Конструкции языка, Optional, типы данных в Swift

План

- Типы данных в Swift
- Конструкции языка
- Value & Reference types
- Optional
- Перерыв
- Разберем на практике (темы которые вызовут вопросы)
- + протоколы, extensions, enums, access levels

Категории типов данных

- Named
- Compound

Категории типов данных: Named

- protocol
- struct
- class
- enum

- Bool
- Int & UInt
- Decimal, Float, Double
- String & Character
- Array, Set, Dictionary

Категории типов данных: Compound

- tuple (.., (.., ..))
- function () -> Void

Basics

Constants and Variables

```
let maximumNumberOfLoginAttempts = 10
var currentLoginAttempt = 0
```

```
var x = 0.0, y = 0.0, z = 0.0
```

Naming Constants and Variables

```
let π = 3.14159
let 你好 = "你好世界"
let 🐶🐮 = "dogcow"
```

Type Annotations

```
var welcomeMessage: String
welcomeMessage = "Hello"
```

Printing Constants and Variables

```
print(friendlyWelcome)
// Prints "Bonjour!"
print("The current value of friendlyWelcome is \(friendlyWelcome)")
// Prints "The current value of friendlyWelcome is Bonjour!"
```

Semicolons

```
let cat = "🐱"; print(cat)
// Prints "🐱"
```

Basics

Tuples

```
let http404Error = (404, "Not Found")
// http404Error is of type (Int, String), and equals (404, "Not Found")
```

```
print("The status code is \(http404Error.0)")
// Prints "The status code is 404"
print("The status message is \(http404Error.1)")
// Prints "The status message is Not Found"
```

```
var someTuple = (top: 10, bottom: 12)  // someTuple is of type (top: Int,
  bottom: Int)
someTuple = (top: 4, bottom: 42) // OK: names match
someTuple = (9, 99)              // OK: names are inferred
someTuple = (left: 5, right: 5)  // Error: names don't match
```

Type Aliases

```
typealias AudioSample = UInt16
```

```
typealias Point = (Int, Int)
let origin: Point = (0, 0)
```

Optionals

```
var surveyAnswer: String?
// surveyAnswer is automatically set to nil
```

```
var optionalInteger: Int?
var optionalInteger: Optional<Int>
```

```
var serverResponseCode: Int? = 404
// serverResponseCode contains an actual Int value of 404
serverResponseCode = nil
// serverResponseCode now contains no value
```

```
let possibleNumber = "123"
let convertedNumber = Int(possibleNumber)
// convertedNumber is inferred to be of type "Int?", or "optional Int"
```

Basics

Optionals

```
if convertedNumber != nil {
    print("convertedNumber has an integer value of \(convertedNumber!).")
}
// Prints "convertedNumber has an integer value of 123."
```

```
optionalInteger = 42
optionalInteger! // 42
```

Optional Binding

```
if let constantName = someOptional {
    statements
}
```

```
if let actualNumber = Int(possibleNumber) {
    print("The string \"\(possibleNumber)\" has an integer value of \
  (actualNumber)")
} else {
    print("The string \"\(possibleNumber)\" could not be converted to an
  integer")
}
// Prints "The string "123" has an integer value of 123"
```

Nil coalescing

```
let defaultColorName = "red"
var userDefinedColorName: String?   // defaults to nil

var colorNameToUse = userDefinedColorName ?? defaultColorName
```

Basics

Implicitly Unwrapped Optionals

```
let possibleString: String? = "An optional string."
let forcedString: String = possibleString! // requires an exclamation mark

let assumedString: String! = "An implicitly unwrapped optional string."
let implicitString: String = assumedString // no need for an exclamation
  mark
```

Functions

Defining and Calling Functions

```
func greet(person: String) -> String {
    let greeting = "Hello, " + person + "!"
    return greeting
}
```

```
print(greet(person: "Anna"))
// Prints "Hello, Anna!"
print(greet(person: "Brian"))
// Prints "Hello, Brian!"
```

Functions with Multiple Return Values

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
```

Functions

Functions Argument Labels and Parameter Names

