Swift III.: Protokoly & Generické programování

Programování zařízení Apple (IZA)

Martin Hrubý, FIT VUT, 2023/24

Něžný úvod do Protokolů & Extensions

Protokol = sada hlaviček funkcí

- **Delegátství** - objekt `A.delegate` referencuje `B:Protokol` a posílá mu zprávy protokolu `Protokol`
- **DataSource** - forma delegátství, kdy je cílem model obsahu.
- … a dnes i nové pohledy na protokol.

Extension = vlastnost jazyka, dodatečné doplnění rozhraní.

Pozn.: organizace zdrojových textů (hlavičky, extensions, packages - public).

- speciálně pro Extensions…! Může dojít ke značnému zmatení čtenáře.

Extension

Smalltalk world (image a jeho svět). Objective-C.

Nyní i ve Swiftu.

```
extension Int {
    // NELZE doplnit data, POUZE kód
    var kratDva: Int { self * 2 }
}

//
let a = 10
let b = a.kratDva
```

Smím dodat nové funkce (vypočtené props) do

- tříd, struktur, enum - rozšiřuji interface
- protokolů - **co budou značit funkcionality dodané do protokolů???**

Přístup na atributy (prvky programu)

Rozdílné u aplikačního programu a knihovny:

- `public` (implicitní pro program)
- `private`
- `fileprivate`
- `internal`

Knihovna je odlišný modul, takže z pohledu programu:

- jsou všechny její prvky `internal`
- proto od počátku psát `public`
    - zejména implicitní konstruktory struktur

Extension struct & class

```
class TRIDA {
    // atributy
    var cislo: Int
}

//
extension TRIDA {
    // metody instanční a statické
    func hello() { }
    static func helloStatic() { }

    // property, avšak bez rozšíření dat
    var cisloAsString: String {
        //
        get { return "\(cislo)" }
        set {
            if let _c = Int(newValue) {
                cislo = _c
            }
        }
    }
}
```

Užití Extension struct & class

- rozšíření knihovních typů
- organizace zdrojáku - do souborů, do tematických celků

Příklad:

- deklarace třídy - `trida.swift`
- metody tematického celku "model" - `trida-model.swift`
- extension pro každý protokol zvlášť

Extension struct & class

Extension je jediný způsob dodání metod do `enum`.

```
enum Message {
    case none
    case normal(String)
}

//
extension Message: CustomStringConvertible {
    //
    var description: String {
        // self...
        switch self {
            case .none: return "none"
            case .normal(let X):
                return "normalni:\(X)"
        }
    }
}
```

Protokol

Definice: Protokol je pojmenovaná sada hlaviček funkcí.

```
protocol KamHlasimUdalosti /* : AnyObject */ {
    //
    func startuju()
    func hotovyVypocet(vysledek: Data)

    //
    var mamPokracovat: Bool { get }
}
```

Původním smyslem protokolů bylo **delegátství**. Objekt objektu hlásí zprávy z protokolu, případně žádá vstupy (**DataSource**).

Implementovat protokol

```
// class, struct (, enum)
class ViewController : KamHlasimUdalosti {
    //
    func startuju() { print("Uz to jede...") }
    func hotovyVypocet(vysledek: Data) { ... }
    var mamPokracovat: Bool { true }
}
```

Typičtěji však via Extension.

```
//
class ViewController { ... }
// Objective-C: tzv. category
extension ViewController : KamHlasimUdalosti {
    //
    func startuju() { print("Uz to jede...") }
    func hotovyVypocet(vysledek: Data) { ... }
    var mamPokracovat: Bool { true }
}
```

Properties v protokolu

Deklarujeme povinnost zavést properties (get/set) na straně implementátora.

```
protocol MujProtokol {
    // get
    var mamPokracovat: Bool { get }

    // get i set
    var seznam: [String] { get set }
}
```

Hodnota typu "protokol"

