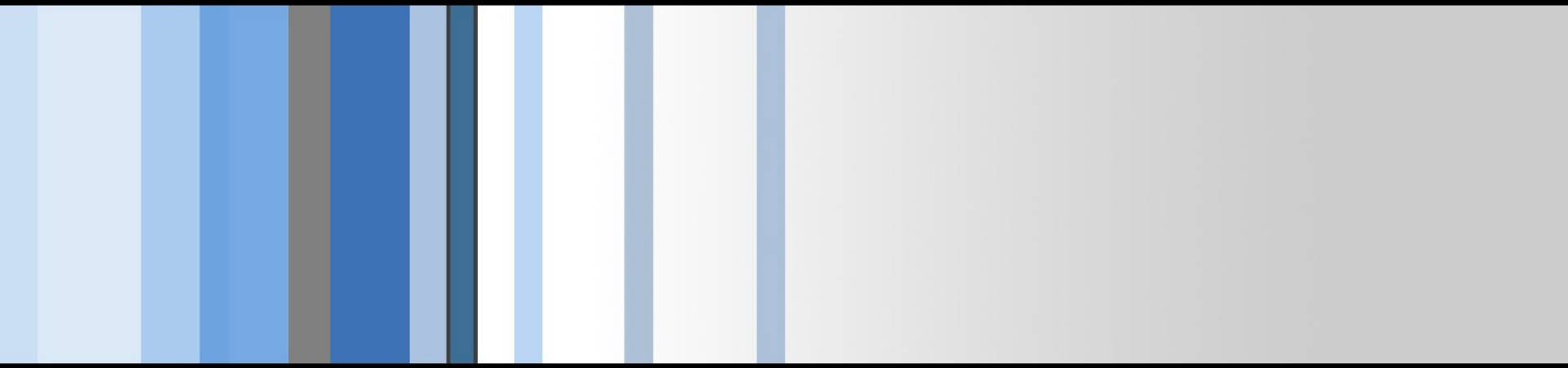
Class 6

# CSC 496: iOS App Development
# Swift Fundamentals: Structs, Computed Property

Si Chen (schen@wcupa.edu)



# 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

- Structs, short for structures, allow us to group related types of data together. This is very useful when developing apps in iOS. It helps in organizing our code, making it more readable, and reusing it in different parts of the app.
- **1. Declaration**: To declare a struct, we first use the keyword '**struct**', followed by *the name of the struct*, and then we enclose the properties and methods of the struct in braces **{}**. Here's how you do it:

```
struct Student {
    var name: String
    var age: Int
}
```

In this example, we've declared a struct Student that has two properties, name and age.

# Structs

- **2. Initialization**: Structs in Swift have an automatic member wise initializer, which you can use to initialize the struct. Here's how you do that:

```
let student = Student(name: "John Doe", age: 20)
```

Here, we've created an instance of Student struct.

- **3. Accessing Properties**: To access the properties of a struct, you can use the dot syntax.

```
print(student.name) // Outputs "John Doe"
```

# Structs

- **2. Initialization**: Structs in Swift have an automatic member wise initializer, which you can use to initialize the struct. Here's how you do that:

```
let student = Student(name: "John Doe", age: 20)
```

Here, we've created an instance of Student struct.

- **3. Accessing Properties**: To access the properties of a struct, you can use the dot syntax.

```
print(student.name) // Outputs "John Doe"
```

# Structs

- **4. Methods**: Structs can also have methods in Swift. Yes, you heard it right! Here's how to declare a method inside a struct:

```
struct Student {
    var name: String
    var age: Int

    func description() -> String {
        return "\(name) is \(age) years old!"
    }
}
```

Here, we've added a method description to our Student struct. Let's call description method using an instance of Student:

```
let student = Student(name: "John Doe", age: 20)
print(student.description()) // Outputs "John Doe is 20 years old!"
```

# Structs

- **5. Mutability: By default, struct instances are immutable**, meaning we can't change their properties once they are created. However, we can make them mutable by declaring the instance with **'var**' instead of **'let**'. Here's an example:

```
struct Car {
    var make: String
    var model: String

    func honk() {
      print("Beep beep! I'm a \(make) \(model).")
    }
}

let myCar = Car(make: "Toyota", model:"Corolla")
myCar.honk()  // This will print: Beep beep! I'm a Toyota Corolla.
```

```
var myCar = Car(make: "Toyota", model:"Corolla")
myCar.make = "Honda"  // This will change the make of myCar to "Honda"
```

