Swift II.

IZA, Martin Hrubý, FIT VUT, 2018

Literatura

- Skupina objc.io — knihy, články. Doporučuji.
- Fórum předmětu — sbírky zajímavých odkazů.
- Projekt do předmětu:
  - implementační — aplikace pro iOS/macOS/tvOS/watchOS,
  - implementační — aplikace na opensource verzi Swiftu,
  - studijní — odhalování temných stránek Swiftu. :)

Poznámka ke kolekcím

- Swiftové kolekce mají charakter struktur, tzn. kopírovací sémantika je záměrná.
- Každé navázání objektu kolekce do let/var značí kopii kolekce.
- Copy-on-write.
- Kolekce NS… z původního Foundation ponechávají sémantiku Obj-C kolekcí.
- Swift je jazyk pro rozlišení mutable/immutable.

let & var pro kolekce

```
// pole je mutable
var pole = [1, 2, 3]

// vznikne KOPIE, ktera je imutable
// tj. neni to predani reference na "pole"
let ipole = pole

// pole a ipole jsou odlisne objekty
pole[1] = 10

// tiskne [1, 10, 3]
print(pole)

// tiskne [1, 2, 3]
print(ipole)
```

NSMutableArray

```
import Foundation

var a = NSMutableArray()
var b = a
let c = a

a.add(1)
a.add(2)

print(a)
print(b)
print(c)

// je OK
c.add(4)
```

```
var a = NSMutableArray()
// NSArray.init(array: [Any]) !!!
let c = NSArray(array: a)
// typova konverze
let d = a as NSArray

a == c // true
a === c // false
a == d // true
a === d // true

a.add(1)
// nelze
c.add(2)
// nelze
d.add(3)
```

Přechod Obj-C / Swift kolekce

```
import Foundation

// je to [Any]
var a = NSMutableArray()
//
a.add("hello")
a.add(3)

// b ma byt typu [Int]
// havarie programu
var b = a as! [Int]
//
b.append(4)

// c je [Int]?
var c = a as? [Int]
// c je nil
c?.append(5)
```

Konstrukce Swiftu

- Řídicí (!!! ”i”) příkazy — if, while, switch…
- Procedury, funkční bloky, apod.
- Třídy, struktury a výčty (enum):
  - třídy (class) — hodnota (objekt) se předává referencí. RefCounter může být > 1.
  - struct/enum — hodnota se předává kopií hodnoty. RefCounter je max 1.

Řídicí příkazy

- if else — podmínka musí být typu BOOL.
- for in — iterování přes zadaný rozsah.
- while — obvyklý význam.
- switch — případy ”case” nevyžadují ”break”.
- Řídicí příkazy ve variantě ”let”.

if else

- if COND {stm} [else]
- if let (navázení optionals, případně enum).
- if cond1, cond2, … {}
- Bloky pro if-true, else musí být vždy v {} !!!
- guard — Zvláštní případ ”if”.

if let / var

- Konstrukce if let _a = a { STM } provede:
  - Pokus o navázání ”_a” na ”a” (optional unwrapping),
  - zavede do bloku { STM } objekt ”_a” s platnou hodnotou.
- if var _a = a …

guard

- guard COND else { …; return/throw/break }
- tělo ”else” musí obsahovat příkaz pro opuštění současného bloku kódu.
- guard let _x = x zavádí do bloku konstantu _x.
- Typicky testování vstupů funkce. Lze použít i pro ukončení cyklu apod.
- Dobrý návyk pro programování.

guard, příklad

```
//
func stringify(c: Int?) -> String {
    //
    guard let _c = c, _c > 0 else {
        print("NE"); return "null"
    }
    //
    return String(_c)
}

//
let num = "1234"

//
if var n = Int(num) {
    // "n" je lokalne v bloku
    // promennou
    n = n + 1
}
```

Testování vstupů funkcí

- Funkci (…) -> (…) berme jako kód, který se musí bránit před okolním kódem.
- Nadmíra guard-testů může protáhnout program.
- Objektové API — veřejné metody, skryté metody.
- Rozlišení metod vnějších a vnitřních (primitivn.)
- metoda(…), _metoda(…), __metoda(…).

for in

- for I in RANGE { stm }
  - I — je zde immutable (let)
  - RANGE — kolekce nebo rozsah
- Poznámka: for-in přes kolekci, která se v ”stm” modifikuje (add, delete).
  - Nikdy! Mazání: raději selekce do nového pole.
  - Ve Swiftu RANGE kopíruje (copy-on-write) hodnotu případné kolekce.