Hodnota typu `KamHlasimUdalosti` je "cosi", komu lze poslat zprávu z protokolu. Víc o něm nevíme.

```
func nastartuj(p: KamHlasimUdalosti) {
    //
    p.startuju()

    //
    while p.mamPokracovat == true {
        //
        p.hotovyVypocet(vysledek: ...)
    }
}
```

Hodnota typu "protokol"

Pokud hrozí referenční cyklus.

```
class TRIDA {
    // hodnota typu KamHlasimUdalosti
    // weak => protocol je AnyObject
    weak var delegate: KamHlasimUdalosti?

    //
    func hlaseni() {
        //
        delegate?.startuju()
    }
}
```

Producent-Konzument, ref cyklus

Může žít `Pracovnik` déle než `Zadavatel`? Naprosto obvyklé.

```
// Producent nějakých událostí, dat
class Pracovnik {
    // delegátská reference
    weak var delegate: KamHlasimUdalosti?
    // ...
    func delamPraciAsyncAPakVolam() {
        // ...
        delegate?.mamHotovo(prac: self, vysledky)
    }
}
// Konzument. Je VLASTNÍKEM producenta
class Zadavatel {
    var pracovnik: Pracovnik?
    //
    func hokna() {
        pracovnik = Pracovnik(delegate: self)
        pracovnik.delamPraciAsyncAPakVolam()
    }
}
```

Prod-Konzument, stav konzumenta

Typicky akce na tlačítko.

```
//
class MujViewController: UIViewController {
    var pracovnik: Pracovnik?
    //
    func akceNaTlacitko() {
        // stav objektu!
        // disable() na tlačítko, nejak ošetřit
        guard pracovnik == nil else { ... }

        //
        pracovnik = Pracovnik(delegate: self)
        pracovnik.delamPraciAsyncAPakVolam()
    }
}
```

Delegátství je relikt z UIKit

UIKit=hromady protokolů roztodivných jmen:

- `NSFetchedResultsControllerDelegate`
- `UITableViewDataSource`

Ve SwiftUI je tento koncept už mimo.

Delegátské protokoly postupně nahrazovány closures.

Prod-Konzument, @escaping closures

```
//
class Pracovnik {
    // delegátská reference !!! REF-CYCLE
    var delegate: (Results) -> ()
    // ...
    func delamPraciAsyncAPakVolam() {
        // ...
        delegate(results)
    }
    init(delegate: @escaping (Results)->()) { self.delegate = delegate }
}
// Konzument. Je VLASTNÍKEM producenta
class Zadavatel {
    var pracovnik: Pracovnik?
    //
    func hokna() {
        pracovnik = Pracovnik { result in
            // nekam si to zapíšu
            self.pracovnik = nil
        }
        //
        pracovnik.delamPraciAsyncAPakVolam()
    }
}
```

Escaping closure

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

```
//
class Pracovnik {
    // delegátská reference
    var delegate: (Results) -> ()
}

// Konzument. Je VLASTNÍKEM producenta
class Zadavatel {
    var pracovnik: Pracovnik?
    //
    func hokna() {
        pracovnik = Pracovnik { [weak self] result in
            guard let _self = self else {...}
            // nekam si to zapíšu
            _self.cosikdesi = result
            _self.pracovnik = nil
        }
    }
}
```

Referencování producenta

Referenci na `Pracovnik` je třeba někde držet. Typicky:

- explicitně v uložené prop objektu
- automaticky v nějaké frontě (OperationQueue, GCD)

Delegátství (varianty):

- `weak var delegate: TYP?` ; `TYP` je referencovatelné
- `var delegate: (Result)->()` a `[weak self]`

Pozn.: (!!!) objekty musí komunikovat výhradně přes hlavní vlákno (bude jasné později).

Protokol jako vlastnost

`Self` je symbol pro TYP implementátora.