```
func someFunction(firstParameterName: Int, secondParameterName: Int) {
    // In the function body, firstParameterName and secondParameterName
    // refer to the argument values for the first and second parameters.
}
someFunction(firstParameterName: 1, secondParameterName: 2)
```

```
func someFunction(argumentLabel parameterName: Int) {
    // In the function body, parameterName refers to the argument value
    // for that parameter.
}
```

Omitting Argument Labels

```
func someFunction(_ firstParameterName: Int, secondParameterName: Int) {
    // In the function body, firstParameterName and secondParameterName
    // refer to the argument values for the first and second parameters.
}
someFunction(1, secondParameterName: 2)
```

Default Parameter Values

```
func someFunction(argumentLabel parameterName: Int = 12) {
    // In the function body, parameterName refers to the argument value
    // for that parameter.
}
```

Functions

In-Out Parameters

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// Prints "someInt is now 107, and anotherInt is now 3"
```

Function Types

```
func addTwoInts(_ a: Int, _ b: Int) -> Int {
    return a + b
}
```

```
var mathFunction: (Int, Int) -> Int = addTwoInts
```

Closures

Closure Expression Syntax

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

```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

Inferring Type From Context

```
reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 } )
```

Implicit Returns from Single-Expression Closures

```
reversedNames = names.sorted(by: { s1, s2 in s1 > s2 } )
```

Shorthand Argument Names

```
reversedNames = names.sorted(by: >)
```

Trailing Closures

```
reversedNames = names.sorted() { $0 > $1 }
```

```
reversedNames = names.sorted { $0 > $1 }
```

Enums

Enumeration Syntax

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

```
var directionToHead = CompassPoint.west
```

```
directionToHead = .east
```

Matching Enumeration Values with a Switch Statement

```
directionToHead = .south
switch directionToHead {
case .north:
    print("Lots of planets have a north")
case .south:
    print("Watch out for penguins")
case .east:
    print("Where the sun rises")
case .west:
    print("Where the skies are blue")
}
// Prints "Watch out for penguins"
```

Matching Enumeration Values with a Switch Statement

```
let somePlanet = Planet.earth
switch somePlanet {
case .earth:
    print("Mostly harmless")
default:
    print("Not a safe place for humans")
}
// Prints "Mostly harmless"
```

Iterating over Enumeration Cases

```
enum Beverage: CaseIterable {
    case coffee, tea, juice
}
let numberOfChoices = Beverage.allCases.count
print("\(numberOfChoices) beverages available")
// Prints "3 beverages available"
```

Raw Values

```
enum ASCIIControlCharacter: Character {
    case tab = "\t"
    case lineFeed = "\n"
    case carriageReturn = "\r"
}
```

Structures and Classes

Definition Syntax