Rozsah

- Rozsah je objekt, tj. lze ho uložit do proměnné.
  - let rozsah = 1…1000; rozsah.contains(20) -> true
- a…b — od ”a” po ”b” včetně.
- a..<b — ”b” není zahrnuto.
- stride(from: to: by:)
- kolekce ve for-in generuje rozsah.

Vlastní iterátor

```
struct MujIterator : IteratorProtocol
{
    //
    let iterme: MojeTrida
    var n = 0

    //
    init(_ iterme: MojeTrida) {
        self.iterme = iterme
    }

    mutating func next() -> Int? {
        //
        let thisn = n
        //
        guard thisn < iterme.size
            else { return nil }
        //
        n += 1
        //
        return thisn
    }
}
```

```
class MojeTrida : Sequence {
    //
    var size : Int {
        //
        return 10
    }

    //
    func makeIterator() ->
       MujIterator {
        //
        return MujIterator(self);
    }
}

let mujv = MojeTrida()

for i in mujv {
    //
    print("Hodnota \(i)")
}
```

while

- while COND {}
- repeat {} while COND
- Kdo pamatuje pascalovský repeat-until?

```
//
let mujv = MojeTrida()
// iterator musi byt mutable
var mujIt = mujv.makeIterator()
//
while let i = mujIt.next(), … {
    //
}
```

switch

- C-kové ”break” je zde implicitní!
- Fallthrough explicitně.
- Patterns.

```
switch some value to consider {
case value 1:
    respond to value 1
case value 2,
     value 3:
    respond to value 2 or 3
default:
    otherwise, do something else
}
```

switch, pattern matching

```
let approximateCount = 62
let countedThings = "moons orbiting Saturn"
let naturalCount: String
switch approximateCount {
case 0:
    naturalCount = "no"
case 1..<5:
    naturalCount = "a few"
case 5..<12:
    naturalCount = "several"
case 12..<100:
    naturalCount = "dozens of"
case 100..<1000:
    naturalCount = "hundreds of"
default:
```

switch, pattern matching

```
let anotherPoint = (2, 0)

switch anotherPoint {

case (let x, 0):

    print("on the x-axis with an x value of \
           (x)")

case (0, let y):

    print("on the y-axis with a y value of \
           (y)")

case let (x, y):

    print("somewhere else at (\(x), \(y))")

}

// Prints "on the x-axis with an x value of 2"
```

switch, case, let, where

```
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"
```

Vyskakování z cyklu

- break, continue, fallthrough, return, throw
  - fallthrough — neprověřuje platnost dalších case!
- pojmenované bloky
  - labelName: { blok }
  - platí pro příkazy if, while, switch

fallthrough

```
let v = 1

// tiskne 1 a 2
switch v {
case 1:
    print("1"); fallthrough
case 2:
    print("2")
case 3:
    print("3")
default:
    print("def")
}
```

Pojmenované bloky

```
ZACATEK: while COND1 {
    //
    while COND2 {
        //
        while COND3 {
            //
            if cosiSpatne {
                break ZACATEK
            }

            //
            if necoJineho {
                continue ZACATEK
            }
        }
    }
}
```

Funkce

- Základní koncept strukturovaného programování: pod-programy, procedury, funkce.
- Funkce se volá, má parametry, návratový typ.
- Funkci vykonává *nějaké vlákno*.

Pojmenování parametrů funkce

- Swift zavádí dvojí pojmenování parametrů:
  - refCounting v ARC. Parametr funkce je lokální prom, zvyšuje refCount.
  - Vnitřní pro potřeby funkce — důvod je zřejmý.
  - Vnější pro potřeby volání funkce — pak tento *"label"* rozšiřuje terminologii v programu*, **rozšiřuje jméno funkce***.
  - label lze vynechat — "_"
- Tzn. při volání je povinné pojmenovávání parametrů.

