Swift I.: Úvod do jazyka

Programování zařízení Apple (IZA)

Martin Hrubý, FIT VUT, 2023/24

Úvod

Jazyk Swift je v zásadě obvyklý moderní programovací jazyk obsahující:

- globální funkce a proměnné/konstanty
- třídy (OOP), struktury, výčtový typ (enum)
- příkazy: přiřazení, if-else, while, for, switch
- skoky: continue, break, return
- exceptions

Takže na první pohled *nic extra nového*.

Připomenutí: Terminologie OOP

Třídy (a struktury) mají:

- instanční a třídní atributy (data)
- instanční a třídní metody (funkce/procedury)
- konstruktor a destruktor (metody)

V jazyce Swift (a Objective-C) nad to zavádíme pojem *Property*

- uložená property
- vypočtená property

Uložená property (data)

```
class TRIDA {
    // uložené property
    let a = 10
    var b: String = "pocatecni hodnota"

    // vypočtená property
    var nejakeCislo: Int { return a * 20 }

    // abstrakce nad property b
    var jmeno: String {
        get { b }
        set { b = newValue }
    }
}
```

Property je atribut, na který se přistupuje voláním get/set přístupových metod. Rozšířená funkcionalita (KVC, KVO).

Uložená property (data)

Obecně připouštíme, že se při get/set přístupu na property "může stát něco dalšího" nad rámec primitivní operace.

Budu stále opakovat: pozor, kterým vláknem na objekt přistupujeme.

- CoreData - get/set provádí přístup na disk
- UIKit - get/set zasahují do GUI
- podobně SwiftUI
- obecně rozesílání zpráv napříč aplikací

Property v "tečkové notaci"

```
//
let objekt = TRIDA()

// volám getter
print(objekt.a)

// setter "b", setter "jmeno"
objekt.b = "ahoj"
objekt.jmeno = "pepa"
```

Property je:

- primárně - get/set nad nějakým jménem
- sekundárně - uložení dat ve strukturovaném typu

Property v pojetí protokolů

```
protocol HavingAName {
    // požaduji getter nad jménem "name"
    var name: String { get }
}

extension HavingAName {
    // každý implementátor HavingAName má
    // (libovolně implementovaný) get pro "name"
    func sayHello() {
        print("Hello", name)
    }
}
```

Property v pojetí protokolů

```
struct Neco: HavingAName {
    // mám tedy implicitní get/set,
    // tj. i ten požadovaný get
    var name: String
}

struct NecoJineho: HavingAName {
    // chce get (name->String), dostane get
    var name: String { return "me" }
}

struct NecoZaseJineho: HavingAName {
    // tohle už ne...
    func name() -> String { return "asdf" }
}
```

Primitivní observing property

Tradice z Objective-C.

```
class TRIDA {
    //
    var hodnota: Int {
        //
        didSet {
            // newValue, oldValue
            print("Zmenili mi hodnotu")
        }
    }
}
```

Data

(Kvazi-)primitivní datové typy:

- Bool - není "céčkové" přetypování všeho na Bool.
- Int, Double - explicitní konverze.
- String.

Konstrukce datové položky (lokální, globální, property):

- let - položka je nadále *immutable* (neměnná)
- var - položka je nadále *mutable*

```
let a = 129
var b = "ahoj"
let pravda = true
```

let/var deklarace

```
let NAME [:type] = initValue
```

Dogma č. 1: Každá deklarace vyžaduje počáteční hodnotu.

Výjimky:

- ta se dodá v init(...) struct/class
- explicitní typ je Optional<Type>, **Type?**
- explicitní typ je **Type!**

Kompilátor Swiftu obsahuje *typovou inferenci*, tj automaticky let/var přiřadí datový typ z výrazu počáteční hodnoty.

let deklarace (konstanta)

```
let cislo = 10 // bude Int
let jineCislo: Double = 3
let jesteJine = 3.14 // bude Double
let vypocteno = nejakaFunce(argumenty volani)
```

Jsou to immutable prvky, tj nelze je přepsat.

```
cislo = 1234 // error
```

Konvence: Je přirozené a normální mít v programu konstanty.

Proměnné (var) musí mít zdůvodnění.

Swift se rád tváří jako *funkcionální jazyk*.

