Swift II.

IZA, Martin Hrubý, FIT VUT, 2020
http://perchta.fit.vutbr.cz/vyuka-iza

Knihovny

- Swift Standard Library — kolekce, koncepty.
- Foundation — společná funkcionalita pro všechny platformy.
- UIKit — vývoj iOS aplikací/Storyboard.
- SwiftUI — struct-View deklarativní zápis.

SwiftUI, motivační demo

```
struct PrvniObrazovka : View {
    //
    @Environment(\.managedObjectContext) var managedObjectContext
    //
    @FetchRequest(
        entity: Sesit.entity(),
        sortDescriptors: [NSSortDescriptor(keyPath: \Sesit.name, ascending: true)]
    ) var sesity: FetchedResults<Sesit>
    //
    var body: some View {
        //
        NavigationView {
            //
            List {
                //
                ForEach(sesity, id:\.self) { nb in
                    //
                    NavigationLink(destination: PrvniObrazovkaDetail(nb)) {
                        //
                        SesitRow(sesit: nb)
                    }
                }
            }
```

Přehled o Swiftu

- Příkazy — if/switch, guard, while, for-in.
- Základní použití kolekcí.
- Konstrukce tříd, struktur, enum.
- *Extensions*.
- *Protokoly*.
- *Generické programování*.

Příkaz ”if”

- if COND,COND2,… stm [else stm2]
  - let _const = optional-expr; var _var = optional-expr
  - Není céčkové ”(expr)”. Podobně středník.
- Deklarace ”guard”
  - Deklarace guard vytváří novou let/var v celém kontextu bloku.
  - Smyslem je **přehledně testovat vstupní argumenty** (funkcí).
  - ”Escape” sekce guardu. Častokrát ”abort”.

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.

if let, guard demo

```swift
// [ = nil] implicitne
// !!! víme, že A je 1) platná hodnota 2) nil
var A : Trida?

// A != nil, do _A se provede kopie!
// tj. pripadne refCount++
if let _A = A, _A.cosi() > 10 {
    //
    print("A je \(_A)")
}

// vraci UITableViewCell?
guard let cell = table.dequeTableViewCell() else {
    abort("Cele je to spatne");
}
```

Podtržítkové názvosloví…

- Rozlišujte metody / proměnné:
  - interní, velmi interní — _interniMetoda(), __velmiInterni()
  - veřejné — vratHodnotu()
  - vstupní parametry — func ahoj(pozdrav: String?)
  - otestované parametry s důvěryhodnou hodnotou (je v očekávaném rozsahu)
  - … pro běh ve vedlejším vlákně
  - … vyhazující výjimky

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;
}
```

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

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

Použití stráže…

```swift
// 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)
```

Příkaz for-in

- Iterování přes něco iterovatelného:
  - implementuje protokoly *Sequence* a *IteratorProtocol*,
  - kolekce (array/slice, set, map),
  - rozsah — 0…10, 0..<20.
- for i in cosi:Sequence { / / i }
  - for i in [1,2,3,4]; for i in 0…100; for i in ARR;
  - var iter = ARR.makeIterator(); while let _v = iter.next() {};

IteratorProtocol

```swift
//
public protocol IteratorProtocol {
    // The type of element traversed by the iterator.
    associatedtype Element

    //
    public mutating func next() -> Self.Element?
}

// odvozujeme od protokolu "IteratorProtocol"
struct MujIterator : IteratorProtocol {
    // specifikujeme typ hodnoty iterovani
    typealias Element = Int
}
```

Iterátor

```
// odvozujeme od protokolu "IteratorProtocol"
struct MujIterator : IteratorProtocol {
    // specifikujeme typ hodnoty iterovani
    typealias Element = Int
    // zrejme nejde bez nejakeho stavu
    var mujVnitrniStav : Element = 0
    // funkce pro vystup dalsi hodnoty
    // nil == konec
    mutating func next() -> Int? {
        //
        if mujVnitrniStav < 10 {
            //
            let retValue = mujVnitrniStav
            //
            mujVnitrniStav += 1;
            //
            return retValue
        }
        //
        return nil
    }
}
//
var mujI = MujIterator()
//
while let _t = mujI.next() {
    //
    print(_t)
}
```

Iterátor, víc swiftově…

