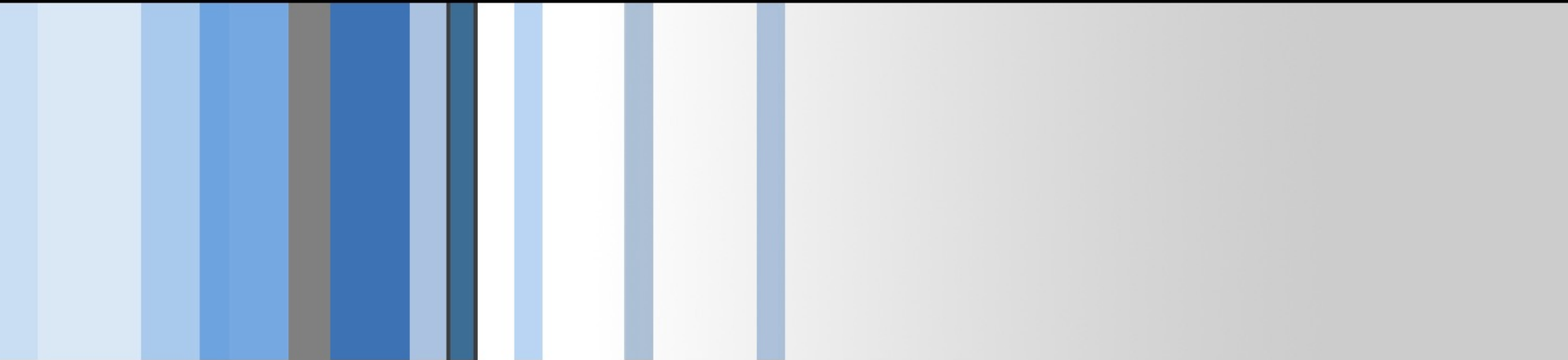


# CSC 496: iOS App Development

## Swift Fundamentals: Enums, Closures

Si Chen (schen@wcupa.edu)



# Review

- Unlike a class, struct, or enum, a **protocol** is a blueprint of methods, properties, and other requirements that suit a particular action or piece of functionality.
- They're like the rules of the game, setting the standards for how classes, structures, and enumerations should appear and behave.

## Protocol Syntax

Swift protocols have a specific syntax, similar to what you would see when defining a class or structure. However, protocols won't provide any implementation for the requirements they define.

```
protocol SomeProtocol {  
    // protocol definition goes here  
}
```

# Adopting Protocols

- A class, structure, or enumeration can adopt a protocol by listing its name after their own, separated by a colon :
- Look at this simple example:

```
struct SomeStructure: FirstProtocol, AnotherProtocol {  
    // structure definition goes here  
}
```

Here, 'SomeStructure' is adopting and conforming to 'FirstProtocol' and 'AnotherProtocol'.

# Adopting Protocols: Example

```
protocol Fordable {  
    var hasFourWheels: Bool { get }  
    func startEngine()  
}
```

In this protocol, **Fordable**, there are a variable **hasFourWheels** and a function **startEngine()**. Any type that adopts this protocol will need to implement these two properties.

```
class Car: Fordable {  
    var hasFourWheels: Bool = true  
    func startEngine() {  
        print("Engine started!")  
    }  
}
```

# Protocol Inheritance

- Swift protocols can inherit from one or many protocols. This enables you to build on top of existing protocols to provide more specialized behavior. Let's see how it's done.

```
protocol InheritingProtocol: SomeProtocol, AnotherProtocol {  
    // protocol definition goes here  
}
```

```
protocol Vehicle {  
    var hasWheels: Bool { get }  
}  
  
protocol Fordable: Vehicle {  
    var hasFourWheels: Bool { get }  
}
```

In this scenario, Fordable is inheriting Vehicle protocol. Therefore, anyone who will adopt Fordable protocol will need to fulfill the requirements of Vehicle as well.