- hodnotu spíše trasnformujeme na jinou hodnotu (map),
- než bychom hodnotu modifikovali in-situ.

var deklarace (proměnná)

```
var cislo = 1
var nejakeCislo: Int? // počátek implicitně [=nil]
```

Proměnné jsou *mutable*, tj smím dále změnit hodnotu:

```
cislo = 1234
nejakeCislo = 50

// Je typu Optional<Int>, tj zahrnuje i nil
nejakeCislo = nil
```

Programové konstrukce: *if-...-else*

```
if <Hodnota-Bool> {
    // then ...
} [ else { ... } ]
```

Logické operátory:

- &&, ||
- <=, <, >, apod...

Typické AND operátorem ",":

```
// cosi && necoJineho ...
if cosi, necoJineho, aJeste {

}
```

Optional (wrapped) typ

Swift pracuje s pojmem **hodnota** a trvá na tom, aby byla vždy za všech okolností u atributu **nastavena**.

Pokud není přesto známa, pak musí být atribut označen jako *Optional*.

```
enum Optional<Type> {
    case value(Type)
    case nil
}

//
var optInt: Int? // implicitně := nil
```

V *optInt* hodnota **JE**, jenom je **nil**. Musíme její přítomnost testovat.

Conditional unwrapping

```
//
var optInt: Int? // implicitně := nil

// optional unwrapping
if let platnaHodnota = optInt {
    //
    print(platnaHodnota)
}
```

Konstrukce `let CIL = neco?` je predikát, který uspěje (je `true`), pokud lze rozbalit obsah `neco` na platnou hodnotu.

Důležité: unwrapping provede **kopii hodnoty** a uloží ji do `CIL`.
V těle "if" máme celou dobu garanci platnosti hodnoty `platnaHodnota`.

Conditional unwrapping, kopie hodnoty

```
//
var optInt: Int? // implicitně := nil

// optional unwrapping
if let platnaHodnota = optInt {
    // nelze !!!!, je immutable
    platnaHodnota = 1234
}

// optional unwrapping
if var platnaHodnota = optInt {
    // lze, ovsem NEzapisuje do optInt
    platnaHodnota = 1234
}
```

Conditional unwrapping, kopie hodnoty

Provede se tedy kopie hodnoty. Jak za chvíli uvidíme:

- Provede se paměťová kopie (struct/enum) typů předávaných hodnotou.
- Provede se inkrement reference counteru u referencovaných hodnot.

```
var hodnota: Typ?
//
if let platna = hodnota {
    // hodnota je platna po dobu if-body
    DispatchQueue.global().async {
        // tj i v navazujícím výpočtu v GLOBAL queue
        let neco = funkce(platna)
    }
}
```

Příkaz guard-else

Motivace: na začátku bloku (funkce) chceme testovat Optional argumenty a vytvořit z nich platné hodnoty.

```
func ahoj(cislo: Int?, jmeno: String?) {
    //
    guard let _cislo = cislo, let _jmeno = jmeno
    else { return }

    // zavedeni novych jmeno do prostoru
    print(_cislo)
    print(_jmeno)
}
```

Příkaz guard zavede do bloku **nové let konstanty**.

Pozn.: možno mít i `guard var _cislo=cislo` ...

Příkaz guard-else

Else sekce příkazu *guard* musí řízení programu jakkoli dostat pryč z **aktuálního bloku kódu**.

- return - typické pro skok ven z funkce
- break, continue - pryč z těla cyklu
- throw - výjimka
- fatalError("hláška")

```
while true {
    //
    guard cosi < 10 else { break }
}
```

Pokročilý guard, weak self

```
udelejNeco(args: ...) { [weak self] result in
    // žije ještě ten volající?
    guard let _self = self else { return }

    // jo, žije...předám výsledek
    _self.vysledek = result
}
```

Volání `udelejNeco(...)` s předáním closure (callback) s referencí `self`, ovšem `weak self`, tj "až to doděláš a budeš chtít volat můj callback, a já nebudu už existovat, tak ten weak self bude nil".

Trocha hazardu, operátor "!"

Forced unwrapping. Pokud rozbalení neuspěje, máme seg-fault.

```
var cosi: Int?

...

// unwrapping
let vysledek = func(cosi!)
```

Je rozumné použít jenom **v okamžicích dobré jistoty platnosti hodnoty**.