```
// odvozujeme od protokolu "IteratorProtocol"
struct MujIterator : IteratorProtocol {
    // specifikujeme typ hodnoty iterovani
    typealias Element = Int
    // zrejme nejde bez nejakeho stavu
    var mujVnitrniStav : Element = 0
    // funkce pro vystup dalsi hodnoty
    // nil == konec
    mutating func next() -> Int? {
        //
        guard mujVnitrniStav < 10 else {
            return nil
        }
        // !!!
        defer { mujVnitrniStav += 1 }
        //
        return mujVnitrniStav
    }
}
```

Nekončící iterátor

```
//
struct MyRND : IteratorProtocol {
    //
    typealias Element = Double

    //
    private var x : UInt = 0

    //
    mutating func next() -> Double? {
        // &+, &* -- pripousti preteceni cisel
        x = (x &* 69069) &+ 1

        //
        return Double(x) / (Double(UInt.max) + 1)
    }
}
```

Protokol Sequence

```
// zaklad protokolu Sequence
public protocol Sequence {
    // nad typem Element
    associatedtype Element
    // s iteracnim typem Iterator
    // a podminkou, ze ...
    associatedtype Iterator: IteratorProtocol
        where Iterator.Element == Element
}
```

- Sekvence implementuje for-in operaci.
  - Alokace iterátoru. Potom *while let i = iter.next {}*
- Další operace: filter, map, …

```swift
// Rozdil: struct/class
struct RNDSequence: Sequence, IteratorProtocol {
    //
    private var x : UInt = 0
    private var ic: Int = 0
    //
    mutating func next() -> Double? {
        // dam si tam nejaky pevny bod...
        guard ic < 10 else { return nil }; ic += 1
        // &+, &* -- pripousti preteceni cisel
        x = (x &* 69069) &+ 1
        //
        return Double(x) / (Double(UInt.max) + 1)
    }
    //
    func makeIterator() -> RNDSequence {
        //
        print("Calling makeIterator()")
        //
        return self
    }
}
// hodnota "sekvence" (class/struct)
var threeToGo = RNDSequence()
// for in se v prekladaci generuje na:
var tempIter = threeToGo.makeIterator()
// pruchod cyklem, telo cyklu
while let _val = tempIter.next() { print(_val) }
//
for i in threeToGo { print(i) }
//
for i in threeToGo { print(i) }
```

Rozsahy, slices

- 0…100 — začíná hodnotou 0, končí 100.
- 0..<100 — 100 není zahrnuto.
- let pole = [1,2,3,4,5]
- let slice = pole[0..<2]
  - pod-pole zadaného pole.
  - Abstrakce nad polem.

Přehled kolekcí

- Kolekce jsou homogenní (na rozdíl od Obj-C).
- Array<Element> — podobně jako std::vector
  - C++: navíc ještě std::deque, std::list
- Set<Element> — Element:Hashable
  - C++: std::set, std::unordered_set
- Map<Key,Value> — Key:Hashable,Equitable
  - C++: std::map, std::unordered_map
- Časová složitost operací! CPU-cache.

Přístup na prvky kolekce

- Iterování přes pole. Metody *filter* a *map*.
- Funkcionální přístup k programování.
- Subscript:
  - index-sekvenční přístup — rozsah pole, exception.
  - přístup do dictionary/map, klíč-hodnota — vrací Hodnota?
- Index-sekvenční přístup v programu minimalizovat!!!

Array, konstrukce

```
//
struct Person { let name: String }
//
let names = ["Petr", "Jan", "Filip"]
// = Array<Person>()
var people: [Person] = []

