Swift III.

Výčtový typ enum,
Protokoly, Extensions, Generické programování

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

Enum

- Typ nabývá hodnoty A nebo B nebo C…pod A,B,C si dosaďme cokoli.
- Výčet může být:
  - konečný,
  - parametrický,
  - rekurzivní.

Enum: demo

```
//
enum Colors {
    //
    case red, green, blue
}

//
func something(color:Colors) {}

// je znamy kontext
let p: Colors = .red
// kontext musi byt explicitne urcen
let p2 = Colors.green

// kontext je znamy
something(color: .blue)
```

Raw values...

```
//
enum Colors : String {
    //
    case red = "Red", green = "Green", blue="Blue"
}

//
print(Colors.red.rawValue)

// je typu Colors?
if let _fromraw = Colors(rawValue: "Green") {
    //
    print(_fromraw)
}
```

Testování hodnoty enum

- if case [let] VZOR = hodnota {}
- switch EXPR case .alt:

```
// je znamy kontext
let p: Colors = .red
// lze navazat "p" na vzor case .red
if case .red = p {
    //
}
// if p == .red {}
extension Colors {
    //
    var isRed: Bool { if case .red = self { return true}; return
false }
}
```

```swift
struct Project { let name: String }
// Parametrizovatelny enum (skoro jako union v C)
enum TODO {
    // parametr: volitelne lze pojmenovat
    case work(Project)
    case beer(count:Int, kind: String)
    case sleep
}
//
let w = TODO.work(Project(name: "..."))
let beers = TODO.beer(count: 3, kind: "Radegast")
// case let .work(_)
if case let .work(P) = w {
    //
    print("Working on \(P.name)")
}
//
extension TODO : CustomStringConvertible {
    //
    var description: String {
        //
        switch self {
        case .sleep:
            return "Sleep"
        case .beer(let X, _):
            return "Drink \(X) beers"
        case .work(let P):
            return "Working on \(P.name)"
        }
    }
}
```

```swift
struct Project { let name: String }
// Parametrizovatelny enum
enum TODO {
    // parametr: volitelne lze pojmenovat
    case work(Project)
    case beer(count:Int, kind: String)
    case sleep
    // rekurzivni hodnota
    indirect case haveABreak(minutes:Int, andThen: TODO)
}
//
extension TODO : CustomStringConvertible {
    //
    var description: String {
        //
        switch self {
        case .sleep:
            return "Sleep"
        case .beer(let X, _):
            return "Drink \(X) beers"
        case .work(let P):
            return "Working on \(P.name)"
        case let .haveABreak(minutes: M, andThen: Then):
            return "Short nap\(M) and later \(Then)"
        }
    }
}
//
print(TODO.haveABreak(minutes: 10, andThen: TODO.sleep))
```

Protokoly

- Protokol je *sada hlaviček funkcí,* tzn. nezavádí přímo uložené atributy (properties).
- Protokol *může dědit* z jiného protokolu.
- Protokol může sám *vytvářet nějaké chování* (extensions).
- Protokol může definovat *asociovaný typ*.
- Omezení na třídní protokoly.

Základní využití protokolu

- Komunikace mezi dvěma objekty (nemusí znát svoji strukturu).
  - protocol *myProt* {}
  - var p: *myProt* = …
  - proměnná *p* vyjadřuje "kdokoli implementující *myProt*".
- Delegátství — obousměrná komunikace.
- Data Source — jednosměrná komunikace.

```swift
protocol MyConsumerProtocol : class {
    //
    func workStarted(from:Producer);
    func workProgress(from:Producer, results:[Int]);
    func workTerminated(from:Producer);
}

//
class Producer {
    // paměťový model ”weak”
    weak var delegate: MyConsumerProtocol?

    //
    func startMyWork() {
        // synchronni zprava
        delegate?.workStarted(from: self)
    }

    //
    func workInProgress(results: [Int]) {
        // TODO: proc je tohle chybne? Rozbor…
        if let _del = delegate {
            // asynchronni zprava
            DispatchQueue.main.async {
                // pozor, muze dorazit "az po smrti" prijemce
                _del.workProgress(from: self, results: results)
            }
        }
    }
}
```

