Espresso is ready!'
makeCoffee(type: "Cappuccino") //Output: 'Your Cappuccino is ready!'
```

Functions

Variadic Parameters

- Have you ever wanted to accept an unknown number of arguments in your function?
- Swift has you covered with **Variadic Parameters** -- A variadic parameter accepts zero or more values of a specified type. In Swift, you can indicate a variadic parameter by inserting three period characters (...) after the parameter's type name. Let's look at an example:

```
func sum(numbers: Int...) -> Int {
    var total = 0
    for number in numbers {
        total += number
    }
    return total
}

print(sum(numbers: 1, 2, 3, 4))
```

Functions

In-Out Parameters

- Sometimes, we want a function to modify one or more of its parameters.
- In Swift, **function parameters are constants by default**. But with in-out parameters, a function can change the value that was passed in, and those changes will persist after the function has finished! Let's see an example how to use an in-out parameter:

```
func doubleInPlace(number: inout Int) {
    number *= 2
}

var myNum = 10
doubleInPlace(number: &myNum)
print(myNum)
```

This will output: **20**
In this case, you see **myNum** was doubled inside the function, and it retained the changed value after the function is called. This is the power of in-out parameters

Functions: Conclusion

- Functions are a building block of almost all programming languages.
- Allowing functionality to be defined and reused.

```
func nameOfFunction(parameter1: ParamaterType1,
parameter2: ParamaterType2, …) -> OutputType {
    // Function's implementation
    // If the function has an output type
    // the function must return a valid value
    return output
}
```

```
3  func fullName(givenName: String, middleName: String,
       familyName: String) -> String {
4      return "\(givenName) \(middleName) \(familyName)"
5  }
6
7  let myFullname = fullName(givenName: "Kevin", middleName:
       "David", familyName: "Moon")
8
9  print(myFullname)
```

Question 1: Fahrenheit to Celsius Converter

- Write a Swift function named **convertToCelsius** that takes a single argument, a **Double** representing a temperature in Fahrenheit. The function should return a **Double** representing the equivalent temperature in Celsius.
- The formula to convert from Fahrenheit to Celsius is

$$(Fahrenheit - 32) \times \frac{5}{9}$$

Example Usage:

```
let tempInCelsius = convertToCelsius(fahrenheit: 98.6)
print(tempInCelsius)  // Output should be 37.0
```

Question 2: Fibonacci Sequence Generator

- Write a Swift function named **generateFibonacci** that takes a single argument, an **Int n**, and returns an array of **Ints** containing the first **n** numbers in the Fibonacci sequence. The Fibonacci sequence starts with the numbers 0 and 1, and each subsequent number is the sum of the two preceding ones (0, 1, 1, 2, 3, 5, 8, ...).
- **Function Signature:**

```swift
func generateFibonacci(n: Int) -> [Int] {
    // Your code here
}
```

- **Example Usage:**

```swift
let fibonacciNumbers = generateFibonacci(n: 6)
print(fibonacciNumbers)  // Output should be [0, 1, 1, 2, 3, 5]
```

Nested Functions

Declaration of Nested Functions

- Nested functions, also known as Inner functions, are functions declared within other functions. Think of it like boxes inside. Here's a basic structure of a nested function:

```
func outerFunction() {
    func innerFunction() {
        // code
    }
    // code
}
```

- In the above code, **innerFunction()** is nested inside **outerFunction()**. The **innerFunction()** is only visible within the scope it is defined, which means we can only call **innerFunction()** inside **outerFunction()**. Trying to call **innerFunction()** outside **outerFunction()** will throw an error.

Nested Functions

Calling Nested Functions

- How to call a nested function? You just call it like you would any other function, however the scope is important. Here's an example:

```
func outerFunction() {
    func innerFunction() {
        print("Hello, World! 🌍")
    }

    innerFunction()
}

outerFunction()
```

- When we run outerFunction(), it also executes innerFunction() which prints Hello, World! 🌍. However we cannot call innerFunction() independently.

Nested Functions

Use Cases for Nested Functions

- Now, you might wonder why we need nested functions in Swift. There are several reasons that encourage the use of nested functions:
  - **Encapsulation**: Nested functions are a way to hide functionality that isn't necessary for the outside world to see or use.
  - **Readability**: Grouping related code together can make it much easier to understand.
  - **Prevents namespace pollution**: The scope of nested functions is limited to the enclosing function, which can prevent potential name collisions.