// !!! vzdy, kdyz dopredu znam velikost
people.reserveCapacity(names.count)
//
for n in names {
    //
    people.append(Person(name: n))
}
// Pozn. NSMutableArray -> NSArray
// .map { n in return Person(name: n) }
let funcStyle = names.map { Person(name: $0) }
// selekce
let shortNames = names.filter { $0.count <= 3 }
// enumerace: (index, hodnota)
for (idx, n) in people.enumerated() {
    //
    print("Person: \(n), index: \(idx)")
}
```

subscript

```
//
let pole = [1,2,3,4,5]
// subscript
let prvni = pole[1]
// exception
let nejaky = pole[10000]

// dictionary
let dict: [Int:Int] = [ 1:2, 2:3]
let val = dict[1]
// je Optional(Int) !!!
print(val)
//
if let _tested = dict[1] {
    //
}
```

Uživatelský sub-script

```
//
struct MojeData {
    //
    subscript(_ idx: Int) -> String {
        // nejaky rozumny kod...
        return "ahoj"
    }
}

//
let m = MojeData()
let v = m[1]
```

```swift
struct Matrix {
    let rows: Int, columns: Int
    var grid: [Double]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
// template Array
        grid = Array(repeating: 0.0, count: rows * columns)
    }
    func indexIsValid(row: Int, column: Int) -> Bool {
        return row >= 0 && row < rows && column >= 0 && column < columns
    }
    subscript(row: Int, column: Int) -> Double {
        get {
            assert(indexIsValid(row: row, column: column), "Index out of range")
            return grid[(row * columns) + column]
        }
        set {
            assert(indexIsValid(row: row, column: column), "Index out of range")
            grid[(row * columns) + column] = newValue
        }
    }
}
```

Slices, demo

```
//
let names = ["Petr", "Jan", "Filip"]

// slice1[0] je seg-fault, zaciname [1]
let slice1 = names[1...]

// enumerace: (index, hodnota)
// !!! enumerated() vsak posloupnost precisluje
// generuje [(Int, Value)] vlastnim pruchodem
for (idx, n) in slice1.enumerated() {
    //
    print("Person: \(n), index: \(idx)")
}
```

Démonický switch

- Rozdíl od C — implicitně ”nepropadává” přes case. Explicitní ”fallthrough”.
- case je strukturovaný vzor.

```
//
switch a {
case 1:
    print("jedna")
case 0...100:
    print("patri do intervalu");
    fallthrough
case 3:
    print("tri")
default:
    print("neco jineho")
}
```

Konstrukce tříd

- Objektově-orientované programování.
- Jméno třídy, nadřazená třídy, protokoly.
- Atributy:
  - instanční/třídní(statické),
  - uložené/vypočtené/lazy.
  - metody.
  - inicializační metody.
  - deinit.

Datové atributy třídy / class

```
//
class TRIDA {
    // lze je zapsat JEDNOU (behem inicializace)
    let konstA = 1
    let konstB: Int
    let konstC: Int?
    let konstD: Int! // trochu mimo
    //
    var promA = "Ahoj"
    var promB: Int
    var promC: Int?
    var promD: Int!

    //
    init() {
        // vynucena inicializace
        konstB = 0
        konstC = nil
        konstD = 9875
        // vynucena inicializace
        promB = 2222
        promC = 5678
    }
}
```

statické / třídní atributy

```
//
class TRIDA {
    // tradicni singleton
    static let shared = TRIDA()

    //
    static func STARTUP() {
        //
        shared.startup()
    }

    //
    func startup() {
        // inicializuj "shared"
    }
}
```

lazy var

```
//
class TRIDA {
    // kod se vykona pouze jednou!
    lazy var dokument : String? = {
        // otevri soubor
        // nacti dokument
        print("Nacitam dokument ze souboru")
        return "nejaky obsah"
    }()
}
//
let t = TRIDA()
// pristup na lazyvar, vykona se jeji init-kod
let doc = t.dokument
// dale je volne zapisovatelna
t.dokument = nil
// a hodnota zustava nil
let doc2 = t.dokument
```

Vypočtená property

- Je to fundamentálně funkce, která však nemá argumenty.
- Odvozuje svůj obsah z kontextu ”self”.

demo…

