Learning Swift Language on Ubuntu:
Functions
Closures
Enumeration
Structures and Classes
Properties

implementation

```
// functions 関数
// https://docs.swift.org/swift-book/LanguageGuide/Functions.html

// すべての関数には型がある。関数はネストできる。
// すべての関数には名前がある。

func func0(x0: Int, x1: Int) {
    print(x0 * x1)
}
func0(x0: 2, x1: 3)
// func0(x1: 2, x0: 3)  // 引数の順番を守らないと compile-time error。

// 文字列を取り文字列を返す関数を定義する。
func greet(person: String) -> String {
    let greeting = "Hello, " + person + "!"
    return greeting
}
// argument label を用いて引数を与える。
print(greet(person: "Anna"))
print(greet(person: "Brian"))

func greetAgain(person: String) -> String {
```

```
    return "Hello again, " + person + "!"  // 一旦定数に束縛することをせずに return する。
}
print(greetAgain(person: "Anna"))

// パラメータのない関数。
func sayHelloWorld() -> String {
    return "hello, world"
}
print(sayHelloWorld())

// 複数のパラメータを持つ関数。
func greet(person: String, alreadyGreeted: Bool) -> String {
    if alreadyGreeted {
        return greetAgain(person: person)
    } else {
        return greet(person: person)
    }
}
print(greet(person: "Tim", alreadyGreeted: true))

// 返り値はなくてもいい。
func greet1(person: String) {
// func greet1(person: String) -> () {
    print("Hello, \(person)!")
}
print(greet1(person: "Dave"), ())  // -> ()

func printAndCount(string: String) -> Int {
    print(string)
    return string.count
}
func printWithoutCounting(string: String) {  // 返り値はない
    let _ = printAndCount(string: string)    // 返り値を無視している
}
// printAndCount(string: "hello, world")  // 返り値 12 を無視している。warning
let _ = printAndCount(string: "hello, world")  // warning を抑制する
let _ = printAndCount(string: "hello, world")  // _は複数回定義できる
printWithoutCounting(string: "hello, world")

// タプルによって複数の値を返す。
```

```
func minMax(array: [Int]) -> (min: Int, max: Int) {  // 名前付きタプルを返す
    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)
}
let bounds = minMax(array: [8, -6, 2, 109, 3, 71])
print("min is \(bounds.min) and max is \(bounds.max)")

// optional tuple return types
func minMax1(array: [Int]) -> (min: Int, max: Int)? {
    print("optional tuple return types")
    if array.isEmpty { return nil }
    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)
}
if let bounds = minMax1(array: [8, -6, 2, 109, 3, 71]) {
    print("min is \(bounds.min) and max is \(bounds.max)!")
}
print(minMax1(array: [8, -6, 2, 109, 3, 71]) as Any)
print(minMax1(array: [8, -6, 2, 109, 3, 71])!)

