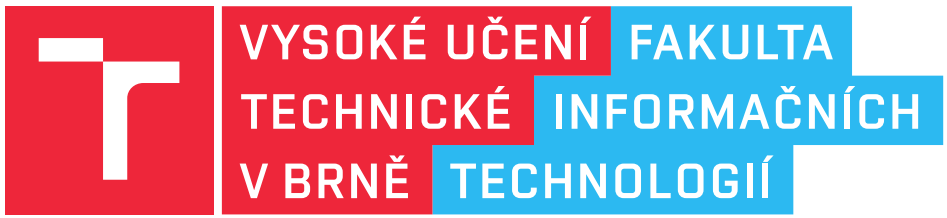
IZA – Swift II.
Advanced constructions

Filip Klembara

filip@klembara.pro

xklemb00@stud.fit.vutbr.cz



February 20, 2019

- **struct** without name

```swift
let a = ("Hello", 3, 5.67)

print(a.0) // Hello
print(a.1) // 3
let b: (hello: String, num: Int) = ("Hi!", 3)

print(b.hello) // Hi!

var mt: (hello: String, num: Int)
mt.number = 10
mt.hello = "Wazaaa"
print(mt) // (hello: "Wazaaa", num: 10)
```

- Closed Range Operator: `a...b`

```
0...5 // 0, 1, 2, 3, 4, 5

let names = ["Adam", "Bob", "Carol", "Denis"]

let n = names[0...2]
print(n) // ["Adam", "Bob", "Carol"]
```

- Half-Open Range Operator: `a..<b`

```
0..<5 // 0, 1, 2, 3, 4

let names = ["Adam", "Bob", "Carol", "Denis"]

let n = names[0..<2]
print(n) // ["Adam", "Bob"]
```

- One-Sided Ranges: `...b`, `a...`, `..<b`

```
..<3 // startIndex...2

let names = ["Adam", "Bob", "Carol", "Denis"]

let n1 = names[...2]
print(n1) // ["Adam", "Bob", "Carol"]
let n2 = names[2...]
print(n2) // ["Carol", "Denis"]

let n3 = names[..<2]
print(n3) // ["Adam", "Bob"]
```

- Is everything ok?

```
let a = [1, 2, 3]
let b = a[1...]
print(a)
print(b)
print(a[0])
print(b[0])
```

- Index is out of range... Why?

```
let a = [1, 2, 3]
let b = a[1...]
print(a) // [1, 2, 3]
print(b) // [2, 3]
print(a[0]) // 1
print(b[0]) // fatal error - out of range
```

- startIndex
- endIndex

```swift
let a = [1, 2, 3]
let b = a[1...]
print(a) // [1, 2, 3]
print(b) // [2, 3]
print(type(of: b))   // ArraySlice<Int>
print(a.startIndex) // 0
print(a.endIndex)   // 3
print(b.startIndex) // 1
print(b.endIndex)   // 3
```

- **for** element **in** collection {...}
- **for var** element **in** collection {...}

```
let numbers = [1, 2, 3]
for var number in numbers {
    number *= 2
    print(number)
}
// 2
// 4
// 6

let people = [("John", "Smith"),
    ("Marry", "Ryan")]
for (name, surname) in people {
    print("\(surname) \(name)")
}
// Smith John
// Ryan Marry
```

- What if collection in **`for-in`** is modified?

```
var numbers = [1, 2, 3]
for number in numbers {
    if number == 2 {
        numbers.append(4)
    }
    print(number)
}
print("numbers is \(numbers)"
```

- copy-on-write

```
var numbers = [1, 2, 3]
for number in numbers {
    if number == 2 {
        numbers.append(4)
    }
    print(number)
}
print("numbers is \(numbers)"

// 1
// 2
// 3
// numbers is [1, 2, 3, 4]
```

While, repeat-while

- `while condition {...}`
- `while let a = optionalValue {...}`
- `while var a = optionalValue {...}`
- `repeat {...} while condition`

C

```c
typedef enum {
    red,
    green,
    blue,
} Color;

Color clr = red;
```

Swift

```swift
enum Color {
    case red
    case green
    case blue
}

let clr1 = Color.red
let clr2: Color = .red
```

Enum - Asociated values

Swift

```
enum Barcode {
    case upc(Int, Int, Int, Int)
    case qrCode(String)
    case none
}

var productBarcode
    = Barcode.qrCode("*Le Code")

productBarcode = .none
```

C?