```
struct Resolution {
    var width = 0
    var height = 0
}
class VideoMode {
    var resolution = Resolution()
    var interlaced = false
    var frameRate = 0.0
    var name: String?
}
```

Structure and Class Instances

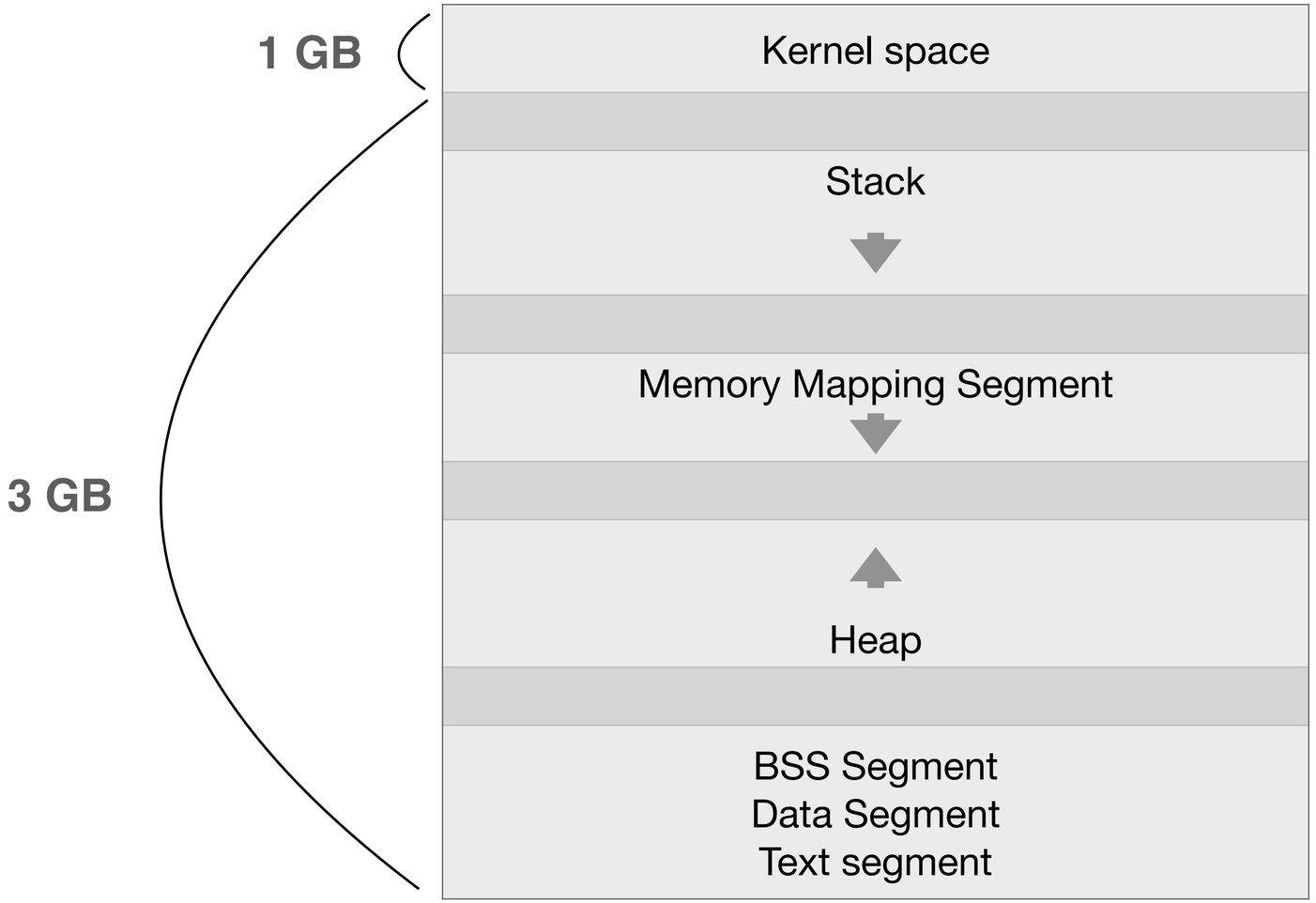
```
let someResolution = Resolution()
let someVideoMode = VideoMode()
```

```
someVideoMode.resolution.width = 1280
print("The width of someVideoMode is now \
  (someVideoMode.resolution.width)")
// Prints "The width of someVideoMode is now 1280"
```

Memberwise Initializers for Structure Types

```
let vga = Resolution(width: 640, height: 480)
```

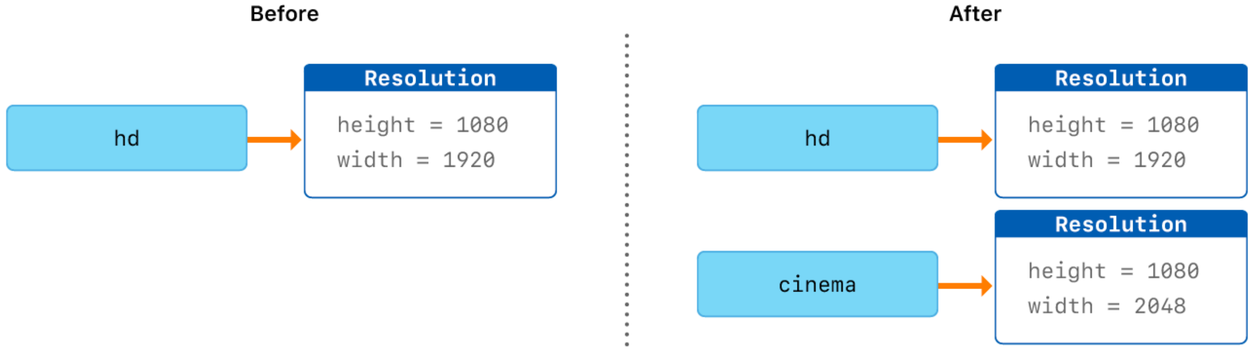
Virtual Address Space



Structures and Classes

Structures and Enumerations Are Value Types

