

Swift I.

Základy

IZA, Martin Hrubý, FIT VUT, 2019

Úvod

- Jazyk Swift vytváří Apple pro programování svých zařízení, tj. "dělají ho pro sebe".
 - Existuje však open-source varianta (Linux).
 - Objective-C (+Foundation) žilo v izolaci.
- Swift je vysoce produktivní nástroj programování.
 - Lze převzít inspiraci pro vlastní styl programování.
 - Budujeme si programátorské návyky. To je cíl!

Objective-C versus Swift

- Objective-C tolerovalo:
 - NE-inicializaci proměnných (malloc()) beztak nuluje paměť).
 - Poslání zprávy NULL-referenci.
- Swift dělá pravý opak:
 - Překladač kontroluje inicializaci proměnných.
 - NULL (nil) je prvkem explicitního datového typu (optional).
 - Explicitní testy na NULL.
 - Swift usiluje o bezpečné programování. Mnoho projevů.

Ošetření NULL-reference

```
// Objective-C
// muze byt i NULL
TRIDA *obj = ziskejProMeObjekt();
// nezkouma se NULL
// obj==NULL => implicitne vyraz nabyva hodnoty 0
if ([obj posliZpravu] > 10) {
    // neco delej
}

// Swift
var *obj = ziskejProMeObjekt()
// testuju NNull obj
if let _obj = obj {
    // posilam zpravu overene referenci
    if (_obj.posliZpravu() > 10) {
        //
    }
}
// posilam pokusne zpravu...vizme dale
if let _result = obj?.posliZpravu(), _result > 10 {
    //
}
```

```
class TRIDAB {  
    // muze byt chyba pri prekladu  
    // zatim (!!!) chybi inicializace property "prom"  
    var prom : String  
}  
  
// C/C++, (TRIDA *) je datovy typ "ukazatel na objekt TRIDA"  
// tento typ v sobe __zahrnuje__ hodnotu NULL  
TRIDAB *objektTRIDA = nullptr;  
  
// nelze bez inicializace  
var objekt : TRIDAB;  
  
// nelze, datovy typ TRIDAB je "hodnota typu objekt TRIDAB"  
// tento datovy typ v sobe __NEzahrnuje__ hodnotu NULL  
objekt = nil
```

Prvky bezpečného programování

- Striktní typové kontroly. *Int není Bool!*

```
int a = cosi;
//
while (a /* >0 ???/ ) {
  //
  a = a - neco;
}
```

- Syntactic sugar pro různé konstrukce:

- Systém výjimek. try / throw metody a volání.
- @escaping — bloky / closures (bude později),
- optional — hodnota smí obsahovat NULL,
- override — dědičnost, přetěžování funkcí,
- mutating — explicitní zápis do *const*-hodnoty.

Základní fenomény jazyka

- Pojetí hodnoty a způsob jejího předávání.
- Konstantnost a proměnlivost — čeho.
- Pojetí strukturované informace.
- Pozn.: C++ versus Swift
 - jaký je v C++ rozdíl mezi *struct* a *class*?
 - jaký je v C++ význam *enum*?

struct / class / enum

- struct — strukturovaná data, která chci někomu předat. Záznam smí mít odvozené properties (v C++ — const metody). Dočasná data.
- class — OOP, dědičnost, objekty s delším životem, předávání zpráv, provázání objektů (objektová paměť).
- enum — výčet (obor) hodnot. (ve Swiftu strukturovaných).

Držení / evidence hodnoty

```
// globalni promenna
var globka = 10
var globPole = [1,2,3,4]

//
class TRIDA {
    // property tridy
    var propka = "Ahoj"

    //
    func metoda(arg:Int) {
        // lokalni promenna
        var lokka = "lokalni data"
    }
}
```

1) Inicialiace

- Vztah *Identifikator* přiřad' obsah. "*var prom = 10*"
- Swift klade důraz na to, aby identifikátory (proměnné) měly jasně určený obsah:
 - tj. usiluje o minimalizaci výskytu neinicializované hodnoty,
 - normálně by k tomu nemělo dojít...ale může (!/?).
- Překladač Swiftu si to bude vynucovat.
 - Bude vás to štvát, ale zvyknete si.
 - Rada: nesnažte se s tím chytračit, zanedbávat to...