Enum - Asociated values

Swift

```swift
enum Barcode {
    case upc(Int, Int, Int, Int)
    case qrCode(String)
    case none
}

var productBarcode
    = Barcode.qrCode("*Le Code")

productBarcode = .none
```

C

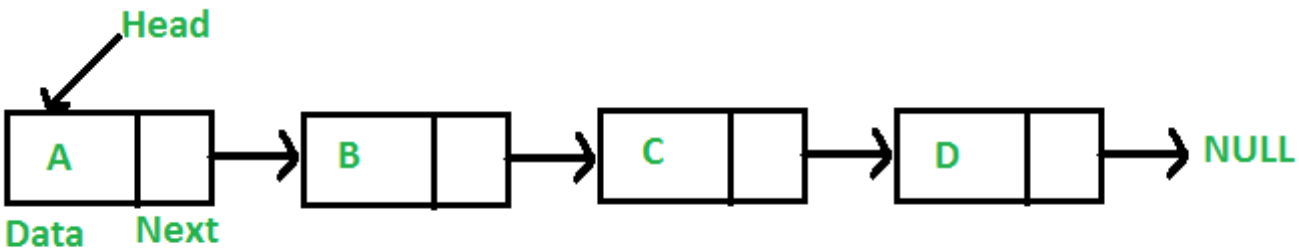
```c
typedef struct {
    enum {
        B_upc,
        B_qrCode,
        B_none
    } typeCase;
    union {
        int upc[4];
        char * qrCode;
    } value;
} Barcode;
```

- Explicitly (Implicitly) Assigned Raw Values

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

let _c = ASCIIControlCharacter(rawValue: "\t")
let c = _c! // Optional
print(c.rawValue) // prints \t
```

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

print(Planet.venus.rawValue) // 2

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

print(CompassPoint.south.rawValue) // "south"
```

- How to implement linked list?



- How to implement linked list?
- **`struct`** with **`Optional`**?

```
struct LinkedList {
    var next: LinkedList?
    let value: Int
}
```

- How to implement linked list?
- **`struct`** with **`Optional`**?

```
struct LinkedList {
    var next: LinkedList?
    let value: Int
}
```

Value type 'LinkedList' cannot have a stored property that recursively contains it

- How to implement linked list?
- **`class`** with **`Optional`**?
- But... is **`class`** what we really want?

```
class LinkedList {
    init(value: Int) {
        self.value = value
        next = nil
    }
    var next: LinkedList?
    let value: Int
}
```

- How to implement linked list?
- Solution: use **`indirect case`** in **`enum`**

```
enum LinkedList {
    indirect case node(value: Int,
                       next: LinkedList)
    case last(value: Int)
}
```

- No implicit fallthrough (implicit break)
- All cases must be handled (**`default`**)

```
switch variable {
    case value1:
        // respond to value1
    case value2, value3:
        // respond to value2 or value3
    default:
        // otherwise, do something else
}
```

```swift
let i = 5
var description = "variable i is"
switch i {
case 2, 3, 5, 7, 11, 13, 17, 19:
    description += " a prime number, and also"
    fallthrough
default:
    description += " an integer."
}
print(description)
// variable i is a prime number, and also an
   integer.
```

```swift
let somePoint = (1, 1)
switch somePoint {
case (0, 0):
    print("\(somePoint) is at the origin")
case (_, 0):
    print("\(somePoint) is on the x-axis")
case (0, _):
    print("\(somePoint) is on the y-axis")
case (-2...2, -2...2):
    print("\(somePoint) is inside the box")
default:
    print("\(somePoint) is outside of the box")
}
```

```
let code = Barcode.qrCode("Swift is fun!")

switch code {
    case .upc(_, _, let n3, let n4):
        print("\(n3), \(n4)")
    case .qrCode, .none:
        print("QR code or no code")
}
```

```swift
let yetAnotherPoint = (1, -1)
switch yetAnotherPoint {
case let (x, y) where x == y:
    print("\(x), \(y) is on the line x == y")
case let (x, y) where x == -y:
    print("\(x), \(y) is on the line x == -y")
case let (x, y):
    print("\(x), \(y) is just some arbitrary
        point")
}
// Prints "1, -1 is on the line x == -y"
```

