Property

강사 주영민

프로퍼티

- 변수의 다른 이름
- 클래스, 구조체, 열거형등 전체적으로 사용되는 변수를 프로퍼티라고 부른다.

프로퍼티의 종류

- 저장 프로퍼티 (Stored Properties)
- 연산 프로퍼티 (Computed Properties)
- 타입 프로퍼티 (TypeProperties)

저장 프로퍼티(Stored Properties)

- 가장 일반적인 프로퍼티
- 값을 저장하는 용도로 사용된다.
- 클래스, 구조체에서만 인스턴스와 연관된 값을 저장한다.
- 초기값을 설정 할 수 있습니다.

저장 프로퍼티(Stored Properties)

```
struct FixedLengthRange {
    var firstValue: Int
    let length: Int
}
```

```
var rangeOfThreeItems:FixedLengthRange
            = FixedLengthRange(firstValue: 0, length: 3)
```

지연 저장 프로퍼티 (Lazy Stored Properties)

- 지연 저장된 속성은 처음 프로퍼티가 사용되기 전 까지 초기값이 계산되지 않은 특성을 가지고 있는 프로퍼티이다.

- 지연 저장 속성은 **lazy** keyword를 선언 앞에 작성한다.

- let은 지연 저장 프로퍼티로 설정할 수 없다.

- 초기화하는데 오래걸리거나, 복잡한 초기화 과정이 있는 변수의 경우 지연저장을 사용하면 좋다 .

연산 프로퍼티(Computed Properties)

- 실제로 값을 저장하지 않지만, get, set키워드를 통해서 값을 간접적으로 설정하거나 받을 수 있다.
- 클래스, 구조체, 열거형에서 사용할수 있는 프로퍼티이다.

연산 프로퍼티 예제

```
struct Point {
    var x = 0.0, y = 0.0
}
struct Size {
    var width = 0.0, height = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = origin.x + (size.width / 2)
            let centerY = origin.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set(newCenter) {
            origin.x = newCenter.x - (size.width / 2)
            origin.y = newCenter.y - (size.height / 2)
        }
    }
}
```

연산 프로퍼티 예제

- 연산프로퍼티 자신을 바로 사용하면 어떻게 될까요?

```
struct Rect {
    //var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = center.x + (size.width / 2)
            let centerY = center.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set(newCenter) {
            center = newCenter
        }
    }
}
```

Setter ValueName 미 지정

- set의 값이름 미지정시 newValue 가 기본 값으로 사용된다.

```
struct Rect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = origin.x + (size.width / 2)
            let centerY = origin.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set {
            origin.x = newValue.x - (size.width / 2)
            origin.y = newValue.y - (size.height / 2)
        }
    }
}
```

Read Only 연산 프로퍼티

- 읽기 전용 연산프로퍼티 작성시 get 키워드 없이 바로 작성할수 있다.

- 쓰기 전용 연산 프로퍼티는 작성할수 없다.

```
struct Cuboid {
    var width = 0.0, height = 0.0, depth = 0.0
    var volume: Double {
        return width * height * depth
    }
}
```

Property Observers

- 프로퍼티 값의 변화를 감시하고 그에 따라 대응한다.
- 초기값이 설정된 저장 프로퍼티에서 사용 가능하다. .
- 프로퍼티의 값이 설정될때마다 호출된다.
- didSet, willSet 키워드를 통해 값 변화의 직전 직후를 감지 할수 있다.
- 값이름 미지정시 oldValue, newValue가 기본값으로 지정된다.

Property Observers 예제

```
var changeValue:Int = 0{
    didSet(oldV){
        print("oldValue \(oldV)")
    }
    willSet(willV)
    {
        print("newValue \(willV)")
    }
}

changeValue = 4
```

타입 프로퍼티(Type Properties)

- 인스턴스의 속성이 아닌, 타입에 따른 속성을 정의 할수 있다.
- static 키워드를 사용해서 값타입에서 타입 프로퍼티를 설정할수 있으며, class 키워드를 사용해서 클래스 타입에서 타입 프로퍼티를 설정할 수 있다.
- 값을 가져올때는 클래스의 이름을 통해서 가져올 수 있다.