# Optional Protocol Requirements

- In Swift, protocols can also have optional requirements, which are not required to be implemented by a type in order to conform to the protocol. These optional requirements are marked by the **optional** keyword.

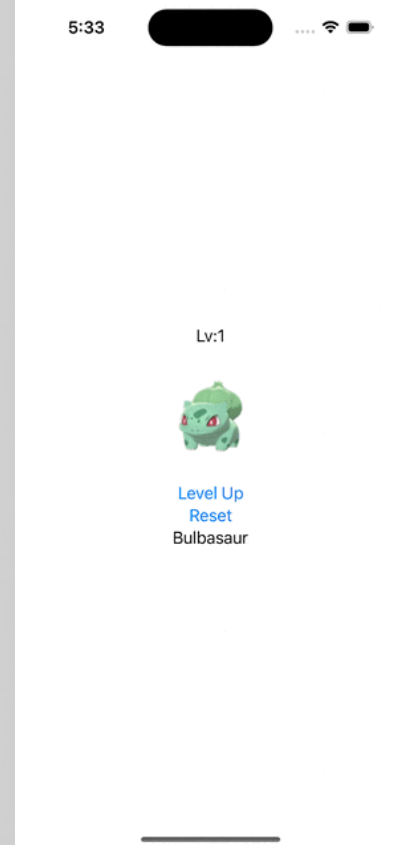
```
@objc protocol Moveable {  
    @objc optional func startMoving()  
    @objc optional func stopMoving()  
}
```

- Here **startMoving()** and **stopMoving()** are not mandatory for any type that adopts **Moveable** protocol.

**Scenario:** You have a custom class, and you want your SwiftUI view to update when an attribute of the class changes.

**Problem with Bulbasaur Class in SwiftUI:** Simply updating the class attribute won't update the UI.

- **ObservableObject:** A protocol that SwiftUI uses to re-render a view when the data it observes changes.
- **@Published:** A property wrapper that marks properties of an ObservableObject to notify the UI of changes.





# Understanding SwiftUI's Data Flow - ObservableObject and @Published

## ▪ How to Implement ObservableObject

1. Conform your class to the **ObservableObject** protocol.
2. Use **@Published** to annotate attributes that, when changed, should trigger a UI update.

```
class Bulbasaur: ObservableObject {  
    // Attributes  
    @Published var id: Int  
    @Published var level: Int  
}
```

Use **@ObservedObject** or **@StateObject** to bind an instance of your class to the view.

```
@StateObject var b = Bulbasaur()
```

- **@ObservedObject**: Use when your observable object is passed from a parent view.
- **@StateObject**: Use when your observable object is created within the view.

# Enums

- Enumerating values with enums
  - Enumerations are a programming construct that lets you define a value type with a finite set of options.
  - Similar to enum type in Java

```
// Enum
enum Title{
    case mr
    case mrs
    case mister
    case miss
    case dr
    case prof
    case other
}
let title1 = Title.dr
```

In many programming languages, including C, enum are defined as a type definition on top of an **integer**.

In Swift, enums **do not need** to represent **integers** --> they do not need to be backed by any type

# Enums: Raw Values and Functions

- In Swift, you can assign raw values to enumeration cases when you define the enumeration.

```
enum Rating: Int{
    case worst // Inferred as 0
    case bad // Inferred as 1
    case average // Inferred as 2
    case good // Inferred as 3
    case best // Inferred as 4
}

let r1 = Rating.good
print(r1)
print(r1.rawValue)
```

```
enum TitleNew: String{
    case mr = "Mr"
    case mrs = "Mrs"
    case mister = "Mister"
    case miss = "Miss"
    case dr = "Dr"
    case prof = "Prof"
    case other // Inferred as "other"

    func isProfessional() -> Bool {
        return self == TitleNew.dr || self == TitleNew.prof
    }
    // feels more appropriate as a computed property
    var isProfessional2:Bool{
        return self == TitleNew.dr || self == TitleNew.prof
    }
}
```

- We can also add functions or computed property into enums

# Enums: Associated Values

**Associated Values** Enums in Swift can store associated values, providing additional information about each case. Check out this example:

## Example: Dungeon Room Enumeration

Create an enumeration called **DungeonRoom** with cases for different types of rooms: **treasureRoom**, **monsterRoom**, and **emptyRoom**.