// 面倒な気がする。
// minMax を使う側はこの場合、[Int]#isEmpty ではないと知っている。
// しかし、minMax のなかで未知の問題が起きないとも限らないから minMax()!とまでしたくない。
// 本当にしたいことは、minMax() に [] を渡したときにはコンパイルエラーとして、
```

```
// [] 以外を渡したときにはそうでなくしてほしいということだ。
// つまり、minMax() の引数は array: [Int] というより、where !array.isEmpty なわけである。
// しかし言語仕様などのためにそうはいかないから、minMax() は定義域に [] を含むことになる。
// 結果として、入力 [] について出力 nil が返される。結果として正常な使用者にも影響が生じる。
// 1 つの方法は、minMax1 に [] が与えられたときには nil を返さず実行時エラーにすることだろう。
// すると、コードは簡潔になる。
// optional にある思想の 1 つは、実行時エラーにしないということだろうか。
// というのも、使用者はもしかしたら回復のすべを持っているのかもしれない。
// ならば、すべての関数（の返り値）を Optional にするか？
// しかしすべての関数の引数を Optional にまでしないのだから、nil のまますべて進行するのではない。
// minMax(array: [Int]) とすることで、API はユーザに対して [] を受け取ると宣言してしまう。
// 宣言しておいて、[] を渡されたときには注意書き違反だと実行時エラーを出してそれでいいのか。
// エラーが結果としての nil で表されるとどこに問題があったのかわかりにくい。
// というのも、開発中の大半のエラーは、なんでもない typo などで生じるだろう。
// 他の言語で、precondition() を用いる方法も一般的である。
// また別の方法は、デフォルト値として (0, 0) などを返すことである。
// それは問題を最も隠蔽する面はあるが、過大な定義域について出力の型を堅持することができる。
// 実装時にコントロールできない条件については Optional は適しているだろう。
// 正常系を外れたときには、例外を投げるか、Optional で nil を返すか、実行時エラーとする。
// コードの上では、ある制約を満たしていると明らかであっても、言語はそのすべてを理解できない。
// ゆえに、理想的に堅牢にはできない。
// 実行時エラーを見たくないというのは、考え方としてはありうる。
// もし関数が Optional なら、実行時エラーにするかどうかはユーザが選べる。
// 実行時エラーにしたければ、ただ f()!を多用すればよいと考えられる。
// いかなるときにある関数が nil を返すかは、ドキュメントで謳う形になるだろう。
// しかし実行時エラーにするかどうかをユーザに選ばせるという意味では、f()!は好ましくない。
// やはり、nil のまま進行させることになる。どこからを実行時エラーとすべきか、ということになる。
// モジュールでの異常値がプログラム全体を停止させるに値しないなら assert() を使う。
// その異常値ではどうせ処理ができないなら precondition() を使う。

// https://stackoverflow.com/questions/30094582/swift-ounchecked-and-assertions
// You should definitely not compile -Ounchecked for release in order to use
// precondition only while testing. Compiling -Ounchecked also disables
// checks for things like array out of bounds and unwrapping nil, that could
// lead to some pretty nasty production bugs involving memory corruption.

// 引数の定義域を理想的に制約できないことによる入力値には precondition() でいいのではないか。
// 辞書の要素アクセスすら Optional なので、Optional を汎用すればいいのかもしれない。!を多用する。

// Place parameters that don't have default values at the beginning of a
```

```
// function's parameter list, before the parameters that have default values.

// default 値のある引数について、ラベルで区別できるので前方にも置ける。
// 引数ラベルを無名にするなども、呼び出しが曖昧にならない限り許される。
func f1(x: Int = 1, y: Int) {
    print(x * y)
}
f1(x:2, y:3)
f1(y:3)

// variadic parameters 可変長パラメータ
// パラメータ number の型は [Double] となる。
func arithmeticMean(_ numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}
print(arithmeticMean(1, 2, 3, 4, 5),
      arithmeticMean(3, 8.25, 18.75))

// 引数ラベルを_としなかった場合:
func f2(x: Int...) {
    for element in x {
        print("\(element) ", terminator:"")
    }
    print()
}
f2(x: 1, 2, 3)  // 先頭にだけ引数ラベルを与えれば動くようだ。

// inout 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)")

func addTwoInts(_ a: Int, _ b: Int) -> Int {
    return a + b
}
func multiplyTwoInts(_ a: Int, _ b: Int) -> Int {
    return a * b
}
func printHelloWorld() {
    print("hello, world")
}
print(type(of: addTwoInts),
      type(of: multiplyTwoInts),
      type(of: printHelloWorld))
var mathFunction: (Int, Int) -> Int = addTwoInts
print("Result: \(mathFunction(2, 3))")
mathFunction = multiplyTwoInts
print("Result: \(mathFunction(2, 3))")

// 高階関数
func printMathResult(_ mathFunction: (Int, Int) -> Int,
                     _ a: Int,
                     _ b: Int) {
    print("Result: \(mathFunction(a, b))")
}
printMathResult(addTwoInts, 3, 5)

func stepForward(_ input: Int) -> Int {
    return input + 1
}
func stepBackward(_ input: Int) -> Int {
    return input - 1
}
// 関数を返す関数。
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    return backward ? stepBackward : stepForward
}
// nested function
func chooseStepFunction1(backward: Bool) -> (Int) -> Int {
    func stepForward(input: Int) -> Int { return input + 1 }
```

```