```
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd
```



Classes Are Reference Types

```
let tenEighty = VideoMode()
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25.0
```

```
let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30.0
```

Structures and Classes

- Наследование
- Осуществление вызовов методов (method dispatch)
- Расположение в виртуальном адресном пространстве
- Копирование при новом инстансе (inout, mutating)
- Copy on Write (для Swift foundation Array)
- Иммутабельность структур

Structures and Classes

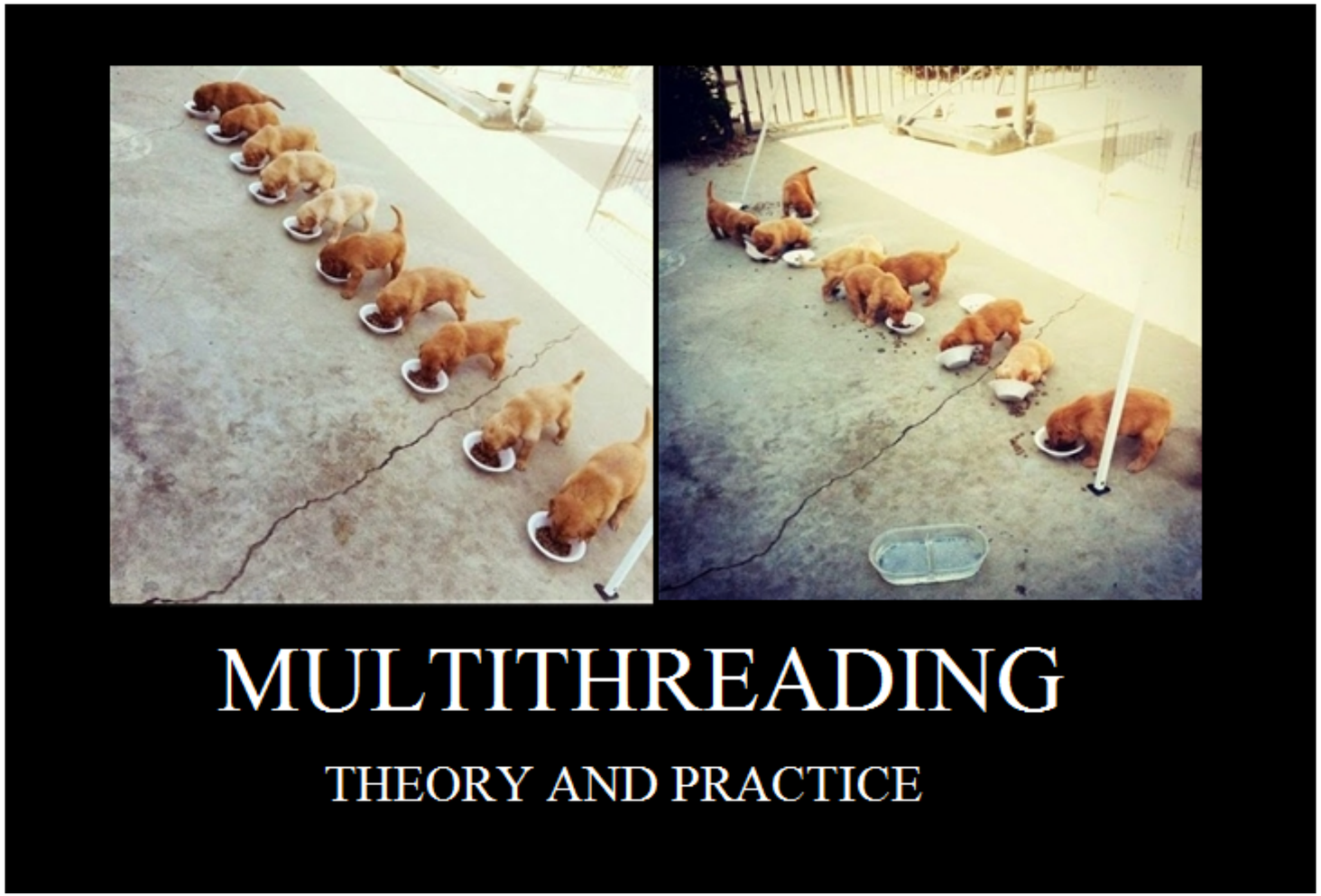