- Each **treasureRoom** contains an integer value representing the amount of gold. Each **monsterRoom** contains a string representing the type of monster.
- Write a function **describeRoom** that takes a **DungeonRoom** as an argument and returns a string describing the room.

# Enums: Associated Values

```
enum DungeonRoom {  
    case treasureRoom(Int)  
    case monsterRoom(String)  
    case emptyRoom  
}  
  
func describeRoom(room: DungeonRoom) -> String {  
    switch room {  
    case .treasureRoom(let goldAmount):  
        return "This is a treasure room with \(goldAmount) gold coins."  
    case .monsterRoom(let monsterType):  
        return "This is a monster room with a \(monsterType)."  
    case .emptyRoom:  
        return "This is an empty room."  
    }  
}  
  
// Usage:  
let description = describeRoom(room: .monsterRoom("Dragon"))  
print(description) // Output: "This is a monster room with a Dragon."
```

# Exercise: Direction Enumeration

## Direction Enumeration:

- Create an enumeration called `Direction` with four possible cases: north, south, east, and west.
- Assume a player is at position (0, 0) on a 2D grid. Write a function **`movePlayer`** that takes a `Direction` and a distance as arguments and returns the new position of the player.

# Exercise: Player Action Enumeration

## Player Action Enumeration:

Create an enumeration called `PlayerAction` with cases for different player actions: attack, defend, heal, and escape.

- Each attack action has an associated integer value representing the attack power. Each defend action has an associated integer value representing the defense power.
- Write a function `performAction` that takes a `PlayerAction` as an argument and prints a description of the action performed.

```
enum Direction {
    case north
    case south
    case east
    case west
}

func movePlayer(direction: Direction, distance: Int) -> (Int, Int) {
    var x = 0
    var y = 0

    switch direction {
    case .north:
        y += distance
    case .south:
        y -= distance
    case .east:
        x += distance
    case .west:
        x -= distance
    }

    return (x, y)
}
```



```
enum PlayerAction {  
    case attack(Int)  
    case defend(Int)  
    case heal  
    case escape  
}  
  
func performAction(action: PlayerAction) {  
    switch action {  
    case .attack(let power):  
        print("Player attacks with \(power) power.")  
    case .defend(let defense):  
        print("Player defends with \(defense) defense.")  
    case .heal:  
        print("Player heals.")  
    case .escape:  
        print("Player escapes.")  
    }  
}
```

- Closures are also referred to as **anonymous functions**
  - Closures are **functions without a name**
    - Can take a set of input parameters
    - Can return an output
    - Can be assigned, stored, passed around, and used as input and output to functions

# Closures

- **Syntax:** At its core, a closure is a block of code that can be called in your program, somewhat **similar to functions**. However, they can be written in a more **concise** and **readable** way!
- Here is the basic syntax of a swift closure:

```
{ (parameters) -> return type in  
    statements  
}
```

**Example:**

```
let sumClosure = { (a: Int, b: Int) -> Int in  
    return a + b  
}  
let result = sumClosure(5,8)    // 13
```

In this example, **sumClosure** is a closure that takes two integers and returns the sum of them.

# Closures

```
// 1. No input, no output
let printProfDetails: () -> Void = {
    let name = PersonName(givenName: "Si", middleName: "", familyName: "Chen")
    print(name.fullName())
}
printProfDetails()
```

```
// 2. No input, String output
let createProf: () -> String = {
    let name = PersonName(givenName: "Si", middleName: "", familyName: "Chen")
    return name.fullName()
}
let prof1 = createProf()
print(prof1)
```

```
// 3. Double input, Circle output
let createACircle: (Double) -> Circle = {
    r1 in
    let c = Circle(r: r1)
    return c
}
let c2 = createACircle(100)
print(c2.getCircumference())
```

# Closures

- **Capture Lists:** Closures can "capture", or store, references to variables and constants. This is useful when you want a closure to remember those values and use them later even if the original value has been destroyed or changed.
- Here's how it looks:

```
var number = 5
let incremter = { [number] in
    print(number + 1)
}
number = 6
incremter()
```

Even though number changed after the incremter closure was created, the closure still prints 6 because it captured the original value of number when the closure was created.