- How to break loop?

```
while true {
    switch value {
    case 2:
        continue // we want to continue
    case 3:
        break    // we want to break loop
    default:
        break
    }
}
```

- Labeled statements

```
myWhile: while true {
    switch value {
    case 2:
        continue myWhile // continue `myWhile`
            loop
    case 3:
        break myWhile  // break `myWhile` loop
    default:
        break
    }
}
```

Deallocation

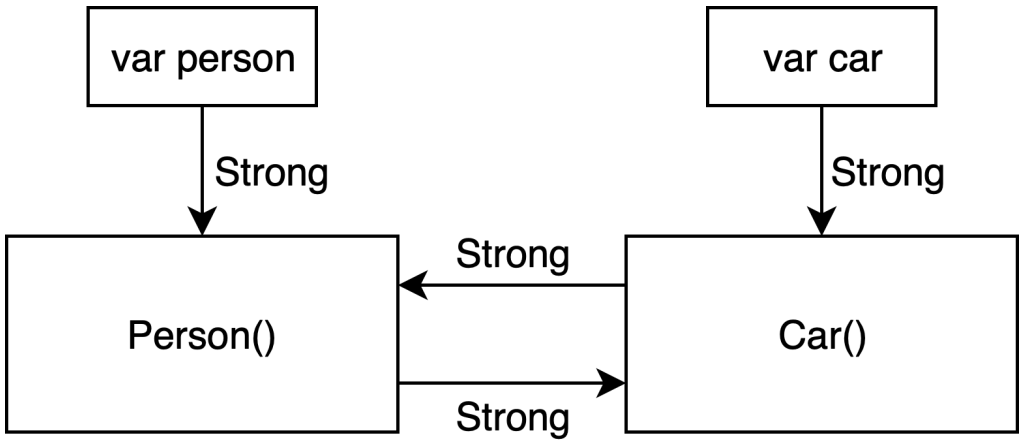
- C++ – delete
- Java – garbage collector
- Swift – ?

- ARC
- Strong vs weak reference

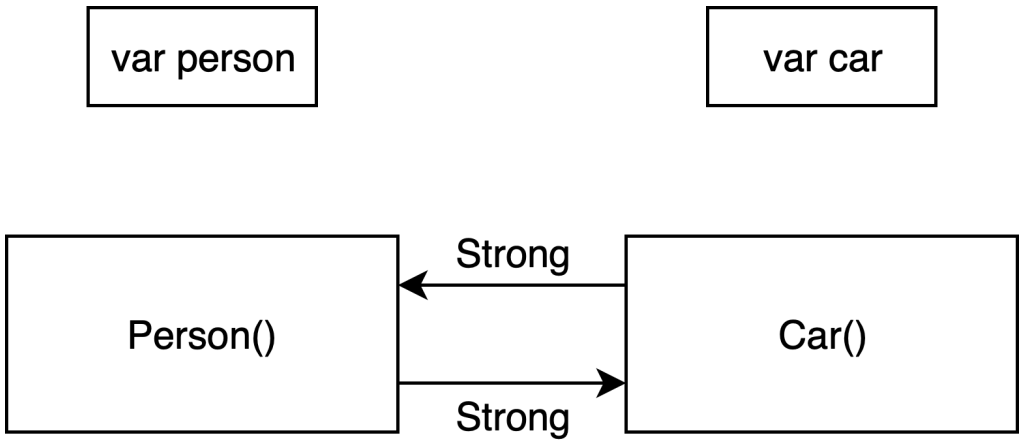
```
class A {
    init() {
        print("init")
    }
    deinit {
        print("deinit")
    }
}

var a1: A? = A()
var a2: A? = a1
a1 = nil
print("a1 == nil")
a2 = nil
```

```swift
class Person {
    var car: Car?
    deinit { print("deinit person") }
}

class Car {
    var owner: Person?
    deinit { print("deinit car") }
}

var person: Person? = Person()
var car: Car? = Car()
```

```
person!.car = car
car!.owner = person
```