Dogma č. 1

- Veškeré proměnné a properties (instanční proměnné) mají inicializovanou hodnotu.
- Properties mají "povolen odklad". Lze provést v `init()`.
- Inicializace `class/struct`.

```
// lze  
var a = 1  
let A = 2  
var b : Int = 3  
let B : Int = 3
```

```
// nelze, chybi pocatecni hodnota  
var a : Int  
let A : Int // !!!
```

2) Mutable / Immutable

- Objective-C, Swift.
- Mutable — věc je *modifikovatelná*.
- Immutable — věc je *NEmodifikovatelná*, tj. *konstantní*.
- Let / var deklarace:
 - proměnné / konstanty lokální (ve funkcích), globální.
 - properties u class / struct.

Proměnné a konstanty

- Deklarace prostřednictvím *let* a *var*.
 - `let A = 3 //` typově určí jako `Int`
 - `let B : Double = 3 //` explicitně určím typ `Double`
 - `var C = 4 // Int`
- `A`, `B` jsou *immutable* — v daném kontextu "`A`" představuje *konstantu* `Int(3)`.
- `C` je *mutable* — je *proměnná*.
- Optimalizace kódu překladačem.

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 kolekce

- Objective-C: NSArray / NSMutableArray.
 - Rozdílná implementace, odlišná čas. složitost přístupu!
 - Typicky: konverze NSMutableArray na NSArray!
- Je rozdíl "let A = [1,2,3]" a "var B = [1,2,3]".
 - A je immutable, B je mutable (přidávat / mazat / přepisovat).
- Optimalizace výsledného kódu:
 - A.count — nahraditelné Int(3)
 - A[0] — nahraditelné Int(1).

Dilema "kdy let / var"

- Implicitně vše "let", pokud není důvod modifikovat obsah.
 - let property mi "zakáže" si náhodou přepsat obsah.
- Výjimky:
 - Obsah je "živý", mění se často.
 - Obsah je zatím neznámý, doplní se později.
- Bude plynout z praktičtějších ukázek...

3) Předávání hodnoty

- Dvě varianty (předává se hodnota...):
 - *Referencovatelná* — platí pro objekty, tj. instance tříd (*class*).
Hodnota je reference na objekt. Předává se reference. refCount.
Pozn.: dále pak funkce a closures.
 - *NE-referencovatelná* — platí pro *vše ostatní*. Hodnota je obsah paměti. Předává se kopií obsahu. Nemá refCount. Představme si to **jako registr procesoru počítače**.
- Kdy se předává hodnota:
 - Přiřazením (let/ var). Zapsáním do kolekce.
 - Předáním jako argumentu volání funkce. Speciálně: *inout*.

Primitivní datové typy

- Int, Double, Bool (true / false). String.
 - Jsou implementovány jako *struct*. Mají metody.
 - Předávají se hodnotou, tj. kopírují se.
- Primitivně (syntax) však lze zadávat konstanty:
 - 1, 2, 3.14, ...
 - "ahoj"
 - true, false (v Objective-C: YES/NO).

Předání hodnoty kopií...

```
var x = 42
var y = x
//
y += 1
```

- Nikoho nepřekvapí, že hodnota "x" se kopíruje do proměnné "y".
- "Int(42)" není referencovatelný objekt.
- Stejně je to se strukturami "struct/enum".
 - Struct je prostě hodnota, jenom je strukturovaná.
 - Překladač preferuje alokovat *struct* ze zásobníku! Uklízení paměti. Super-rychlá alokace/dealokace. No refCounting!

```
//  
class TRIDA {  
    //  
}  
  
//  
struct STR {  
    //  
}  
  
//  
func funkceA(obj: TRIDA, str: STR) {  
    // obj zvysuje refCount a predava se referenci  
    // str je KOPIE hodnoty  
}  
  
//  
let obj = TRIDA()  
let str = STR()  
//  
let obj2 = obj // refCount++  
let str2 = str // kopie obsahu  
  
// refCount++, kopie obsahu  
funkceA(obj: obj, str: str)
```