    func stepBackward(input: Int) -> Int { return input - 1 }
    return backward ? stepBackward : stepForward
}
var currentValue = 3
// let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)
let moveNearerToZero = chooseStepFunction1(backward: currentValue > 0)
print("Counting to zero:")
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")

do {
    let x = 89
    print(x)
}
// print(x)
```

implementation

```
// closures クロージャ
// https://docs.swift.org/swift-book/LanguageGuide/Closures.html

let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
print(names)

// 普通の関数定義による整列関数。
func backward(_ s1: String, _ s2: String) -> Bool {
    return s1 > s2
}
var reversedNames = names.sorted(by: backward)
print(reversedNames)

// 無名関数による実装:
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})
print(reversedNames)
```

```
// 型推論
// 引数に渡される無名関数については常に型推論できる。
reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 } )
print(reversedNames)

// 暗黙の return
reversedNames = names.sorted(by: { s1, s2 in s1 > s2 } )
print(reversedNames)

// shorthand argument name
reversedNames = names.sorted(by: { $0 > $1 } )
print(reversedNames)

// この場合は既存の関数を渡しても同じである。
reversedNames = names.sorted(by: >)
print(reversedNames)

// trailing closures
reversedNames = names.sorted() { $0 > $1 }
print(reversedNames)

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

let digitNames = [
    0: "Zero", 1: "One", 2: "Two",   3: "Three", 4: "Four",
    5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"
]
let numbers = [16, 58, 510]
let strings = numbers.map { (number) -> String in
    var number = number
    var output = ""
    repeat {
        // number の第 1 桁を取得し辞書を通してすぐに force-unwrap する。
        output = digitNames[number % 10]! + output  // 文字列の先頭に追加していく
        number /= 10    // 第 1 桁を捨てる
    } while number > 0  // 右から左にすべての桁を処理する
    return output
}
```

```
print(strings)



func makeIncrementer(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    // クロージャ。amout と runningTotal を補足する。
    func incrementer() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementer
}
let incrementByTen = makeIncrementer(forIncrement: 10)
print(incrementByTen())
print(incrementByTen())
print(incrementByTen())
let incrementBySeven = makeIncrementer(forIncrement: 7)
print(incrementBySeven())
print(incrementBySeven())
print(incrementBySeven())
print(incrementByTen())

// A closure is said to escape a function when the closure is passed as an argument
// to the function, but is called after the function returns.
// ある関数呼び出しが終了したのちに渡した関数が使われうるなら渡した関数は escaping している。

var completionHandlers: [() -> Void] = []
func someFunctionWithEscapingClosure(completionHandler: @escaping () -> Void) {
    completionHandlers.append(completionHandler)
}
func someFunctionWithNonescapingClosure(closure: () -> Void) {
    closure()
}
class SomeClass {
    var x = 10
    func doSomething() {
        someFunctionWithEscapingClosure { self.x = 100 }  // self. が必要
        someFunctionWithNonescapingClosure { x = 200 }
    }
```

```
}

let instance = SomeClass()   // インスタンスを生成する
instance.doSomething()
print(instance.x)            // 200; プロパティの状態を見る
completionHandlers.first?()  // escaping なクロージャによる更新
print(instance.x)            // 100

// autoclosure
// An autoclosure lets you delay evaluation, because the code inside isn't run
// until you call the closure.

var customersInLine = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
print(customersInLine.count)  // 5
let customerProvider = { customersInLine.remove(at: 0) }
print(customersInLine.count)  // 5
print("Now serving \(customerProvider())!")  // Chris
print(customersInLine.count)  // 4

// customersInLine is ["Alex", "Ewa", "Barry", "Daniella"]
func serve(customer customerProvider: () -> String) {
    print("Now serving \(customerProvider())!")