- hodnota je zcela jistě lokální a má pouze jednoho majitele
- raději nedělat. Máme jiné možnosti.
- Každopádně má **definované chování**.

Funkce, metody, closures

Spustitelný blok kódu `(ARGS)->(RETURN)` rozlišujeme na:

- pojmenovaný - funkce/metoda,
- nepojmenovaný - funkční objekt, lambda výraz, closure.

Funkce a metody:

- název
- argumenty
- návratový typ
- specifikátory - override, throws, async

Funkce, hlavička funkce

Smalltalk je pra-předek Swiftu (ze strany Objective-C). Proto konvence v hlavičce funkce.

```
func nazevFunkce([ argumenty ]) [ -> NavratovyTyp ] {
    // telo funkce
    return hodnota
}
```

- argumenty jsou pojmenované
- při volání funkce se musí explicitně pojmenovat
- label jména argumentu, _

Funkce, pojmenování argumentů

```
func ahoj(cislo: Int, jmeno: String) {
    //
    print(cislo); print(jmeno)
}

...
ahoj(cislo: 3, jmeno: "pepa")
```

Jména argumentů jsou součástí jména funkce!

Funkce, pojmenování argumentů

```
func ahoj(vesele jmeno: String) { print("Cauuuu", jmeno) }
func ahoj(smutne jmeno: String) { print("no...", jmeno) }
func ahoj(_ jmeno: String) { print(jmeno) }

...

//
ahoj(vesele: "Petre")
ahoj(smutne: "Lojzo")
ahoj("asdfg")
```

Argument tedy má:

- jméno pro interní použití
- jméno pro externí použití (label)
- lze mít prázdný label, _

Je to užitečné při konstrukci objektu

```
class TRIDA {
    //
    let cislo: Int

    //
    init(absolutne v: Int) { cislo = v }
    init(nasobne v: Int) { cislo = v * 100000 }
}

//
let a = TRIDA(absolutne: 10)
let b = TRIDA(nasobne: 5)
// je syntakticka zkratka
let a2 = TRIDA.init(absolutne: 10)
// Objective-C
var a3 = [[TRIDA alloc] initAbsolutne: 10]
```

Výchozí hodnota argumentu funkce

Jazykové okénko: default -> implicitní -> výchozí

```
func nejaka(cislo: Int, dodatek: String? = nil) {
    //
}

//
nejaka(cislo: 5, dodatek: "ahoj")
nejaka(cislo: 1)
```

Funkce, návratová hodnota

- Příkazem `return`.
- Tělo funkce je jediný výraz patřičného typu.

```
func fa() -> Int { return 10 }
func fb() -> Int { 10 }

// nelze:
func fc() -> Int {
    // nejaka cinnost
    print("asdf")

    // chyba, musí být "return 10"
    10
}
```

Funkce, návratová hodnota

- Kompilátor trvá na návratové hodnotě, a současně…
- Statická analýza programu dokáže zjistit, že k ní dochází za všech okolností.

```
func ahoj() -> Int {
    //
    if cosi {
        return 10
    } else {
        return 20
    }

    // statická analýza překladače nebude remcat
}
```

Středníky

Jazyky:

- bigotně středníkové - C/C++
- bigotně nestředníkové
- liberální (např Swift)
- podivné (např Go)

Sazba zdrojového textu není součástí syntaxe jazyka.

```
// go-lang. Závorka MUSÍ být na řádku hlavičky
func funkce() {
    //
}
```

Struktury - struct

Struct je strukturovaná hodnota (record), která má právě jednoho vlastníka, tj nelze ji referencovat.

- Předává se kopií.
- Překladač připouští zjednodušenou syntaxi pro init metodu.

```
struct STR {
    let a: Int
    let b: String

    // init(...) je implicitní
}

//
let a = STR(a: 10, b: "ahoj")
var b = STR(a: 1234, b: "neenene")
let c = b // kopie
```

Inicializace strukturované hodnoty

- metoda `init(...)`. Inicializátor.
- metoda `deinit { }`. Destruktor.

Kolem metody `init(...)` u tříd bývala v raných verzích Swiftu trocha teorie. Autory Swiftu to už naštěstí přešlo.