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))
```

Formáty hlaviček funkcí

- Statické/instanční funkce.
- Protokolem NELZE přikázat zavedení proměnné do implementujícího typu.
  - Lze přikázat setter/getter property, tj vynutit si "dojem existence uloženého atributu"
  - var atribut: Int { get set }

getter / setter v protokolu

```swift
//
protocol PROT {
    //
    var anItem: Int { get set }
}

//
struct ImplPROT : PROT {
    //
    var anItem: Int
}

//
struct ImplPROT2 : PROT {
    //
    private var __internal: Int = 10
    //
    var anItem: Int {
        //
        get { return __internal }
        set(newValue) { __internal = newValue }
    }
}
```

Protokol zavádí vlastní chování

```
//
protocol PROT {
    //
    var anItem: Int { get set }
}
// smim na urovni protokolu dodat chovani!
extension PROT {
    // abstraktni metoda pracujici nad
    // vlastnostmi danymi protokolem
    func printInternals() {
        // self existuje, ale immutable
       // mutating func…
        print(anItem)
    }
}
// instancovatelny implementator
struct ImplPROT : PROT { var anItem: Int }
// je typu PROT
let w: PROT = ImplPROT(anItem: 100)
//
w.printInternals()
```

DataSource protokoly v UIKit

- UITableViewDataSource
- UICollectionViewDataSource
- apod.

Protokoly z Objective-C

- Protokol v Obj-C zaváděl povinné a *volitelné* hlavičky funkcí.
  - NSObjectProtokol — responds(to: Selector!)
- Swift "optional" hlavičky nedovoluje, tudíž se při potřebě "optional" vrací k Obj-C.
- Obj-C protokol je pouze třídní!!
  - (musí ho implementovat class)

Demo: objc protokol

```
// Protokol v duchu Objective-C
@objc protocol MyDataSource {
    // implementator tohle implementuje volitelne
    @objc optional func numberOfSections() -> Int;
    // tohle implementovat musi
    func mustImplement();
}

// tridni protokol
class DS: MyDataSource {
    // z protokolu povinne
    func mustImplement() {
        //
    }
}

//
let ds = DS()
// test na "responds(to:)" a pripadna invokace
ds.numberOfSections?()
```

NSObject / NSObjectProtocol

- Vlastnosti NSObject:
  - NSKeyValueCoding,
  - NSKeyValueObserving,
  - responds(to:)
- Selector — datově vyjádřené volání funkce.

NSObject, metaprotokol

```
// NSObject - Foundation
class P : NSObject {
    // metoda a() je v ramci Obj-C metaprotokolu
    @objc func a() {}
}
//
let p = P()
// NSObject...
if p.responds(to: #selector(P.a)) {
    //
    print("Rozumi zprave a()")
}
// hodnota typu identifikace metody
let sel = Selector("a")
//
if p.responds(to: sel) {
    //
    print("Stale rozumi")
}
```

Šablonové protokoly

- Protokol nemůže být *šablonový*, ale …
- Protokol smí definovat *associated-type*.
  - associatedtype X
  - Interpretace: hlavičky funkcí předpokládají typ X. Význam typu X naplní ten, kdo bude protokol implementovat.

associated type

```
//
protocol Container {
    // napojeny/pracovni/asociovany/navazany typ
    associatedtype Item

    //
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }
}

//
struct MyCollection: Container {
    //
    typealias Item = String
    //
    mutating func append(_ item: Item) { }
    var count: Int { return 1 }
    subscript(i: Int) -> Item { return "Ahoj...." }
}
```

associated type

```swift
// protokol zavadi asociovany typ
protocol PROT {
    //
    associatedtype Element
}

