Extension

Outline

1. Overview
2. Extension Syntax
3. Computed Properties
4. Initializers
5. Methods
6. Subscripts
7. Nested Types

1. Overview

- Extensions add new functionality to an existing class, structure, enumeration, or protocol type. This includes the ability to extend types for which you do not have access to the original source code (known as *retroactive modeling*).

- In Swift, you can even extend a protocol to provide implementations of its requirements or add additional functionality that conforming types can take advantage of.

- Extensions can add new functionality to a type, but they cannot override existing functionality.

1. Overview (cont)

- Extensions in Swift can:
  - ➔ Add computed instance properties and computed type properties
  - ➔ Define instance methods and type methods
  - ➔ Provide new initializers
  - ➔ Define subscripts
  - ➔ Define and use new nested types
  - ➔ Make an existing type conform to a protocol

2. Extension Syntax

- Declare extensions with the **`extension`** keyword:

```
extension SomeType {
    // new functionality to add to SomeType goes here
}
```

- An extension can extend an existing type to make it adopt one or more protocols. To add protocol conformance, you write the protocol names the same way as you write them for a class or structure:

```
extension SomeType: SomeProtocol, AnotherProtocol {
    // implementation of protocol requirements goes here
}
```

3. Computed Properties

- Extensions can add computed instance properties and computed type properties to existing types:

```
extension Double {
    var km: Double { return self * 1_000.0 }
    var m: Double { return self }
    var cm: Double { return self / 100.0 }
    var mm: Double { return self / 1_000.0 }
    var ft: Double { return self / 3.28084 }
}
let oneInch = 25.4.mm
print("One inch is \(oneInch) meters")
// Prints "One inch is 0.0254 meters"
let threeFeet = 3.ft
print("Three feet is \(threeFeet) meters")
// Prints "Three feet is 0.914399970739201 meters"
```

4. Initializers

- Extensions can add new initializers to existing types. This enables you to extend other types to accept your own custom types as initializer parameters, or to provide additional initialization options that were not included as part of the type's original implementation.

```
let rect = Rect(origin: Point(x: 2.0, y: 2.0),
                size: Size(width: 5.0, height: 5.0))

extension Rect {
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size: size)
    }
}

let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
                      size: Size(width: 3.0, height: 3.0))
// centerRect's origin is (2.5, 2.5) and its size is (3.0, 3.0)
```

4. Initializers (cont)

- Extensions can add new convenience initializers to a class, but they cannot add new designated initializers or deinitializers to a class. Designated initializers and deinitializers must always be provided by the original class implementation.

```
let heartImage = UIImage(named: "heart")
let heartImageView = UIImageView(image: heartImage)
heartImageView.translatesAutoresizingMaskIntoConstraints = false
heartImageView.contentMode = .scaleAspectFit

extension UIImageView {
    convenience init?(named name: String, contentMode: UIViewContentMode) {
    guard let image = UIImage(named: name) else {
      return nil
    }

    self.init(image: image)
    self.contentMode = contentMode
    translatesAutoresizingMaskIntoConstraints = false
  }
}

let star = UIImageView(named: "star", contentMode: .scaleAspectFit)
```

4. Initializers (cont)

- If you use an extension to add an initializer to a value type that provides default values for all of its stored properties and does not define any custom initializers, you can call the default initializer and memberwise initializer for that value type from within your extension's initializer.

```
struct Point {
    var x: Double
    var y: Double
}

extension Point {
    init() {
        self.init(x: 0, y: 0)
    }
}

let point = Point()
print("The point is at (\(point.x), \(point.y))")
```

5. Methods

- Extensions can add new instance methods and type methods to existing types:

```
extension Int {
    func repetitions(task: () → Void) {
        for _ in 0..<self {
            task()
        }
    }
}

3.repetitions {
print("Hello!")
}

// Hello!
// Hello!
// Hello!
```

5. Methods (cont)

- Instance methods added with an extension can also modify (or mutate) the instance itself. Structure and enumeration methods that modify **`self`** or its properties must mark the instance method as **`mutating`**, just like mutating methods from an original implementation.

```
extension Int {
    mutating func square() {
        self = self * self
    }
}

var someInt = 3
someInt.square()
print(someInt) // Prints "9"
```

6. Subscripts

- Extensions can add new subscripts to an existing type:

```
extension Int {
    subscript(digitIndex: Int) → Int {
        var decimalBase = 1
        for _ in 0..<digitIndex {
            decimalBase *= 10
        }
        return (self/decimalBase) % 10
    }
}
print(746381295[0]) // Prints "5"
```

7. Nested Types

- Extensions can add new nested types to existing classes, structures, and enumerations:

```swift
extension Int {
    enum Kind {
        case negative, zero, positive
    }

    var kind: Kind {
        switch self {
        case 0:
            return .zero
        case let x where x > 0:
            return .positive
        default:
            return .negative
        }
    }
}
```

```swift
func printIntegerKinds(_ numbers: [Int]) {
    for number in numbers {
        switch number.kind {
        case .negative:
            print("- ", terminator: "")
        case .zero:
            print("0 ", terminator: "")
        case .positive:
            print("+ ", terminator: "")
        }
    }
    print("")
}
printIntegerKinds([3, 19, -27, 0, -6, 0, 7])
// Prints "+ + - 0 - 0 + "
```

Question & Answer?