```
func makeIncrementer(incrementAmount: Int) -> () -> Int {
    var total = 0
    func incrementer() -> Int {
        total += incrementAmount
        return total
    }
    return incrementer
}

let incrementByTwo = makeIncrementer(incrementAmount: 2)

print(incrementByTwo()) // prints 2
print(incrementByTwo()) // prints 4
```

Nested Functions

```
func makeIncrementer(incrementAmount: Int) -> () -> Int {
    var total = 0
    func incrementer() -> Int {
        total += incrementAmount
        return total
    }
    return incrementer
}

let incrementByTwo = makeIncrementer(incrementAmount: 2)

print(incrementByTwo()) // prints 2
print(incrementByTwo()) // prints 4
```

In the above example, **incrementer()** is a nested function within the **makeIncrementer()** function. This function generates and returns another function that increments total by a specified amount.

Exercise 1: Nested Multiplier Function

Objective:

- To understand the concept and usage of nested functions in Swift by implementing a function that multiplies two numbers.

Description:

- Create a nested function called **multiplier** inside a function called **calculate** that takes two parameters **a** and **b**. The nested function multiplier should multiply **a** and **b** and return the result. Call the calculate function with any two numbers and print the result.

```swift
func calculate(a: Int, b: Int) -> Int {
    func multiplier() -> Int {
        return a * b
    }

    return multiplier()
}

let result = calculate(a: 5, b: 6)
print("Result: \(result)")
```

Exercise 2: Cumulative Multiplication Function

Objective:

- To explore more advanced use-cases of variable capturing for maintaining state in nested functions.

Description:

- Create a function called **makeMultiplier** that takes an integer parameter **multiplyAmount**. Inside it, define a nested function called multiplier that multiplies a running total (stored in the outer function) by **multiplyAmount**. The **makeMultiplier** function should return this nested function.

Example Usage:

```
// Your implementation for 'makeMultiplier' goes here

let multiplyByThree = makeMultiplier(multiplyAmount: 3)

print(multiplyByThree()) // Should print 3
print(multiplyByThree()) // Should print 9
```

ContentView.swift (Pokédex version 2)

```
struct ContentView: View {
    @State private var PokemonID = "1"

    var body: some View {
        VStack {
            Text("Pokedex Ver 2.0")
                        .font(.custom("Pokemon-Pixel-Font", size: 36))

            TextField("Pokemon ID:", text: $PokemonID)
                    .multilineTextAlignment(.center)
                    .keyboardType(.numberPad)
                    .font(.custom("Pokemon-Pixel-Font", size: 36))

            Image(PokemonID)

            if PokemonID == "151" || PokemonID == "150" {
                Image("rare")
                    .resizable()
                    .frame(width: 100, height: 100)
            }

        }
        .padding()
    }
}
```

Structs

- Bundling values into structs
  - **Class** objects are great for encapsulating data and functionality within a unifying concept.
  - However, not everything is an object
    - We may need to represent data that is logically grouped together
    - But there isn't much more than that → Use **Structs**
- Structs are **value types**, not classes

```
struct PersonName{
    // three properties to PersonName
    let givenName: String
    let middleName: String
    var familyName: String

    // add a method to combine the three properties into a fullName String
    func fullName() -> String{
        return "\(givenName) \(middleName) \(familyName)"
    }

    // provide a method to change the family name property
    // and prefix this method with mutating keyword
    mutating func change(familyName: String){
        self.familyName = familyName
    }
}
```

```
// create a person name
// aside from using the struct keyword instead of class.
// the definition of a class and a struct are almost idential
var alissaName = PersonName(givenName: "Alissa", middleName: "May", familyName: "Jones")
```

Structs has Value-type Semantics

```
// e.g.,
var alissaCurrentName = alissaName // VERY IMPORTANT!
print(alissaCurrentName.fullName())

alissaName.change(familyName: "Chen")
print(alissaName.fullName())
print(alissaCurrentName.fullName())
```

```
Alissa May Jones
Alissa May Chen
Alissa May Jones
```

// Struct a value-type semantics --> when you mutate a struct, you **create a copy** of the struct with the changed properties

Question 1: Coordinate Point Representation

- Create a Swift struct named Point that has two properties: x and y, both of type Double.
- Also, write a function within the struct named distanceToOrigin that calculates the distance of the point to the origin (0, 0) using the formula: $\sqrt{x^2 + y^2}$

- **Example Usage:**