# Closures

- **Capture Lists:** Closures can "capture", or store, references to variables and constants. This is useful when you want a closure to remember those values and use them later even if the original value has been destroyed or changed.

```
func makeIncrementer() -> () -> Int {  
    var counter = 0  
    let incrementer: () -> Int = {  
        counter += 1  
        return counter  
    }  
    return incrementer  
}
```

In this case, every call to **makeIncrementer()** would return a new **incrementer** closure that has its own counter capture list.

# Closures

- **Escaping Closures** Closures that are passed as arguments to a function, but are called after the function is done executing are known as **escaping closures**. Escaping closures are written with a **@escaping** annotation.

```
func doSomething(escapingClosure: @escaping () -> Void) {  
    DispatchQueue.main.async {  
        escapingClosure()  
    }  
}
```

In this function, the escapingClosure might be called after doSomething() returns. Hence, it escapes the function's scope.

# Closures

- **Escaping Closures** often use as method parameters
  - Can be really useful when we want to be notified when a long-running task is completed
- Example: We want to save the details of our **Circle** object to a remote database.
  - We may want to be notified when this process has completed.
    - Execute some additional code → Printing a completion message, update UI, ...etc
  - Passing a closure to execute on completion

```
var saveHandler: ((Bool) -> Void)? // closure, take a Bool to indicate whether the save was a success

func saveToRemoteDatabase(handler: @escaping (Bool) -> Void){ // @escaping, a modifier, tells the compiler rather than using the closure within this method,
                                                                // we intend to store the closure and use it later, the closure will be *escaping* the scope of
                                                                this method

    saveHandler = handler
    // send circle object to remote database
    // once remote save is complete, it calls saveComplete(Bool)
    saveComplete(success: true)
}

func saveComplete(success: Bool)
{
    saveHandler?(success)
}
```



# Closures Examples

- Imagine we are developing an iOS App for a restaurant, and we need a piece of code that can handle food orders. A customer can tap on a food item to place an order.
- Now, how would you write this code using closures in Swift?

```
var orderFood: (String) -> Void = { foodItem in  
    print("Order received for \(foodItem)")  
}
```

Now, whenever a customer taps on a food item, you could execute this closure:

```
orderFood("Pizza")
```

In the console, you would then see: "Order received for Pizza".

# Closures Examples

- Imagine we are developing a food delivery app and need a function to calculate the **total cost** of an order.
- This includes the **sum of the prices of each food item, plus a delivery fee**.
  - The sum and fee will vary, so we want to use a closure to perform this calculation.
- We can visualize this as follows:

How to implement this closure?

```
var foodPrices = [10, 15, 20]
var deliveryFee = 5

var totalCost: ([Int], Int) -> Int?
```

Here, we have an array of food prices `foodPrices`, a fixed delivery fee `deliveryFee`, and a closure `totalCost` that takes an array of integers and an integer, and returns an integer.

# Closures Examples

- Here's how we can solve it:

- Step 1: We assign a block of code that performs the necessary calculation to totalCost.

```
totalCost = {  
    prices, fee in  
    var sum = 0  
    for price in prices {  
        sum += price  
    }  
    return sum + fee  
}
```

- Step 2: We can now use this closure to calculate the total cost of an order.

```
let cost = totalCost?(foodPrices, deliveryFee)  
print("The total cost of the order is: \(cost)")
```