# mutating keyword

- In the **Car**, instances are immutable by default unless explicitly declared mutable with the **var** keyword.
- However, even if an instance is mutable, **any method** within the struct that aims to modify the properties of the struct must be marked with the **mutating** keyword.
- In our example, the **honk()** method merely accesses the **make** and **model** properties without modifying them, so it doesn't need to be marked as **mutating**.
- However, if you **have a method that tries to change the value** of **make** or **model**, you will need to use the **mutating** keyword like so:

```
mutating func updateMake(newMake: String) {
    self.make = newMake
}
```

# Structs: Conclusion

- 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")
```

# 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

- **Declaration**: A simple way to declare a Computed Property in Swift looks like this:

```
var name: Type {
    get {
        // return computed value
    }
    set(newValue) {
        // set value
    }
}
```

You can see above, we use the **getter** to access the value of the property and the **setter** to set or modify it.

## Declaration: Example

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.

# Computed Property

- **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!

# Computed Property

- **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!

# Property Observers

- **Property Observers:** Swift offers two kinds of property observers **willSet** and **didSet**, which get called before and after a property's value gets changed. Useful to track the changes!

```
var score: Int = 0 {
    willSet(newScore) {
        print("About to set score to \(newScore)")
    }
    didSet {
        print("Score was just set from \(oldValue) to \(score)")
    }
}
```

# 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:**

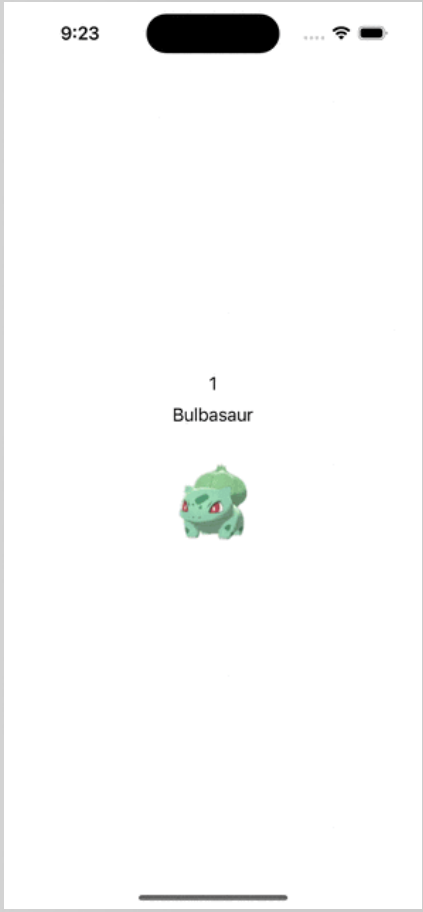
Open the **Pokemon.Swift** file in the project. Locate the array **firstGenPokemonNames** which contains the names of the first-generation Pokémon.

- **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.

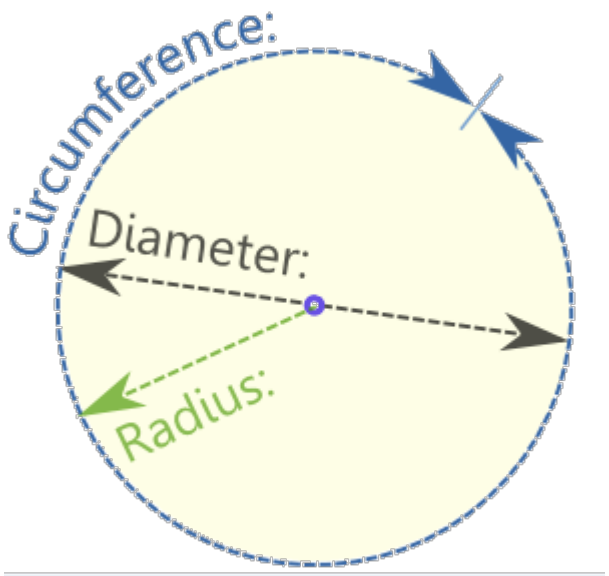
# Classes

```
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<br>- PI : double = 3.14159 |
| + Circle(r : double)<br>+ setRadius(r : double) : void<br>+ getRadius() : double<br>+ getArea() : double<br>+ getDiameter() : double<br>+ getCircumference() : double |

# Structs vs Classes

- In Swift, structs and classes have a lot in common:
  - both can have properties and methods.
- But there are some key differences you should be aware of:
  - Unlike classes, structs are **always copied** when they are **passed around your code**.
  - Structs can't be subclassed.

# Q & A