Důsledky pro ne-referencovatelné

- **Nelze získat ukazatel** na ne-referencovatelnou hodnotu.
 - Máme garanci, že *struct* jsou "objekty s refCountem=1".
- *struct* je pouze obalení několika *properties*.
- "Potřebuju ti říct 5 čísel, tak ti je zabalím do struktury a tu ti pošlu".
 - Co když jsou *properties* referencovatelné typy? **Advanced...**

Předávání hodnoty `Array<ELEM>`

- **Array je struct.** Copy sémantika.
 - Dále všechny kolekce včetně `String`.
- **Důsledky:**
 - předávám-li pole, pak by se kopíroval obsah paměti.
 - provozně neefektivní — vede na **copy-on-write**.
 - technická implementace kolekce: datový obsah, metadata.

```
// mutable array
var pole : [Int] = []

// inArray je kopie
func statistika(_ inArray: [Int]) {
    //
}

//
pole.append(1)
pole.append(contentsOf: [2,3,4])
pole.count

//
statistika(pole)

// !!! iterator provadi kopii pole
for i in pole {
    //
}
}
```

Kdy struct, kdy class?

- Pokud má být kus paměti referencován ze dvou míst (sdílená paměť vláken), pak *class*. Lock!
- Kolekce struktur / referencí. Sorting atd.
- Trvám na identitě (pak *class*) nebo na informaci (*struct*).
- Je informace příliš velká? Nechci kopírovat MB
- V dalších komplexnějších příkladech...

Copy-sémantika má i nevýhody

```
//  
class TRIDA {}  
  
//  
struct MojeDATA {  
    //  
    let cislo: Int  
    let pole: [Int]  
    let jmeno: String  
    let objekt: TRIDA  
}  
  
func necoDelej(inp: MojeDATA) {  
    // kopie inp.cislo  
    // refCount++ interne v inp.pole  
    // refCount++ interne v inp.jmeno  
    // refCount++ inp.objekt  
}
```

Strukturované datové typy

- class, struct, enum — třída, struktura, výčet.
- Mohou mít metody.
- class umožňuje dědičnost.
- class / struct mohou mít *uložené* properties.
- class / struct — odlišně dogmatická inicializace.
- class — mutable self. struct / enum — immutable

Demo class / struct

```
// class/struct
class TRIDA {
    //
    let propertyConst = 10
    var propertyVar : String

    //
    func metoda(parametr:Int) -> Int {
        //
        return parametr;
    }

    //
    init(initVal: String) {
        // self.
        propertyVar = initVal;
    }
}

//
let objekt = TRIDA(initVal: "Ahoj")
```

Inicializace class / struct

- class / struct musí mít "těsně po konstrukci objektu" všechny properties s inicializovanou hodnotou.
 - To kontroluje překladač.
 - Počáteční hodnota nebo explicitní *init()*, včetně "let" properties. Více *init()* -> všechny musí inicializovat vše.
- struct — překladač připouští / generuje implicitní konstruktor.

Implicitní konstruktor struct

```
//  
struct Person {  
    //  
    let age : Int  
    //  
    let name : String  
}  
  
//  
let person = Person(age: 42, name: "Petr")
```

- Takto se smí deklarovat a instancovat pouze struct.
- V případě class chyba při překladu, chybí *init(.)*.

Mutability of self

- *self/this* (C++).
- *self* třídy je implicitně *mutable*.
- *self* struktury je **IMPLICITNĚ** *immutable*.
 - nelze totiž určit, zda-li je obsah struktury držen jako *let* nebo jako *var*. Je nutné zajistit konstantnost hodnoty.
 - lze explicitně povolit modifikátorem *mutating* pro funkci.

struct je strukturovaná hodnota

- Je třeba zdůrazňovat, že STR() není objekt!
- "A" reprezentuje obsah struktury a je "let".
- koncept struct nutí přemýšlet, kdy class a kdy struct...

```
//  
struct STR {  
    //  
    var prom : Int = 1  
}
```

```
//  
let A = STR()  
var B = STR()
```

```
// nelze  
A.prom = 10  
// lze  
B.prom = 20
```

```

// self je mutable
class TRIDA {
    //
    var a = 3
    let b = "Ahoj"

    //
    func zmense() {
        //
        self.a = 1234
    }
}

