# Learning Swift Language on Ubuntu:
# Deinitialization
# Optional Chaining
# Error Handling
# Type Casting
# Nested Types
# Extensions
# Protocols

## abstract 要旨

Protocol という章の@objc 属性に関する例示については、(「import Foundation」したとしても、) Ubuntu において正常に動かなかった。

## 4 種類のエラー

Optional の使い方について。エラーはすべて nil で流す感じで、開発時には assert() で落とせばよいのではないか。

https://github.com/apple/swift/blob/master/docs/ErrorHandlingRationale.rst

というものがあるらしい。エラーには 4 種類ある。simple domain error と recoverble error と universal error と logic failure である。simple domain error には Optional 型で nil 値を返し、recoverble error には例外を投げ、universal error は fatalError() で終了し、logic failure は precondition() すればいい。なおエラーの原因が 1 つではない場合に、recoverble error として例外を投げることが適している。原因が 1 つならば Optional が便利である。

## Optional についてのある考え方

こういう考え方ができるのではないだろうか。つまり、生じうるエラーの種類が 1 種類になるまで、処理を細かく関数らに分解する。すると、1 個の関数は 1 個のエラーを生じることになる。そのエラーを Optional 型の返り値 nil によって表現する。関数は、どのような入力に対して自らが nil を返すかを、コメントにおい

て明示する。ユーザはそれを読んでその関数を利用する。

返り値が Optional であることは、API とユーザの間のコミュニケーションであり、nil が返りうることについてユーザの注意を喚起しているにすぎない。ユーザは、決して nil が返らないように引数を処理してから関数に渡す。このようにして、関数が nil を返さないことは、API のコメントとユーザの思考によって確認され、機械的には確認されない。

ユーザは自らの責任において関数の返り値をすぐに bang (x!) する。nil を bang して実行時エラーになるなら、API の説明か自身の考えたロジックに欠陥があったということになる。1 つ 1 つの bang について、ユーザはそれが正当である理由をすぐ近くにコメントで説明する。「bang because ...」といった記述が並ぶことになる。

エラーを生じるうる複数の関数を利用する関数で生じうるエラーが 1 種類だとは考えられない。よって関数のエラーの種類を 1 種類にするとは、普遍的なまでの原則ではない。単に、とても重要な指針と見なすのみである。

ネットワークアクセスなど、いかにも多様なエラーが生じそうな場合には、例外によって呼び出し元とコミュニケーションする。そうではなく、上記の趣旨がうまく適合しない場合には、precondition() によってプログラムをエラー終了してしまう。そのようにして、正常系でないコードを各部分に最小に押し込めることによって、コードの簡潔さを保ち、総合的な生産性に寄与する。特に巨大ないし mission critical なプログラムについては別である。

## implementation

```
// deinitialization
// https://docs.swift.org/swift-book/LanguageGuide/Deinitialization.html

// deinitializer は deinit {}と書く。クラスにのみ使える。
// Automatic Reference Counting（ARC）という仕組みがある。
// deinit {}を自ら呼ぶことはできない。サブクラスの deinit {}が先に呼ばれる。

class Bank {
    static var coinsInBank = 10_000
    /// coins 単位だけお金を引き出す。ただし銀行にある以上には引き出せない。
    static func distribute(coins numberOfCoinsRequested: Int) -> Int {
        let numberOfCoinsToVend = min(numberOfCoinsRequested, coinsInBank)
        coinsInBank -= numberOfCoinsToVend  // 残高を減少する
        return numberOfCoinsToVend
    }
    /// coins 単位だけお金を預ける。
    static func receive(coins: Int) {
        print("Bank.receive()")
        coinsInBank += coins                // 残高を増加する
    }
```

```
}
class Player {
    var coinsInPurse: Int  /// 所持金
    init(coins: Int) {
        coinsInPurse = Bank.distribute(coins: coins)
    }
    func win(coins: Int) {
        coinsInPurse += Bank.distribute(coins: coins)
    }
    deinit {
        print("Player#deinit()")
        Bank.receive(coins: coinsInPurse)
    }
}

var playerOne: Player? = Player(coins: 100)
print("A new player has joined the game with \(playerOne!.coinsInPurse) coins")
print("There are now \(Bank.coinsInBank) coins left in the bank")

playerOne!.win(coins: 2_000)
print("PlayerOne won 2000 coins & now has \(playerOne!.coinsInPurse) coins")
print("The bank now only has \(Bank.coinsInBank) coins left")

playerOne = nil  // プレイヤーがゲームを離れる。参照カウントが 0 になる。
print("PlayerOne has left the game")
print("The bank now has \(Bank.coinsInBank) coins")
```

