Swift II.: OOP, Enum, Closures, Protokoly, Extensions

Programování zařízení Apple (IZA)

Martin Hrubý, FIT VUT, 2022/23

Úvod

Koncepty programování ve Swiftu:

- Objektově orientované programování (OOP)
- Protokolově orientované programování (POP)
- Closures - dynamika v programu
- Extensions & Protokoly - abstrakce v programu

Na Swift můžeme nahlížet jako na:

- nástroj programování iOS aplikací
- vysoce abstraktní způsob formulace algoritmů (např POP)

Obecná koncepce programování v iOS

UIKit/MVC:

- Třídy (Model, ViewController, View).
- (neřízená) Komunikace mezi objekty. Bez metodiky.

SwiftUI/MVVM

- Data = struktury
- Chování = protokoly
- objekty & closures - referencovatelné hodnoty = druh "pojiva/lepidla" v koncepci programu

Začátečnický kurz. **Příliš to nepřehánět s abstrakcemi.**
Typizované konstrukce.

Referencovatelná hodnota = pojivo

- UI programu je implementováno *struct*.
- Dvě *struct* si spolu "nemůžou povídat", protože se navzájem nemohou oslovit/referencovat.
- Používají tedy "prostředníka" - referencovatelnou hodnotu, kterou mohou sdílet.

Budeme tedy nacházet prostředníky komunikace:

- objekty, closures
- property wrappers
- Binding<Typ>(get: set:)

Demo "pojiva"

```
struct Obrazovka: View {
    // property wrapper vytváří abstrakci nad "obsah"
    @State var obsah = "nejaky string"

    //
    func akce() {
        // syntaktická zkratka: $obsah
        volamNekoho(data: Binding(get: { self.obsah },
        set: { self.obsah=$0 }))
    }
}
```

- @State v sobě ukrývá referenci na string.
- closures get/set jsou referencovatelné hodnoty
- !!! self struktury není referencovatelné, ale odkazuje na *obsah*

Pokud berete SwiftUI jako dogma, nepátráte po smyslu...

Úvod do tříd (classes)

U třídy nás zajímají dva fenomény:

- OOP - dědičnost, polymorfizmus, zapouzdření (abstrakce, rozhraní)
- referencovatelnost objektů = pojivo

V moderním Swiftu však můžeme chování specifikovat protokoly

- Protocol oriented programming.
- Třídy/struktury/enum nechť ho implementují.
- Abstraktní sdílená funkcionalita v protokolu.

Srovnání s UIKitem a hierarchií tříd (UIViewController, UIView).

Syntaxe tříd

```
class TRIDA [:Nadtřída] [:protokoly,...] {
    // properties – uložené, vypočtené, lazyvar
    // metody
    // init metody
    // deinit
}
```

- třída zavádí dědičnost
    - není povinná - Objective-C, NSObject.
- referencovatelný mutable self (na rozdíl od struct/enum)
- lazyvar

Properties třídy

```
class TRIDA {
    // uložená property (bez Objective-C sémantiky)
    let cislo: Int
    var jmeno: String

    // vypočtená property - je var !
    // je to funkce bez parametrů
    var vetsiCislo: Int { return cislo * 1000 }

    // () - evokuje spuštění metody
    lazyvar obsahSouboru: Data = {
        // získej nějak obsah
    }()

    // init - počáteční hodnota props
    // při instanciaci. Dogma č. 1
}
```

Statické atributy třídy

Sémantika static uložených properties:

- jiné jazyky - vypočte se při startu aplikace před voláním main(). Nejsme schopni určit pořadí.
- Swift - vypočte se při **prvním volání get**

```
class Model: ObservableObject {
    // pozor na vlákna
    static let shared = Model()

    //
    init() {
        // tady se spouští datová operace
    }
}
```

Objective-C/Swift **miluje singletony** (shared, default).