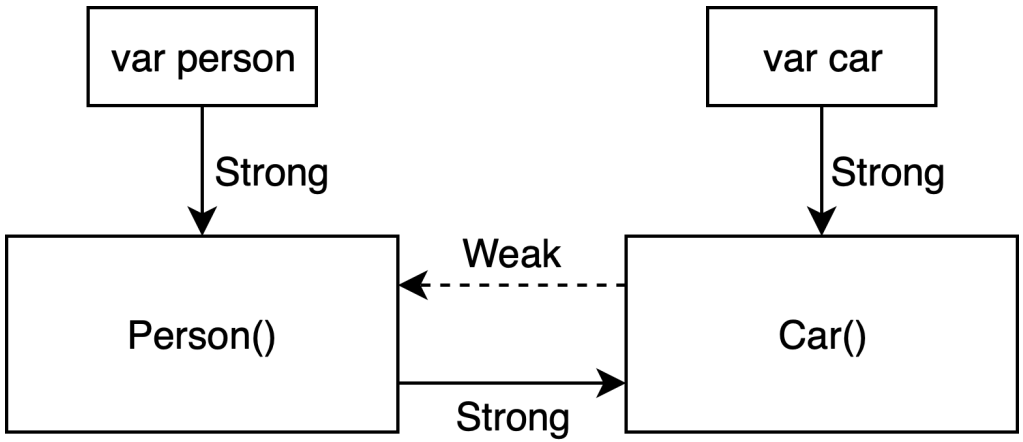


```
person = nil
car = nil
```

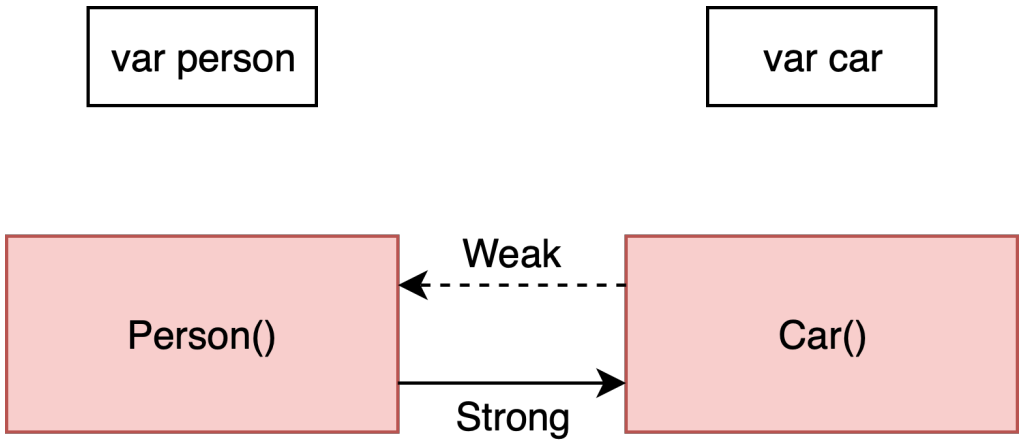


```swift
class Person {
    var car: Car?
    deinit { print("deinit person") }
}

class Car {
    // owner is weak reference
    weak var owner: Person?
    deinit { print("deinit person") }
}

var person: Person? = Person()
var car: Car? = Car()
```

```
person!.car = car
car!.owner = person
```



```
person = nil
car = nil
```



- **weak – Optional**

```
class A {
    func hello() {
        print("Hello")
    }
    deinit {
        print("deinit")
    }
}

var a: A? = A()
weak var weakA = a // Optional

a!.hello() // Hello
weakA!.hello() // Hello
a = nil
print(weakA) // nil
```

- **unowned**
- **unsafe unowned**

```
class A {
    var k = 4
    deinit { print("deinit") }
}

var a: A? = A()
// Not Optional, can be constant
unowned let unownedA = a!
unowned(unsafe) let nonoA = a! // BIG NO NO!

print(a!.k) // 4
print(unownedA.k) // 4
print(nonoA.k) // 4
a = nil
print(nonoA.k) // Maybe 4? Maybe fatal error?
print(unownedA.k) // fatal error
```

- Blocks with tuple as parameter and return value
- Closures can capture and store references
- Can infer types
- Can use shorthand arguments
- Trailing closure syntax

```
{ [references] (tupleWithArgs) -> returnType in
    statement1
    statement2
    ...
    statementn
    return value1
}
```