## implementation

```
// optional chaining
// https://docs.swift.org/swift-book/LanguageGuide/OptionalChaining.html

class Person {
    var residence: Residence?
}
class Residence {
    var numberOfRooms = 1
}
```

```
let john = Person()
// let roomCount = john.residence!.numberOfRooms  // 実行時エラー
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")  // 実行される
}
john.residence = Residence()
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")  // 実行される
} else {
    print("Unable to retrieve the number of rooms.")
}

// より階層化された場合を考える:
class Person1 {
    var residence: Residence1?  /// 住んでいる家
}
/// 住居
class Residence1 {
    var rooms = [Room1]()  /// この家にある部屋ら
    /// この家にある部屋の個数
    var numberOfRooms: Int {
        return rooms.count
    }
    /// 整数の添字によってこの家のそれぞれの部屋を表す
    subscript(i: Int) -> Room1 {
        get {
            return rooms[i]
        }
        set {
            rooms[i] = newValue
        }
    }
    /// 部屋の個数を出力する
    func printNumberOfRooms() {
        print("The number of rooms is \(numberOfRooms)")
    }
    var address: Address1?  /// この家の所在地
}
```

```
/// 家のそれぞれの部屋
class Room1 {
    let name: String  /// 部屋の名前
    init(name: String) { self.name = name }
}
/// 所在地
class Address1 {
    var buildingName: String?    /// 建物の名前
    var buildingNumber: String?  /// 建物の番号
    var street: String?          /// 通りの名前
    /// この所在地にある建物の識別子
    func buildingIdentifier() -> String? {
        if let buildingNumber = buildingNumber, let street = street {
            return "\(buildingNumber) \(street)"
        } else if buildingName != nil {
            return buildingName
        } else {
            return nil
        }
    }
}

let john1 = Person1()
if let roomCount = john1.residence?.numberOfRooms {  // residence は nil である
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")  // 実行される
}

let someAddress = Address1()
someAddress.buildingNumber = "29"
someAddress.street = "Acacia Road"
john1.residence?.address = someAddress  // residence は nil だから右辺は評価されない。

// 右辺が評価されていないことを確認する
/// 所在地を作って返す
func createAddress() -> Address1 {  // 実行されない
    print("Function was called.")   // 実行されない

    let someAddress = Address1()
```

```
    someAddress.buildingNumber = "29"
    someAddress.street = "Acacia Road"

    return someAddress
}
john1.residence?.address = createAddress()

// 返り値のないメソッドについての optional chaining:
if john1.residence?.printNumberOfRooms() != nil {  // 成功なら ()、失敗なら nil だ。
    print("It was possible to print the number of rooms.")
} else {
    print("It was not possible to print the number of rooms.")  // 実行される
}

// optional chaining への代入は Void?型の値を返すので、代入が成功したか判定できる。
if (john1.residence?.address = someAddress) != nil {  // residence は nil なので失敗する。
    print("It was possible to set the address.")
} else {
    print("It was not possible to set the address.")  // 実行される
}

// optional chaining を通じた添字へのアクセス:
if let firstRoomName = john1.residence?[0].name {  // residence は nil なので失敗する。
    print("The first room name is \(firstRoomName).")
} else {
    print("Unable to retrieve the first room name.")  // 実行される
}
john1.residence?[0] = Room1(name: "Bathroom")  // residence は nil なので代入できない。

let johnsHouse = Residence1()
johnsHouse.rooms.append(Room1(name: "Living Room"))
johnsHouse.rooms.append(Room1(name: "Kitchen"))
john1.residence = johnsHouse

if let firstRoomName = john1.residence?[0].name {  // 今や residence は nil ではない。
    print("The first room name is \(firstRoomName).")  // 実行される
} else {
    print("Unable to retrieve the first room name.")
}
```

```
// 添字の返り値が Optional 型の場合:
var testScores = ["Dave": [86, 82, 84], "Bev": [79, 94, 81]]
print(testScores)
testScores["Dave"]?[0] = 91   // 成功する
testScores["Bev"]?[0] += 1    // 成功する
testScores["Brian"]?[0] = 72  // キー Brian はないので代入は失敗する。
print(testScores)

// optional chaining を重ねても Optional は 1 層でありつづける。

if let johnsStreet = john1.residence?.address?.street {  // address == nil
    print("John's street name is \(johnsStreet).")
} else {
    print("Unable to retrieve the address.")  // 実行される
}

let johnsAddress = Address1()
johnsAddress.buildingName = "The Larches"
johnsAddress.street = "Laurel Street"
john1.residence?.address = johnsAddress

if let johnsStreet = john1.residence?.address?.street {
    print("John's street name is \(johnsStreet).")  // 実行される
} else {
    print("Unable to retrieve the address.")
}

if let buildingIdentifier = john1.residence?.address?.buildingIdentifier() {
    print("John's building identifier is \(buildingIdentifier).")
}

// メソッドの返り値も optional chaining できる。
if let beginsWithThe =
        john1.residence?.address?.buildingIdentifier()?.hasPrefix("The") {
    if beginsWithThe {
        print("John's building identifier begins with \"The\".")  // 実行される
    } else {
        print("John's building identifier does not begin with \"The\".")
    }
}
```