# Exercise: Calculating Damage

- Create a closure named **calculateDamage** that takes two arguments: an integer representing the player's attack power, and a float representing the enemy's defense factor (a number between 0 and 1). The closure should return an integer representing the damage dealt to the enemy.

```
let calculateDamage: (Int, Float) -> Int = { attackPower, defenseFactor in  
    // Your code here  
}
```

# Exercise: Checking Level Up

## Checking Level Up:

- Create a closure named **checkLevelUp** that takes two arguments: an integer representing the player's current level, and an integer representing the player's current experience points. Assume 100 points are needed to level up. The closure should return a **boolean** indicating whether the player levels up.

```
let checkLevelUp: (Int, Int) -> Bool = { currentLevel, currentXP in  
    // Your code here  
}
```

# Exercise: High Score Tracker

## High Score Tracker

- You are developing a simple game where players can achieve high scores. You need a way to track the highest score achieved so far even after a new game starts. Create a closure that can update and provide the highest score whenever called.

```
var highestScore = 0

let trackHighScore: (Int) -> Int = { newScore in
    // Your code here to compare and possibly update the highest score
}

// Usage:
// Assume these calls happen at different times during game play
let newHighScore1 = trackHighScore(100) // Highest score so far: 100
let newHighScore2 = trackHighScore(150) // Highest score so far: 150
let newHighScore3 = trackHighScore(120) // Highest score remains 150
```

# JSON

- **JSON** (JavaScript Object Notation) is a lightweight data-interchange format.
  - Other options: XML, YAML,...
- JSON is built on two structures:
  - A collection of name/value pairs. In various languages, this is realized as an *object*, record, struct, dictionary, hash table, keyed list, or associative array.
  - An ordered list of values. In most languages, this is realized as an *array*, vector, list, or sequence.

```
{
  "title": "Design Patterns",
  "subtitle": "Elements of Reusable Object-Oriented Software",
  "author": [
    "Erich Gamma",
    "Richard Helm",
    "Ralph Johnson",
    "John Vlissides"
  ],
  "year": 2009,
  "weight": 1.8,
  "hardcover": true,
  "publisher": {
    "Company": "Pearson Education",
    "Country": "India"
  },
  "website": null
}
```

# How to fetch and parse data from the API with Swift

## ■ 1. Build a data model (based on the structure of the JSON)

```
[ "Names", [ "Bulbasaur", "Ivysaur", "Venusaur", "Charmander", "Charmeleon", "Charizard", "Squirtle", "Wartortle", "Blastoise", "Caterpie", "Metapod", "Butterfree", "Weedle", "Kakuna", "Beedrill", "Pidgey", "Pidgeotto", "Pidgeot", "Rattata", "Raticate", "Spearow", "Fearow", "Ekans", "Arbok", "Pikachu", "Raichu", "Sandshrew", "Sandslash", "Nidoran ♀", "Nidorina", "Nidoqueen", "Nidoran ♂", "Nidorino", "Nidoking", "Clefairy", "Clefable", "Vulpix", "Ninetales", "Jigglypuff", "Wigglytuff", "Zubat", "Golbat", "Oddish", "Gloom", "Vileplume", "Paras", "Parasect", "Venonat", "Venomoth", "Diglett", "Dugtrio", "Meowth", "Persian", "Psyduck", "Golduck", "Mankey", "Primeape", "Growlithe", "Arcanine", "Poliwag", "Poliwhirl", "Poliwrath", "Abra", "Kadabra", "Alakazam", "Machop", "Machoke", "Machop", "Bellsprout", "Weepinbell", "Victreebel", "Tentacool", "Tentacruel", "Geodude", "Graveler", "Golem", "Ponyta", "Rapidash", "Slowpoke", "Slowbro", "Magnemite", "Magnetron", "Farfetch'd", "Doduo", "Dodrio", "Seel", "Dewgong", "Grimer", "Muk", "Shellder", "Cloyster", "Gastly", "Haunter", "Gengar", "Onix", "Drowzee", "Hypno", "Krabby", "Kingler", "Voltorb", "Electrode", "Exeggcute", "Exeggutor", "Cubone", "Marowak", "Hitmonlee", "Hitmonchan", "Lickitung", "Koffing", "Weezing", "Rhyhorn", "Rhydon", "Chansey", "Tangela", "Kangaskhan", "Horsea", "Seadra", "Goldeen", "Seaking", "Staryu", "Starmie", "Mr. Mime", "Scyther", "Jynx", "Electabuzz", "Magmar", "Pinsir", "Tauros", "Magikarp", "Gyarados", "Lapras", "Ditto", "Eevee", "Vaporeon", "Jolteon", "Flareon", "Porygon", "Omanyte", "Omastar", "Kabuto", "Kabutops", "Aerodactyl", "Snorlax", "Articuno", "Zapdos", "Moltres", "Dratini", "Dragonair", "Dragonite", "Mewtwo", "Mew", ] ]
```

```
struct MyModel: Decodable {
    let firstString: String
    let stringArray: [String]

    init(from decoder: Decoder) throws {
        var container = try decoder.unkeyedContainer()
        firstString = try container.decode(String.self)
        stringArray = try container.decode([String].self)
    }
}
```