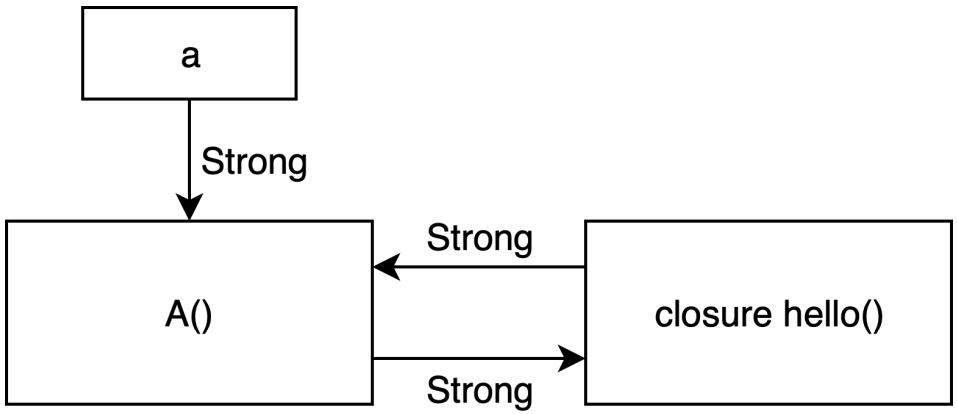
- Closures can capture and store references

```
var h = "Hello"
let hello = { (name: String) in
    print("\(h) \(name)")
}

hello("Tom") // Hello Tom
h = "Hi"
hello("John") // Hi John
```

- Closures can capture and store references
- Explicit `self` in class

```
class A {
    let name: String
    init (name: String ) { self.name = name }
    lazy var hello: () -> Void = {
        print("Hello \(self.name)")
    }

    deinit {
        print("deinit")
    }
}

var a: A? = A(name: "John")

a?.hello() // Hello John
a = nil
```

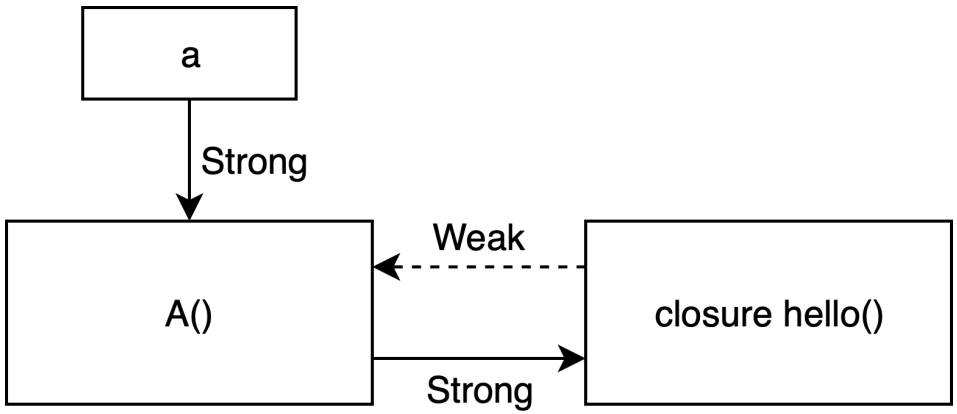
- Closures can capture and store references
- Explicit `self` in class
- `A` will never be deallocated



- Solution

```
class A {
    let name: String
    init (name: String ) { self.name = name }
    lazy var hello: () -> Void = {
        [unowned self] in

            print("Hello \(self.name)")
    }

    deinit { print("deinit") }
}

var a: A? = A(name: "John")

a?.hello() // Hello John
a = nil // deinit
```

- Closures can capture and store references
- Explicit `self` in class
- Reference type in closures



```swift
let arr = [3, 1, 2]

// func sorted(by areInIncreasingOrder:
//     (Int, Int) -> Bool) -> [Int]
arr.sorted(...)
```

```swift
let arr = [3, 1, 2]
// last argument is a closure
arr.sorted(by: {
    (left: Int, right: Int) -> Bool in

        return left > right
})

// return expression and type can be inherited
arr.sorted { left, right in left > right }

// automatically provided shorthand argument
   names
arr.sorted { $0 > $1 }

// function is a special case of closure, > is
   a function
arr.sorted(by: >)
```

