Learning Swift Language on Ubuntu:
Generics
Automatic Reference Counting
Memory Safety
Access Control
Advanced Operators

strong/weak reference

強い参照と弱い参照、さらに unowned な参照というものがあるらしい。ARC (Automatic Reference Counting) は強い参照が 0 個になったオブジェクトをデストラクトする仕組みなので、強い参照で参照循環 (retain cycle) になっているとメモリリークとなるらしい。

強参照循環は悪だろうか。GC (Garbage Collection) のうち ARC でないものを単に GC と呼ぶことにすれば、GC は参照が到達可能なものを残す仕組みだろう。到達可能だとはある意味では主観的だ。（というのはつまり、相手からすれば到達可能でなくなるのはこちらであるから。） 中心となるグラフからある部分が切り離されたときに、そこにある強参照循環を解いてやれば、ARC によって残りは deinit されることになるのだろう。しかしそのような、自ら GC を実装するようなアプローチはまず考えなくていいだろう。だから強参照循環は悪いものだと考えられる。

すべての参照を弱参照にしてはどうだろうか。それはできない。ただちに deallocate されてしまうからだ。ゆえに素朴に考えると、存在すべきオブジェクトらについて強参照は木構造になっていると考えられる。有向非巡回グラフ (Directed Acyclic Graph, DAG) という考え方のほうが近いかもしれない。問題は、あるオブジェクトの参照を親から行うか子から行うかだろう。それは時々の設計だろう。

弱い参照とは別に、unowned なるものがある。これは弱い参照の一種だと考えていいのではなかろうか。implicitly unwrapped なものだと考えられる。古典的な参照は nil を許しただろう。だから nil かどうかの検査すらできない unowned な切れた参照はより厄介かもしれない。しかしそんな事態を防ぐのは設計の責任だ。

将来の構成の変化に備えて、weak な参照を多用してしまえという考え方もあるようだ。短所は、Optional が氾濫してコードが劣化することらしい。Optional は可能なら最小にすべきものだと思われる。そして、将来的な構成の変化を事実上考えなくていい状況はあるだろう。unowned は let つまり定数にできて好ましい。

循環参照が生じない限りにおいて強参照を使えばいい。循環参照が生じて、明らかに永続的な包含関係があるときは、unowned let を使えばいい。チュートリアルには unowned(unsafe) のほうがパフォーマンスがよいとあるが、さほど人気はないようだ。循環参照が生じるが包含関係にないときは、weak var を使えばいい。

強参照によって存在を保つのはそのときには、上層の責任だ。

implementation

```
// generics
// https://docs.swift.org/swift-book/LanguageGuide/Generics.html

// 総称関数を用いない場合、実装が同じ関数を型ごとに定義せざるをえない:
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)")

// generics function 総称的な関数
// type parameter 型パラメータ T を用いる
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let temporaryA = a
    a = b
    b = temporaryA
}
someInt = 3
anotherInt = 107
swapTwoValues(&someInt, &anotherInt)
print(someInt, anotherInt)

var someString = "hello"
var anotherString = "world"
swapTwoValues(&someString, &anotherString)
print(someString, anotherString)

func swapTwoValues1<T>(_ a: inout T, _ b: inout T) { (a, b) = (b, a) }
someInt = 3
anotherInt = 107
swapTwoValues1(&someInt, &anotherInt)
print(someInt, anotherInt)
```

```
someInt = 3
anotherInt = 107
swap(&someInt, &anotherInt)
print(someInt, anotherInt)

// naming type parameters
// Dictionary<Key, Value>や Array<Element>のように、型パラメータには有意義な名前が望ましい。
// しかし特に適当なものがない場合には慣習的に、T、U、V などとする。

// generic type 総称関数とは別に総称型がある。
// stack は push と pop ができるデータ構造である。

// 総称的でない型によるスタックの実装:
struct IntStack {
    var items = [Int]()
    mutating func push(_ item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
}

// 総称的な Stack 型:
struct Stack<Element> {
    var items = [Element]()
    mutating func push(_ item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
}
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")
stackOfStrings.push("cuatro")
print(stackOfStrings.items)
let fromTheTop = stackOfStrings.pop()
```

```
print(fromTheTop, stackOfStrings.items)

// 総称型を extension するときは、宣言しなくても同じ型変数（ここでは Element）が使える:
extension Stack {
    /// トップにある要素を返す。ただし、もし空なら nil を返す。
    var topItem: Element? {
        return items.isEmpty ? nil : items[items.count - 1]
    }
}
if let topItem = stackOfStrings.topItem {  // optional 束縛
    print("The top item on the stack is \(topItem).")
}

// type constraints 型制約
// 例えば Dictionary オブジェクトの要素のキーは Hashable でなければならない。
// type constraints in action

/// 配列 in における ofString の位置を返す。存在しなければ nil を返す。
func findIndex(ofString valueToFind: String, in array: [String]) -> Int? {
    for (index, value) in array.enumerated() {  // 各要素について
        if value == valueToFind {               // 見つけた
            return index
        }
    }
    return nil  // 見つからなかった
}
let strings = ["cat", "dog", "llama", "parakeet", "terrapin"]
if let foundIndex = findIndex(ofString: "llama", in: strings) {
    print("The index of llama is \(foundIndex)")
}

// 総称関数で演算子==を用いるには型が Equatable でなければならない:
func findIndex<T: Equatable>(of valueToFind: T, in array:[T]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