```
let point = Point(x: 3, y: 4)
let distance = point.distanceToOrigin()
print(distance)  // Output should be 5.0
```

Question 2: Simple Bank Account

- Create a Swift **struct** named **BankAccount** that has a balance property of type **Double**. Write two methods within the struct:

1. **deposit(amount: Double):** Adds the amount to the balance. Return the new balance.
2. **withdraw(amount: Double):** Subtracts the amount from the balance. If the withdrawal amount is greater than the balance, return **nil**; otherwise, return the new balance.

- **Struct Definition:**

```
struct BankAccount {
    var balance: Double

    mutating func deposit(amount: Double) -> Double {
        // Your code here
    }

    mutating func withdraw(amount: Double) -> Double? {
        // Your code here
    }
}
```

Computed Property

- In Swift, a computed property doesn't store a value. Instead, it provides a getter and an optional setter to retrieve and set other properties and values indirectly.
- Computed properties are used when the property's value is **derived or calculated from other properties' values** or **needs to be set dynamically**.

Computed Property

Syntax

The basic syntax of a computed property involves using a code block { } after the property name to include a get block and optionally, a set block.

```
struct Rectangle {
    var width: Double
    var height: Double

    var area: Double {
        get {
            return width * height
        }
        set(newArea) {
            // For simplicity, assume a square shape for the new area
            width = sqrt(newArea)
            height = sqrt(newArea)
        }
    }
}
```

In this example, area is a computed property that calculates its value by multiplying width and height. You can also set area, and doing so will update width and height accordingly.

Lazy Initialization 😴

- Sometimes, it might be kin on resources if you compute a complex property upfront. Swift allows for lazy initialization of properties with the lazy keyword. This means that the computation of the property is delayed until it is first accessed. Consider:

```
class ComplexOperation {
    lazy var expensiveValue: Int = {
        // Some expensive computation
        return 4 // Placeholder value
    }()
}
```

- Here, **expensiveValue** won't be computed until the first time it's accessed. This can make your app more efficient by spreading out the work over time!

Lazy Initialization 😴

- Sometimes, it might be kin on resources if you compute a complex property upfront. Swift allows for lazy initialization of properties with the lazy keyword. This means that the computation of the property is delayed until it is first accessed. Consider:

```
class ComplexOperation {
    lazy var expensiveValue: Int = {
        // Some expensive computation
        return 4 // Placeholder value
    }()
}
```

The parentheses at the end of the expensiveValue computed property is known as a "closure expression" and it allows the property to have a default value that is computed lazily.

- Here, **expensiveVal**
  accessed. This can n e
  work over time!

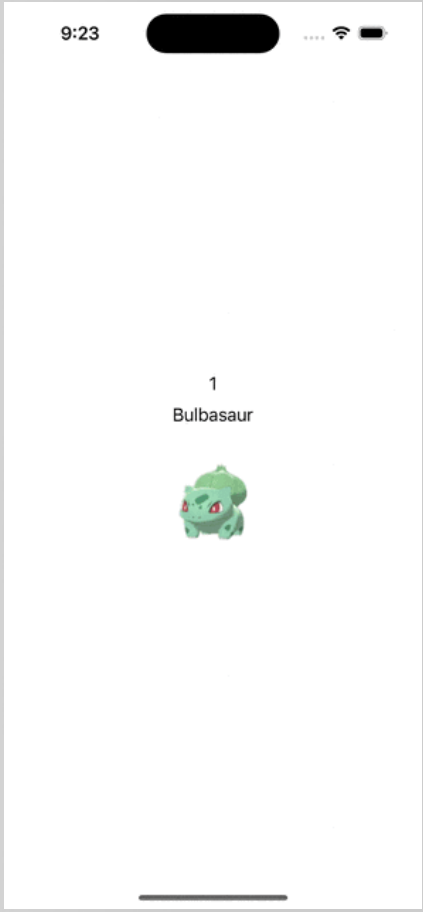
Pokédex version 2 Solution (using computed property)

```
import SwiftUI

struct ContentView: View {
    @State private var PokemonID = "1"

    var pokemonID_num: Int {
        get {
            return min(max((Int(PokemonID) ?? 1) - 1, 0), 150)
        }
        set(newID) {
            PokemonID = String(newID + 1)
        }
    }

    var body: some View {
        VStack {
            Text("Pokedex Ver 2.0")
                .font(.custom("Pokemon-Pixel-Font", size: 36))