# How to fetch and parse data from the API with Swift

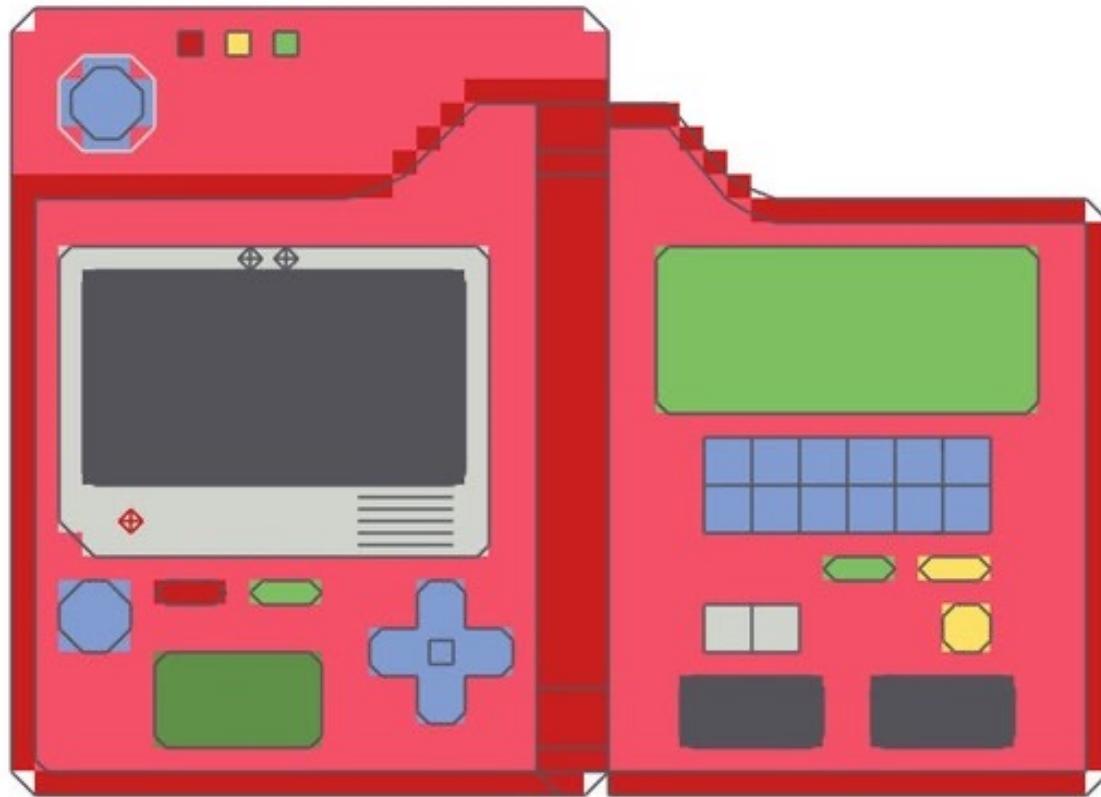
- 2. Create a Class to fetch API Data and decode it based on the model

```
class APIData: ObservableObject{
    var pokemonName: [String] = []
    init(){
        fetchAPIData{ (PokemonData) in
            self.pokemonName = PokemonData.stringArray
        }
    }

    func fetchAPIData(completionHandler: @escaping (MyModel) -> Void){
        let url = URL(string: "https://cs.wcupa.edu/schen/ios2022/pokemon.html")!
        var task = URLSession.shared.dataTask(with: url) {(data, response, error) in
            guard let data = data else { return }
            do{
                let pokemonData = try JSONDecoder().decode(MyModel.self, from: data)
                completionHandler(pokemonData)
            }
            catch{
                let error = error
                print(error.localizedDescription)
            }
        }.resume()
    }
}
```