Использовать value тип когда:

- Сравнение сущностей через == имеет смысл
- Копии должны иметь независимое состояние
- Данные будут использованы между несколькими потоками

Использовать reference тип когда:

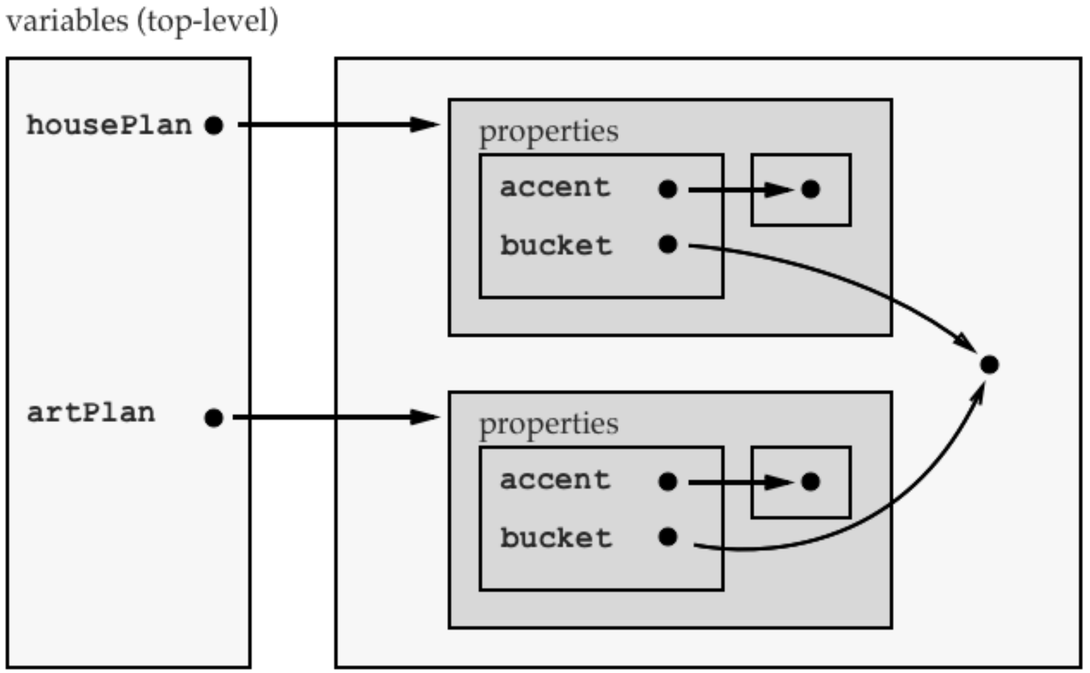
- Сравнение сущностей через === имеет смысл
- Sharing состояний приветствуется

Structures and thread safety



Value types with Reference semantic

- Избегайте использование таких типов, если они не поддерживают COW
- Обманчивы и могут послужить огромным плацдармом ошибок для неопытного программиста



Properties

Stored Properties