// sablonova implementace strukturou
struct PP<Element> : PROT {
    //
    let data: [Element]
}

// typovou inferenci odvozeno/specifikovano
let x = PP(data: [1,2,3,4])
// explicitne specifikovano
let x2 = PP<String>(data: ["a","b"])
```

Extensions

- Pochází z Objective-C.
- Dovoluje rozšířit rozhraní enum / class / struct o další metody.
  - Přidání metod. Moduly, tematické balíky metod.
  - Přidání implementace protokolu.
  - Nedovoluje rozšířit uložené atributy. Nelze "zvětšit velikost objektu".

Extensions, motivační demo

```
// Formatovac datumu + jeho konfigurace
let __mujDF = DateFormatter()
//
__mujDF.dateStyle = .medium
__mujDF.timeStyle = .short

// globalni funkce pro prevod Date na String
func asString(date dt: Date) -> String { return __mujDF.string(from: dt) }

// rozsireni knihovni tridy o vypoctenou property
extension Date {
    // instancni metoda tridy Date
    var asString : String { return __mujDF.string(from: self) }
}

//
let dt = Date()

//
print(asString(date: dt))
print(dt.asString)
```

demo

```
// Definice tridy a ulozenych atributu
class MyClass {
    //
    var aNumber: Int = 3
}

//
extension MyClass : CustomStringConvertible{
    //
    public var description: String { "MyClass: \(aNumber)" }
}
```

Protokolové programování

```swift
// Muzeme protokolem dodat i implementovane funkce
protocol ForALL {
    //
    var name: String { get }
}
// Deklaruji dva typy trid: producenty/konzumenty
protocol Producent {}
protocol Consument : ForALL {} // odvozuje/dedi od ForALL
// extension protokolu s podminkou na cilovy typ
// cilovy typ: zde "Self"
extension ForALL where Self: Producent, Self:ForALL {
    //
    func work() { print("Prace producenta \(name)") }
}
// cilovy typ: zde "Self"
extension ForALL where Self: Consument {
    //
    func work() { print("Prace konzumenta \(name)") }
}
//
struct ImplProducent : Producent, ForALL {
    //
    var name: String { return "Struct-PROD" }
}
//
struct ImplConsument : Consument {
    //
    var name: String { return "Struct-CONS" }
}
//
ImplProducent().work()
ImplConsument().work()
```

Šablony a koncepty

- Smalltalk: třída je objekt. Jazyk Self.
- Šablonová funkce globální a instanční.
  - self,
  - Self — nahrazuje třídu, která implementuje protokol.
  - Self.self
  - T.Type — datový typ objektu.
  - Trida.self —

…

- Int.self == Int
- type(of: 3) == Int

Jednoduchá šablonová funkce

```
//
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let temporaryA = a
    a = b
    b = temporaryA
}

//
var aNumber = 1
var bNumber = 2
//
swapTwoValues(&aNumber, &bNumber)
```

...

```swift
// typ musi mit init90
protocol Constructible {
    init()
}
// T musi implementovat Constructible
// type: T -- chci hodnotu typu T
// type: T.Type -- chci datovy typ T
func construct<T:Constructible>(type: T.Type) -> T {
    //
    return T()
}
// demo
struct S1:Constructible { init() {} }
//
let s = construct(type: S1.self)
```

...

```swift
//
struct Record : Codable {
    //
    let num: Int
}

// zakoduj hodnotu
let v = Record(num:2)
let e = JSONEncoder().encode(v)
// dekoduj hodnotu
let j = JSONDecoder().decode(Record.self, from: e)
// func decode<T>(_ type: T.Type, from data: Data) throws -> T where T : Decodable
```

```swift
//
class TRIDA<TTT> {
    //
    func neco<T>(value: T) where T : CustomStringConvertible{
        //
        print("Hodnota \(value) je typu \(T.self)")
    }