Úvod do init(...) metod, konstrukce

```
class TRIDA {
    //
    let foreverConst = 1
    let cislo: Int // [?, !]
    let jmeno: String // [?, !]
    var mutableCislo: Int = 10
    //
    init(...) { ... }
}
```

- props bez počáteční hodnoty => init(...)
- props s ?/! typem => implicitně nil počáteční hodnota
- init metod smí být více, každý MUSÍ komplet inicializovat obsah properties
- struct smí mít implicitní `init`, třídy musí explicitně

Poznámka k Type? a Type!

```
// Optional type, ?, !
var cislo: Int?
var jineCislo: Int!
```

- Type? - Wrapped type, nutný unwrapping (if let, guard let)
- Type! - je totožné s Type? (obsahuje nil), pouze kompilátor nevyžaduje unwrapping (je automatický)

```
// Optional type, ?, !
if let _cislo = cislo { }

// Type! je technicky Type? bez dozoru
cislo = nil; jineCislo = nil
// lze obojí
let kopie = jineCislo // bez dozoru. Seg-F

if let _jineCislo = jineCislo {}
```

Init, sémantika

Init je metoda s pravidly (statická kontrola překladačem):

- naplnit všechny počáteční hodnoty atributů
- `let` property lze v `init` zapsat pouze jednou
- volat případný `super.init(...)`
- pak smí přistupovat na `self` (get/set na atribut, volání metody)

Init, sémantika

```
class TRIDA: NadTrida {
    //
    let cislo: Int
    var jmeno: String

    // z argumentu, nebo v metodě
    init(...) {
        // poprvé smím
        cislo = 1
        // podruhé: error
        cislo = 10
        // inicializace POVINNÁ
        jmeno = "je var, takze ..."
        // až teď smím volat super.init
        super.init(...)
        // obsah naplněn, self je kompletní
        self.volamMetoduObjektu()
    }
}
```

Připomenutí let a var

Při návrhu třídy si musím uvědomit, zda-li se atribut:

- během života objektu NEmění - pak je `let`
- mění, pak smí být `var`

Programátorova sebe-kontrola.

Perlička z UIKitu: životní cyklus UIViewController.

- lazy vyhodnocování obsahu View
- svým způsobem až SwiftUI naplňuje původní záměr Swiftu.

Volání super.init

Až po dokončení inicializace atributů `TRIDA`.

Srovnejme pro zajímavost:

- C++ - voláním konstruktoru `NadTrida` se zahajuje konstruktor `TRIDA`
- Simula67 - namísto konstruktoru je "main procedura třídy", v ní příkaz `inner`.
- ... probublá se až do nejvyšší nadtřídy, v ní se spustí její `main` a v místech `inner` se volá `main` dědice, případně hierarchicky dál

Volání super.init

Inicializátor je metoda, tj smí být při dědičnosti přetížen.

```
class TRIDA {
    //
    let cislo: Int
    //
    init(cislo: Int) { self.cislo = cislo }
}

//
class SubTrida: TRIDA {
    //
    let dalsi: Int
    // musím dodat "dalsi"
    override init(cislo: Int) {
        //
        dalsi = 123; super.init(cislo: cislo)
    }
    // nový init pro oba atributy
    init(cislo: Int, dalsi: Int) {
        self.dalsi = dalsi
        super.init(cislo: cislo)
    }
}
```

Convenience init()

Doposud jsme mluvili o primárních inicializátorech (Designated Initializers).

- Občas potřebujeme v rámci inicializace provést nějakou návaznou činnost.
- tj. nejprve inicializovat objekt
- pak nad ním něco "iniciačního" provést

Motivace: při větším množství init metod se hodí si v nich udělat "pořádek", znovupoužitelnost kódu:

- primární init-y
- navazující init-y

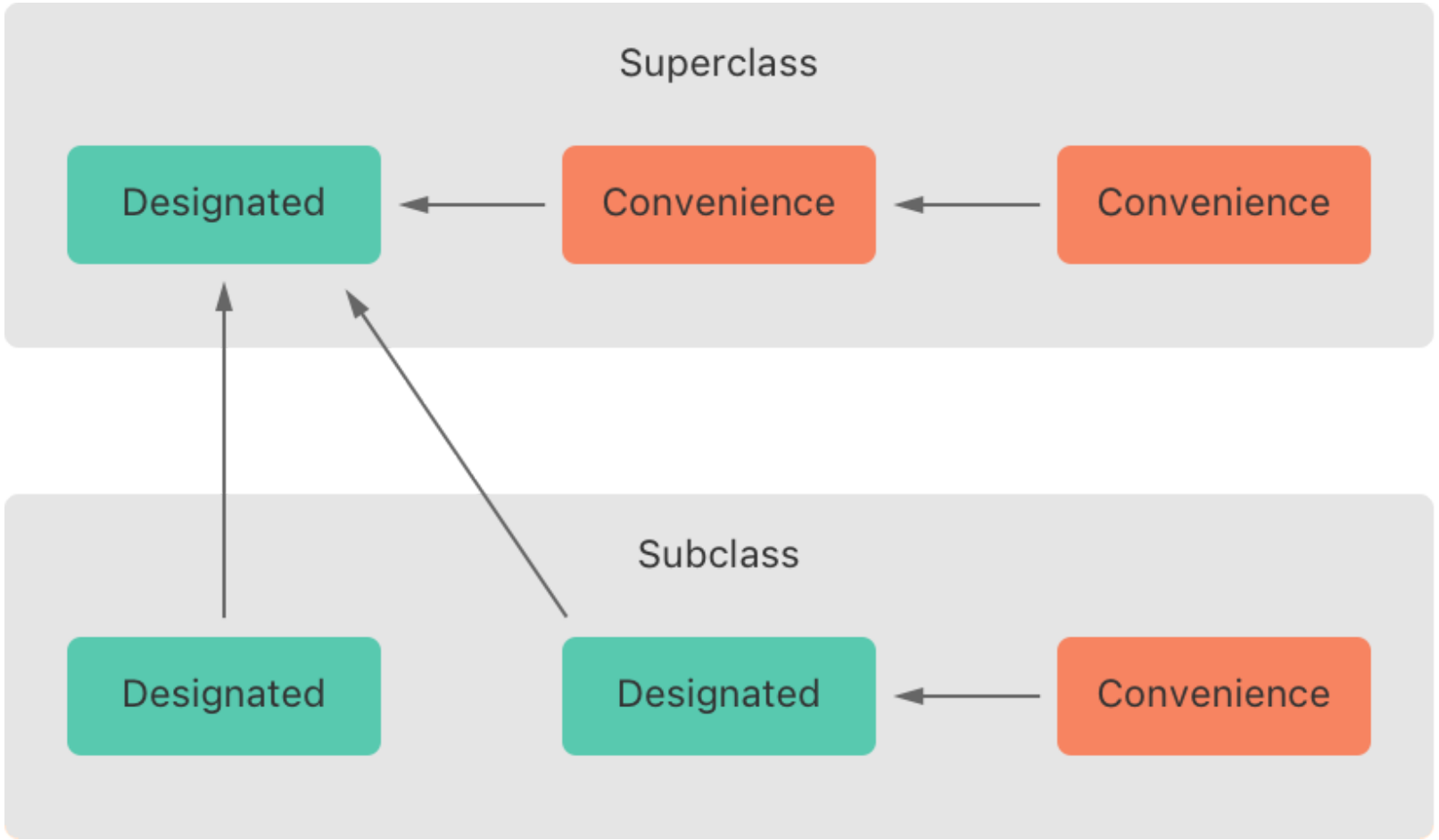
Convenience init()

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

    // primarni/designated init
    init(cislo: Int) {
        //
        self.cislo = cislo
    }

    // navazny init
    convenience init(cislo: Int) {
        //
        init(cislo: cislo)
        //
        // nejaka dalsi akce
    }
}
```

Convenience init()



Convenience init() - konvenčně

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

    // primarni/designated init
    init(cislo: Int) {
        //
        self.cislo = cislo
    }

    //
    func dodatecneInitAkce() -> TRIDA {
        // neco
        return self
    }
}

//
let o = TRIDA(cislo: 10).dodatecneInitAkce()
```

Required init()

```
class TRIDA {
    //
    let cislo: Int
    // klasicky primarni init
    init(cislo: Int) { self.cislo = cislo }
    // jiny init s komplexnim chovanim
    required init(fromJSON: String) {
        // inicializuj se z JSON kodu
    }
}

class OdvozenaTRIDA: TRIDA {
    // je vynucena pritomnost tohoto init
    required init(fromJSON: String) {
        // inicializuj se z JSON kodu
        super.init(fromJSON: fromJSON)
    }
}
```

Required init()

Když nějaký protokol vynutí přítomnost konkrétního `init`, pak toto kaskádově platí i v sub-classes.

```
//
protocol JSONable {
    //
    init(fromJSON: String)
}

//
class TridaA: JSONable {
    // stava se zde REQUIRED
    required init(fromJSON: String) {
        //
    }
}
```

Destruktor

Destruktor je relikt starých jazyků. Kontext AutoreleasePool.

```
class TRIDA {
    //
    deinit {
        //
        print("So long and thanks for all the fish!")
    }
}
```

Užití (které mě napadá):

- uzavření souborů a kanálů
- odhlášení z KVO, odhlášení z NotificationCenter
- nulování weak referencí (později)

Jinak je dynamická paměť spravována automaticky.