Funkce

```
func adding(a: Int, b: Int) -> Int {
    //
    return a + b;
}

// parametry musi byt pojmenovane
let resa = adding(a: 2, b: 10)

func adding2(labelA a: Int, labelB b: Int) -> Int {
    //
    return a + b;
}

//
let resb = adding2(labelA: 2, labelB: 10)

func adding3(_ a: Int, _ b: Int) -> Int {
    //
    return a + b;
}

//
let resb = adding3(2, 10)
```

```swift
func delej(a: Int) { print("Prvni") }
func delej(necoJineho a: Int) { print("Druhy") }
func delej(_ a: Int) { print("Treti") }

//
delej(a: 1)
delej(necoJineho: 2)
delej(3)


  // nums - vystupuje interne jako [Int]
  // k zamysleni: neni lepsi deklarovat rovnou?
  func mdelej(_ nums: Int...) {
      //
      print("Mam \(nums.count) parametru")
  }

  //
  func mdelej(postaru nums: [Int]) {
      //
      print("Mam \(nums.count) parametru")
  }

  mdelej(1,2,3,4)
  mdelej(postaru: [1,2,3,4])
```

Názvosloví metod

- Používání label názvů parametrů má smysl především u konstruktoru (inicializačních metod) tříd.
- Pochází z doby Objective-C. Různé způsoby konstrukce objektu.
  - dateWithString, dateWithDate, dateWith…
  - Swift zavádí pouze ”init”, init(withDate,…), init(withString)

”inout” parametr

- Parametr funkce je součástí výstupu funkce.
- Držme se funkcionálního přístupu: vstupy a výstupy.

```
func delej(a: Int, b: inout Int) -> Int {
    //
    b = 1000
    //
    return 3;
}

// Musi byt var, let je error
var xy : Int = 10

//
delej(a: 33, b: &xy)
print(xy)
```

```swift
func delej(a: Int, b: Int) -> (vystup: Int, dalsi: Int) {
    //
    return (3, 1000);
}

//
print(delej(a: 33, b: 11))
```

Kde najdeme funkce

- Globální funkce.
- Metody tříd — instanční, třídní (statické).
- Properties tříd — gettery / settery.
- Bloky (closures).

Funkce jako hodnota

- Funkce je určena argumenty a návratovým typem.
- Funkci (tj. kód) lze předávat jako data.
- Syntaxe: (args) -> RET
  - (Int, Int) -> Int

```swift
typealias skupinaMetod = (Int, Int) -> Int

//
func solution(problem: Int) -> skupinaMetod {
    //
    func plus(a: Int, b: Int) -> Int { return a + b}
    //
    func minus(a: Int, b: Int) -> Int { return a - b}

    //
    switch problem {
    case 1:
        return plus
    default:
        return minus
    }
}

let mySol = solution(problem: 1)
// note: zmizla navesti parametru
let result = mySol(1, 2)
// vraci: 21
let result2 = solution(problem: 1)(1,20)
```

Bloky (closures)

- Funkční bloky (argumenty, return) zasaditelné do kontextu.
  - Bloky se předávají referencí.
  - Bloky zvyšují ref-count i svého kontextu.
- Lze je předávat jako data (do zpráv).
- Syntaxe: ”{” (args)->(return) in stm ”}”
- @escaping — lze předat blok a opustit funkci

Blok / Funkční objekt refCount

```
class TRIDA {
    //
    var zz : Int = 100

    //
    func hej() -> () -> Int {
        //
        func hejz() -> Int { return zz }

        //
        return hejz
    }
}

//
let t = TRIDA()
let h = t.hej()
// tiskne 100
let v = h()
//
t.zz = 123456
// tiskne 123456
let v2 = h()
```

Trailing closures

- Pokud je poslední argument funkce blokem, lze redukovat syntaxi volání funkce.

```
func gogo(closure: ()->Void) {
    print("Do something")
    closure()
}

// volam gogo
gogo(closure: { print("Hello") })

// volam gogo
gogo {
    print("Hello again")
}
```

@escaping

```
// nevyžaduje @escaping
func ge(v: ()->()) {
    return v();
}

class Dd {
    var x = 1

    func dosam() -> () -> () {
        ge {
            x = 2
        }
        // blok je "escaping"
        return { () -> () in
            self.x = 1
        }
    }
}

var p = Dd()
var d = p.dosam()
```