- **private**
- **fileprivate**
- **internal** (**default**)
- **public**
- **open**
- (**set**) variants

```
fileprivate let defaultName = "Johny"

public struct Person {
    internal init(name: String? = nil) {
        self.name = name ?? defaultName }
    public private(set) var name: String
}

extension Person {
    mutating func set(name: String) {
        self.name = name
    }
}
```

Property observers

- **willSet**
- **didSet**

```
class Person {
    var hello: String? {
        willSet {
            print("hello will be \(newValue)")
        }
        didSet {
            print("new value is \(hello)")
            if let h = hello {
                print(h)
            }
        }
    }
}
```

- **get**
- **set**

```
class View {
    private var background: Background!
    init (...) { ... }
    var bgColor: Color {
        get {
            return background.color
        }
        set {
            background.setNew(color: newValue)
        }
    }
}
```

- **subscript**

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

```
enum JSON {
    case dictionary([String: JSON]), array([JSON])
    case int(Int), bool(Bool), double(Double)
    case string(String)

    subscript(key: String) -> JSON? {
        guard case .dictionary(let dic) = self else {
            return nil
        }
        return dic[key]
    }

    subscript(index: Int) -> JSON? {
        guard case .array(let arr) = self else {
            return nil
        }
        return arr[index]
    }
}

let json = JSON.array([JSON.dictionary([
    "person":JSON.dictionary(["name": "John"])])])
json[0]?["person"]?["name"]?.stringValue
```

```swift
@dynamicMemberLookup
enum JSON {
    case ...

    var stringValue: String? {
        guard case .string(let str) = self else {
            return nil
        }
        return str
    }

    subscript(key: String) -> JSON? { ... }

    subscript(index: Int) -> JSON? { ... }

    subscript(dynamicMember member: String) -> JSON? {
        return self[member]
    }
}

json[0]?["person"]?["name"]?.stringValue

json[0]?.person?.name?.stringValue // Optional "John"
```

Problem – Units in game

How to implement units for game where:

- Static defense – can attack, be damaged
- Warriors – can attack, be damaged, move
- Critters – can move, be damaged

Solution?

Requirements for:

- Methods (Java interface), Properties
- Inits

```
protocol Unit: AnyObject {
    var maxHP: Int { get }
    var hp: Int { get }
    var position: (Int, Int) { get set }
    init(position: (Int, Int))
}
protocol Movable: Unit {
    func move(at: (Int, Int)) -> Bool
}
protocol Damageable: Unit {
    var armor: Int { get set }
}
protocol Attacker: Unit {
    var damage: Int { get set }
    var range: Int { get }
}
```

- Extends **`class`**, **`struct`**, **`enum`**, **`protocol`**
- Extension can't contains stored properties or overrides

```
extension Unit {
    var isAlive: Bool { return hp > 0 }
}

extension Damageable {
    func take(damage: Int) {
        let dmg = damage - armor
        hp -= dmg > 0 ? dmg : 1
    }
}

extension Attacker {
    func attack(_ unit: Damageable) {
        // TODO: check range
        unit.take(damage: damage)
    }
}
```

```
class Tower: Damageable, Attacker {
    let maxHP = 200
    var hp = 200

    var armor = 2
    var damage = 7
    let range = 400
    var position: (Int, Int)

    required init(position: (Int, Int)) {
        self.position = position
    }
}
```

```
class Warrior: Damageable, Attacker, Movable {
    var armor = 1
    var damage = 5
    let range = 50
    let maxHP = 50
    var hp: Int
    var position: (Int, Int)

    required convenience init(position: (Int, Int)) {
        self.init(hp: 50, position: position)
    }

    init(hp: Int, position: (Int, Int)) {
        self.hp = hp
        self.position = position
    }

    func move(at newPosition: (Int, Int)) -> Bool {
        self.position = newPosition
        return true
    }
}
```