```
let doubleIndex = findIndex(of: 9.3, in: [3.14159, 0.1, 0.25])
print(doubleIndex as Any)
let stringIndex = findIndex(of: "Andrea", in: ["Mike", "Malcolm", "Andrea"])
print(stringIndex as Any)

// associatedtype
protocol Container {
    associatedtype Item
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }
}
struct IntStack1: Container {
    var items = [Int]()
    mutating func push(_ item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
    // typealias Item = Int  // turns abstract type into a concrete type.
    mutating func append(_ item: Int) {  // Int が Container の Item だろうと推論される。
        self.push(item)
    }
    var count: Int {
        return items.count
    }
    subscript(i: Int) -> Int {
        return items[i]
    }
}
struct Stack1<Element>: Container {
    var items = [Element]()
    mutating func push(_ item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
    mutating func append(_ item: Element) {  // Element が Item だと自動的に推論される。
```

```
        self.push(item)
    }
    var count: Int {
        return items.count
    }
    subscript(i: Int) -> Element {
        return items[i]
    }
}

// extending an existing type to specify an associatedtype
// 既存の Array はそもそも Container たる要件を満たしているので宣言さえすればいい:
extension Array: Container {}

// adding constraints to an associatedtype
protocol Container1 {
    associatedtype Item: Equatable
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }
}

// using a protocol in its associated type's constraints
protocol SuffixableContainer: Container {
    associatedtype Suffix: SuffixableContainer where Suffix.Item == Item
    /// 末尾の size 個の要素を返す。
    func suffix(_ size: Int) -> Suffix
}
extension Stack1: SuffixableContainer {
    func suffix(_ size: Int) -> Stack1 {  // Suffix は具体的には Stack1 だと推論される。
        var result = Stack1()
        for index in (count-size)..<count {  // 末尾の size 個それぞれについて
            result.append(self[index])
        }
        return result
    }
}
var stackOfInts = Stack1<Int>()
stackOfInts.append(10)  // [10]
stackOfInts.append(20)  // [10, 20]
```

```
stackOfInts.append(30)  // [10, 20, 30]
let suffix = stackOfInts.suffix(2)
print(suffix)  // [20, 30]

// 上の例では Stack1 の associatedtype はまた Stack1 であったが、そうでなくてもいい。
// 例えば次の例では、IntStack1 の associatedtype Stack1<Int>は制約を満たしている。
// 制約とはつまり、ともに SuffixableContainer であって、要素の型が Int 同士等しいということだ。
extension IntStack1: SuffixableContainer {
    func suffix(_ size: Int) -> Stack1<Int> {
        var result = Stack1<Int>()
        for index in (count-size)..<count {
            result.append(self[index])
        }
        return result
    }
}