```

```

// self je immutable
struct STR {
    //
    var a = 3
    let b = "Ahoj"

    //
    func zmense() {
        // error
        self.a = 1234
    }

    //
    mutating func zmense2() {
        //
        self.a = 5678
    }
}

var a = STR()
let b = STR()
//
a.zmense2() // lze
b.zmense2() // nelze, error

```


Úvod do konstrukce funkcí

- Funkce je pojmenovaný blok kódu, který:
 - má vstupní argumenty,
 - má možnost vracet hodnotu,
 - je umístěn v nějakém kontextu (globální, metoda třídy, vnořená metoda).
- Časem budeme s funkcemi pracovat jako s daty.
- Hlavička funkce umožňuje dvojí jméno arg:
 - pro vnitřní použití, pro vnější použití (label argumentu).

Pojmenovávání argumentů funkcí

- `func funkce(label name: Type) -> RetType {}`
- `func funkce(name: Type) -> RetType {}`
- `func funkce(_ name: Type) -> RetType {}`
- Při volání je nutno explicitně pojmenovat argumenty:
 - buď explicitním "label",
 - nebo implicitním jménem "name",
 - nebo explicitně NEpojmenovat argument.
- **argumenty jsou z pohledu funkce konstanty (let).**

```
// argument "a" je pro vnitrni i vnejsi pouziti
func funkceA(a:Int, s:String) -> String {
    //
    return "\ (a) je \ (s)"
}

// pri volani je NUTNO pojmenovat argumenty
let resultA = funkceA(a: 10, s: "Ahoj")

// labelProA je vnejsi jmeno argumentu,
// "a" je vnitrni jmeno argumentu
func funkceB(labelProA a: Int) -> Int {
    //
    return a + 2;
}

//
let resultB = funkceB(labelProA: 1)

//
func funkceC(_ innerName: Int) -> Int {
    //
    return innerName;
}

print(funkceC(10))
```

- Použití explicitního "label" nám rozšiřuje jmenný prostor v programu.
- Typicky u inicializátoru/konstruktoru init()

```
// label rozširuje jmeno metody
struct Distance {
    //
    let distInM : Double
    //
    init(meters value: Double) {
        //
        distInM = value
    }
    //
    init(yards value: Double) {
        //
        distInM = value * 0.9144
    }
}
//
let d1 = Distance(meters: 10)
let d2 = Distance(yards: 20)
```

Optional<Type>

- *var obj = TRIDA() // class TRIDA {};*
- Nelze teď přiřadit: *obj = nil;*
- Datový typ *obj* je "instance TRIDA". Hodnota *nil* tam nepatří.
- Potřebujeme vyjádřit "necht' *obj* je objekt-TRIDA nebo *nil*". Chceme rozšířit obor hodnot *obj*.

Optional<Type>

- Nad typem TRIDA zbudujeme nadřazený typ Optional<TRIDA>, syntaxe: *TRIDA?*
- Proměnná "var prom : TRIDA?" nabývá hodnot buď:
 - nil,
 - nebo reference na objekt TRIDA.
- Optional je vlastně *enum* (později).
- Říkáme tomu "wrapped value".

Demo, optional

```
//  
class TRIDA {  
    //  
}  
  
// !!! je implicitne = nil  
var prom : TRIDA?  
  
// instancuji a referencuji objekt  
prom = TRIDA()  
  
// de-referencuji  
prom = nil
```

- Sláva, můžu "ne-inicializovat" proměnnou.
 - ...ona je inicializovaná hodnotou *nil*...
 - Má to však svoje důsledky... :(

```
//  
class TRIDA {  
    //  
    var attr : Int = 3  
}
```

- `var prom: TRIDA? = TRIDA()`
- Zdůrazněme: `prom` není typu `TRIDA`
- Hodnotu typu `TRIDA` dostaneme buď:
 - unwrapping...
 - Násilnou konverzí pomocí operátoru `!!`, tj. `prom!!`, v C++ je to unární `*`, popř. `->`.
 - Testem s pokusem navázat obsah `prom` na hodnotu `TRIDA`, tj. `if let...`, `guard let...`

Operátor !