타입 프로퍼티 예제

```
struct AudioChannel {
    static let level = 10
    static var maxLevel = 0
    var currentLevel: Int = 0 {
        didSet {
            if currentLevel > AudioChannel.level
            {
                currentLevel = AudioChannel.level
            }
            if currentLevel > AudioChannel.maxLevel
            {
                AudioChannel.maxLevel = currentLevel
            }
        }
    }
}
```

Method

- 메서드는 특정 타입에 관련된 함수를 뜻합니다.
- 함수의 문법과 같다.
- 인스턴스의 기능을 수행하는 인스턴스 메서드와 타입자체의 기능을 수행하는 타입 메서드로 나눌수 있습니다.

self Property

- 모든 인스턴스는 self 프로퍼티를 가지고 있다.
- self프로퍼티는 자기 자신을 가르키고 있는 프로퍼티 이다.
- 이를 사용해서 인스턴스 메소드 안에서 자기 인스턴스에 접근 할수 있다.

```
struct Point {
    var x = 0.0, y = 0.0
    func isToTheRightOf(x: Double) -> Bool {
        return self.x > x
    }
}
```

Value Type 프로퍼티 수정

- 기본적으로 구조체와 열거형의 값타입 프로퍼티는 인스턴스 메소드 내에서 수정이 불가능 하다.

- 그러나 특정 메소드에서 수정을 해야 할경우에는 mutating 키워드를 통해 변경 가능하다.

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveBy(x deltaX: Double, y deltaY: Double) {
        x += deltaX
        y += deltaY
    }
}
```

타입 메서드

- 타입 프로퍼티랑 마찬가지로 타입 자체에서 호출이 가능한 메서드.

- 메서드 앞에 static키워드를 사용하여 타입메서드를 작성할수 있다. 타입 프로퍼티와 마찬가지로 클래스에서는 class키워드를 사용해 타입메서드를 표현한다.

- 타입 메소드 안에서의 self는 인스턴스가 아닌 타입을 나타낸다.

열거형

강사 주영민

Fast campus

열거형(enumeration)

- 그룹에 대한 연관된 값을 정의하고 사용가능한 타입
- 다른 언어와 달리 항목 그자체가 고유의 값으로 해당 항목에 값을 매칭 시킬 필요가 없다.(C계열 언어는 Int타입의 값이 매칭됨
- 원시값(rawValue)이라는 형태로 실제 값(정수, 실수, 문자등)을 부여 할수 있다.
- 열거형의 이니셜라이즈를 정의 할수 있으며, 프로토콜 준수, 연산프로퍼티, 메소드등을 만들수 있습니다.

열거형 문법

```
enum <열거형 이름> {

    case <열거 항목1>
    case <열거 항목2>
    case <열거 항목3>
}
```

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

```
enum Planet {
    case mercury, venus, earth,
         mars, jupiter, saturn,
         uranus, neptune
}
```

열거형 값 지정

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

*각 case값만 들어 갈수 있으며, 선언 후 점(.)문법을 통해 쉽게 다른 값을 설정 할수 있다.

Switch문 사용

```
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")
}
```

*열거형 모든 case가 제공될때 default값은 제공될 필요가 없다.

Associated Values

```
enum Barcode {
    case upc(Int, Int, Int, Int)
    case qrCode(String)
}
```

Associated Values

```
enum Barcode {
    case upc(Int, Int, Int, Int)
    case qrCode(String)
}
```

<연관 열거형 값지정>

```
var productBarcode = Barcode.upc(8, 85909, 51226, 3)

productBarcode = .qrCode("ABCDEFGHIJKLMNOP")
```

Associated Values

```
enum Barcode {
    case upc(Int, Int, Int, Int)
    case qrCode(String)
}
```

<연관 열거형 값 불러오기>

```
switch productBarcode {
    case .upc(let numberSystem, let manufacturer, let product, let check):
        print("UPC: \(numberSystem), \(manufacturer), \(product), \(check).”)

    case .qrCode(let productCode):
        print("QR code: \(productCode).")
}
```

Associated Values

```
enum Barcode {
    case upc(Int, Int, Int, Int)
    case qrCode(String)
}
```

<Pattern Matching>

```
let productBarcode = Barcode.upc(8, 85909, 51226, 3)

if case let Barcode.upc(8, companyCode, productCode, 3) =
productBarcode
{
    //정상 바코드
    print(companyCode)
    print(productCode)
}
```

Raw Values

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

```
enum Planet: Int{
    case mercury=1, venus, earth,
    mars, jupiter, saturn,
    uranus, neptune
}
```

```
enum CompassPoint: String {
    case north, south, east, west
}
```

Raw Values

- .rawValue 프로퍼티를 통해 원시값을 가져올수 있다.