- **as**
- **as**?
- **as**!

```
let u1: Unit = Warrior(hp: 25, position: (1, 1))
let u2: Unit = Tower(position: (2, 2))
let u3 = Warrior(position: (1, 2))
let u4 = Warrior(position: (2, 1))

guard let attacker = u1 as? Attacker else {
    fatalError()
}
let target = u2 as! Damageable
attacker.attack(target)
u3.attack(target)
(u4 as Attacker).attack(target)
print(target.hp) // 191
```

- **typealias** Element
- **func** next() **->** Element?

```
struct Countdown: IteratorProtocol {
    var time: Element
    typealias Element = Int
    mutating func next() -> Element? {
        guard time >= 0 else {
            return nil
        }
        defer { time -= 1 }
        return time
    }
}

var countdown = Countdown(time: 5)
while let t = countdown.next() {
    print("t = \(t)")
}
```

Countdown using closure

```
func countdown(time: Int) -> () -> Int? {
    var t = time
    return {
        guard t >= 0 else {
            return nil
        }
        defer {
            time -= 1
        }
        return time
    }
}

let next = countdown(time: 5)
while let t = next() {
    print("t = \(t)")
}
```

- `for _ in Sequence {...}`

```
extension Countdown: Sequence { }

for i in Countdown(time: 3) {
    print("i = \(i)")
}
```

```
struct Stack {
    typealias Element = Int
    private var stack: [Element]
    init(stack: [Element]) {
        self.stack = stack.reversed()
    }

    mutating func push(_ element: Element) {
        stack.append(element)
    }
    mutating func pop() -> Element? {
        return stack.popLast()
    }
}
```

```
extension Stack: Collection {
    typealias Index = Int
    var startIndex: Index {
        return stack.startIndex }
    var endIndex: Index {
        return stack.endIndex }

    subscript(index: Index) -> Element {
        get {
            let i = stack.endIndex - index - 1
            return stack[i]
        }
    }

    func index(after i: Index) -> Index {
        return stack.index(after: i)
    }
}
```

Collection protocol implements:

- map, filter, comapctMap, ... methods
- first, isEmpty, count, ... properties

```
let stack = Stack(stack: [1, 2, 3, 4])
print(stack.first!)
print(stack[3])
print(stack.reversed())
print(stack.map { $0 * 2 })
```

Allows writing generic algorithms

```
func fiveTs<T>(value: T) -> [T] {
    return [T](repeating: value, count: 5)
}

func double<NumberType: Numeric>(number:
    NumberType) -> NumberType {
    return number * 2
}

let a = fiveTs(value: 3)
print(a)
let b = fiveTs(value: "Hello")
print(b)

let c: Float = double(number: 2.3)
print(c)
```

```
struct Stack<T> {
    typealias Element = T
...

let stack = Stack(stack: [1, 2, 3, 4])
print(stack.first!)
print(stack[3])
print(stack.reversed())
print(stack.map { $0 * 2 })

let stack2 = Stack(stack: ["A", "B", "C"])
print(stack2.first!)
print(stack2[2])
print(stack2.reversed())
print(stack2.map { $0.lowercased() })

let stack3 = Stack<Double>(stack: [])
```

```
enum LinkedList<T: Equatable> {
    indirect case node(value: T,
                       next: LinkedList)
    case last(value: T)
}
```

```swift
func divide(_ a: Double, by b: Double) ->
    Double? {
     guard b != 0 else {
         return nil
     }
     return a / b
}

print(divide(5, by: 2)) // Optional(2.5)
print(divide(5, by: 0)) // nil
```

```swift
enum MyError: Error {
    case divisionByZero
}

func divide(_ a: Double, by b: Double)
    throws -> Double {

        guard b != 0 else {
            throw MyError.divisionByZero
        }
        return a / b
}

print(try divide(5, by: 2)) // 2.5
print(try divide(5, by: 0)) // throws error
```

```
do {
    print(try divide(5, by: 2)) // 2.5
    print(try divide(5, by: 0))
} catch MyError.divisionByZero {
    print("division by zero")
} catch let error {
    print("Unknown error \(error)")
}
```

- **`try`**`?` – There can be an error but we don't care
- **`try`**`!` – We are absolutely sure there will be no error

```
print(try? divide(5, by: 2)) // Optional(2.5)
print(try? divide(5, by: 0))  // nil

print(try! divide(5, by: 2)) // 2.5
print(try! divide(5, by: 0))  // fatal error
```

Questions?