// generic where clause
/// C1 と C2 の各要素が同じ順でそれぞれ等しければ真を返す。
/// Container でありさえすれば、C1 と C2 の型は異なっていていい。
func allItemsMatch<C1: Container, C2: Container>
    (_ someContainer: C1, _ anotherContainer: C2) -> Bool
    where C1.Item == C2.Item, C1.Item: Equatable {
        if someContainer.count != anotherContainer.count {
            return false  // そもそもコンテナの大きさが異なる。
        }
        for i in 0..<someContainer.count {
            if someContainer[i] != anotherContainer[i] {
                return false  // ある要素が存在して等しくなかった。
            }
        }
        return true  // コンテナの大きさが等しく各添字の要素が互いに等しい。
}
var stackOfStrings1 = Stack1<String>()
stackOfStrings1.push("uno")
stackOfStrings1.push("dos")
stackOfStrings1.push("tres")
var arrayOfStrings = ["uno", "dos", "tres"]
if allItemsMatch(stackOfStrings1, arrayOfStrings) {
    print("All items match.")  // 実行される
```

```
} else {
    print("Not all items match.")
}

// extensions with a generic where clause
// Stack1 型を（Element が Equatable なものについてのみ）extension する。
extension Stack1 where Element: Equatable {
    func isTop(_ item: Element) -> Bool {
        guard let topItem = items.last else {
            return false  // [Element]#last は要素がないとき nil。
        }
        return topItem == item  // トップにある要素が item かどうか。
    }
}
if stackOfStrings1.isTop("tres") {
    print("Top element is tres.")  // 実行される
} else {
    print("Top element is something else.")
}

struct NotEquatable { }
var notEquatableStack = Stack1<NotEquatable>()
let notEquatableValue = NotEquatable()
notEquatableStack.push(notEquatableValue)
// notEquatableStack.isTop(notEquatableValue)  // Error
// error: argument type 'NotEquatable' does not conform to expected type 'Equatable'

// protocol を where で拡張する:
extension Container where Item: Equatable {
    func startsWith(_ item: Item) -> Bool {
        return count >= 1 && self[0] == item
    }
}
if [9, 9, 9].startsWith(42) {
    print("Starts with 42.")
} else {
    print("Starts with something else.")  // 実行される
}

extension Container where Item == Double {  // 具体的な型で制約する。
```

```
    func average() -> Double {
        var sum = 0.0
        for index in 0..<count {
            sum += self[index]
        }
        return sum / Double(count)
    }
}
print([1260.0, 1200.0, 98.6, 37.0].average())  // 648.9

// associated types with a generic where clause
protocol Container2 {
    associatedtype Item
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }

    associatedtype Iterator: IteratorProtocol where Iterator.Element == Item
    func makeIterator() -> Iterator
}

// protocol を継承した子 protocol で associatedtype を制約する where 節はこう書ける:
protocol ComparableContainer: Container where Item: Comparable { }

// generic subscript 総称的添字
extension Container {
    // 添字として Sequence 型を取る。ただしその Iterator の Element は Int でなければならない。
    // なお self の添字が Int であることは、Container プロトコルの定義がすでに言っている。
    subscript<Indices: Sequence>(indices: Indices) -> [Item]
        where Indices.Iterator.Element == Int {
            var result = [Item]()
            for index in indices {
                result.append(self[index])
            }
            return result
    }
}
print(stackOfStrings1)          // ["uno", "dos", "tres"]
print(stackOfStrings1[[0, 1]])  // ["uno", "dos"]
```

implementation

```
// automatic reference counting
// https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html

// Automatic Reference Counting (ARC)
// ARC はクラスのインスタンスにのみ適用される。構造体と列挙型は pass by value なので無関係だ。

protocol CountRef: AnyObject {}
extension CountRef {
    func countRef() {
        print("(strong, owned, weak) =",
              _getRetainCount(self),
              _getUnownedRetainCount(self),
              _getWeakRetainCount(self))
    }
}

class Person: CountRef {
    let name: String  // stored constant property
    init(name: String) {
        self.name = name
        print("\(name) is being initialized")
    }
    deinit {
        print("\(name) is being deinitialized")
    }
}
var reference1: Person?
var reference2: Person?
var reference3: Person?
reference1 = Person(name: "John Appleseed")
reference1?.countRef()
// 現在、John Appleseed への強参照は 1 個である。
reference2 = reference1
reference3 = reference1
reference1?.countRef()
// 現在、John Appleseed への強参照は 3 個である。
reference1 = nil
```