```
let earthsOrder = Planet.earth.rawValue
// earthsOrder is 3

let sunsetDirection = CompassPoint.west.rawValue
// sunsetDirection is "west"
```

Initializing from a Raw Value

- 원시값 열거형에서는 초기화 함수를 통해 instance를 만들수 있다. (rawValue:값 지정으로 인해 생성)

- 초기화를 통해 만든 인스턴스는 옵션널 변수로 만들어 진다.

```
enum Planet: Int{
    case mercury=1, venus, earth,
    mars, jupiter, saturn,
    uranus, neptune
}
```

```
let possiblePlanet:Planet? = Planet(rawValue: 1)
```

Recursive Enumerations

- 재귀열거형은 다른 인스턴스 열거형이 Associated Values로 사용되는 열거형이다.
- indirect 키워드를 통해 순환 열거형을 명시할수 있으며, 특정 항목만 사용시 case 앞에, 열거형 전체에 사용될 때는 enum 키워드 앞에 붙이면 된다.

```
enum ArithmeticExpression {
    case number(Int)
    indirect case addition(ArithmeticExpression,
                           ArithmeticExpression)
    indirect case multiplication(ArithmeticExpression,
                           ArithmeticExpression)
}

indirect enum ArithmeticExpression {
    case number(Int)
    case addition(ArithmeticExpression, ArithmeticExpression)
    case multiplication(ArithmeticExpression,
                           ArithmeticExpression)
}
```

Recursive Enumerations 예제

(5 + 4) * 2.

```
let five = ArithmeticExpression.number(5)
let four = ArithmeticExpression.number(4)

let sum = ArithmeticExpression.addition(five, four)
let product = ArithmeticExpression.multiplication(sum,
ArithmeticExpression.number(2))
```

Recursive Enumerations 예제

```
let five = ArithmeticExpression.number(5)
let four = ArithmeticExpression.number(4)

let sum = ArithmeticExpression.addition(five, four)
let product = ArithmeticExpression.multiplication(sum,
ArithmeticExpression.number(2))

func evaluate(_ expression: ArithmeticExpression) -> Int {
    switch expression {
    case let .number(value):
        return value
    case let .addition(left, right):
        return evaluate(left) + evaluate(right)
    case let .multiplication(left, right):
        return evaluate(left) * evaluate(right)
    }
}

print(evaluate(product))
```

다양한 예제

```
//기본 연관 값 열거형
enum KqueueEvent {
    case UserEvent(identifier: UInt, fflags: [UInt32], data: Int)
    case ReadFD(fd: UInt, data: Int)
    case WriteFD(fd: UInt, data: Int)
    case VnodeFD(fd: UInt, fflags: [UInt32], data: Int)
    case ErrorEvent(code: UInt, message: String)
}
//중첩 열거형
enum Wearable {
    enum Weight: Int {
        case Light = 1
        case Mid = 4
        case Heavy = 10
    }
    enum Armor: Int {
        case Light = 2
        case Strong = 8
        case Heavy = 20
    }
    case Helmet(weight: Weight, armor: Armor)
    case Breastplate(weight: Weight, armor: Armor)
    case Shield(weight: Weight, armor: Armor)
}
```

다양한 예제 - 함수

```
enum Wearable {
    enum Weight: Int {
        case Light = 1
    }
    enum Armor: Int {
        case Light = 2
    }
    case Helmet(weight: Weight, armor: Armor)
    func attributes() -> (weight: Int, armor: Int) {
        switch self {
        case .Helmet(let w, let a):
            return (weight: w.rawValue * 2,
                    armor: a.rawValue * 4)
        }
    }
}
let woodenHelmetProps = Wearable.Helmet(weight: .Light,
                                        armor: .Light).attributes()
print (woodenHelmetProps)
```

다양한 예제 - 함수

```
enum Device {
    case iPad, iPhone, AppleTV, AppleWatch
    func introduced() -> String {
        switch self {
        case .AppleTV:
            return "\(self) was introduced 2006"
        case .iPhone:
            return "\(self) was introduced 2007"
        case .iPad:
            return "\(self) was introduced 2010"
        case .AppleWatch:
            return "\(self) was introduced 2014"
        }
    }
}
print (Device.iPhone.introduced())
```

다양한 예제 - 연산프로퍼티

```
enum Device {
    case iPad, iPhone
    var year: Int {
      switch self {
      case .iPhone:
          return 2007
      case .iPad:
          return 2010
      }
   }
}

print (Device.iPhone.year)
```

에러처리

강사 주영민

예외처리

- 프로그램의 오류 조건에 응답하고 오류 조건에서 복구하는 프로세스입니다

- Swift는 런타임시 복구 가능한 오류를 던지고, 포착하고, 전파하고, 조작하는 기능을 제공합니다.

- 에러는 Error 프로토콜을 준수하는 유형의 값으로 나타냅니다. 실제로 Error프로토콜은 비어 있으나 오류를 처리할수 있는 타입임을 나타냅니다.

열거형의 에러 표현

- 열거형은 에러를 표현하는데 적합합니다.