Paměťové modely uložených props

Platí pouze !!! pro classes (class type protocols)

Uložená property obsahuje hodnotu:

- s kopírovací sémanitkou - pak není žádný paměťový model (hodnotu musí mít, `Optional` - má případně nil)

... S referencovatelnou sémantikou, pak

- drží referenci na jiný objekt
- vzniká vztah vlastnictví a ten chceme ovlivnit

Odbočka: C++

```
class TridaCPP {
    //
}

// pointer bez přívlastků
typedef TridaCPP *TridaCPP_wptr;

// čítaná reference
typedef std::shared_ptr<TridaCPP> TridaCPP_ptr;

// ...
auto p = new TridaCPP();
auto pstrong = std::make_shared<TridaCPP>();
```

Odbočka: C++

```
class JinaTrida {
    //
    TridaCPP &refNaObjekt;

    //
    JinaTrida(TridaCPP &r) : refNaObjekt(r) {}
}
```

Často vídám předání reference, ale výjimečně uložení reference.

Paměťové modely uložených props

```
class TRIDA {
    // kopírovací sémantika, hodnota tam JE
    var hodnota: Int|String|struct|?|!

    // ref sémantika. Implicitně strong-reference
    var delegate: NejakaTrida
}
```

Objekt `TRIDA` referencuje jiný objekt `NejakaTrida`.

Strong reference: objekt `NejakaTrida` referencovaný přes `delegate` bude žít, dokud žije jeho vlastník, tj objekt `TRIDA`.

Opakování: reference

Předně: rozdíl mezi *pointer* a *reference*.

- Pointer = paměťová adresa + pointerová aritmetika.
- Reference = paměťová adresa, způsob vlastnictví.

Vlastnictví:

- strong - dokud cíl referencuju, cíl žije
- weak - cíl referencuju, co když během toho zanikne?

Implementace strong reference:

- reference counter + garbage collecting
- +/- refCounteru - typicky atomická operace, zámek, režie

Vedlejší efekty strong reference

```
let a = TRIDA(); // refCount := 1
let b = a; // refCount++
volamFunkci(a)
// arg je dalsi vlastnik, tj refCount++
// pri navratu refCount--
func volamFunkci(_ arg: TRIDA) {}
```

- časová režie na reference counting - vadí? Zámky.
- referenční cyklus - velmi typické pro "delegátství"
- referencovaný cíl prostě jednou zanikne, potřebuju se to dozvědět

Referenční cyklus

```
class Pracovnik {
    var delegate: Zadavetel?
    //
    func delamPraciAsyncAPakVolam() {
        // ...
        delegate?.mamHotovo(prac: self, vysledky)
    }
}
// ViewController, který typicky zanikne
class Zadavatel {
    var pracovnik: Pracovnik?
    // ... před dokončením zadané hokny
    func hokna() {
        pracovnik = Pracovnik(delegate: self)
        pracovnik.delamPraciAsyncAPakVolam()
    }
    func mamHotovo(prac: Pracovnik, _ vysledky:...) {
        guard pracovnik===prace else { fatalError()}
        // dealokuji, rozpojuji, ...
        pracovnik?.delegate = nil; pracovnik = nil
    }
}
```

Vedlejší efekty strong reference, C++

Toto ve Swiftu **NELZE**.

```
// predam referenci na reference counter :)
// nezvysuju ref count
void neco(const std::shared_ptr<TRIDA> &o) {
    //
    o->cosiZavolej()
}

// vytvářím objekt s reference countingem
auto _ref = std::make_shared<TRIDA>(...)
// zvyšuju refCount
auto _kopie = _ref;
// předávám referenci na referenci
neco(_ref)
```

Weak reference

Objekt `A.vazba` referencuje `TARGET`

```
class A {
    // je VAR, let je bez diskuzí...
    weak var vazba: TARGET?
}
```