```
reference2 = nil
reference1?.countRef()
reference3 = nil

// strong reference cycle
// 強参照循環は、例えば２つのオブジェクトが互いを参照するときに起こる。

// 互いを参照することで強参照循環を起こしてしまう誤った実装の例:
class Person1: CountRef {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment1?
    deinit { print("\(name) is being deinitialized") }
}
class Apartment1: CountRef {
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person1?
    deinit { print("Apartment \(unit) is being deinitialized") }
}
var john: Person1?
var unit4A: Apartment1?
john = Person1(name: "John Appleseed")
unit4A = Apartment1(unit: "4A")
john?.countRef(); unit4A?.countRef();
// john = nil; unit4A = nil;  // ここで nil にすれば deinit する。

john!.apartment = unit4A
unit4A!.tenant = john
john?.countRef(); unit4A?.countRef();
john = nil; unit4A = nil;  // nil にしても（強参照循環のため）deinit しない。
// memory leak が起きたことになる。

// resolving strong reference cycles between class instances
// クラスインスタンス間の強参照循環を解決する
// weak reference と unowned reference というものが提供される。

// 参照先の生存時間が自分よりも短いときには weak reference を使うといい。
// 参照先の生存時間が短いか同じときには unowned reference を使うといい。
// weak reference は deinit されたとき nil になるので Optional な変数にする。定数にはできない。
```

```
// Apartment2 から Person2 への参照を弱参照に修正した:
class Person2: CountRef {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment2?
    deinit { print("\(name) is being deinitialized") }
}
class Apartment2: CountRef {
    let unit: String
    init(unit: String) { self.unit = unit }
    weak var tenant: Person2?
    deinit { print("Apartment \(unit) is being deinitialized") }
}
var john2: Person2?
var unit4A2: Apartment2?
john2 = Person2(name: "John Appleseed")
unit4A2 = Apartment2(unit: "4A")
john2!.apartment = unit4A2
unit4A2!.tenant = john2
john2?.countRef(); unit4A2?.countRef();
john2 = nil    // deinit()
unit4A2 = nil  // deinit()

// 参照カウント方式でないガーベジコレクションを用いるシステムでは、弱参照がキャッシュに使われる。
// その弱参照は、メモリ容量が不足したときに破棄される。参照カウントの弱参照はただちに破棄される。
// よって ARC の弱参照は、いわゆる GC における caching の用途には適してない。

// unowned reference
// unowned reference は nil にならない。しかし deinit 済みの参照にアクセスしたら実行時エラーだ。

class Customer {
    let name: String
    var card: CreditCard?
    init(name: String) {
        self.name = name
    }
    deinit { print("\(name) is being deinitialized") }
}
class CreditCard {
```

```
    let number: UInt64
    // unowned let customer: Customer
    unowned(unsafe) let customer: Customer
    init(number: UInt64, customer: Customer) {
        self.number = number
        self.customer = customer
    }
    deinit { print("Card #\(number) is being deinitialized") }
}
var john3: Customer?
john3 = Customer(name: "John Appleseed")
john3!.card = CreditCard(number: 1234_5678_9012_3456, customer: john3!)
john3 = nil  // deinit

// unowned(unsafe) と書けば unsafe unowned reference が使える。実行時のチェックがなくなる。
// チェックがなくなるとはつまり、すでに存在しない参照先を読み書きしても実行時エラーにならない。

// unowned references and implicitly unwrapped optional properties

class Country {
    let name: String
    // self が init() されるまでは City() に self を渡せないので let にはできない。
    var capitalCity: City!  // implicitly unwrapped optional property
    init(name: String, capitalName: String) {
        self.name = name
        self.capitalCity = City(name: capitalName, country: self)
    }
}
class City {
    let name: String
    unowned let country: Country  // unowned reference property
    init(name: String, country: Country) {
        self.name = name
        self.country = country
    }
}
var country = Country(name: "Canada", capitalName: "Ottawa")
print("\(country.name)'s capital city is called \(country.capitalCity.name)")

// strong reference cycles for closures
```

```
// クラスのプロパティにあるクロージャが self を補足すると、強参照循環が起こっている。
// closure capture list という機能がある。