- Operátor "!" nazvěme:
 - "jsem si jistý sám sebou",
 - "prubnu to, v nejhorším ta appka slítne",
 - "nikdy jsem se seg-faultu nebál".
 - "tu hodnotu jsem tam dal / testoval, tak tam musí být"
- *prom!* konvertuje hodnotu z TRIDA? na TRIDA,
 - nebo havaruje program...
 - rada: nikdy nad proměnnou nadřazené úrovně deklarace...

`if let _obj = obj {}`

- Větvení programu příkazem `if`, jehož testovaná část je (tzv. *optional binding*):
 - `true` — pokud je `obj != nil`, pak jako vedlejší efekt kopíruje hodnotu do `_obj`, což nám **zvyšuje refCount na objekt!**
 - `false` — `obj` je `nil`.
- V těle příkazu `if` pro `true` je lokálně platná konstanta `_obj`.
 - zdůrazněme: nová proměnná / konstanta.

```
//  
class TRIDA {  
    //  
    var attr : Int = 3  
}  
  
//  
var prom : TRIDA? = TRIDA()  
  
// jsem drsnak  
prom!.attr = 1  
  
// drsnak, co testuje  
if prom != nil {  
    // ... a tak doufa, ze tu stale je  
    prom!.attr = 2  
}  
  
// if let konstrukce  
if let _prom = prom {  
    //  
    _prom.attr = 3  
}
```

Operátor “??”

- Syntaxe: výrazPodmínka ?? hodnota
 - Typicky: let a = neco ?? value
- Sémantika:
 - je-li *výrazPodmínka* not-nil, pak výsledná hodnota výrazu je *výrazPodmínka*,
 - jinak *hodnota*.

Optional chaining, *prom?.attr*

- *prom* je typu *TRIDA*?
 - property "*attr*", je *Int*
- Tečková notace jako v Objective-C. Getter / Setter
- *prom?.attr* — zkratka pro "je-li *prom* nenulové, pak volej getter *attr*, jinak vrať *nil*".
 - Zcela evidentně výraz končí buď hodnotou *Int* nebo *nil*, tj. je ve výsledku typu *Int*?
 - *prom?.attr = 100* je podmíněné volání setteru.

Optional se nám propaguje dál...

- *obj?.a* je zkratka pro *volání* getteru, pokud je *obj* *ne-NULL*. Lze libovolně řetězit (*obj?.a?.b?.c*)
- *obj* je typu TRIDA?
- *obj!.a* je Int
- *obj?.a* je Int?
- *obj!.b* je Int?
- *obj!.b!* je Int (a krom toho velké zlo...)

```
//  
class TRIDA {  
    //  
    var a : Int = 10  
  
    //  
    var b : Int?  
}  
//  
var obj : TRIDA? // = nil
```

Další projevy ?/!

- Operátor `as`, `as?`, `as!` — typová konverze.
 - `as` — staticky zjevná možnost konverze.
 - `as?` — dynamicky se zkusí, popř. vrací `nil`.
 - `as!` — dynamicky se zkusí, popř. havaruje.
- Konstrukce `try`, `try?`, `try!` — vstup do bloku `try`.
- `init?(...)` — inicializace objektu s možností vracet `nil`.

init?()

```
//  
class TRIDA {  
    // bude inicializovan v init()  
    var value: Int  
    // podmienka konstrukcie/inicializace objektu  
    init?(_ withValue: Int) {  
        //  
        if withValue < 0 { return nil }  
        //  
        self.value = withValue  
    }  
}  
  
//  
var o1 : TRIDA? = TRIDA(10)  
// _o2 je potvrzena hodnota  
if let _o2 = TRIDA(3) {  
    //  
    print("Kvalitni objekt: \(_o2.value)")  
}
```


Příkaz *guard*, motivační demo

```
//  
func soucet(param1: Int?, param2: Int?) -> Int? {  
  //  
  if let _param1 = param1 {  
    //  
    if let _param2 = param2 {  
      //  
      return _param1 + _param2;  
    }  
  }  
  
  // nebo lepe...  
  if let _param1 = param1, let _param2 = param2 {  
    //  
    return _param1 + _param2;  
  }  
  
  //  
  return nil;  
}
```

```
// takove ne-Swiftove...
func soucet(param1: Int?, param2: Int?) -> Int? {
    //
    if (param1 == nil || param2 == nil) {
        //
        return nil
    }

    //
    return param1! + param2!
}
```