Tady uz začíná mít protokol **generické chování**.

```
protocol Comparable {
    // Self = typ mě, až to budu implementovat
    static func < (l: Self, r: Self) -> Bool
}
```

Implementátor garantuje binární operaci `<` nad sebou.

```
// NELZE
protocol Comparable {
    // NELZE - zkusme si to analyzovat/zdůvodnit
    static func < (l: Comparable, r: Comparable) -> Bool
}
// Pokud bychom toto chtěli, pak šablonovitě
```

Protokol jako vlastnost

```
struct Event: Comparable {
    //
    let time: Date
    let priority: Int

    //
    static func < (l: Event, r: Event) -> Bool {
        //
        if l.time == r.time {
            //
            return l.priority > r.priority
        }

        //
        return l.time < r.time
    }
}
```

Protokol -> extension důsledky

Implementátor garantuje binární operaci `<` nad sebou.

Abstraktní chování založené na protokolu.

```
// Array je šablona s typovým argumentem Element
struct Array<Element> {}

// vkládám EXTENSION nad Array pro
extension Array where Element: Comparable {
    //
    var sorted: [Element] { ... s pomocí < }
}
```

Protokol jako vlastnost

Swift Standard Library. (pak je ještě Foundation)

- `Equatable`, `==`, abstraktně sekvenční vyhledávání
- `Identifiable`, `id: Hashable`
- `Hashable` -> hash hodnota pro `Self`
- `CustomStringConvertible`, `description: String`
- `Codable` -> JSON encoder/decoder

Identifiable

```
struct Record: Identifiable {
    // libovolně implementované get "id"
    // splňující Hashable
    let id = UUID()
    let jmeno: String
}

//
struct Person: Identifiable {
    // String je Hashable
    let rodneCislo: String
    var id: String { rodneCislo }
}

//
class TRIDA: Identifiable {
    // ... je Hashable
    var id: ObjectIdentifier { ObjectIdentifier(self) }
}
```

Hashable

```
//
struct Clovek: Hashable {
    //
    let rc: String
    let jmeno: String
    // nacpi to do Hasher, on z toho udela HASH value
    func hash(into hasher: inout Hasher) {
        //
        hasher.combine(rc)
        hasher.combine(jmeno)
    }
}

// struct Set<Element> where Element: Hashable {}
typealias SetOfClovek = Set<Clovek>
typealias MapOfClovek = [Clovek:String]
```

CustomStringConvertible

```
//
struct Clovek: CustomStringConvertible {
    //
    let rc: String
    let jmeno: String

    // vyuzito i pri ladeni v XCode
    var description: String {
        // formátování stringu
        "Clovek \(jmeno), RC: \(rc)"
    }
}

//
let c = Clovek(...)

//
print(c)
```

Třídní protokol

Potřebuji někdy přikázat, aby případný implementátor byl třídou, tj měl referencovatelnou podobu.

```
// tento protokol bude mit třídní chování
protocol JenomProTridy: AnyObject {
    //
}
```

`self:Self` je mutable, referencovatelné

Dědičnost protokolů

Protokoly mohou slučovat vlastnosti/požadavky jiných protokolů.

Proč bychom to chtěli? Protokol vyjadřuje:

- delegátství, data source (typické pro UIKit)
- zobecnitelnou vlastnost - Comparable, Hashable, Identifiable
- požadavek na rozhraní někoho

Pak lze vytvářet abstraktní sdílené chování.

```
// protokol Codable je sloučení...
protocol Codable: Encodable & Decodable {
    //
}
```

Obecné chování pro implementátory protokolů

```
protocol HavingAName {
    //
    var name: String { get }
}

//
extension HavingAName {
    //
    func hello() { print("Hello \(name)")}
}

//
struct Person: HavingAName {
    //
    let name: String
}
```

Dědičnost protokolů

Kolem každého protokolu vzniká nějaká abstraktní funkcionalita a model chování. To může mít hierarchii.

```
protocol NotOrdered { ... }
// nějaké abstraktní chování
extension NotOrdered {
    //
    func cosi() {}
}

// dědím a rozšiřuji
protocol Ordered: NotOrdered {
    //
}
// dodám další abstraktní chování do protokolu
extension Ordered {
    //
    func necoDalsihi() {
        //
    }
}
```

Dědičnost struktur a enum

```
struct StrA: Ordered {
    //
}
```

`StrA` implementuje protokol **a současně získává jeho chování** (z extensions). Struktury ve Swiftu nemají dědičnost.