```
struct FixedLengthRange {
    var firstValue: Int
    let length: Int
}
var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3)
// the range represents integer values 0, 1, and 2
rangeOfThreeItems.firstValue = 6
// the range now represents integer values 6, 7, and 8
```

Stored Properties of Constant Structure Instances

```
let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4)
// this range represents integer values 0, 1, 2, and 3
rangeOfFourItems.firstValue = 6
// this will report an error, even though firstValue is a variable property
```

Lazy Stored Properties

```
class DataManager {
    lazy var importer = DataImporter()
    var data = [String]()
    // the DataManager class would provide data management functionality
  here
}
```

Computed Properties

```
struct Cuboid {
    var width = 0.0, height = 0.0, depth = 0.0
    var volume: Double {
        return width * height * depth
    }
}
let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0, depth: 2.0)
print("the volume of fourByFiveByTwo is \(fourByFiveByTwo.volume)")
// Prints "the volume of fourByFiveByTwo is 40.0"
```

Property Observers

```
class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            print("About to set totalSteps to \(newTotalSteps)")
        }
        didSet {
            if totalSteps > oldValue  {
                print("Added \(totalSteps - oldValue) steps")
            }
        }
    }
}
let stepCounter = StepCounter()
stepCounter.totalSteps = 200
// About to set totalSteps to 200
// Added 200 steps
```

Property wrappers

```
@propertyWrapper
struct TwelveOrLess {
    private var number: Int
    init() { self.number = 0 }
    var wrappedValue: Int {
        get { return number }
        set { number = min(newValue, 12) }
    }
}
```

```
struct SmallRectangle {
    @TwelveOrLess var height: Int
    @TwelveOrLess var width: Int
}

var rectangle = SmallRectangle()
print(rectangle.height)
// Prints "0"

rectangle.height = 10
print(rectangle.height)
// Prints "10"

rectangle.height = 24
print(rectangle.height)
// Prints "12"
```

Property wrappers

- Реализовать propertyWrapper, который предназначен для String, и при присваивании нового значения конкатенирует его со старым, добавляя пробел

Пример данных:

@StringConcatenation var name: String

name = «Alex»

name = «Magnusson»

print(name) -> Alex Magnusson

Protocols

Protocol Syntax

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

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

```
class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol {
    // class definition goes here
}
```

Property and Method Requirements

```
protocol SomeProtocol {
    var mustBeSettable: Int { get set }
    var doesNotNeedToBeSettable: Int { get }
}
```

```
protocol RandomNumberGenerator {
    func random() -> Double
}
```

```
protocol Togglable {
    mutating func toggle()
}
```

Class-Only Protocols

```
protocol SomeClassOnlyProtocol: AnyObject, SomeInheritedProtocol {
    // class-only protocol definition goes here
}
```

Optional Protocol Requirements

```
@objc protocol CounterDataSource {
    @objc optional func increment(forCount count: Int) -> Int
    @objc optional var fixedIncrement: Int { get }
}
```

```
class Counter {
    var count = 0
    var dataSource: CounterDataSource?
    func increment() {
        if let amount = dataSource?.increment?(forCount: count) {
            count += amount
        } else if let amount = dataSource?.fixedIncrement {
            count += amount
        }
    }
}
```

Extensions

Extension Syntax

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

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

Computed Properties

```
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"
```

Mutating Instance Methods

```
extension Int {
    mutating func square() {
        self = self * self
    }
}
var someInt = 3
someInt.square()
// someInt is now 9
```

Providing Default Protocol Implementations

```
extension PrettyTextRepresentable  {
    var prettyTextualDescription: String {
        return textualDescription
    }
}
```

Optional

- Зачем нужен Optional?

```
@frozen public enum Optional<Wrapped> : ExpressibleByNilLiteral {

    /// The absence of a value.
    ///
    /// In code, the absence of a value is typically written using the `nil`
    /// literal rather than the explicit `.none` enumeration case.
    case none

    /// The presence of a value, stored as `Wrapped`.
    case some(Wrapped)
```

- Обеспечивает безопасную работу с переменными в момент компиляции программы. Вместо проверки на nil (null pointer), разработчик должен извлечь данные из Optional value, если он это не сделает программа не скомпилируется, в отличие от Runtime exception.

Access Control

- open
- public
- internal
- fileprivate
- private

```
open class User {
    open func login() { }
    public func playGame() { }
    public init() { }
}
```

Access Control

- Практика: понимаем разницу на уровне таргетов

Что почитать

- Типы - Документация Apple
- struct vs class
- Apple Docs

Вопросы

- Зачем нужен Optional?
- В чем отличие struct от class?
- Зачем нужны property observers?
- Что такое Copy on Write?

Домашнее задание

- https://github.com/apple/swift/blob/main/stdlib/public/core/Optional.swift - К ознакомлению
- Реализовать COW в своей структуре со свойством reference type (isKnownUniquelyReferenced)
- До 30 июня включительно