## implementation

```
// error handling
// https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html

// ゲームのなかの自動販売機のエラーを表す列挙型:
enum VendingMachineError: Error {
    case invalidSelection
    case insufficientFunds(coinsNeeded: Int)
    case outOfStock
}

// コインが 5 枚足りないことを表すエラー:
// throw VendingMachineError.insufficientFunds(coinsNeeded: 5)

// throws と書かれた関数は throwing function でありエラーを（外に）投げることができる。
// throws のない関数がエラーを投げたなら（中で）自ら処理せねばならない。

struct Item {
    var price: Int  /// 価格
    var count: Int  /// 個数
}
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  /// 投入されたコインの個数
    func vend(itemNamed name: String) throws {
        // guard 文を使う
        guard let item = inventory[name] else {  // name という名のアイテムはない
            throw VendingMachineError.invalidSelection
        }

        guard item.count > 0 else {  // item はもう 1 つもない
            throw VendingMachineError.outOfStock
        }
```

```
        guard item.price <= coinsDeposited else {  // item を売るにはコインが足りない
            throw VendingMachineError.insufficientFunds(
                    coinsNeeded: item.price - coinsDeposited)  // 不足分を算出する
        }
        coinsDeposited -= item.price  // 売った分のコインを減らす
        var newItem = item
        newItem.count -= 1
        inventory[name] = newItem
        print("Dispensing \(name)")   // name を販売する
    }
}

/// key の好きなスナックは value である。
let favoriteSnacks = [
    "Alice": "Chips",
    "Bob": "Licorice",
    "Eve": "Pretzels",
]
// エラーをさらに上に投げる関数:
/// person の好きなスナックを vendingMachine で購入する。
func buyFavoriteSnack(person: String, vendingMachine: VendingMachine) throws {
    let snackName = favoriteSnacks[person] ?? "Candy Bar"
    try vendingMachine.vend(itemNamed: snackName)
}

// throwing initializer
struct PurchasedSnack {
    let name: String
    init(name: String, vendingMachine: VendingMachine) throws {
        try vendingMachine.vend(itemNamed: name)
        self.name = name
    }
}

// catch する例:
var vendingMachine = VendingMachine()
vendingMachine.coinsDeposited = 8
// Alice は Chips が好きだが、Chips を買うにはコインが 2 個足りない。
do {
    try buyFavoriteSnack(person: "Alice", vendingMachine: vendingMachine)
```

```
    print("Success! Yum.")
} catch VendingMachineError.invalidSelection {
    print("Invalid Selection.")
} catch VendingMachineError.outOfStock {
    print("Out of Stock.")
} catch VendingMachineError.insufficientFunds(let coinsNeeded) {  // 実行される
    print("Insufficient funds. Please insert an additional \(coinsNeeded) coins.")
} catch {
    print("Unexpected error: \(error).")  // 暗黙なローカル変数 error
}

/// 栄養補給する
func nourish(with item: String) throws {
    do {
        try vendingMachine.vend(itemNamed: item)
    } catch is VendingMachineError {
        print("Invalid selection, out of stock, or not enough money.")
    }
}
do {
    try nourish(with: "Beet-Flavored Chips")  // 存在しない item
} catch {
    print("Unexpected non-vending-machine-related error: \(error)")
}

// try?によってエラーを Optional に変換できる。
// try!によってエラーを実行時エラーにして無視できる。

// エラーと無関係に defer 文を使ってもいい。
```