// クラスとクロージャによって強参照循環が起こりメモリリークしている例:
/// HTML 要素を表すクラス。
class HTMLElement: CountRef {
    let name: String   // 要素の名前。h1、p、br など。
    let text: String?  // この要素のなかのテキスト。<p>text</p>などの内側部分。
    // init() 完了後に self が使えるので lazy とする。
    lazy var asHTML: () -> String = {  // クロージャ
        // このクロージャは self を補足する。
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }
    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }
    deinit {
        print("\(name) is being deinitialized")
    }
}
let heading = HTMLElement(name: "h1")
let defaultText = "some default text"
heading.asHTML = {
    return "<\(heading.name)>\(heading.text ?? defaultText)</\(heading.name)>"
}
print(heading.asHTML())  // <h1>some default text</h1>

var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
paragraph!.countRef()
print(paragraph!.asHTML())  // <p>hello, world</p>
paragraph!.countRef()
paragraph = nil

// resolving strong reference cycle for closures
// 強参照循環をなくしメモリリークを解決したバージョン:
```

```
class HTMLElement1: CountRef {
    let name: String
    let text: String?
    lazy var asHTML: () -> String = {
        [unowned self] in  // closure capture list
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }
    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }
    deinit {
        print("\(name) is being deinitialized")
    }
}
var paragraph1: HTMLElement1? = HTMLElement1(name: "p", text: "hello, world")
paragraph1!.countRef()
print(paragraph1!.asHTML())
paragraph1!.countRef()
paragraph1 = nil  // 無事に deinit() される。
```

implementation

```
// memory safety
// https://docs.swift.org/swift-book/LanguageGuide/MemorySafety.html

var one = 1                     // メモリに書き込む
print("We're number \(one)!")  // メモリを読み取る

/// than+1 を返す。
func oneMore(than number: Int) -> Int {
    return number + 1
}
var myNumber = 1
myNumber = oneMore(than: myNumber)
```

```
print(myNumber)  // 2

// A function has long-term write access to all of its in-out parameters.

// conflicting access to in-out parameters:
var stepSize = 1
func increment(_ number: inout Int) {
    number += stepSize
}
// increment(&stepSize)  // 実行時エラー

var copyOfStepSize = stepSize  // 別のメモリ領域に一時的にコピーする
increment(&copyOfStepSize)     // コピーされた側をコピー元（stepSize）で更新する
stepSize = copyOfStepSize      // オリジナルなメモリ領域に更新を反映する
print(stepSize)                // 2

// passing a single variable as the argument for multiple in-out parameters
// of the same function produces a conflict:
func balance(_ x: inout Int, _ y: inout Int) {
    let sum = x + y
    x = sum / 2
    y = sum - x
}
var playerOneScore = 42
var playerTwoScore = 30
balance(&playerOneScore, &playerTwoScore)     // OK
// balance(&playerOneScore, &playerOneScore)  // コンパイルエラー
print(playerOneScore, playerTwoScore)  // 36 36

// conflicting access to self in methods
struct Player {
    var name: String
    var health: Int
    var energy: Int

    static let maxHealth = 10
    mutating func restoreHealth() {
        health = Player.maxHealth
    }
}
```

```
extension Player {
    mutating func shareHealth(with teammate: inout Player) {
        balance(&teammate.health, &health)
    }
}
var oscar = Player(name: "Oscar", health: 10, energy: 10)
var maria = Player(name: "Maria", health: 5, energy: 10)
oscar.shareHealth(with: &maria)     // OK
// oscar.shareHealth(with: &oscar)  // コンパイルエラー
print(oscar, maria)

// conflicting access to properties

var playerInformation = (health: 10, energy: 20)
// balance(&playerInformation.health, &playerInformation.energy)  // 実行時エラー

var holly = Player(name: "Holly", health: 10, energy: 10)
// balance(&holly.health, &holly.energy)  // 実行時エラー

// ローカル変数ならアクセスが競合してないことをコンパイラが証明できる。
func someFunction() {
    var oscar = Player(name: "Oscar", health: 10, energy: 10)
    balance(&oscar.health, &oscar.energy)  // OK
}
someFunction()

// ある構造体の異なる部分へのアクセスがコンパイラに許される条件は次の 3 つだ。
// computed プロパティや class プロパティではなく stored プロパティにのみアクセスしている。
// 構造体がグローバル変数の値ではなくローカル変数の値である。
// 構造体がクロージャに補足されてないか、nonescaping なクロージャにのみ補足されている。
```

implementation

```
// access control
// https://docs.swift.org/swift-book/LanguageGuide/AccessControl.html

// Indeed, if you are writing a single-target app, you may not need to
// specify explicit access control levels at all.
```