Sémantika:

- uložená property musí být `Optional`, tj připouští `nil`
- navázání **nezvyšuje** refCount `TARGET`
- při zániku `TARGET` provede Swift-runtime automaticky nulování `vazba`
- implementace: `TARGET` si drží kolekci weak-vlastníků

Unowned/Unsafe reference

Weak reference = režie. Když chci odhodit tuto režii, tj:

- `TARGET` zanikne, nepředá zprávu
- `vazba` není nulována, tj ukazuje chybně do paměti

Jak se to v runtime ošetří při přístupu `vazba`:

- `unowned` - Swift-runtime garantuje exception
- `unowned(unsafe)` - žádná garance

Předávání referencovatelné hodnoty

Se beztak vždy odehrává strong formou.

```
// cosi: refCount++
func udelej(cosi: TARGET) {}

class A {
    // je VAR, let je bez diskuzí...
    weak var vazba: TARGET?

    //
    func predej() {
        // tady je _v opět strong ref
        if let _v = vazbe {
            // tj refCount++
            udelej(cosi: _v)
        }
    }
}
```

OOP ve SwiftUI

Model-View-ViewModel architektura:

- Model - struct, class, enum
- View - struct
- ViewModel - struct zaobalující objekty

UIKit - je OOP knihovna
SwiftUI - je POP (Protocol Oriented P.) knihovna, silně šablonově orientovaná

Nejvíc OOP bude v Combine a to bude "template hell" :)

Closures - { i,j in ... }

Closures: Úvod

Funkce - globální, lokální (metoda), referencovatelná.

```
// argument: Int, vrací: Int
typealias MojeFunkce = (Int) -> (Int)
```

Funkce/lambda výraz/closures jsou **referencovatelné objekty**.

- Funkci mohu předat jako data.
- Uložit jako data (do proměnných, kolekcí).
- Zavolat.

Funkce = pojmenovaný lambda výraz (nebo closure).

Strukturované programování

Co je funkce? Co je **ideál** funkce?

- blok kódu se vstupy a výstupy.
- nemá vedlejší efekt, tj stav programu.

S přechodem k OOP toto modifikujeme:

- metoda smí (je určena) měnit stav svého objektu (self).
- metodu volám v kontextu jejího `self`.

Takže budou existovat lambda výrazy, které:

- pochází z nějakého kontextu,
- … a chtějí ho modifikovat.

Lambdy (zatím bez významu closure)

```
// (Int)->Int
let l1 = { (i: Int) in return i * 10 }

// () -> (), Void->Void
let l2: ()->() = { print("hello") }

// při volání NEpojmenovávám argumenty
print(l1(10))
//
l2()

// minimalistická syntaxe
// typová inference -> "i" je Int
let l3 = { i in return i * 10 }
let l4 = { return $0 * 10 }
let l5 = { $0 * 10 }
```

Lambda jako argument funkce

```
func delej(cislo: Int, blk: (Int)->(Int)) {
    // volám blok
    print(blk(cislo))
}

// volání
delej(cislo: 10, blk: { (i:Int) in i + 1 })
delej(cislo: 10, blk: { i in i + 1 })
delej(cislo: 10, blk: { $0 + 1 })
```

Trailing Closure

Syntaktická zkratka. Velmi rozšířeno.

Definice: Je-li poslední argument hlavičky volání funkce lambda, pak lze argument "položit" za hlavičku volání.

```
func delej(cislo: Int, blk: (Int)->(Int)) {
    // volám blok
    print(blk(cislo))
}

// volání s trailing closure
delej(cislo: 10) { (i:Int) in i + 1 }
delej(cislo: 10) { i in i + 1 }
delej(cislo: 10) { $0 + 1 }
```

Trailing Closure

Poznat, že se jedná o volání funkce.

```
func delej(blk: (Int)->(Int)) {
    // volám blok
    print(blk(1))
}

// volání s trailing closure
delej { (i:Int) in i + 1 }
delej { i in i + 1 }
delej { $0 + 1 }

// async je funkce
DispatchQueue.main.async { print("Hello) }
```

Co je closure?