## implementation

```
// type casting
// https://docs.swift.org/swift-book/LanguageGuide/TypeCasting.html

/// デジタルメディアのライブラリの要素を表すクラス。
class MediaItem {
    var name: String
    init(name: String) {
```

```
        self.name = name
    }
}

/// 映画を表すクラス
class Movie: MediaItem {
    var director: String  /// 映画監督
    init(name: String, director: String) {
        self.director = director
        super.init(name: name)
    }
}

/// 歌を表すクラス
class Song: MediaItem {
    var artist: String  /// アーティスト
    init(name: String, artist: String) {
        self.artist = artist
        super.init(name: name)
    }
}

let library = [  // [MediaItem] 型へと型推論される。
    Movie(name: "Casablanca",              director: "Michael Curtiz"),
    Song (name: "Blue Suede Shoes",        artist: "Elvis Presley"),
    Movie(name: "Citizen Kane",            director: "Orson Welles"),
    Song (name: "The One And Only",        artist: "Chesney Hawkes"),
    Song (name: "Never Gonna Give You Up", artist: "Rick Astley")
]
// MediaItem 型の要素の元の型を扱うには downcast する必要がある。

// is (type check operator) によってあるインスタンスの型があるサブクラスかわかる。
var movieCount = 0  // 映画の個数
var songCount = 0   // 歌の個数
for item in library {
    if item is Movie {
        movieCount += 1
    } else if item is Song {
        songCount += 1
    }
```

```
}
print("Media library contains \(movieCount) movies and \(songCount) songs")

// as (type cast operator) によって downcast ができる。
for item in library {
    // optional binding を使う。
    if let movie = item as? Movie {
        print("Movie: \(movie.name), dir. \(movie.director)")
    } else if let song = item as? Song {
        print("Song: \(song.name), by \(song.artist)")
    }
}

// type casting for Any and AnyObject
// Any は何でも。AnyObject は任意のクラスを表す。
var things = [Any]()
things.append(0)                                                    // Int
things.append(0.0)                                                  // Double
things.append(42)                                                   // Int
things.append(3.14159)                                              // Double
things.append("hello")                                              // String
things.append((3.0, 5.0))                                           // (Double, Double)
things.append(Movie(name: "Ghostbusters", director: "Ivan Reitman"))  // Movie
things.append({ (name: String) -> String in "Hello, \(name)" })       // (String) -> String
print(things)
// for x in things { print(type(of: x)) }

// switch 文で型でマッチして処理を振り分ける。
for thing in things {
    switch thing {
    case 0 as Int:
        print("zero as an Int")
    case 0 as Double:
        print("zero as a Double")
    case let someInt as Int:
        print("an integer value of \(someInt)")
    case let someDouble as Double where someDouble > 0:
        print("a positive double value of \(someDouble)")
    case is Double:
        print("some other double value that I don't want to print")
```

```
    case let someString as String:
        print("a string value of \"\(someString)\"")
    case let (x, y) as (Double, Double):
        print("an (x, y) point at \(x), \(y)")
    case let movie as Movie:
        print("a movie called \(movie.name), dir. \(movie.director)")
    case let stringConverter as (String) -> String:
        print(stringConverter("Michael"))
    default:
        print("something else")
    }
}
```

