Swift I.

IZA, Martin Hrubý, FIT VUT, 2018

Úvod

- Představen na WWDC 2014.
- Argumentem bylo zpřístupnění programování širší veřejnosti. Bezpečnější programování.
- Verze Swiftu — 1, 2, 3, 4.
- Na první pohled: Syntaktický wrap nad Objective C plus pár vylepšení konceptu.
- Koexistence s Objective-C, C. Bridges.

Obecné principy jazyků

- Bezpečnější programování — programátorské návyky.
- Dynamická alokace paměti. Vlastnictví objektů.
- Datové struktury.
- (Volání metod. Zasílání zpráv.)

Paměť aplikace

- V programu rozlišujeme data organizovaná v:
  - Datovém segmentu aplikace, statická, globální.
  - Dynamicky přidělovaná z haldy (heap, alloc, new, …) — low-memory warning, uvolňování paměti.
  - Dynamicky alokovaná na zásobníku.
- Bylo by dobré vědět, odkud se bere paměť pro data programu (specificky vztah zásobník/halda).

```
int *dejPole()
{
    int pole[] = {1,2,3,4};

    //
    return pole;
}

int *dejPoleJinak()
{
    static  int pole[] = {1,2,3,4};

    //
    return pole;
}

typedef std::vector<int> Intv;
//
Intv    dej() {
    Intv _a = {1,2,3,4,5};
    //
    return _a;
}
//
Intv res = dej();
```

```
int     Vypocet(DatovaStruktura dt, int idx)
{
    if (idx <= 0) return 0;

    //
    return expr(Vypocet(dt, idx-1));
}

int     Vypocetc(const DatovaStruktura &dt, int idx)
{
    if (idx <= 0) return 0;

    //
    return expr(Vypocetc(dt, idx-1));
}
```

Dynamická alokace paměti

- Proč dynamická alokace paměti:
  - Neznám předem rozsah problému a potřebnou paměť.
  - Chci **odložit** alokaci/inicializaci nějakého objektu.
- Implementace: heap/halda, new/delete.
  - Zamykání haldy. Více-vláknové programy.
  - Inicializace (nulování) získaného bloku.
  - Testování ”nezískání” bloku.
  - Rada: minimalizovat počet míst v programu s operací ”new”.
  - Horší je to s operací ”delete”…

Property, lazy vyhodnocení

```
@synthesize doc = _doc;

-(Document *) doc {
    //
    if (_doc == NULL) {
        _doc = [[Document alloc] init];
        [_doc readAndInit];
    }
    //
    return _doc;
}

//
self.doc.posliZpravu();
//
_doc = NULL;
```

Problém s vlastnictvím objektu

- Především: rozdíl mezi ”pointer” a ”reference”.
- Rozdíl:
  - Objekt A vlastní objekt B (silná/strong reference).
  - Objekt A referencuje objekt B (slabá/weak reference).
- Vlastník objektu B je zodpovědný za jeho dealokaci. Když končí objekt A, ukončí to i B.
- Problém: když má být vlastníků víc, předání vlastnictví.

Strom vlastnictví, objektová paměť

- A má vlastnit B,
- C má vlastnit B.
- Kdo vlastní A a C? Nadřazený objekt X.
  - X tedy bude vlastnit B. A a C budou pouze referencovat (weak).
- Velmi problematické: A vlastní B, B vlastní A.

Proměnná ”TRIDA *b”

- Vztah vznikající: TRIDA *b = [TRIDA alloc];
  - Data reference ”b” — paměťové místo. ”b” je ”držák reference”
  - Objekt třídy TRIDA — typicky z heapu.
- ”b” je opět v nějakém kontextu vyšší struktury:
  - Je instanční proměnnou objektu (nebo třídní proměnnou).
  - Je lokální proměnnou procedury (tj. na zásobníku).
  - Je globální proměnnou programu. Minimalizovat glob. data.
  - Je prvkem kolekce libovolného typu.

