코드 확장

Subscript

Subscript

- 클래스, 구조체, 열거형의 collection, list, sequence의 멤버에 접근 가능한 단축문법인 Subscript를 정의 할수 있다.
- Subscript는 별도의 setter/getter없이 index를 통해서 데이터를 설정하거나 값을 가져오는 기능을 할 수 있다.
- Array[index] / Dictionary[“Key”] 등의 표현이 Subscript이다.

문법

```
subscript(index: Type) -> Type {
    get {
        // return an appropriate subscript value here
    }
    set(newValue) {
        // perform a suitable setting action here
    }
}
```

.....

```
subscript(index: Type) -> Type {
    // return an appropriate subscript value here
}
```

*연산 프로퍼티와 문법이 같음

예제 - Array

```
class Friends {
    private var friendNames:[String] = []

    subscript(index:Int) -> String
    {
        get {
            return friendNames[index]
        }
        set {
            friendNames[index] = newValue
        }

    }
}


let fList = Friends()
fList[0] = "joo"
```

예제 - struct

```
struct TimesTable {
    let multiplier: Int
    subscript(index: Int) -> Int {
        return multiplier * index
    }
}
```

```
let threeTimesTable = TimesTable(multiplier: 3)
print("six times three is \(threeTimesTable[6])")
```

예제 - 다중 parameter

```
struct Matrix {
    let rows: Int, columns: Int
    var grid: [Double]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        grid = Array(repeating: 0.0, count: rows * columns)
    }

    subscript(row: Int, column: Int) -> Double {
        get {
            return grid[(row * columns) + column]
        }
        set {
            grid[(row * columns) + column] = newValue
        }
    }
}

var metrix = Matrix(rows: 2, columns: 2)
metrix[0,0] = 1
metrix[0,1] = 2.5
```

Extension

Extensions

- Extensions 기능은 기존 클래스, 구조, 열거 형 또는 프로토콜 유형에 새로운 기능을 추가합니다
- Extensions으로 할수 있는것은…

1. Add computed instance properties and computed type properties
2. Define instance methods and type methods
3. Provide new initializers
4. Define subscripts
5. Define and use new nested types
6. Make an existing type conform to a protocol

문법

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

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

유형 : Compute 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"
```

유형 : init

```
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)
    }
}
```

유형 : method

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


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

유형 : mutating method

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

var someInt = 3
someInt.square()
```

유형 : Subscript

```
extension Int {
    subscript(digitIndex: Int) -> Int {
        var decimalBase = 1
        for _ in 0..<digitIndex {
            decimalBase *= 10
        }
        return (self / decimalBase) % 10
    }
}

746381295[0]
// returns 5
746381295[1]
// returns 9
746381295[2]
// returns 2
746381295[8]
// returns 7
```

유형 : Nested Types

```
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
        }
    }
}
```

Generic

Generic

- 어떤 타입에도 유연한 코드를 구현하기 위해 사용되는 기능
- 코드의 중복을 줄이고, 깔끔하고 추상적인 표현이 가능하다.

왜 Generic을 사용하는가?

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

두 Int를 받아 서로 바꿔주는 스왑함수가 있다.
우리는 Int 외에도 Double, String 등 다양한 타입의 데이터를 스왑하고 싶다면 어떻게 해야될까?

Generic을 사용한 swap함수

```
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

```
var someInt = 3
var anotherInt = 107
swapTwoValues(&someInt, &anotherInt)
// someInt is now 107, and anotherInt is now 3

var someString = "hello"
var anotherString = "world"
swapTwoValues(&someString, &anotherString)
// someString is now "world", and anotherString is now "hello"
```

Framework확인

- Array / Dictionary 파일 확인하기

Type Parameters

- 제넥릭에 사용된 “T”는 타입의 이름으로 사용되었다기 보다는 placeholder 역할로 사용되었다.

- 타입은 꺾쇠<> 로 감싸 표시한다.

- 타입의 이름은 보통 사용되는 속성에 맞게 네이밍 할수 있으나 아무런 연관이 없는 타입의 경우에는 T,U,V 같은 알파벳으로 사용된다.

Generic만들기 : Stack