}
// 明示的な closure
serve(customer: { customersInLine.remove(at: 0) } )  // Alex
func serve(customer customerProvider: @autoclosure () -> String) {
    print("Now serving \(customerProvider())!")
}
// 暗黙な closure つまり autoclosure
serve(customer: customersInLine.remove(at: 0))  // Ewa

// 属性@autoclosure と@escaping を同時に用いる。
var customerProviders: [() -> String] = []      // 関数の配列
func collectCustomerProviders(_ customerProvider: @autoclosure @escaping () -> String) {
    customerProviders.append(customerProvider)  // 配列に関数を追加する
}
collectCustomerProviders(customersInLine.remove(at: 0))  // autoclosure が escaping する
collectCustomerProviders(customersInLine.remove(at: 0))  // autoclosure が escaping する
print("Collected \(customerProviders.count) closures.")  // 2
for customerProvider in customerProviders {
```

```
    // escaping されていたそれぞれの autoclosure が実行される。
    print("Now serving \(customerProvider())!")
}
```

implementation

```
// enumeration 列挙
// https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html

enum CompassPoint {
    case north  // 各 enumeration case らを定義する
    case south
    case east
    case west
}
print(CompassPoint.north)

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

var directionToHead = CompassPoint.west
print(directionToHead)
directionToHead = .east
print(directionToHead)
let directionToHead1: CompassPoint = .north
print(directionToHead1)

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

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

// CaseIterable プロトコルを enumeration に設定する。
enum Beverage: CaseIterable {
    case coffee, tea, juice
}
let numberOfChoices = Beverage.allCases.count
print("\(numberOfChoices) beverages available")
for beverage in Beverage.allCases {
    print(beverage)
}

// associated value
enum Barcode {
    case upc(Int, Int, Int, Int)
    case qrCode(String)
}
// インスタンス化
var productBarcode = Barcode.upc(8, 85909, 51226, 3)
print(productBarcode)
productBarcode = .qrCode("ABCDEFGHIJKLMNOP")
print(productBarcode)

// switch 文では、associated value らを束縛できる。
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).")
}
// let や var は手前にまとめて書ける。
switch productBarcode {
case let .upc(numberSystem, manufacturer, product, check):
```

```
    print("UPC : \(numberSystem), \(manufacturer), \(product), \(check).")
case let .qrCode(productCode):
    print("QR code: \(productCode).")
}

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

// implicitly assigned raw value
enum Planet1: Int {
    case mercury = 1, venus, earth, mars, jupiter, saturn, uranus, neptune
}
enum CompassPoint1: String {
    case north, south, east, west
}

let earthsOrder = Planet1.earth.rawValue
print(earthsOrder)      // 3
let sunsetDirection = CompassPoint1.west.rawValue
print(sunsetDirection)  // west

// initializing from a row value
let possiblePlanet = Planet1(rawValue: 7)  // failable initializer
print(possiblePlanet!, possiblePlanet == Planet1.uranus)

let positionToFind = 11
if let somePlanet = Planet1(rawValue: positionToFind) {
    switch somePlanet {
    case .earth:
        print("Mostly harmless")
    default:
        print("Not a safe place for humans")
    }
} else {
    print("There isn't a planet at position \(positionToFind)")  // 実行される
}
```

```
// recursive enumeration 再帰的列挙
enum ArithmeticExpression {
    case number(Int)
    indirect case addition(ArithmeticExpression, ArithmeticExpression)
    indirect case multiplication(ArithmeticExpression, ArithmeticExpression)
}
// すべてのケースについて indirect にするには enum 全体を indirect 修飾すればいい。
indirect enum ArithmeticExpression1 {
    case number(Int)
    case addition(ArithmeticExpression1, ArithmeticExpression1)
    case multiplication(ArithmeticExpression1, ArithmeticExpression1)
}

// (5 + 4) * 2
let five = ArithmeticExpression1.number(5)
let four = ArithmeticExpression1.number(4)
let sum = ArithmeticExpression1.addition(five, four)
let product = ArithmeticExpression1.multiplication(sum, ArithmeticExpression1.number(2))
print(product)
func evaluate(_ expression: ArithmeticExpression1) -> 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))  // 18
```

implementation

```
// structures and classes 構造体とクラス
// https://docs.swift.org/swift-book/LanguageGuide/ClassesAndStructures.html

// In practice, this means most of the custom data types you define will be
// structures and enumerations.