V C++, Obj-C, Swift, Java …

- V iOS-app typicky alokace NSOperation, NSTimer, nějakého UI-prvku…

```
-(void) metoda
{
    TRIDA *b = [[TRIDA alloc] init];
    // konfigurace "b"
    ...
    //
    return;
}
```

Reference counting

- Objekt B bude mít *int counter* čítač, kolikrát je referencován. NSObject.
- Referencování — inkrementace counter.
- DEreferencování — dekrement counter.
- Po dosažení counter==0, dealokace.
  - Implementace — okamžitá, odložená (pool).
  - Dealokace objektu — další časová režie.
- Pomohli jsme si?

Problémy ref-countingu.

- Manuální incr/decr čítače — zodpovědnost programátora (retain/release, autoreleasepool).
  - při volání, noveJmeno — inkrementuje ref-count?
- Paralelizmus — výlučnost přístupu na counter.
- Časová režie — incr/decr.

```
-(void) setJmeno: (NSString *)noveJmeno
{
    [noveJmeno retain];
    [_jmeno release];
    //
    _jmeno = noveJmeno;
}
```

Automatic RefCounting (ARC)

- V běžné iOS aplikaci můžeme problémy ref-countingu zanedbat. Pak je ARC jasná volba.
- Automatizuje retain/release a pool.
- Inkrementace:
  - Přiřazení reference,
  - předání ref jako argumentu volání metody (režie!).
- Dekrement:
  - Nulování reference.

ARC v C++

```
// wrapper nad pointerem
typedef std::shared_ptr<Trida> Trida_ptr;
typedef Trida *Trida_wptr;
//
Trida_ptr _ptr = std::make_shared<Trida>(...);

void metoda(const Trida_ptr &aTrida)
{
    //
    Trida_wptr _wp = &*aTrida;
}
```

Datové struktury v programech

- Kdo chce zvládnout programování, musí zvládnout (abstraktní) datové struktury.
- Obecně:
  - strukturované typy (struct, class, …),
  - kolekce (obvykle homogenní) — co je prvkem kolekce? Kolekce objektů typu A nebo kolekce referencí (různých). Rozdíl Objective-C a Swift.
  - Heterogenní pole (Obj-C) je odbytá struktura.
  - Diskuze: jak registrovat balík objektů různých typů?

Typologie kolekcí

- Pole (vector) — NSArray, NSMutableArray.
- Seznam (list) — rozdíly vector/list?
- Množina (set, unordered_set) — NSSet.
  - Hashvalue versus operátor <
- Slovník (map) — NSDictionary.
- Varianty: mutable / imutable.
  - Rozdíly v implementaci mutable/imutable pro typy kolekcí.

Práce s kolekcí

- Přidávání, odebírání. Vkládání.
- Vyhledávání — sekvenční, binární.
- Řazení — lze quicksort?.
- Iterování přes kolekci.
- Zjištění prvního/posledního prvku.
- Praxe:
  - Fáze budování kolekce, fáze čtení kolekce.
  - Pokud lze, konvertujte na NSArray/vec.

Koncept Swiftu

- Opět hybridní v datových typech.
- Lepší práce s primitivními typy — Int, Double, Bool
- Staví na konceptu ”hodnota” — tj. existující objekt.
  - Hodnotu lze obalit do ”optional value”.
  - !!! Takže už se nemluví o referenci na objekt (explicitní *)

Bezpečný kód se Swiftem

- Inicializace proměnných.
- Kontrola indexů do polí. Kontrola přetečení číselných typů.
- ”Optionals” — reference má hodnotu NULL explicitně.
- ARC.
- Další…

Strukturované typy

- class — třída, instanciací vzniká objekt (hodnota). Hodnota (objekt) se předává referencí.
- struct — struktura. Hodnota se při předávání kopíruje. Nelze vyjádřit explicitně pointer na strukturu (nízkoúrovňové programování).
  - UnsafeMutablePointer
- enum — výčet hodnot.