Použití bloků

```
// Data pro sort
var dt = [2,3,1,2,10,8,-1]
// Funkce pro porovnani
func fcmp(_ a:Int, b:Int) -> Bool { return a < b }
// Blok pro porovnani
let bcmp = { (a:Int, b:Int) -> Bool in return a < b }

//
let sortedf = dt.sorted(by: fcmp)
//
let sortedb = dt.sorted(by: bcmp)
let sortedb2 = dt.sorted(by: { (a:Int, b:Int) -> Bool in a < b })
let sortedb3 = dt.sorted { (a:Int, b:Int) -> Bool in a < b }
let sortedb4 = dt.sorted(by: { a,b in a < b })
let sortedb5 = dt.sorted(by: { $0 < $1 })
let sortedb6 = dt.sorted { $0 < $1 }
let sortedb7 = dt.sorted(by: <)
```

Enumeration

- Datový typ provádějící výčet možných hodnot.
- Hodnoty jsou typicky vyjádřeny identifikátorem.
- Pojetí C/C++: jsou to integer konstanty.

```
enum Seasons {
    case spring, summer
    case autumn
    case winter
}

let a = Seasons.spring
let b : Seasons = .summer
var c : Seasons = .autumn
//
c = .winter
```

```
enum Seasons : String {
    case spring = "Spring"
    case summer = "Summer"
}

let a = Seasons.spring
let aa = a.rawValue
let b = Seasons(rawValue: "Summer")
let c = Seasons(rawValue: "ddd") // nil
```

Parametrický enum

```
enum Message {
    // primitivni
    case ok, failed
    // parametricke
    case number(Int)
    case text(String)
    // parametricke s pojmenovanim
    case pack(text: String, value: Double, count: Int)
    // rekurzivni
    indirect case complex(message: Message, count: Int)
}

let m = Message.text("Hello")
let n = Message.pack(text: "Hello", value: 3.1, count: 10)
let o = Message.complex(message: .number(3), count: 10)
//
```

Zjišťování hodnoty enum

```
switch o {
case .ok, .failed:
    print("OK/failed")
case .number(let NX):
    print("Number \(NX)")
case .pack(let text, _, _):
    print("Text \(text)")
case let .complex(msg, cnt):
    print("Complex, \(cnt)")
default:
    print("DDD")
}
```

Testování enum

```
if case Message.ok = o {
    print("Je to OK")
}

if case Message.complex(let message, let count) = o {
    print("Je to Complex, cnt=\(count)")
}

if case let Message.text(txt) = m {
    print("TXT-Message \(txt)")
}
```

Extending enum

```
extension Message {
    //
    var descriptor : String {
        return "Message"
    }
    //
    func doSomething() {
        switch self {
        case .ok:
            print("OK")
        default:
            print("DDD")
        }
    }
    //
    mutating func makeOK() {
        self = .ok
    }
}

print(m.descriptor)
m.doSomething()
```

Optional je enum

```
enum MujOptional<TYPE> {
    //
    case none

    //
    case some(Wrapped : TYPE)
}

let xy = MujOptional<String>.none
let yz = MujOptional<String>.some(Wrapped: "ahoj")
```

Význam enum

- Výčet: hodnota může být z množiny (nekonečné).
- Fakticky se jedná o výčet odlišných struct1, struct2, struct3…
  - Polymorfní strukturovaná hodnota.
- V konceptu immutable hodnot je enum významný prvek stylu programování!

enum versus OOP

- OO modelování světa: koncepty auto, nákladní auto, osobní auto, … třídy a dědičnost.
- U triviálních objektů lze toto nahradit enum.
- Konvenční provedení (C/C++)? Union?

```
enum Auto {
    case prosteNejakeAuto
    case nakladniAuto(nosnost: Double, delka: Double)
    case osobniAuto(pocetMist: Int, objemKufru: Double)
}

extension Auto {
    //
}
```

Struktury a třídy

- struct a class — instanční proměnné, metody, *extensions*, subscript, init, protokoly.
- Rozdíly:
  - struct — předává se hodnotou (refCount==1)
  - class — dědičnost
  - dědičnost — implikuje polymorfizmus, přetypování
- životní cyklus objektu — konstrukce, život, destrukce. Struct pouze drží hodnotu.

extensions (class, struct, enum)

- Lze přidávat do existujících tříd/struktur/enum další metody (srovnejme s dědičností).
- Pouze metody (a vypočtené properties).
- Pochází už z Objective-C.

Instanční proměnné struktur

- Uložené hodnoty.
- Musí mít počáteční hodnotu (být inicializovány).
  - Objective-C: instanční proměnné a properties
  - Swift: pouze properties
- Struktura generuje implicitní init. Při inicializaci nutno specifikovat
- Optional value.