```
//
class TRIDA {
    //
    private var _dokument: String? = nil

    // vypoctena property
    var dokument: String {
        //
        if _dokument == nil {
            // vytvor nejaky obsah (soubor)
            _dokument = "nejaky obsah"
        }

        //
        return _dokument!
    }

    // neni thread-safe!!!
    func uvolniDokument { _dokument = nil }
}
```

getter / setter property

```
//
class TRIDA {
    //
    private var _dokument: String = "no-cont"

    //
    var dokument: String {
        //
        get {
            //
            return "Nejak zformatovany \(_dokument)"
        }

        //
        set(newValue) {
            //
            _dokument = newValue
        }
    }
}
```

Key-Value Observing

```
//
class TRIDA {
    //
    var dokument: String = "nictuneni" {
        //
        willSet(newValue) {
            //
            print("Nekdo chce zapsat hodnotu \(newValue)")
        }
        //
        didSet {
            //
            print("Doslo k prepsani hodnoty")
        }
    }
}

//
let p = TRIDA()
// volani setteru! Vzdy berme jako spusteni nejakeho kodu.
p.dokument = "ahoj"
```

NSKeyValueCoding protokol

```
//
class TRIDA : NSObject {
    // musi byt explicitne oznaceno jako KVC
    @objc var dokument: String = "nictuneni"
}

//
let p = TRIDA()
p.dokument = "ahoj"
// Objective-C getter, je String?
print(p.value(forKey: "dokument"))
// Objective-C setter
p.setValue("ahoj", forKey: "dokument")
```

Paměťové modely

- Jaký vztah vyjadřujeme datovým atributem?
  - var jmeno = hodnota
- strong reference — refCount++. ARC.
- Platí výhradně pro uložené properties:
  - weak reference — weak var: Type?
  - unowned var: Type
  - unowned(unsafe) var: Type

Weak var p: Type? = cíl

```
class TRIDA {
   weak var delegate: MujProtokol?
}
```

- Nezvyšuje refCount.
- Musí být *Type?*, protože připouštíme vynulování.
- Pokud *cíl* zanikne, pak runtime automaticky vynuluje ”p”.
- Implementace. Režie.
- Typické pro vztah ”delegate”.

Rozpojení ref-cyklu

```
//
class PROD {
    // reference cycle !!!
    var delegate: CONS?
}

//
class CONS {
    // reference cycle !!!
    var prod: PROD?
    //
    func start() {
        //
        prod = PROD(); prod?.delegate = self
    }
    // explicitne, rozhodnu se rozpojit
    func konec() {
        //
        prod?.delegate = nil
    }
}
```

Unowned

- *unowned let delegate: Type*
- Musí být naplněno hodnotou. Nelze *Type?*
- Nezvyšuje refCount.
- Pokud cíl zanikne, zanechá po sobě v paměti ”značku”. Při přístupu je garantován seg-fault.
- Předpokládá se, ”že mě můj vlastník dealokuje dřív, než si všimnu, že už sám není…”.
  - … nemám však rozběhnutá žádná vlákna…

Demo

```
//
class Master {
    //
    var servant: Servant?