## implementation

```
// nested types
// https://docs.swift.org/swift-book/LanguageGuide/NestedTypes.html

// ネストした列挙型を定義して利用する。
struct BlackjackCard {
    enum Suit: Character {
        case spades = "♠", hearts = "♡", diamonds = "♢", clubs = "♣"
    }
    enum Rank: Int {
        case two = 2, three, four, five, six, seven, eight, nine, ten
        case jack, queen, king, ace  // rawValue を持たない
        struct Values {              // さらにネストした構造体
            let first: Int, second: Int?  // 第２要素はないことがある。
        }
        var values: Values {
            switch self {
            case .ace:
                return Values(first: 1, second: 11)
            case .jack, .queen, .king:
                return Values(first: 10, second: nil)
            default:
                return Values(first: self.rawValue, second: nil)
            }
        }
```

```
    }
    let rank: Rank, suit: Suit
    var description: String {
        var output = "suit is \(suit.rawValue),"
        output += " value is \(rank.values.first)"
        if let second = rank.values.second {
            output += " or \(second)"
        }
        return output
    }
}
let theAceOfSpades = BlackjackCard(rank: .ace, suit: .spades)
print("theAceOfSpades: \(theAceOfSpades.description)")

// refering to nested types
let heartsSymbol = BlackjackCard.Suit.hearts.rawValue
print(heartsSymbol)
```

## implementation

```
// extensions
// https://docs.swift.org/swift-book/LanguageGuide/Extensions.html

// extension により computed プロパティを（Double 型に）追加する。
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  // リテラルに対して呼び出せる。
print("One inch is \(oneInch) meters")
let threeFeet = 3.ft
print("Three feet is \(threeFeet) meters")
let aMarathon = 42.km + 195.m
print("A marathon is \(aMarathon) meters long")

// ただし extension はストアドプロパティやプロパティオブザーバは追加できない。
```

```
// extension によって convenience initializer を追加できる。

struct Size {
    var width = 0.0, height = 0.0
}
struct Point {
    var x = 0.0, y = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
}

// default initializer を使う
let defaultRect = Rect()
// memberwise initializer を使う
let memberwiseRect = Rect(origin: Point(x: 2.0, y: 2.0),
                          size: Size(width: 5.0, height: 5.0))
// extension する
extension Rect {
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        // memberwise initializer を使う。
        self.init(origin: Point(x: originX, y: originY),
                  size: size)
    }
}

extension Int {
    /// self 回 task() する。
    func repetitions(task: () -> Void) {
        for _ in 0..<self {
            task()
        }
    }
    func repetitions1(_ task: @autoclosure () -> Void) {
        for _ in 0..<self {
            task()
```

```
        }
    }
}
3.repetitions {
    print("Hello!")
}
// 3.repetitions1(print("Hello1!"))
// 3.repetitions1({ print("Hello1!"); print(89)}())
// 3.repetitions { print("Hello!"); print(89)}

// 値を変更する extension
extension Int {
    mutating func square() {
        self = self * self
    }
}
var someInt = 3
print(someInt)
someInt.square()
print(someInt)

// 添字を extension する
extension Int {
    subscript(digitIndex: Int) -> Int {
        var decimalBase = 1
        for _ in 0..<digitIndex {
            decimalBase *= 10
        }
        return (self / decimalBase) % 10
    }
}
print(746381295[0], 746381295[1], 746381295[2], 746381295[8],
      746381295[9], 0746381295[9])

// nested type を extension する
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
        }
    }
}
/// numbers の各要素の Kind を文字列で出力する。
func printIntegerKinds(_ numbers: [Int]) {
    for number in numbers {
        switch number.kind {
        case .negative:
            print("- ", terminator: "")
        case .zero:
            print("0 ", terminator: "")
        case .positive:
            print("+ ", terminator: "")
        }
    }
    print("")
}
printIntegerKinds([3, 19, -27, 0, -6, 0, 7])
```