```

```
struct Resolution {
    var width = 0  // stored porperty
    var height = 0
}
class VideoMode {
    var resolution = Resolution()
    var interlaced = false
    var frameRate = 0.0
    var name: String?
}
let someResolution = Resolution()  // initializer syntax
let someVideoMode = VideoMode()
print(someResolution, someVideoMode)

// dot syntax ドット文法
print("The width of someResolution is \(someResolution.width)")  // 0
print("The width of someVideoMode is \(someVideoMode.resolution.width)")  // 0

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

// memberwise initializers for structure types
let vga = Resolution(width: 640, height: 480)
print(vga)

// structures and enumerations are value types 構造体と列挙は値型である。
// 配列と辞書は構造体である。
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd
cinema.width = 2048
print("cinema is now \(cinema.width) pixels wide")
print("hd is still \(hd.width) pixels wide")

// 列挙も値型だから、pass-by-value される。
enum CompassPoint {
    case north, south, east, west
    mutating func turnNorth() {
```

```
        self = .north
    }
}
var currentDirection = CompassPoint.west
let rememberedDirection = currentDirection
currentDirection.turnNorth()
print("The current direction is \(currentDirection)")
print("The remembered direction is \(rememberedDirection)")

// クラスは参照型なので pass-by-reference される。
let tenEighty = VideoMode()
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25.0

let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30.0
print("The frameRate property of tenEighty is now \(tenEighty.frameRate)")  // 30.0

// identity operators
// identical かどうかを Bool で返す。
if tenEighty === alsoTenEighty {
    print("tenEighty and alsoTenEighty refer to the same VideoMode instance.")
}
```

implementation

```
// properties プロパティ
// https://docs.swift.org/swift-book/LanguageGuide/Properties.html

// Stored Property と Computed Property がある。
// type property はインスタンスではなく型に属する。

// stored property

struct FixedLengthRange {
    var firstValue: Int  // 変数属性
    let length: Int      // 定数属性
```

```
}
var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3)
print(rangeOfThreeItems)
rangeOfThreeItems.firstValue = 6
print(rangeOfThreeItems)


// stored properties of constant structure instances
let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4)
// rangeOfFourItems.firstValue = 6  // 定数は変更できない。

// lazy stored properties
class DataImporter {  // 初期化の重いクラス
    var filename = "data.txt"
    init() { print("Initializing a DataImporter instance.... Done.") }
}
class DataManager {
    // DataManger が init 完了した時点では未定なので lazy 対象は var とせねばならない。
    lazy var importer = DataImporter()  // 必要になるまでインスタンス化しない
    var data = [String]()
}
let manager = DataManager()
manager.data.append("Some data")
manager.data.append("Some more data")
print(manager.importer.filename)

// Note
// If a property marked with the lazy modifier is accessed by multiple
// threads simultaneously and the property has not yet been initialized,
// there is no guarantee that the property will be initialized only once.