V důsledku: Lze u struktur imitovat dědění metod a atributů, tj. OOP. Uvidíme to i SwiftUI.

V tomto smyslu jsou abstrakce prostřednictvím protokolů obecnější než OOP.

OOP, POP

OOP = třída `TRIDA` deklaruje rozhraní a pak ho následně sama (sobě si) implementuje.

- … dále provádíme odvozování podtříd …

POP = protokol `Something` deklaruje rozhraní.

- k němu extension dodává obecnou funkcionalitu,
- protokol se dále odvozuje v rámci dědičnosti,
- a někdo ho implementuje.

Náročnější téma.

POP-like styl v C++

```
class AbstractClass {
    // protokol pro implementatora
    virtual void cosiUdelej() = 0;

    // obecne chovani
    void    pracuj() { cosiUdelej(); }
}

// implementator
class Specificke: AbstractClass {
    //
    virtual void cosiUdelej() { ...}
}
```

POP-like styl v C++ (Swift ekvivalent)

```
// rozhrani
protocol CosiProtokol {
    func cosiUdelej()
}

// obecne chovani pro implementatory CosiProtokol
extension CosiProtokol {
    func pracuj() { cosiUdelej(); }
}

// implementator
class Specificke: CosiProtokol {
    func cosiUdelej() { ... }
}
```

Generické programování

Protokoly: `associatedtype`

Úvod

Je třeba zopakovat, že protokol dává možnost deklarovat abstraktní chování platné pro všechny jeho implementátory.

```
protocol HavingAName {
    //
    var name: String { get }
}

//
extension HavingAName {
    //
    func hello() { print("Hello \(name)")}
}
```

SwiftUI View

```
public protocol View {

    /// The type of view representing the body of this view.
    ///
    /// When you create a custom view, Swift infers this type from your
    /// implementation of the required ``View/body-swift.property`` property.
    associatedtype Body : View

    /// The content and behavior of the view.
    ///
    /// When you implement a custom view, you must implement a computed
    /// `body` property to provide the content for your view. Return a view
    /// that's composed of built-in views that SwiftUI provides, plus other
    /// composite views that you've already defined:
    ///
    ///     struct MyView: View {
    ///         var body: some View {
    ///             Text("Hello, World!")
    ///         }
    ///     }
    ///
    /// For more information about composing views and a view hierarchy,
    /// see <doc:Declaring-a-Custom-View>.
    @ViewBuilder var body: Self.Body { get }
}
```

Generické programování

Základem generického programování **je kopírování zdrojáku v okamžiku překladu**.

Generický typ (šablona/template): jako MAKRO.

- preprocessor překladu, generování variant
- (až na pár okamžiků, kde jsou virtuální funkce…)

Programátorský vtip ;)

```
#define true (rand() > 10)
```

Generické programování

Projdeme si to po částech:

- template globální funkce
- template struct/class/enum
- template instanční metody (včetně exensions)
- šablonovité **protokoly** (associated types)

C-makro (jako demo...)

```
#define F(Type) void Funct(Type arg) { ... }

// Instancuj pro Type=int, double, ...
F(int)
F(double)
//
```

Princip generických prog. jazyků je v automatizovaném generování instancí (klonů.)