```
// module とは、import キーワードの対象になるものである。
// 5 個の access level がある。open、public、internal、fileprivate、private である。
// private は型の中から参照できる。fileprivate はファイルの中から参照できる。
// internal はモジュールの中から参照できる。public はモジュールの中から継承できる。
// open はモジュールの外から継承できる。

// Swift 2 までは public、internal、private の 3 つしかなかった。
// そこでの public と private は今でいう open と fileprivate であった。
// つまり Swift 3 で public と private が追加された。
// Swift 4 で private の仕様が変わり extension を許すようになった。

// Open access is the highest (least restrictive) access level and private
// access is the lowest (most restrictive) access level.
// アクセスレベルが「高い」とは広範囲から見えることであり、「低い」とはローカルなことだ。

public class SomePublicClass {}
internal class SomeInternalClass {}
fileprivate class SomeFilePrivateClass {}
private class SomePrivateClass {}

public var somePublicVariable = 0
internal let someInternalConstant = 0
fileprivate func someFilePrivateFunction() {}
private func somePrivateFunction() {}

// func someFunction() -> (SomeInternalClass, SomePrivateClass) {  // コンパイルエラー
private func someFunction() -> (SomeInternalClass, SomePrivateClass) {
    return (SomeInternalClass(), SomePrivateClass())
}

// ゲッタよりもセッタのアクセスレベルを高くする:
struct TrackedString {
    private(set) var numberOfEdits = 0
    var value: String = "" {
        didSet {
            numberOfEdits += 1
        }
    }
}
var stringToEdit = TrackedString()
```

```
stringToEdit.value = "This string will be tracked."
stringToEdit.value += " This edit will increment numberOfEdits."
stringToEdit.value += " So will this one."
print("The number of edits is \(stringToEdit.numberOfEdits)")  // 3

// さらにゲッタにもアクセスレベルを設定する:
public struct TrackedString1 {
    public private(set) var numberOfEdits = 0
    public var value: String = "" {
        didSet {
            numberOfEdits += 1
        }
    }
    public init() {}
}

// private members in extensions
protocol SomeProtocol {
    func doSomething()
}
struct SomeStruct {
    private var privateVariable = 12
}
extension SomeStruct: SomeProtocol {
    func doSomething() {
        print(privateVariable)  // 同じファイルなので private メンバにアクセスできる。
    }
}
SomeStruct().doSomething()  // -> 12
```

implementation

```
// advanced operators
// https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html

// 高度な演算子がある。ビット演算子、オーバーフロー演算子、ユーザ定義演算子だ。
// すべてのオーバーフロー演算子は&で始まる。

/// 引数を 2 進数表現で印字する。
```

```
func p2<T: BinaryInteger>(_ xs: T...) {
    /// 引数を 2 進数で表した文字列を返す。
    func s2<T: BinaryInteger>(_ x: T) -> String {
        /// 引数が 8 文字以下ならば左を 0 で満たす。
        func pad8(_ s: String) -> String {
            let padLen: Int = 8 - s.count
            let pad: String = String(repeatElement("0", count: padLen))
            return pad + s
        }
        let s: String = String(x, radix: 2)
        return pad8(s)
    }
    for i in 0..<xs.count {
        let s = s2(xs[i])
        switch i {
        case xs.count - 1:
            print(s)
        default:
            print(s, terminator: " ")
        }
    }
}

// NOT
let initialBits: UInt8 = 0b00001111  // 2 進数
let invertedBits = ~initialBits      // NOT 演算
p2(invertedBits)                     // 11110000

// AND
let firstSixBits: UInt8 = 0b11111100
let lastSixBits: UInt8  = 0b00111111
let middleFourBits = firstSixBits & lastSixBits
p2(middleFourBits)

// OR
let someBits: UInt8 = 0b10110010
let moreBits: UInt8 = 0b01011110
let combinedbits = someBits | moreBits
p2(combinedbits)
```

```
// XOR
let firstBits: UInt8 = 0b00010100
let otherBits: UInt8 = 0b00000101
let outputBits = firstBits ^ otherBits
p2(outputBits)

let shiftBits: UInt8 = 4
p2(shiftBits,
   shiftBits << 1,
   shiftBits << 2,
   shiftBits << 5,
   shiftBits << 6,
   shiftBits >> 2)

let pink: UInt32   = 0xCC6699
let redComponent   = (pink & 0xFF0000) >> 16
let greenComponent = (pink & 0x00FF00) >> 8
let blueComponent  =  pink & 0x0000FF
p2(redComponent, greenComponent, blueComponent)