## implementation

```
// protocols
// https://docs.swift.org/swift-book/LanguageGuide/Protocols.html

import Foundation

// protocol に conform している（則っている)、という言い方をする。
// 継承する親クラスは則るプロトコルより先に記載する。

protocol FullyNamed {
    var fullName: String { get }
}
```

```
struct Person: FullyNamed {
    var fullName: String
}
let john = Person(fullName: "John Appleseed")
print(john.fullName)

class Starship: FullyNamed {
    var prefix: String?
    var name: String
    init(name: String, prefix: String? = nil) {
        self.name = name
        self.prefix = prefix
    }
    var fullName: String {
        return (prefix != nil ? prefix! + " " : "") + name
    }
}
var ncc1701 = Starship(name: "Enterprise", prefix: "USS")
print(ncc1701.fullName)

// メソッドのプロトコル
protocol RandomNumberGenerator {
    func random() -> Double
}
// 線形合同法による擬似乱数生成器
class LinearCongruentialGenerator: RandomNumberGenerator {
    var lastRandom = 42.0  // random seed
    let m = 139968.0
    let a = 3877.0
    let c = 29573.0
    func random() -> Double {
        lastRandom = ((lastRandom * a + c).truncatingRemainder(dividingBy:m))
        return lastRandom / m
    }
}
let generator = LinearCongruentialGenerator()
print("Here's a random number: \(generator.random())")
print("And another one: \(generator.random())")

// プロトコルのメソッドを mutating とすれば struct ないし enumeration も準拠できる。
```

```
protocol Togglable {
    mutating func toggle()
}
enum OnOffSwitch: Togglable {
    case off, on
    mutating func toggle() {
        switch self {
        case .off:
            self = .on
        case .on:
            self = .off
        }
    }
}
var lightSwitch = OnOffSwitch.off
print(lightSwitch)
lightSwitch.toggle()
print(lightSwitch)

// クラスの initializer があるプロトコルに準拠するときには required キーワードを書く。
// required 修飾された initializer は任意のサブクラスについて存在を保証することになる。
// final なクラスについてはサブクラスが存在しえないので required を書く必要はない。

// 親クラスの required な初期化子を override する意味では required のみ書けばよい。
// ただし親クラスの（designated）初期化子を override しプロトコルの初期化子を required
// するときには併記する。

// failable initializer requirements
// プロトコル規約 init?は init?でも init でも準拠できる。
// プロトコル規約 init は、init! (implicitly unwrapped failable initializer) または
// init で準拠できる。

// protocols as types プロトコルは型である
/// size 個の面を持つサイコロを表す。
class Dice {
    let sides: Int
    let generator: RandomNumberGenerator  // この型はプロトコルだ
    init(sides: Int, generator: RandomNumberGenerator) {
        self.sides = sides
        self.generator = generator
```

```
    }
    func roll() -> Int {
        // generator.random() は [0, 1)。
        // generator.random() * Double(sides) は [0, sides)。
        // Int(generator.random() * Double(sides)) は [0, sides-1]。+1 して [1, sides]。
        return Int(generator.random() * Double(sides)) + 1
    }
}
var d6 = Dice(sides: 6, generator: LinearCongruentialGenerator())
for _ in 1...5 {  // 5 回試行する
    print("Random dice roll is \(d6.roll())")
}

// delegation 移譲
protocol DiceGame {
    var dice: Dice { get }
    func play()
}
// protocol が class-only（クラス専用）であることを: AnyObject と書いて表す。
protocol DiceGameDelegate: AnyObject {
    func gameDidStart(_ game: DiceGame)
    func game(_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int)
    func gameDidEnd(_ game: DiceGame)
}
class SnakesAndLadders: DiceGame {
    let finalSquare = 25
    let dice = Dice(sides: 6, generator: LinearCongruentialGenerator())
    var square = 0
    var board: [Int]
    init() {
        board = Array(repeating: 0, count: finalSquare + 1)
        board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
        board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
    }
    weak var delegate: DiceGameDelegate?
    func play() {
        square = 0
        delegate?.gameDidStart(self)  // fail gracefully for nil
        gameLoop: while square != finalSquare {
            let diceRoll = dice.roll()