    //
    func haveServant() {
        //
        servant = Servant(myMaster: self)
    }
}

class Servant {
    // weak, unowned
    unowned let master: Master
    //
    init(myMaster: Master) { master = myMaster }
}
```

unowned(unsafe)

- Ekvivalent ”nahaté hvězdičky” v C/C++.
- Naplňuje koncept *unowned*, až na tu značku.
- Důsledek: při zániku cíle runtime nijak stav neošetřuje. Vede na **nedefinované chování**.

Předání reference

- Argumenty volání funkce jsou všechny typu strong reference.
- Referenci nelze předat jinak.
  - každé volání funkce zvyšuje refCount.
  - výjimka: [weak self], bude dále…

```swift
//
class Nekdo {}
//
func volamNekoho(nekdo: Nekdo) {
    //
    let _c = CFGetRetainCount(nekdo)
    //
    print("refCount=\(_c)")
}
//
class Trida {
    //
    weak var nekdo: Nekdo?
    //
    init(nekdo: Nekdo) {
        //
        self.nekdo = nekdo
    }
    //
    func volani() {
        // drzim to "weak", predavam "strong"
        if let _nekdo = nekdo {
            volamNekoho(nekdo: _nekdo)
        }
    }
}

var nekdo: Nekdo? = Nekdo()
var objekt = Trida(nekdo: nekdo!)
// napise refCount=3
objekt.volani()
```

Funkce, Lambdas, Closures

```
typealias MojeFunkce = (Int, String) -> Int?
```

- Funkce:
  - Lze vytvářet globální, lokální/metody (struct/class), vnořené.
  - Lze přidávat do struct/class/enum v rámci Extensions.
  - Mohou být šablonovité.
  - Funkce **je referencovatelný objekt**. Lze ukládat jako data.
  - Funkce se smí vázat na svůj kontext (globální, class — referencovatelný).
  - Funkce je pojmenovaná. Funkce beze jména — lambda výraz.

```swift
// Definice typu lambda vyrazu, vraci Int?
typealias MojeFunkce = (Int) -> Int?
//
class TRIDA {
    func metoda(a:Int) -> Int? {
        //
        return a + 1
    }
    static func metodaStatic(a:Int) -> Int? {
        //
        return a + 1
    }
    func getMe() -> MojeFunkce {
        // vnorene definovana funkce
        func vnorena(a:Int) -> Int? { return 1234; }
        // vracim objekt funkce
        return vnorena
    }
}
//
let obj = TRIDA()
let fce : MojeFunkce = obj.metoda
let fce2 = TRIDA.metodaStatic
let fce3 = obj.getMe()
//
print(fce(1))
print(fce2(3))
print(fce3(4))
```

Lambda výraz

- Základem je definice typu *(…) -> (…)*
  - … a kód je pojmenovaný, pak je to funkce
  - … a kód je nepojmenovaný, pak je to lambda výraz.

```
// explicitne urcim typ a deklaruji hodnotu s explicitne zadanym typem
let lam1 : MojeFunkce = { (a:Int) -> Int? in return a + 2 }
// "a" je Int, tudiz prekladac pochopi (Int)->(Int)
let lam2 = { a in return a + 2 }
// funkce "provolej" je typu MojeFunkce -> ()
func provolej(fce: MojeFunkce) { print(fce(2)) }
//
provolej(fce: lam1)
provolej(fce: lam2)
provolej(fce: { (a:Int) -> Int in return a + 3 })
provolej(fce: { (a:Int) in return a + 4 })
provolej(fce: { a in return a + 5 }) // provolej urci typ pro "a"
provolej(fce: { return $0 }) // provolej urci typ pro $0
provolej { return $0 }
```

Předání lambdy při volání funkce

```
//
func cosiUdelej(a:Int, b: Int, call: (Int)->()) {
    // vypocet
    call(a+b)
}

//
cosiUdelej(a: 10, b: 20) {
    result in print("...a tady je vysledek \(result)")
}
```

Demo — @escaping closure

```
//
let nejakaData = ...