Použití strážce...

```
// konstrukce guard  
func soucet(param1: Int?, param2: Int?) -> Int? {  
    //  
    guard let _p1 = param1, let _p2 = param2 else { return nil }  
  
    //  
    return _p1 + _p2  
}
```

```
// konstrukce guard  
func soucetPos(param1: Int?, param2: Int?) -> Int? {  
    //  
    guard  
        let _p1 = param1,  
        let _p2 = param2,  
        _p1 > 0, _p2 > 0  
        else { return nil }  
  
    //  
    return _p1 + _p2  
}
```

```
soucetPos(param1: 2, param2: nil)
```

Konstrukce guard

- guard COND else { OutOfHere }
- OutOfHere — libovolný kód, který ukončí daný kontext (opustí funkci).
 - return, throw, break, continue, ...
- Oblíbená konstrukce pro test argumentů funkce, zavedení **unwrapped values** ve funkci apod.

class / struct init(), shrnutí

- Dogma č.2: konstruktor `init()` musí zajistit inicializaci všech properties struct / class.
- Struktury mají možnost implicitního konstrukturu.
- Lze si vyhádat i zde *výjimku*:
 - properties typu *var Name: Type!*
 - proměnná *Name* pak je typu *Type* a nemusí se inicializovat.
 - *let Name: Type!* je nejspíš blbost...

```
//  
class TRIDA {  
  //  
  let a = "je proste const"  
  
  // musim dodati v init()  
  let b : Int // ... a dal pak bude const  
  var c : Int // ... a dal pak bude promenna  
  
  // je implicitne := nil, casem dodam obsah  
  var d : String?  
  // nema pocatecni hodnotu, protoze ted nelze zjistit  
  var e : String!  
  
  //  
  init(withB: Int, withC: Int) {  
    //  
    b = withB; c = withC  
  }  
}
```

Vypočtené property

```
//  
class TRIDA {  
    // ulozena property  
    var a = 3  
    // vypoctena (read-only) property, fakticky funkce...  
    var lepsiA : Int { return a * 2 }  
    // privatni datova/ulozena property  
    private var _sg: String = ""  
    // vypoctena property s explicitnim setterem  
    // !!! mozno i pro "struct", mutating  
    var sg : String {  
        //  
        get { return "ahoj:\(_sg)" }  
        set { _sg = newValue }  
    }  
    // vypoctena/ulozena property  
    // vyhodnocuje se pouze jednou  
    lazy var dokument: String = {  
        // nejaka cinnost...  
        // nacti soubor...  
        return "Ahoj\(a),\(\lepsiA)"  
    }()  
}
```

Sledování property (observers)

```
import Foundation
//
class TRIDA {
    //
    var jmeno : String = "Martin" {
        //
        didSet(newJmeno) {
            //
            print("Chteji \(jmeno) prejmenovat na \(newJmeno)")
        }
        //
        didSet {
            //
            print("Driv \(oldValue), ted \(jmeno)")
        }
    }
}

// NSObject, protokol KeyValue Observing
@objc var obsProperty: String = "Ahoj"
}
//
var p = TRIDA()

// didSet, setter, didSet
p.jmeno = "Petr"
```


Dědičnost tříd

```
class TRIDA_A {  
    //  
    let a : Int  
    //  
    func metoda() -> Int { return a }  
    //  
    init(withA: Int) {  
        //  
        a = withA  
    }  
}  
//  
class TRIDA_B : TRIDA_A {  
    //  
    let b : Int  
    //  
    override func metoda() -> Int { return a + b }  
    //  
    init(withA: Int, andB: Int) {  
        //  
        b = andB;  
        // az nakonec...  
        super.init(withA: withA)  
    }  
}
```

Kolekce ve standardu Swiftu

- Array [ELEM], Set, Dictionary [key:value]
 - Set, Dictionary — klíč musí implementovat Hashable.
- Jsou to šablonové struktury (struct!).
 - Generické programování ve Swiftu. Extensions.
- let / var — immutable / mutable.
- Typově vždy homogenní.

Závěr

- Základy Swiftu.
- Příště:
 - Řídicí konstrukce — cykly, iterování přes kolekce, switch.
 - Protokoly.
 - Extensions.
 - Enum.