```

```
            delegate?.game(self, didStartNewTurnWithDiceRoll: diceRoll)
            switch square + diceRoll {
            case finalSquare:
                break gameLoop
            case let newSquare where newSquare > finalSquare:
                continue gameLoop
            default:
                square += diceRoll
                square += board[square]
            }
        }
        delegate?.gameDidEnd(self)
    }
}
class DiceGameTracker: DiceGameDelegate {  // 移譲クラス
    var numberOfTurns = 0
    func gameDidStart(_ game: DiceGame) {
        numberOfTurns = 0
        if game is SnakesAndLadders {  // type check
            print("Started a new game of Snakes and Ladders")
        }
        print("The game is using a \(game.dice.sides)-sided dice")
    }
    func game(_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int) {
        numberOfTurns += 1
        print("Rolled a \(diceRoll)")
    }
    func gameDidEnd(_ game: DiceGame) {
        print("The game lasted for \(numberOfTurns) turns")
    }
}
let tracker = DiceGameTracker()
let game = SnakesAndLadders()
game.delegate = tracker
game.play()

// adding protocol conformance with an extension
// extension によって既存の型をあるプロトコルに準拠させる
protocol TextRepresentable {
    var textualDescription: String { get }
```

```
}
extension Dice: TextRepresentable {
    var textualDescription: String {
        return "A \(sides)-sided dice"
    }
}
let d12 = Dice(sides: 12, generator: LinearCongruentialGenerator())
print(d12.textualDescription)
extension SnakesAndLadders: TextRepresentable {
    var textualDescription: String {
        return "A game of Snakes and Ladders with \(finalSquare) squares"
    }
}
print(game.textualDescription)

// conditionally conforming to a protocol
// generic where clause を使う。
// 要素が TextRepresentable な Array が TextRepresentable であることを定める。
extension Array: TextRepresentable where Element: TextRepresentable {
    var textualDescription: String {
        let itemsAsText = self.map { $0.textualDescription }
        return "[" + itemsAsText.joined(separator: ", ") + "]"
    }
}
let myDice = [d6, d12]  // 要素が TextRepresentable な Array
print(myDice.textualDescription)

// declaring protocol adoption with an extension
// そもそも protocol に準拠している実装を備えたクラス
struct Hamster {
    var name: String
    var textualDescription: String {
        return "A hamster named \(name)"
    }
}
// すでに実装があるので Hamster もまた TextRepresentable であることのみ定めればいい。
extension Hamster: TextRepresentable {}
let simonTheHamster = Hamster(name: "Simon")
// protocol 型で保持する:
let somethingTextRepresentable: TextRepresentable = simonTheHamster
```

```
print(somethingTextRepresentable.textualDescription)

// collectrions of protocol types
let things: [TextRepresentable] = [game, d12, simonTheHamster]
// let things = [game, d12, simonTheHamster]  // 型宣言しないとコンパイルエラー
for thing in things {
    print(thing.textualDescription)
}

// protocol inheritance プロトコルはプロトコルを継承できる
protocol PrettyTextRepresentable: TextRepresentable {
    var prettyTextualDescription: String { get }
}
extension SnakesAndLadders: PrettyTextRepresentable {
    var prettyTextualDescription: String {
        var output = textualDescription + ":\n"
        for index in 1...finalSquare {
            switch board[index] {
            case let ladder where ladder > 0:  // 梯子の根元
                output += "▲ "
            case let snake where snake < 0:    // 蛇の頭
                output += "▼ "
            default:
                output += "○ "
            }
        }
        return output
    }
}
print(game.prettyTextualDescription)

// class-only protocol クラス専用プロトコルは: AnyObject で表せる。
// Use a class-only protocol when the behavior defined by that protocol's
// requirements assumes or requires that a conforming type has reference