Srovnání s Objective C

- Swift je něco jako revize konceptu ObjC.
- ARC implicitně.
- Nelze NULL-ref poslat zprávu.
- Striktní typovanost.
- Templates.
- Gumovější syntaxe — rozhodně není zjednodušením. Středník.

let & var

- Deklarace pojmenované hodnoty s typem — instanční proměnná, lokální/globální proměnná
  - let C = B (— C je immutable, konstanta.)
  - var V = B (— V je mutable.)
- do ”C” už nelze dál zapisovat.
- Hodnota B nemusí být známá v době překladu (vyhodnocuje se dynamicky).

let a var deklarace

```
class TRIDA {
    //
    var muta = 2
    let konsta = 10
}

//
let p = TRIDA()
let k = 1
var v = 2

// lze
v = 10
p.muta = 3

// nelze
k = 10
p = nil
p = TRIDA()
```

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)
```

Držení hodnoty

- Pojem hodnota — proměnná váže nějakou hodnotu (objekt / struktura).
- Datový typ hodnoty.
- Konverze datového typu hodnoty — přetypování.
- Držení hodnoty.

Číselné typy

- Int, UInt, Double, Float. Kdo používá unsigned?
- Bool — true/false. Obj-C (YES/NO).
- Pozor: if, while apod. neuznává C-kovou konverzi číslo na Bool!

```
// chtel tim rict: a > 0 ???
// ono to vsak je a != 0, tj. a==-1 je TRUE
// tj. je-li na zacatku a==-1 => problem
while (a) {
    // !
    a--;
    // !!
    a = a - 2;
    // !!!
    a = a - x;
}
```

Konverze typů, čísla

```
// pokud lze provest implicitni konverzi, provede se
// vysledkem je hodnota typu Int
let a : Int = 12345567

// konverze overitelna v dobe prekladu
// presto je vystupem Int?
let b = Int("123")
// error
let dd = b + 1
let ddx = b! + 1 // ok

// vystupem je Int?, zde zjevne nil
let c = Int("ahoj")
```

Tuples (N-tice)

```
//
let a = ("Ahoj", 3, 5.67)

//
print(a.0)
print(a.1)

let b : (pozdrav: String, cislo: Int) = ("ahoj", 3)

//
print(b.pozdrav)

//
typealias mujTuple = (pozdrav: String, cislo: Int)

//
var mt : mujTuple

mt.cislo = 10
mt.pozdrav = "cau"
```

Tuples — co to je?

- Je to nepojmenovaná struktura.
- Interně je to implementované jako struktura.
- Proč tedy nepsat rovnou strukturu?!
  - Přiznané struct lze dát přístupové metody.

```
struct mujTupleS {
    let pozdrav : String
    let cislo : Int
}

let mts = mujTupleS(pozdrav:
"hello", cislo: 10)
```

```
extension mujTupleS {
    var description : String {
        return "\(pozdrav): \(cislo)"
    }
}
```

Optionals (wrapper)

- Swift staví na pojmu ”hodnota”, tj. reference na prokazatelně existující objekt.
- Někdy potřebujeme říct ”neinicializovaná hodnota”, avšak i takové sdělení musí být explicitní.
  - … nebo: hodnota bude, ale až po inicializaci objektu (IBOutlet).
- Optional: buď hodnota nebo nil. Je to ”enum”.
  - striktně vzato, nil je HODNOTA, specifická konstanta.

Deklarace optionals

- typ?, Int?, String?, …
- deklarace ”?” rozšiřuje obor hodnot o *nil*
  - var a : Int? = nil // ok
  - var b: Int = nil // error
- Testování hodnoty. Přístup.
  - deklarace navázání případné ne-nil hodnoty na jinou proměnnou (virtuálně)

Testování -? hodnoty

```
class TRIDA {
    var a : Int = 1
}

//
var o : TRIDA? = TRIDA()

// testovani
if o != nil {
    // zde mame stale "a", tj.
    // tiskne "Optional(TRIDA)"
    print(o)
}