Dosavadní konvence nad funkcemi:

- funkce má vstup a výstup, kromě toho "nešahá" (čtení, zápis!) na data z okolí (typicky globální data)
- smí volat jiné funkce

Tomu říkáme **funkce nemá vedlejší efekt**.

```
var cislo = 10
var jineCislo = 3
// vedlejší efekt je ZLO
// čtení "cislo"
// zápis "jineCislo" je TUPLOVANÉ ZLO
func udelej() {
    //
    jineCislo = cislo
}
```

Globální data aplikace

Jsou pochopitelně potřeba.

Ale nějak "institucionalizována" :)

```
// typicky v singleton objektech
class GlobAppData {
    //
    static let shared = GlobAppData()

    //
    var hesloOdUzivatele: String = "..."
}

//
let v = GlobAppData.shared.hesloOdUzivatele
```

Co je closure?

- u metod (funkcí v struct/class) vedlejší efekt připouštíme, je zcela normální
- naopak říkáme, že na data objektu se SMÍ sahat výhradně prostřednictvím metod

```
class TRIDA {
    private var cislo = 3
    //
    func nastav(cislo: Int) { self.cislo = cislo }
}
```

Definice: Closure je lambda výraz s vedlejším efektem.

Ok, začíná nová skupina problémů...

Jak vzniká closure?

Typicky "vytrhneme z kontextu" část kódu a někomu ji předáme.

```
class TRIDA {
    //
    var hodnota: Int = 0
    // například akce na nějaké tlačítko
    func prace(vykonavac:Nekdo) {
        // jakože inicializace
        hodnota = 0

        // trailing closure
        vykonavac.pracuj(vstupy) { vysledek in
            // poznačím si výsledek volání "pracuj"
            self.hodnota = vysledek
        }
    }
}
```

Dávám blok kódu, který smí zasahovat do instance třídy (self).

Jak vzniká closure

Proč vlastně closure potřebujeme?

Closures tvoří potenciál pro asynchronní činnost programu (řízeného událostmi).

```
class TRIDA {
    //
    var hodnota: Int = 0

    // například akce na nějaké tlačítko
    func prace(vykonavac:Nekdo) {
        // synchronne....
        hodnota = vykonavac.pracuj(vstupy)
    }
}
```

Jak vzniká closure

Grand Central Dispatch. Multi-vláknovost.

```
class TRIDA {
    //
    var hodnota: Int = 0
    // například akce na nějaké tlačítko
    func prace(vykonavac:Nekdo) {
        // jakože inicializace
        hodnota = 0
        // volám funkci "async", trailing closure
        DispatchQueue.global.async {
            // nejaky vypocet
            let vysledek = pracuj(hodnota)
            // volám funkci "async" na main queue
            DispatchQueue.main.async {
                //
                self.hodnota = vysledek
            }
        }
    }
}
```

Escaping & Non-Escaping Closure

Budeme zkoumat životní cyklus kontextu, ze kterého closure pochází.

Non-Escaping = synchronní volání closure

Escaping = asynchronní volání closure

```
// dostávám blok, který v sobě zavolám (sync)
func sync_call(blk: ()->()) {
    // volám ho...
    blk()
}
```

Escaping & Non-Escaping Closure

```
// dostávám blok, který si "odložím na později"
var prace: [()->()] = []

// je to @escaping closure
func async_call(blk: @escaping ()->()) {
    // nevolám sync, uložím si ho na později
    prace.append(blk)
}

// později někdo...
func pracuj() {
    //
    for i in prace {
        // volej
        i()
    }
}
```

Escaping closure

Closure vytržený ze svého kontextu si musí podržet kontext, ze kterého vychází.

Kontext je struct/class, pak si bere kopii `self`. Kompilátor přikazuje explicitně `self.` nad atributy struct/class.

```
class TRIDA {
    var hodnota = 0
    //
    func akce() {
        // volám nějakou async práci
        volejKnihovnu(arg1) { vysledek in
            // self. je teď nutné
            self.hodnota = vysledek
        }
    }
}
```

Escaping closure

Beru si kopii `self`. Co to znamená? Copy/Ref sémantika.

```
class TRIDA {
    var hodnota = 0
    //
    func zpracujVysledek(i: Int) {
        hodnota = i
    }
    //
    func akce() {
        // volám nějakou async práci
        volejKnihovnu(arg1) { vysledek in
            // self. je teď nutné
            self.zpracujVysledek(i: vysledek)
        }
    }
}
```