// computed properties
struct Point {
    var x = 0.0, y = 0.0
}
struct Size {
    var width = 0.0, height = 0.0
}
struct Rect {
    var origin = Point(), size = Size()  // 保持されるのは原点と大きさ
```

```
    var center: Point {                    // 中心点は computed プロパティ
        get {
            // 原点の x 座標に矩形の幅の半分を加えたものが中心点の x 座標である。
            let centerX = origin.x + (size.width / 2)
            let centerY = origin.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set(newCenter) {
            // 中心点の x 座標から矩形の幅の半分を引いたものが原点の x 座標である。
            origin.x = newCenter.x - (size.width / 2)
            origin.y = newCenter.y - (size.height / 2)
        }
    }
}
var square = Rect(origin: Point(x: 0.0,
                                y: 0.0),
                  size:   Size(width:  10.0,
                               height: 10.0))
print("square.origin is now at (\(square.origin.x), \(square.origin.y))")
let initialSquareCenter = square.center
square.center = Point(x: 15.0, y: 15.0)
print("square.origin is now at (\(square.origin.x), \(square.origin.y))")

// なお set でデフォルト引数 newValue を用いるなら次のようになる。
struct AlternativeRect {
    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 computed properties
struct Cuboid {  // 直方体
    var width = 0.0, height = 0.0, depth = 0.0
    var volume: Double {  // 読み取り専用 computed プロパティ
        return width * height * depth
    }
}
let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0, depth: 2.0)
print("the volume of fourByFiveByTwo is \(fourByFiveByTwo.volume)")  // 40.0

// property observers プロパティオブザーバ
// スーパークラスのイニシャライザが呼ばれたあとのサブクラスのイニシャライザからの更新では、
// スーパークラスのプロパティの willSet や didSet が呼ばれる。

class StepCounter {
    var totalSteps: Int = 0 {
        // willSet(newTotalSteps) {
        //     print("About to set totalSteps to \(newTotalSteps)")
        // }
        willSet {
            print("About to set totalSteps to \(newValue)")
        }
        didSet {
            if totalSteps > oldValue  {  // 増分を出力する
                print("Added \(totalSteps - oldValue) steps")
            }
        }
    }
}
let stepCounter = StepCounter()
stepCounter.totalSteps = 200
stepCounter.totalSteps = 360
stepCounter.totalSteps = 896

// 関数の inout パラメータにオブザーバを持つ他のプロパティを渡すと常に willSet/didSet が呼ばれる。



// global and local variables
// stored プロパティに対して computed プロパティやプロパティオブザーバが行えたことは、
// stored 変数について、computed な変数や変数へのオブザーバとして同様に実装できる。
```

```
// global な定数と変数は常に lazy に compute される。local な定数と変数は決して lazy にはならない。


// type properties; （インスタンスプロパティではないものとしての）型プロパティ

// stored 型プロパティについては初期化するイニシャライザがないので必ず初期値が必要だ。
// stored 型プロパティは常に lazy だが、インスタンスプロパティと異なり、必ず 1 度しか呼ばれない。

// 型プロパティは static ないし class キーワードで修飾する。
struct SomeStructure {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 1
    }
}
enum SomeEnumeration {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 6
    }
}
class SomeClass {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 27
    }
    class var overrideableComputedTypeProperty: Int {
        return 107
    }
}
// インスタンス化せずに使える。
print(SomeStructure.storedTypeProperty)
SomeStructure.storedTypeProperty = "Another value."
print(SomeStructure.storedTypeProperty)
print(SomeEnumeration.computedTypeProperty)
print(SomeClass.computedTypeProperty)

struct AudioChannel {
    static let thresholdLevel = 10
    static var maxInputLevelForAllChannels = 0
```

```
    var currentLevel: Int = 0 {
        didSet {
            if currentLevel > AudioChannel.thresholdLevel {  // セッタでいいのではないか
                currentLevel = AudioChannel.thresholdLevel   // didSet を再発火しない
            }
            if currentLevel > AudioChannel.maxInputLevelForAllChannels {
                AudioChannel.maxInputLevelForAllChannels = currentLevel
            }
        }
    }
}

var leftChannel = AudioChannel()
var rightChannel = AudioChannel()
leftChannel.currentLevel = 7
print(leftChannel.currentLevel)                   // 7
print(AudioChannel.maxInputLevelForAllChannels)  // 7
rightChannel.currentLevel = 11
print(rightChannel.currentLevel)                  // 10; cap された
print(AudioChannel.maxInputLevelForAllChannels)  // 10; 共有されている
// print(leftChannel.maxInputLevelForAllChannels)  // 型メンバはインスタンスから呼べない
```