```
enum VendingMachineError: Error {
    case invalidSelection
    case insufficientFunds(coinsNeeded: Int)
    case outOfStock
}
```

에러발생

- throw 키워드를 통해 에러를 발생 시킵니다.

```
throw VendingMachineError.insufficientFunds(coinsNeeded: 5)
```

에러전달

- 함수의 작성중 에러가 발생할수 있는 함수에는 매개변수 뒤에 throws 키워드를 작성하여 에러가 전달될수 있는 함수를 선언합니다.

```
//에러전달 가능성 함수
func canThrowErrors() throws -> String

//에러전달 가능성이 없는 함수
func cannotThrowErrors() -> String
```

에러처리

- 함수가 에러를 throw하면 프로그램의 흐름이 변경되므로 에러가 발생할 수있는 코드의 위치를 신속하게 식별 할 수 있어야합니다.

- 이 장소를 식별하기 위해 try 나 try?, try!를 사용할수 있습니다.

- 발결된 에러를 처리하기 위해 do-catch 문을 사용해서 에러를 처리 합니다.

do - catch

```
do {
    try expression
    statements
} catch pattern 1 {
    statements
} catch pattern 2 where condition {
    statements
}
```

예제

```
let favoriteSnacks = [
    "Alice": "Chips",
    "Bob": "Licorice",
    "Eve": "Pretzels",
]


func buyFavoriteSnack(person: String, vendingMachine: VendingMachine) throws {
    let snackName = favoriteSnacks[person] ?? "Candy Bar"
    try vendingMachine.vend(itemNamed: snackName)
}
```

```
var vendingMachine = VendingMachine()
vendingMachine.coinsDeposited = 8

do {
    try buyFavoriteSnack(person: "Alice", vendingMachine: vendingMachine)
} catch VendingMachineError.invalidSelection {
    print("Invalid Selection.")
} catch VendingMachineError.outOfStock {
    print("Out of Stock.")
} catch VendingMachineError.insufficientFunds(let coinsNeeded) {
    print("Insufficient funds. Please insert an additional \(coinsNeeded) coins.")
}
```

예제

```
class VendingMachine {
    var inventory = [
        "Candy Bar": Item(price: 12, count: 7),
        "Chips": Item(price: 10, count: 4),
        "Pretzels": Item(price: 7, count: 11)
    ]
    var coinsDeposited = 0

    func vend(itemNamed name: String) throws {
        guard let item = inventory[name] else {
            throw VendingMachineError.invalidSelection
        }
        guard item.count > 0 else {
            throw VendingMachineError.outOfStock
        }

        guard item.price <= coinsDeposited else {
            throw VendingMachineError.insufficientFunds(coinsNeeded: item.price - coinsDeposited)
        }

        coinsDeposited -= item.price

        var newItem = item
        newItem.count -= 1
        inventory[name] = newItem

        print("Dispensing \(name)")
    }
}
```

```
struct Item {
    var price: Int
    var count: Int
}

enum VendingMachineError: Error {
    case invalidSelection
    case
insufficientFunds(coinsNeeded: Int)
    case outOfStock
}
```

Converting Errors to Optional Value

```swift
func someThrowingFunction() throws -> Int {
    // ...
}

let x = try? someThrowingFunction()

let y: Int?
do {
    y = try someThrowingFunction()
} catch {
    y = nil
}
```

Specifying Cleanup Actions (후처리)

- 에러에 의해 함수의 문제가 생기더라도 꼭! 해야할 행동이 있다면!!
- defer 구문은 블럭이 어떻게 종료되던 꼭 실행된다는 것을 보장.

```
func processFile(filename: String) throws {
    if exists(filename) {
        let file = open(filename)
        defer {
            close(file)
        }

        while let line = try file.readline() {
            // Work with the file.
        }
        // close(file) is called here, at the end of the scope.
    }
}
```