Inicializátor: různé způsoby nastavení počátečního stavu strukturované hodnoty.

Explicitní init

```
struct STR {
    //
    let cislo: Int

    // pojmenování argumentů
    init(cislo: Int) {
        //
        self.cislo = cislo
    }
}
```

Optional init

Smalltalk/Objective-C kořeny. Init vracející **nil**.

```
struct STR {
    //
    let cislo: Int

    // výsledkem instanciace je Optional
    init?(fromJSON: String) {
        //
        guard let _cislo = dekoduj(fromJSON)
        else { return nil }

        //
        cislo = _cislo
    }
}

// "v" je typu STR?
let v = STR(fromJSON: "...")

//
if let v = STR(fromJSON: "...") {
    //
}
```

Optional init (konvenčně)

```
struct STR {
    //
    let cislo: Int

    // výsledkem instanciace je Optional
    static func CREATE(fromJSON: String) -> STR? {
        //
        guard let _cislo = dekoduj(fromJSON)
        else { return nil }

        //
        return STR(cislo: _cislo)
    }
}

//
if let v = STR.CREATE(fromJSON: "...") {
    //
}
```

Sémantika struktur

- Na hodnotu *struct* se nahlíží jako na obvykle konstantní balík datových atributů.
- Je to jako "složitější" primitivní typ.
- Konstanta 3 je Int. Nikdo nebude "referencovat trojku".
- Proto je struct pod **hodnotovou/kopírovací sémantikou**.

"Chtěl jsem ti předat pět údajů, tak jsem je zabalil do struct. Tady ji posílám". Obdrží se kopie.

Struktury se předávájí kopií a obvykle jsou immutable. Konstantnost je jejich implicitní chování.

Sémantika struktur

```
struct STR {
    var i: Int
}

//
let a = STR(i: 10)
var b = STR(i: 1234)
//
print(a.i)
```

Nelze

```
a.i = 10
```

Lze

```
b.i = 123
```

Sémantika struktur

```
struct STR {
    //
    var i: Int
}
// "b" je identita hodnoty. Není to ref. na hodnotu!
var b = STR(i: 1234)
// revize hodnoty, znovu-zbudování hodnoty!
b.i = 123
```

Dogma č.2: Aktualizace hodnoty struct se interpretuje jako znovu-vygenerování hodnoty:

- vzniká nová STR(i: 123) s modifikovaným atributem
- ta se zapíše do cíle `var b:STR`
- cíl tudíž musí být `var`

Sémantika struktur

```
struct STR {
    //
    var i: Int
}
//
class TRIDA {
    //
    var hodnota: STR {
        //
        didSet { print("Zmena!") }
    }
}
//
var p = TRIDA(hodnota: STR(i: 1))

// didSet
p.hodnota.i = 10
```

Něžný úvod do kolekcí

Kolekce ve standardní knihovně Swiftu jsou **šablonové struktury**.

- Array<Element> - vektor, std::vector<Element> v C++
- Map<Key:Element> - skoro ... std::map<Key, Element>
- Set<Element> - skoro ... std::set<Element>

Tedy, jejich vlastnosti:

- jsou homogenní (Objective-C: NSArray nad NSObject/id)
- map, set - jsou implementovány hash-table, tj Key/Element musí být Hashable (povíme si později)
- jsou struct, tj. předávají se hodnotou, ale **copy-on-write**

let array // immutable array

```
let pole = [1,2,3]
let pole2: [Int] = [4,5,6,7]
let pole3 = vysledekVolaniFunkce(...)

// subscript
print(pole[0]) // -> Element
print(pole.count) // -> Int
print(pole.isEmpty) // Bool
print(pole.first) // -> Element?
```

Nelze:

```
pole[0] = 123
```

Proč???? Komu to ublíží?! :)

var array // mutable array

```
var pole = Array<Int>()
var pole2 = [Int]()
var pole3: [Int] = []

// je to ale divné....
var pole4 = vysledekVolaniFunkce()
```

- `pole.append(345)`
- `pole.remove(at: Int)`
- `pole.reserveCapacity(100)` - !!! je to vektor !!!
- `pole[0] = 123`

Připomínka: Modifikace hodnoty typu struct jakoby vytvoří zcela novou hodnotu (tu modifikovanou) a tu uloží do cíle, proto cíl musí být `var`.