- zjišťuje se průběžně při překladu
- identifikátory, přemapování

```
F(1) // pro int
F(3.0) // pro double
```

Vytvoření šablony

- Kus kódu (funkce, struct, ...) kde je **typový parametr**.
- Typově parametrizovaný kód.
- Syntaxe `< Param1, Param2, ... >`

Symboly:

- `self` - je reference na sebe.
- `Self` - vyjádření **typu** sebe (třídy, ...)
- `Self.self` - je reference na objekt představující třídu sebe.
- `Param.Type` - vyjádření typu parametru.

Třída (na meta úrovni) je objekt, který v OOP má význam třídy.

Typové symboly

- `Int.self` - vyjadřuju typ `Int` datově
- `Record.self`
- `[Record].self`

Syntaxe:

- když překladač vidí identifikátor (např. `Record`, pak čeká konstrukci hodnoty `Record(...)`)
- když chci předat typ jako hodnotu, pak

```
volaniFunkce(vytvor: Record.self)
// nelze...Record()
volaniFunkce(vytvor: Record)
```

Template funkce (globální)

```
// pozor: toto Swift odmítne...
func plus<X>(a: X, b: X) -> X {
    //
    return a + b
}
```

- `X` je typový parametr.
- `plus<X>` je deklarace šablonové funkce s `X`.
- `a: X` - deklaruji argument `a`, který má nabývat hodnoty `X`.

C++ překladač toto akceptuje (instancuje až na požádání).

Swift chce typovou kontrolu **už na úrovni šablony**.

- neznáme rozsah `X`, není garance `+` obecně.

Template funkce (globální)

Upřesnění typu `X`:

```
//
func plus<X:Numeric>(a: X, b: X) -> X {
    //
    return a + b
}
```

```
func plus<X>(a: X, b: X) -> X where X:Numeric {
    //
    return a + b
}
```

Teď překladač Swiftu typově zkontroluje, že lze šablonu kdykoli instancovat.

Typový constraint

`X:Interface`

- `X` implementuje rozhraní `Interface`, tedy:
- `Interface` je protokol, pak...
- `Interface` je třída

`X==Interface`

- `X` je totožné s `Interface`

Logické spojky: `X:Protocol && X:JinyProtokol`

Strukturovanost: `X.Element==Cosi && ...`

Demo "plus"

```
//
func plus<X:Numeric>(a: X, b: X) -> X {
    //
    return a + b
}

// ok
let _ = plus(1,2)
let _ = plus(1.0,2.0)
// ne-ok
let _ = plus(1, 2.0)
```

Typové symboly v šablonách

```
func dekoduj<T:Codable>(type: T.Type,
                        from: String) -> T {
    //
}
```

- `<T:Codable>` - podmínka na parametr `T`
- `type: T.Type` - je datový typ `T`
- `type: T` - je hodnota typu `T`
- `-> T` - je hodnota typu `T`

```
let a = dekoduj(type: Record.self, from: "...")
```

Typová inference. Většinou zabere. Jinak specifikovat typ "ručně".

```
let a: Record = dekoduj(type: Record.self, from: ...)
```

Příklady

```
func A<X>(a: X) { ... }
func B<X>(b: [X], arg: (X)->(X)) { ...}
```

Vytvoření hodnoty: potřebujeme konstruktor.

```
protocol Construable { init() }

//
func zeros<X:Construable>(m: Int) -> [X] {
    //
    var _out = [X]()

    //
    for _ in 0...m { _out.append( X() ) }

    //
    return _out
}
```

Template struct/class

```
struct Stack<Element> {
    //
    var list = [Element]()
    //
    mutating func push(_ invalue: Element) { ... }
}
```

Užití (typová inference)

```
var stack = Stack<Int>()
var stack2 = Stack<Record>()
//
stack.push(3)
```

Template struct/class

```
struct Stack<Element> {
    // ...
    init(_ type: Element) {}
}
// typova inference, jde to...
var stack = Stack(3)
```

```
struct Stack<Element> {
    // ...
    init(_ type: Element.Type) {}
}

// typova inference, jde to...
var stack = Stack(Int.self)
```

Template struct/class

```
func CR<T>(_ type: T.Type) -> Stack<T> {
    //
    return Stack<T>()
}

// typova inference, jde to...
var stack = CR(Int.self)
```

Šablonové prvky Swift Standard Lib

- kolekce, `Element`
- kódování, protokoly `Encodable`, `Decodable`, `Codable`
- keyPath - větší povídání ...