// semantics rather than value semantics.

// protocol composition プロトコル合成
protocol Named {
    var name: String { get }
}
```

```
protocol Aged {
    var age: Int { get }
}
struct Person1: Named, Aged {
    var name: String
    var age: Int
}
func wishHappyBirthday(to celebrator: Named & Aged) {  // プロトコル合成
    print("Happy birthday, \(celebrator.name), you're \(celebrator.age)!")
}
let birthdayPerson = Person1(name: "Malcolm", age: 21)
wishHappyBirthday(to: birthdayPerson)

class Location {
    var latitude: Double
    var longitude: Double
    init(latitude: Double, longitude: Double) {
        self.latitude = latitude
        self.longitude = longitude
    }
}
class City: Location, Named {
    var name: String
    init(name: String, latitude: Double, longitude: Double) {
        self.name = name
        super.init(latitude: latitude, longitude: longitude)
    }
}
func beginConcert(in location: Location & Named) {  // クラスとプロトコルの合成
    print("Hello, \(location.name)!")
}

let seattle = City(name: "Seattle", latitude: 47.6, longitude: -122.3)
beginConcert(in: seattle)

// checking for protocol conformance
// プロトコルについても is (type check operator) と as (type cast operator) が使える。
protocol HasArea {
    var area: Double { get }
}
```

```
// HasArea なクラスを 2 つ定義する。
class Circle: HasArea {
    let pi = 3.1415927  // 円周率
    var radius: Double  // 半径
    var area: Double { return pi * radius * radius }  // computed property
    init(radius: Double) { self.radius = radius }
}
class Country: HasArea {
    var area: Double  // stored property
    init(area: Double) { self.area = area }
}
class Animal {  // HasArea ではない
    var legs: Int
    init(legs: Int) { self.legs = legs }
}
let objects: [AnyObject] = [  // 任意のクラスのインスタンスを要素とする配列
    Circle(radius: 2.0),      // 面積は 12.5663708 単位
    Country(area: 243_610),   // United Kingdom is 243,610 km^2
    Animal(legs: 4)
]
for object in objects {
    if let objectWithArea = object as? HasArea {  // optional binding
        print("Area is \(objectWithArea.area)")
    } else {
        print("Something that doesn't have an area")
    }
}

// optional protocol requirements
// Objective-C との協調のために optional requirement という仕組みが存在する。
// Objective-C 系のクラスは、optional requirement なプロパティやメソッドを持てる。

// optional requirement はプロトコルそのものとともに@objc 属性で修飾する。
// @objc なプログラムに準拠できるのは、Objective-C のクラスを継承するクラスだけである。
@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) {  // コンパイルエラー
            count += amount
        } else if let amount = dataSource?.fixedIncrement {        // コンパイルエラー
            count += amount
        }
    }
}
class ThreeSource: NSObject, CounterDataSource {
    let fixedIncrement = 3
}
var counter = Counter()
counter.dataSource = ThreeSource()
for _ in 1...4 {
    counter.increment()
    print(counter.count)
}
*/

// Protocol Extensions プロトコルは extension できる。実装を追加する。
extension RandomNumberGenerator {
    func randomBool() -> Bool {
        return random() > 0.5
    }
}
let generator1 = LinearCongruentialGenerator()
print("Here's a random number: \(generator1.random())")
print("And here's a random Boolean: \(generator1.randomBool())")

// providing default implementations
// protocol を extension することでデフォルトの実装を提供できる。

// adding constraints to protocol extensions
// 要素が Equatable な Collection のみここで extension する。
extension Collection where Element: Equatable {
    func allEqual() -> Bool {
        for element in self {
            if element != self.first {
```

```
                return false  // 最初の要素に等しくない要素があれば allEqual() ではない。
            }
        }
        return true  // すべての要素が最初の要素に等しい、つまりすべての要素が等しい。
    }
}
let equalNumbers = [100, 100, 100, 100, 100]
let differentNumbers = [100, 100, 200, 100, 200]
print(equalNumbers.allEqual())      // true
print(differentNumbers.allEqual())  // false
```