Escaping closure

Beru si kopii `weak self`. Co to znamená?

```
class TRIDA {
    var hodnota = 0
    //
    func zpracujVysledek(i: Int) {
        hodnota = i
    }
    //
    func akce() {
        // volám nějakou async práci
        volejKnihovnu(arg1) { [weak self] vysledek in
            // ten objekt self už nemusí existovat
            guard let _self = self else { return }
            _self.zpracujVysledek(i: vysledek)
        }
    }
}
```

Escaping closure

Beru si kopii `self`. COPY sémantika.

```
//
func volejKnihovnu(_ arg: Int, blk: @escaping (Int)->()) {
    //
}
//
struct STR {
    var hodnota = 0
    // struct => self je immutable
    mutating func zpracujVysledek(i: Int) {
        hodnota = i
    }
    //
    mutating func akce() {
        // volám nějakou async práci
        volejKnihovnu(1) { vysledek in
            // self je immutable. ERROR
            self.zpracujVysledek(i: vysledek)
        }
    }
}
```

Non-Escaping pro struct

```
// je SYNC! non-escaping
func volejKnihovnu(_ arg: Int, blk: (Int)->()) {
    //
    blk(123)
}
//
struct STR {
    var hodnota = 0
    // struct => self je immutable
    mutating func zpracujVysledek(i: Int) {
        hodnota = i
    }
    //
    mutating func akce() {
        // volám nějakou SYNC práci
        volejKnihovnu(1) { vysledek in
            // ...
            zpracujVysledek(i: vysledek)
        }
    }
}
```

Closures: závěry

- Kompilátor ve funkci odliší `@escaping` a `non-escaping` closure.
- Tento fakt si vynutí do hlavičky funkce.
- Tento fakt si vynutí do místa volání funkce.
- Escaping closure bude vyžadovat `mutable self`.

Kontextovost překladače Swiftu.

- podobně `throws`

Enumeration

`struct` / `class` / **enum**

Enumeration - výčtový typ

Hodnota může nabývat formy A nebo B nebo C ...

```
enum KodAkce {
    case pomala
    case rychla
}

// Kontextově zřejmé, zkrácená syntaxe
let a: KodAkce = .pomala
let b = KodAkce.pomala

//
if a == .pomala {
    //
}
```

Enumeration: Raw Value

```
// pouze pro primitivní typy
enum KodAkce: Int {
    case pomala = 0
    case rychla = 1
}

// ... je Optional !!!
let a = KodAkce(rawValue: 1)

// syntakticky přípustné, dynamicky neuhlídatelné
let cisloAkce: Int = volaniFunkce() // -> Int
let b = KodAkce(rawValue: cisloAkce)

// test
guard let _b = b else {
    //
    fatalError("Chybna hodnota vstupu")
}
```

Parametrizovatelný Enum

```
enum Akce {
    case nop
    // jeden strukturovaný typ
    case volej(telefon: String)
    // druhý strukturovaný typ
    case precti(String, Date)
    // rekurze...
    indirect case poPauzeUdelej(pauza: Int, akce: Akce)
}

//
let a = Akce.volej(telefon: "123-456-789")
let b: Akce = .poPauzeUdelej(pauza: 2, akce: .nop)
```

Testování/rozbalení hodnoty enum

```
var a: Akce

// trivialni if - test
if a == .nop { ... }

// case - binding - pattern matching
if case .volej(let cislo) = a {
    //
    print("cislo je \(cislo)")
}
// "let" lze "vytknout pred zavorku"
if case let .volej(cislo) = a {
    //
    print("cislo je \(cislo)")
}
```

Testování/rozbalení hodnoty enum

```
var a: Akce

//
switch a {
    case .nop: cosi()
    case .volej(let cislo): cosi()
    // ...
    default:
        fatalError("Implementuj vsechny pripady")
}
```

Pozn.: `default` label je povinný ve `switch`, pokud nejsou ošetřeny všechny případy. Prázdný příkaz `()`.

Závěr

Zůstavají nám (menší) témata:

- exceptions - kódování dat
- async funkce - GCD a multi-threading
- plus tísíc malých detailů do dalších tematických přednášek

A velká témata na příště:

- Protocol Oriented Programming
- šablony, templates, ...