Instanciace struktury

```
struct STR {
    var x = 1
    var y = 2
}
//
let p = STR()

struct STR2 {
    let x : Int
    let y : String
}
//
let p2 = STR2(x: 2, y: "hello")


extension STR {
    var descriptor : String {
        return "x=\(x),y=\(y)"
    }
}

p.descriptor
```

```
struct STR3 {
    let x : Int
    var y : String
    //
    init(x:Int) {
        self.x = x
        self.y = "dd"
    }
}
//
let p3 = STR3(x: 2)
```

”self” struktury je immutable

- refCounter==1, tj. struct má pouze jednoho vlastníka. Struktury NELZE testovat na ===
- Modifikací property na struktuře fakticky konstruujeme novou (celou) hodnotu.
- Interně je ”self” struktury immutable.
  - mutating funkce.

mutating funkce

```
struct HH {
    var x : Int

    func modif() {
        // error, 'self' is immutable
        self.x = 1
    }

    mutating func modif2() {
        //
        self.x = 2
    }
}

//
let h = HH(x: 3)
var m = HH(x: 1)

// nelze, "h" je const
h.modif2()

//
m.modif2()
```

Properties

- Class/struct/enum mohou mít properties (uloženou, vyčíslovanou).
- Properties pracují v systému KVC a KVO.
- Jsou to var, let a lazyvar

```
class TRIDA {
    // nemusi byt ulozena hodnota
    let x = 1
    // hodnotu "let" lze nastavit pouze JEDNOU
    // bud zde = ... nebo v "init"
    let xx : Int

    // vypoctena property, pouze "var"
    // hodnota vznika AZ PO konstrukci
    // read-only, pouze getter
    var y : Int {
        //
        return 3
    }

    //
    var z : Int = 1234

    //
    init(setX x: Int) {
        //
        self.xx = x
    }
}

var p : TRIDA = TRIDA(setX: 3)
```

lazy var, inicializace pouze jednou

```
class TRIDA {
    //
    lazy var xy = 1

    //
    lazy var zz: Int = {
        // nejaky vypocet
        return 1234 + self.xy
    }()
}

//
var p = TRIDA()

// 1235, vola se inicializacni getter
print(p.zz)
// lazy var je mutable
p.zz = 0
// 0
print(p.zz)
```

```
class TRIDA {
    // primitivni hodnota
   (private)
    var _mojeData : String? = nil
    // getter & setter
    var mojeData : String {
        //
        if _mojeData == nil {
            _mojeData = "Ahoj"
        }
        //
        return _mojeData!
    }
    //
    func resetMojeData() {
        //
        self._mojeData = nil
    }
    //
    lazy var pokus: String! = {
        return "Ahoj, pokus"
    }()
}
```

```
var p = TRIDA()

// ok, ok
print(p.mojeData)
print(p.pokus)

//
p.resetMojeData()
p.pokus = nil

// ok, nil (lazyvar
// ne-reinicializuje)
print(p.mojeData)
print(p.pokus)
```

```
class TRIDA {
    //
    var _prop : String = ""
    //
    var prop : String {
        //
        get {
            //
            return "ahoj: \(_prop)"
        }
        //
        set(newValue) {
            //
            _prop = newValue
        }
    }
}

var p = TRIDA()
// "ahoj: "
print(p.prop)
//
p.prop = "svete"
// "ahoj: svete"
print(p.prop)
```

Key Value Observing

```
class TRIDA {
    //
    var prop : String = "ahoj" {
        //
        didSet {
            print("did \(oldValue)")
        }
        //
        willSet(newValue) {
            print("will \(newValue)")
        }
    }
}

var p = TRIDA()
// "ahoj"
print(p.prop)
// "will svete"
// "did ahoj"
p.prop = "svete"
// "svete"
print(p.prop)
```

Observer na proměnné

```
class TRIDA {
    //
    var prop : String = "ahoj"
}

var p = TRIDA() {
    didSet {
        print("Zmena hodnoty p!")
    }
}

// aktivuje se didSet
p = TRIDA()
```

```objc
@implementation TRIDAB
/* @synthesize numberb = _numberb; */

-(int) numberb {
    return _numberb;
}

-(void) setNumberb:(int)v {
    [self willChangeValueForKey: @"numberb"];
    _numberb = v;
    [self didChangeValueForKey: @"numberb"];
}
@end
```

Závěr

- Dostali jsme se až sem?
- Příště MVC…