// registruj blok do vedlejsiho vlakna
DispatchQueue.global().async {
    // ... vypocet mimo main-thread
    let result = vypoctiVeVedlejsimVlakne(vstup: nejakaData);

    // spust v hlavnim vlakne
    DispatchQueue.main.async {
        //
        print("Vysledek \(result)")
    }
}
```

Closures

- Closure je lambda výraz, který je zabalený do kontextu, tj. referencuje svůj (vnější) kontext.
  - ”*self*” objektu, další proměnné (dynamického) kontextu.
- Kdy je closure *@escaping* ?
  - Escaping — předávaný closure nebude okamžitě volán, ale jeho volání bude odloženo.
  - … tj. přeruší se linie volání z místa, kde byl closure vytvořen.
  - … typicky si closure někdo ”uloží” (proměnná, kolekce).
  - Caller v době volání **callbacku** už nemusí existovat nebo chceme rozpojit ref. cyklus.

Closure, základní demo

```
//
class TRIDA {
    //
    var hodnota : Int = 1000
    //
    func metoda(a:Int) -> Int? {
        // hodnota je self.hodnota
        return a + self.hodnota
    }
}
//
var obj : TRIDA? = TRIDA()
// vznika funkcni objekt, ktery musi referencovat "obj"
let lam1 = obj!.metoda
// snizuju retainCount "obj"
obj = nil
print(lam1(2)) // objekt je stale referencovan lambdou
// Model hodnoty "lam1"
struct ModelOfLam1 {
    //
    let mujSelf : TRIDA
    //
    func run(a:Int) -> Int? { return a + mujSelf.hodnota }
}
```

```
//
class Receiver {
    //
    var  todo: [(Int)->Int] = []
    //
    func service(fce: (Int) -> (Int)) {
        // volam blok "fce" ve svem kontextu
        print(fce(3))
    }
    // blok "fce" neni okamzite volan, presto se pouzije (ulozi)
    func serviceDelayed(fce: @escaping (Int) -> Int) {
        //
        todo.append(fce)
    }
}
//
class Caller {
    //
    var localOne = 3
    //
    func mywork(rec: Receiver) {
        // predavam closure do funkce, ktera ho okamzite aktivuje
        rec.service(fce: { a in return a * localOne })
    }
    //
    func myworkDel(rec: Receiver) {
        // parametr "fce" je @escaping, proto musim explicitne
        // psat "self."
        rec.serviceDelayed(fce: { a in return a + self.localOne })
    }
}
//
let receiver = Receiver()
let caller = Caller()
//
caller.mywork(rec: receiver)
caller.myworkDel(rec: receiver)
//
receiver.todo.forEach { blocek in print(blocek(3)) }
```

weak self

```
//
class Receiver {
    // strong ref na funkcni blok
    var callback: ((Int) -> (Int))?
    // blok "fce" neni okamzite volan, presto se pouzije (ulozi)
    func serviceDelayed(fce: @escaping (Int) -> (Int)) {
        // ukladam si blok
        callback = fce
    }
}
//
class Caller {
    //
    var localOne = 3
    //
    func myworkDel(rec: Receiver) {
        // weak self - tvori hodnotu Caller?
        rec.serviceDelayed(fce: { [weak self] a in
            // ... self je optional, test
            guard let _him = self else { return -1 }

            //
            return a + _him.localOne
        })
    }
}
//
let receiver = Receiver()
var caller : Caller? = Caller()
//
caller?.myworkDel(rec: receiver)
caller = nil
//
print(receiver.callback?(10))
```

Closures, závěry

- Každý blok kódu je referencovatelný objekt.
- Pokud je v nějakém kontextu (má *self*), pak předání bloku zvyšuje i refCount toho *self*.
- Escaping, non-escaping.
- Escaping closure musí dávat povinně prefix self.xxx

Třídy: dědičnost

- Všechny metody jsou ”virtual”.
- Přetížení metody:
  - syntaxe vynucuje prefix *override,*
  - super.volaniNadrazeneMetody
- init()
  - super.init(…) se volá na konci metody

```swift
//
class TridaA {
    //
    var prop: Int { return 1 }
    func fun() {}
    //
    init(input: Int) {
        //
        print("InitA: \(input)")
    }
}
//
class TridaB : TridaA {
    //
    override var prop: Int { return super.prop * 2 }
    //
    override init(input: Int) {
        //
        print("TridaB")
        //
        super.init(input: input)
    }
    //
    init(mujNovyInit input: Int) {
        //
        super.init(input: input)
    }
}

//
let x = TridaB(mujNovyInit: 10)
let x2 = TridaB(input: 20)
```

Instanciace třídy, init?