Combine, typické (`async` volání, `Future`):

```
enum Result<Success, Failure> where Failure : Error {
    case success(Success)
    // zjednodušeně
    case error(Failure)
}
```

Šablonové metody strukt. typů

```
class Logging {
    //
    func log<X:CustomStringConvertible>(_ value: X) {
        // zapiš do logu
        print(value.description)
    }
}

//
let l = Logging()

//
l.log(1234)
l.log("hello")
```

Generické protokoly

Associated Type

```
// genericky typ, T
struct Kontejner<T> {
    //
}
```

U struct/enum/class lze deklarovat šablonový typ nad typovým parametrem `T`.

U protokolů to děláme jinak. **Associated (přidružený) types.**

Associated Type

- Protokol nemůže být přímo šablonový.
- Ale může mít šablonové metody.

```
protocol Kontejner {
    // potenciálně různé mínění T, že...
    func add<T>(v: T)
    func remove<T>(v: T)
}

protocol AssocKontejner {
    // jednotné T. Deklarace.
    associatedtype T
    //
    func add(v: T)
    func remove(v: T)
}
```

Associated Type

V tomto smyslu budeme rozlišovat:

- protokoly
- protokoly s asociovaným typem

```
// error, nezname zcela typ hodnoty "k"
func cosi(k: AssocKontejner) {

}
```

a tohle nelze:

```
// error, nezname zcela typ hodnoty "k"
func cosi(k: AssocKontejner<Int>) {

}
```

Associated Type

Typická ukázka. Iterování přes typicky kolekce.

```
//
protocol Iterable {
    // typovy parametr
    associatedtype  Element

    // cílový typ iterování není určen. Je generický.
    mutating func next() -> Element?
}
```

Iterovatelný nad typem `<Element>` je každý, kdo je schopen dodat dalšího v pořadí.

Demo

```
//
struct ArrayIterator<ArrayElement> : Iterable {
    // mapovani typovych parametru explicitne
    typealias Element = ArrayElement
    //
    let on: [ArrayElement]
    //
    mutating func next() -> Element? { ... }
}
```

Mapování typových parametrů implicitně:

```
// typový argument Element je zaveden v Iterable
struct ArrayIterator<Element> : Iterable {
    // ...
    mutating func next() -> Element? { ... }
}
```

Associated Type

- `associatedtype` je typový parametr protokolu
- implementátor ho musí specifikovat (`typealias` , šablonový typ)
- na `associatedtype` můžeme klást požadavky (type constraints) a tvořit abstraktní chování protokolu

Protokol mající `associatedtype` je tzv. **nekompletní protokol**, tj není přímo datovým typem.

- tj, nelze předat hodnotu typu nekompletní protokol (např. `Iterable`)
- lze předat hodnotu typu `some Iterable`

Kompletní/nekompletní protokol

```
// protokol je skořápka chování
protocol Person {
    //
    var name: String { get }
}

// abstraktní chování nad kompletním protokolem
func sayHello(to: [Person]) {
    //
    to.forEach { p in print("Hello \(p.name)")}
}

//
extension Array where Element: Person {
    //
    func sayHello() {
        for i in self { print("hello \(i.name)")}
    }
}
```

Závěr

- Protokoly + šablonové protokoly.
- Extensions.
- Protocol Oriented Programming.

Dnes jsme zanedbali konstrukci `some Protocol`, ale tu si necháme do úvodu SwiftUI (příště).