// navazani.
if let _o = o {
    // sem se vstoupi, pokud o != nil
    // tiskne "(TRIDA)"
    print(_o)
}
```

Znovu: getter / setter

- o.a = 1234; Je volání setteru.
  - -(void) setA: (int) newValue {…}
- let x = o.a; Je volání getteru.
  - -(int) a { … }
- Co když je ”o” typu ”TRIDA?”

Optional chaining

```
class TRIDA {
    var a : Int = 1
}

//
var o : TRIDA? = TRIDA()

// navazani.
if let _o = o {
    //
    _o.a = 1234
}

// zkratka s testovanim "o"
o?.a = 5678

// zkratka s vynucenym pristup
// (hrozi havarie)
o!.a = 1111

// tiskne Optional(1111) !!!
print(o?.a)
```

Datový typ výrazu ”o?.a”

- Připomeňme: přistupujeme na getter/setter.
- o?.a = 1234 je volání setteru, pokud o != nil.
- o?.a — pokud o != nil, pak hodnota (getter), jinak nil. Celkově je to zase ”typ?”, tj. Int?
- o!.a — je hodnota nebo havárie programu.
- Pokud předáváme o?.a dále, tak to někdo jednou bude muset rozseknout…

Návyky s optionals…

- Pišme kód, který dokumentuje sám sebe.
- Zřetelně kódujme okolnosti (if…) a akce.

```
//
func udelej(xy: Int) {
    //
}

// error
// nelze navazat na "xy" funkce
udelej(xy: o?.a)

// podminene provedeni
if let _o = o {
    // _o je garantovana hodnota
    udelej(xy: _o.a)
}
```

```
// NE-swiftove programovani
if o != nil {
    //
    udelej(xy: o!.a)

// pripadne. Zoufalstvi.
    udelej(xy: (o?.a)!)
}
```

Metodická poznámka

- Testování vstupních parametrů funkce.
- Rozhraní objektu a jeho interní metody.
- Rozhraní testuje parametry, předává je po kontrole interním metodám:
- Moje konvence: metody, _metody, __metody

Řídicí příkazy

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

if else

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

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.

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.

while

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

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

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:
  - 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])
```

”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 ”}”

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()
```

Extrémní ukázka…
pro silné povahy

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

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

        //
        return hejz
    }
}
```

```
// t bude optional
var t : TRIDA? = TRIDA()
// vim vsak, ze hodnota existuje
let h = t!.hej()
// zapis do t.zz
t!.zz = 123456
// tiskne 123456
let v2 = h()
// likviduju objekt, refCount==0
t = nil
// drzim funkci objekt TRIDA.hej.hejz
// ktery referencuje objekt puvodne
// vazany TRIDA.zz
let v3 = h()
```

```
func ha() -> ( (Int)->(), () -> Int )
{
    var hodnota = 100

    //
    print("Hodnota je \(hodnota)")

    return ( { (a:Int) -> () in hodnota = a }, { () -> (Int) in return
hodnota })
}

//
let z = ha()
// 100
print(z.1())
//
z.0(1234)
// 1234
print(z.1())
let zz = ha()
// zcela novy objekt ha.hodnota, tiskne 100
print(zz.1())
```

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: <)
```

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")
}
```

```swift
// vola funkci/blok "closure"
func giveme(_ closure: () -> Int) -> Int { return closure() }
// metoda (napr. instancni)
func localone() -> Int
{
    // vznikne objekt, referencovan metodou
    let z = 10
    // blok "blk" zvysi ref-count "z"
    let blk = { () -> Int in return z }
    //
    return giveme(blk)
}

// metoda (napr. instancni)
func localtwo() -> () -> Int
{
    // vznikne objekt, referencovan metodou
    let z = 100
    // blok "blk" zvysi ref-count "z"
    let blk = { () -> Int in return z }
    //
    return blk
}

let val = localone()
let val2 = localtwo()
let val22 = val2()
```

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()
```

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

Instanční proměnné struktur

- Musí mít počáteční hodnotu (být inicializovány).
- 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)
```

Závěr

- Dostali jsme se až sem?
- Příště třídy…