```
//
class Trida {
    //
    var parametr: Int
    //
    init?(input: Int) {
        //
        guard input >= 0 else {
            //
            return nil
        }
        // toto self. muzu
        self.parametr = input
    }
}

// je typu Trida?
let objekt = Trida(input: 2)
//
objekt?.parametr = 3
```

Struktury

- Hodnota typu struct má vždy jenom jednoho vlastníka.
  - Nelze získat ”ukazatel na hodnotu struct”.
- Předáváte struct, provádíte kopii.
  - Při které se podle paměťových modelů pracuje s refCounty na instanční proměnné.

Tuples, demo

```
//
struct Osoba {
    //
    let vek:Int
    let jmeno: String
    let narozen: Date
}
//
typealias OsobaTuple = (vek:Int, jmeno:String, narozen:Date)
//
func vrat() -> (vek:Int, jmeno:String, narozen:Date) {
    //
    return (vek: 10, jmeno: "Jan", narozen: Date())
}
//
func vratS() -> Osoba {
    //
    return Osoba(vek: 10, jmeno: "Jan", narozen: Date())
}
//
let res = vrat()
//
print("\(res.vek) \(res.jmeno)")
```

Přetypování "as"

- Objektový metaprotokol. Objective-C.
  - Zjistit třídu/typ objektu. Rozumí objekt zprávě?
- Co lze přetypovat?
  - Test typu *"is"*.
  - Statické přetypováná (C versus striktně typované jazyky). Dynamické přetypování.
  - Uppercast: *as*, typicky je implicitní.
  - Downcast: *as?*, *as!*
  - Konverze hodnoty.

Testovací operátor ”is”

```
//
class Creature { }
class Human : Creature { }
class Animal : Creature { }
//
let pole: [Creature] = [Human(), Animal()]
var budeCislo = "456"

// is nad primitivnim typem
if 1 is Int { print("Prekvapive, je... ") }
// vysledek Int("123") je Int?
if Int("123") is Int { print("123 je Int") }
// dynamicke vyhodnoceni "is"
if Int(budeCislo) is Int { print("budeCislo je Int") }
//
if pole[0] is Human {
    //
    print("je to clovek...")
}
```

as? demo

```
//
let pole: [Creature] = [Human(), Animal()]
//
for p in pole {
    // dynamicky downcast s vysledkem Human?, ktery dale
    // podminene vazeme na let _h
    if let _h = p as? Human {
        //
        print("Clovek...")
    }
    // switch je prikaz "mnoha tvari"
    switch p {
    case let _p as Human:
        //
        print("Clovek")
    case let _a as Animal:
        //
        print("Zvire")
    default:
        //
        print("Nevime...")
    }
}
```

Co s tím *as!* …?

- Striktní downcast. Při nezdaru program havaruje.
- Motto: kdykoli něco testuji, pak musím vědět, co rozumného má program udělat, když test selže.
  - Situace: knihovní funkce má vrátit objekt třídy XY. Nevrátí XY, ale jiný (např. UITableViewCell). Stav testuji.
  - Vrátí jiný: co rozumného aplikace může provést? Jedině havarovat.
  - Důsledek: *as!* je zde ospravedlnitelný.

as jako datová konverze

- Pro pár tříd z Foundation (NSArray, NSString, NSData, NSError)
  - Nepříliš vážně.
- Upcasting: ”p as Creature”.
- Staticky, pokud překladač dovolí:
  - let p = 1 as Double
  - let pp = 1.3 as Int // error
  - let ppp = Int(1.3) // ppp == 1

Příště…to nejlepší

- Výčtový typ enum.
- Protokoly.
- Extensions.
- Generické programování.