// overflow operator オーバーフロー演算子

var potentialOverflow = Int16.max
// potentialOverflow += 1  // 実行時エラー

var unsignedOverflow = UInt8.max
print(unsignedOverflow)  // -> 255
unsignedOverflow = unsignedOverflow &+ 1
print(unsignedOverflow)  // -> 0

unsignedOverflow = UInt8.min
print(unsignedOverflow)  // -> 0
unsignedOverflow = unsignedOverflow &- 1
print(unsignedOverflow)  // -> 255

var signedOverflow = Int8.min
print(signedOverflow)  // -> -128
signedOverflow = signedOverflow &- 1
print(signedOverflow)  // -> 127
```

```
// 演算子の優先順位と結合方向は原則として C / Objective-C と同じだが、少し改善してある。

// operator methods 演算子メソッド

// 演算子のオーバーロード
/// 2次元のベクトルを表す構造体。
struct Vector2D {
    var x = 0.0, y = 0.0
}
extension Vector2D {
    static func + (left: Vector2D, right: Vector2D) -> Vector2D {
        return Vector2D(x: left.x + right.x,
                        y: left.y + right.y)
    }
}
let vector = Vector2D(x: 3.0, y: 1.0)
let anotherVector = Vector2D(x: 2.0, y: 4.0)
let combinedVector = vector + anotherVector
print(vector, anotherVector, combinedVector)

// 単項演算子
extension Vector2D {
    static prefix func - (vector: Vector2D) -> Vector2D {
        return Vector2D(x: -vector.x,
                        y: -vector.y)
    }
}
let positive = Vector2D(x: 3.0, y: 4.0)
let negative = -positive
let alsoPositive = -negative
print(positive, negative, alsoPositive)

// compound assignment operators 複合代入演算子
extension Vector2D {
    static func += (left: inout Vector2D, right: Vector2D) {
        left = left + right
    }
}
var original = Vector2D(x: 1.0, y: 2.0)
let vectorToAdd = Vector2D(x: 3.0, y: 4.0)
```

```
original += vectorToAdd
print(original, vectorToAdd)

// equivalence operators 等価演算子
extension Vector2D: Equatable {
    static func == (left: Vector2D, right: Vector2D) -> Bool {
        return (left.x == right.x) &&
               (left.y == right.y)
    }
}
let twoThree = Vector2D(x: 2.0, y: 3.0)
let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)
if twoThree == anotherTwoThree {
    print("These two vectors are equivalent.")  // 実行される
}

// 自動的な等価演算子を得られる場合もある。
struct Vector3D: Equatable {
    var x = 0.0, y = 0.0, z = 0.0
}
let twoThreeFour = Vector3D(x: 2.0, y: 3.0, z: 4.0)
let anotherTwoThreeFour = Vector3D(x: 2.0, y: 3.0, z: 4.0)
if twoThreeFour == anotherTwoThreeFour {
    print("These two vectors are also equivalent.")  // 実行される
}

// custom operators
prefix operator +++
extension Vector2D {
    static prefix func +++ (vector: inout Vector2D) -> Vector2D {
        vector += vector
        return vector
    }
}
var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
let afterDoubling = +++toBeDoubled
print(toBeDoubled, afterDoubling)

// precedence for custom infix operators カスタムな中置演算子の優先度
// 優先度は precedence group（優先度グループ）を使って指定する。
```

```
infix operator +-: AdditionPrecedence  // + や-と同じ優先度とする。
extension Vector2D {
    static func +- (left: Vector2D, right: Vector2D) -> Vector2D {
        return Vector2D(x: left.x + right.x,
                        y: left.y - right.y)
    }
}
let firstVector = Vector2D(x: 1.0, y: 2.0)
let secondVector = Vector2D(x: 3.0, y: 4.0)
let plusMinusVector = firstVector +- secondVector
print(firstVector, secondVector, plusMinusVector)

// 前置演算子と後置演算子について優先度を指定することはできない。
// しかし同時に使ったときには後置演算子が先に適用される。
```