            TextField("Pokemon ID:", text: $PokemonID)
                .multilineTextAlignment(.center)
                .keyboardType(.numberPad)
                .font(.custom("Pokemon-Pixel-Font", size: 36))
        }
        .padding()
    }
}
```

In this case, the computed property pokemonID_num is derived from PokemonID. Any time you get or set pokemonID_num, the underlying PokemonID state variable is accessed or modified.

Pokédex version 2

Objective:
To create an iOS app that displays the name and profile picture of a Pokémon based on the user-inputted Pokémon ID.



Instructions:

- **Download and Setup Project:**

Download the **pokedex_ver_2.zip** file from our class website. Unzip the file and open the project in Xcode.

- **Examine the Pokemon.Swift File:**

Open the **Pokemon.Swift** file in the project. Locate the array **firstGenPokemonNames** which contains the names of the first-generation Pokémon.

- **Implement User Input and Display in ContentView.swift:**

In **ContentView.swift**, write code to accomplish the following:

- Use the firstGenPokemonNames array to find the name of the Pokémon corresponding to the entered ID.
- Display the Pokémon's name and its corresponding profile picture based on the entered ID.

For example, if the user inputs Pokémon ID = 1, the app should display "Bulbasaur" along with its profile picture.

Tips:
You may use the .font(.custom("Pokemon-Pixel-Font", size: 16)) modifier to set a Pokemon font.
How would you handle invalid input (try using nil-coalescing operator ??)
How would you handle edge cases where the user input exceeds 151 or falls below 0?

Classes

- Object-oriented programming is a common and powerful programming paradigm.
- Classes in Swift allow you to define blueprints for objects, and they are one of the building blocks of object-oriented programming (OOP).
- Classes can have properties, methods, and initializers, just like structures.
- However, they also offer **additional functionalities** not available in structures, such as inheritance, type casting, and deinitializers.

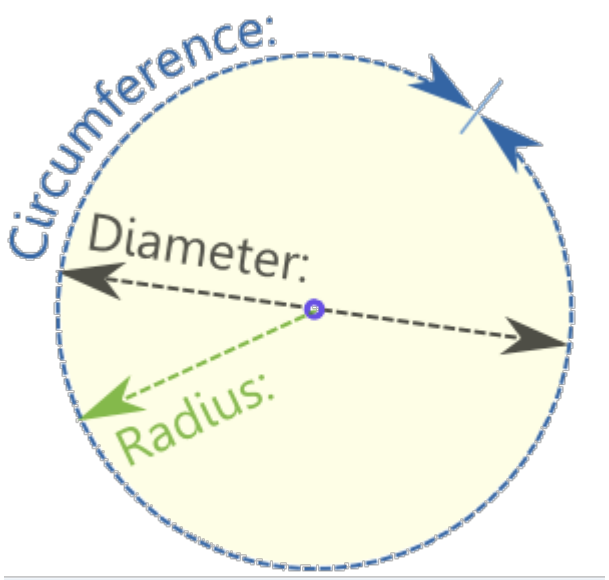
Classes

```
class Person{
    let givenName: String
    let middleName: String
    let familyName: String
    var countryOfResidence: String = "UK"

    // add an initialization method
    init(givenName: String, middleName: String, familyName: String)
    {
        self.givenName = givenName
        self.middleName = middleName
        self.familyName = familyName
    }

    // add a variable as a property of the class, with a computed value
    var displayString: String{
        return "\(self.fullName()) - Location: \(self.countryOfResidence)"
    }

    // add a function within the Person class
    func fullName() -> String{
        return "\(givenName) \(middleName) \(familyName)"
    }
}
```

Classes

```
// create a Friend class that extends the functionality of the Person class
final class Friend: Person{
    // within the Friend class, add a variable property to hold details of where the user met the friend
    var whereWeMet: String?
    override var displayString: String{
        let meetingPlace = whereWeMet ?? "Done't know where we met"
        return "\(super.displayString) - \(meetingPlace)"
    }
}
```

Circle Class



Circle
- radius : double - PI : double = 3.14159
+ Circle(r : double) + setRadius(r : double) : void + getRadius() : double + getArea() : double + getDiameter() : double + getCircumference() : double

Q & A