Dictionary, map klíč:hodnota

```
let x = [Int:String] = [1:"Ahoj", 2:"Cosi"]
var y = [Int:String] = [:]
```

- Key musí být typ implementující `Hashable`
- subscript `x[key] -> Value?`, `if let _x = x[key] {}`
- případně `x[key]!`
- `x.contains(key) -> Bool`

```
// případne for (_, value) ...
for (key, value) in x {
    //
}
```

Předbíháme: ObservableObject

```
class Model: ObservableObject {
    //
    @Published var pole: [Int] = []
}

//
let m = Model()

// observers/sink dostanou zprávu (nové pole)
// odchytí to přes didSet {}
m.pole.append(1)
m.pole[0] = 1234
```

Zpátky ke strukturám

```
struct STR {
    let a = 3 // konst
    let b: Int // init
    let c: Int? // init
    var d = 4 // volitelně init
    var e: Int
    var f: Int?
}

//
let s = STR(b: 1, c: nil, e: 5, f: 120)
```

Optional chaining

Vzpomínky na Objective-C (zpráva NULL referenci).

```
struct STR {
    let a: Int
    let b: Int?
}
let A = STR(a: 3, b: ...)
let B: STR? = volaniNejakeFunkce()
```

Pak Optional tečková notace:

- `A.a` je Int, `A.b` je Int?, `A.b!` je Int (s rizikem)
- `B?.a` je Int?, `B?.b` je Int?, `B?.b!` je magořina :)
- `B!.a` je Int, `B!.b!` je tuplované riziko

Optional chaining

```
struct STR {
    let a: Int
    let b: Int?
}
let A = STR(a: 3, b: ...)
let B: STR? = volaniNejakeFunkce()
```

Pak:

```
if let _value = B?.b {
    // je ok
}

//
guard let _value = B?.b else { return }
```

Optional návratový typ funkce

Velmi inspirativní pro C++ (už je v STL).

```
template<typename T>
struct Optional {
    bool valid = false;
    T value;
    //
    Optional() : valid(false) {}
    Optional(const T &inv) : valid(true), value(inv) {}
    //
    inline bool test() const { return valid; }
    const T & operator()() const { return value; }
};

//
Optional<int> vratCosi() { ... }
```

Instanční metody struktur

- `let/var` vlastnictví struktury -> modifikovatelnost
- `self` struktury je implicitně immutable
- `mutating func` povolí mutable `self` struktury

```
struct STR {
    var cislo = 123

    // funkce je const (jako v C++)
    func dej() -> Int { return cislo }

    // nelze, self je immutable
    func nastav(_ val: Int) { self.cislo = val }

    // lze, podmíněně
    mutating func nastav(_ val: Int) { self.cislo = val }
}
```

Instanční metody struktur

```
struct STR {
    //
    var cislo = 123

    mutating func nastav(_ val: Int) { self.cislo = val }
}
```

Pak:

```
let a = STR()
var b = STR()
// nelze volat z let hodnoty mutating funkci
a.nastav(3)
// lze
b.nastav(567)
```

Předávání hodnoty struct

```
func udelej(s: STR) {
    // dostavam kopii
}

//
let hodnota = STR()
// předávám kopii
udelej(s: hodnota)
```

Připomeňme: Hodnota struct má VŽDY jen jednoho vlastníka.

Připomeňme: SwiftUI je postaveno na strukturách.

SwiftUI demo

```
struct HlavniOkno: View {
    // stavová proměnná okna
    @State var jmeno: String = ""
    // uživatelské rozhraní okna
    var body: some View {
        //
        VStack {
            TextField("zadej jmeno", text: $jmeno)
            Button("odesli") {
                // neco udelej
            }
        }
    }
}
```

Předávání hodnoty, inout

```
func udelej(s: inout STR) {
    // dostavam kopii, vracím kopii
    s.cislo = 1234
}

//
var hodnota = STR()
// operátor & skoro vypadá jako reference! :D
udelej(s: &hodnota)
```

Ne, stane se:

- volání `udelej(s: &hodnota)` předá kopii `hodnota` do funkce
- `s` ve funkci figuruje jako `var`, vrací se kopie
- která přepíše `var hodnota`

Předávání hodnoty, inout konvenčně

```
func udelej(s: STR) -> STR {
    // dostavam kopii, vracím kopii
    s.cislo = 1234; return s
}

//
var hodnota = STR()

//
hodnota = udelej(s: hodnota)
```

inout Uvidíme v Observable modelech

... jinak to příliš smysl nedává ...

```
class Model: ObservableModel {
    //
    private var anies = Set<AnyCancelable>()

    //
    init() {
        //
        nejakyPublisher
            .sink { ... }
            .store(in: &anies)
    }
}
```

Struktura a primitivní referencované hodnoty

```
let s: String = "Ahoj, jak se vede"
// kopie struktury
let s2 = s
```

- `String` je struktura zapouzdřující primitivní implementaci ukazatele na data (malloc, char *, ...)
- ovšem !!! reference-čítanou
- objekt (referencovatelný, čítaný) zapouzdřující `char *`
- `s2` navyšuje při kopii reference counter
- data se nekopírují, není důvod...

Struktura a primitivní referencované hodnoty

```
struct STR {
    //
    let s: String
    var x: String = ""
}

//
let a = STR(s: "Ahoj", x: "vede se?")
let b = a // navyšuje refCount těch dat
var c = a

// dereference předchozího obsahu
// nový obsah
c.x = "neco noveho"
```

Copy-on-write u kolekce, Array

```
// vzniká primitivní datový objekt
// struktura Array<Int> ho zapouzdřuje
let a = [1,2,3,4]

// navyšuji refCount na datový obsah
var b = a

// první změna, copy-on-write
b[0] = 123456
```

V okamžiku zápisu do pole `b[0]=...`

- klonuje se primitivní objekt dat s obsahem
- `b` má vlastní kopii, do té ať si zapíše

Důsledek: je zcela efektivní předávat kolekce hodnotou.

Iterování přes Array

```
let pole: [Int] = ...

// let value: Int
for value in pole {
    // co znamená iterovat?
}
```

- Swift považuje subscript `pole[idx]` za ohavnost :D
- Index-sekvenční přístup do pole je problematická operace
- Přes kolekce zásadně iterujeme: map, filter, ...
- `Array<Element>` implementuje `Sequence`, `IteratorProtocol`

Iterování přes Array

```
let pole: [Int] = ...

// let value: Int
for value in pole {
    // co znamená iterovat?
}

// je ekvivalent
var iterator = pole.makeIterator()

// next() -> Element?
while let value = iterator.next() {
    // vnitřek for cyklu
}
```

IteratorProtocol

```
protocol IteratorProtocol {
    // associatedType Element
    mutating func next() -> Element?
}
// zatím bez šablonových typů...
struct IntArrayIterator : IteratorProtocol {
    //
    let mojeArray: [Int]
    var idx = 0

    //
    mutating func next() -> Int? {
        guard idx < mojeArray.count else { return nil }
        //
        defer { idx += 1 }
        //
        return mojeArray[idx]
    }
}
```

Iterování přes pole

Neiteruji přes `pole`, ale přes **dočasnou kopii pole uloženou v iterátoru**.

```
var pole = [1,2,3,4,5]

//
for value in pole {
    // nedává žádný smysl
    pole.remove(at: ...)
    pole.append(...)
    // value je let
    value = 124 // nelze
}
```

Iterování přes pole

- map
- filter

```
let pole = [1,2,3]

// closure, téma pro příště
let kratDvaPole = pole.map { val in return val * 2 }
let kratDvaPole = pole.map { $0 * 2 }
let filteredPole = pole.filter { $0 > 10 }
```

Iterování přes Range

```
// 0 až 100 včetně
for i in 0...100 {

}

// 0 až 99 včetně
for i in 0..<100 {

}

//
let c = (0...100).filter { $0 % 2 == 0 }
```

Přehled dogmat, závěr, příště

Dogma č. 1: Každá deklarace vyžaduje počáteční hodnotu.

Dogma č.2: Aktualizace hodnoty struct se interpretuje jako znovu-vygenerování hodnoty.

Swift-II: třídy, closures, exceptions
Swift-III: templates, dodatky