# Pokédex Version 3

Task: Build the Pokédex Version 3



## Task: Develop a Pokédex Application

### Objective:

Create a user-friendly mobile application to serve as a Pokédex. The app should display crucial attributes of each Pokémon, including a profile picture.

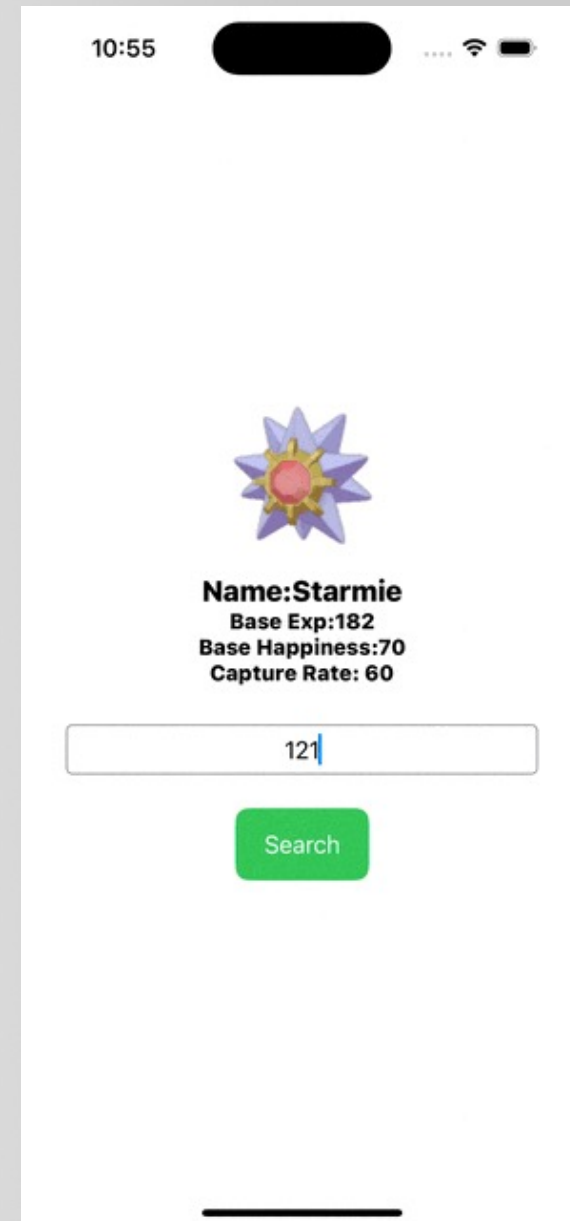
Requirements:

- Core Attributes:** Integrate at least four attributes from the Pokémon API, with "name" being a mandatory field. Other attributes may include weight, height, base experience, etc.

- User Interface:** Develop an intuitive and visually appealing user interface that displays the Pokémon's profile picture alongside its attributes.

- Search Functionality:** Implement a search feature that allows users to input a Pokémon ID and retrieve corresponding information.

- UI Design:** Prioritize aesthetics and user experience. Aim to make the interface polished and visually engaging.



# Q & A