    //
    func sameType<T>(type: T.Type) -> Bool {
        //
        return T.self == TTT.self
    }
}
//
let p = TRIDA<Int>()
//
p.neco(value: "Ahoj")
p.neco(value: 1)
p.neco(value: true)
p.sameType(type: String.self)
```

Šablonové struktury — Stack#1

```
struct Stack1<Element> {

    var items = [Element]()

    mutating func push(_ item: Element) {
        items.append(item)
    }

    mutating func pop() -> Element {
        return items.removeLast()
    }
}
```

Šablonové struktury — Stack#2

```
//
protocol StackProtocol {
    //
    associatedtype Element
    //
    mutating func push(_ item: Element);
    mutating func pop() -> Element;
}

//
struct Stack2<Element> : StackProtocol {
    //
    var items = [Element]()
    //
    mutating func push(_ item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
}

//
var stack = Stack2<String>()
//
stack.push("ahoj")
```

```
//
extension StackProtocol {
    // do protokolu si pridam:
    // var empty: Bool { get }
    mutating func emptyTheStack() {
        //
        while empty == false {
            //
            pop()
        }
    }
}
//
extension Stack1 {
    //
    mutating func emptyTheStack() {
        //
        while items.isEmpty == false {
            //
            pop()
        }
    }
}
```

Protocol, Extension, Šablony

- Struktury nemají dědičnost.
- Pokud chceme tvořit společné chování pro sadu struktur (jakoby nadtřídu), pak je tu extension nad protokolem.
  - Prokotol+Extension — abstraktní nadtřída.
  - Struct1:Protokol, Struct2:… — sdílí abstraktní chování dané Protokolem.
  - Použito v kontejnerech, struct XY: View (SwiftUI), …

Koncepty

- Některé typy a protokoly Swift Standard Library zavádí koncepty (C++20).
- Any — void.
- AnyObject — void, ale musí být instancí třídy.
- var p = [Any]()
  - Do kolekce lze vložit cokoli. Vede k otázce, jak je ”cokoli” implementováno (referencovatelnost).

Koncepty

- *Comparable* — <, <=, >, >=
- *Equatable* — static operator == (l, r)
- *Hashable* (Hasher) — hashValue: Int, hash(on:)
- CustomStringConvertible

Protokoly pro Extensions

- Chtěl bych do *Array<Element>* dodat funkci pro řazení pole.
  - Jaké to klade požadavky na vlastnosti typu *Element*?
  - Musí být schopnost porovnat prvky.

```
// Rozsiruji template-struct Array s asociovanym typem
// Element, kde ovsem Element musi implementovat Comparable
extension Array where Element:Comparable {
    //
    func sorted() -> [Element] {
        // self je [Element]
        // nejaky bubble-sort
    }
}
```

Hashable

```
class Person : Hashable, Equatable {
    //
    let name: String
    let age: Int
    // operatory: =, ==, ===
    static func ==(lhs:Person, rhs:Person) -> Bool {
      // alternativne: lhs === rhs
      return lhs.hashValue == rhs.hashValue
      }
    //
    func hash(into hasher: inout Hasher) {
        //
        print("calling me")
        hasher.combine(name)
        hasher.combine(age)
    }

    //
    init(name: String, age: Int) {
        //
        self.name = name; self.age = age;
    }
}
```

Hashable

```swift
//
class Tridka1 : Hashable, Equatable {
    // deprecated
    var hashValue: Int { return ObjectIdentifier(self).hashValue }
    static func == (lhs: Tridka1, rhs: Tridka1) -> Bool { return lhs === rhs }
}

//
class Tridka2 : Hashable, Equatable {
    //
    func hash(into: inout Hasher) { into.combine(ObjectIdentifier(self))}
    static func == (lhs: Tridka2, rhs: Tridka2) -> Bool { return lhs === rhs }
}
```

Dodatky

- Tuples.
- Instanciace tříd.
- Přetypování.
- Výjimky (exception, try, throw, catch) — necháme to na přednášku o kódování dat.

Tuples, demo, *typealias*

```
//
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)")
```

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)
```

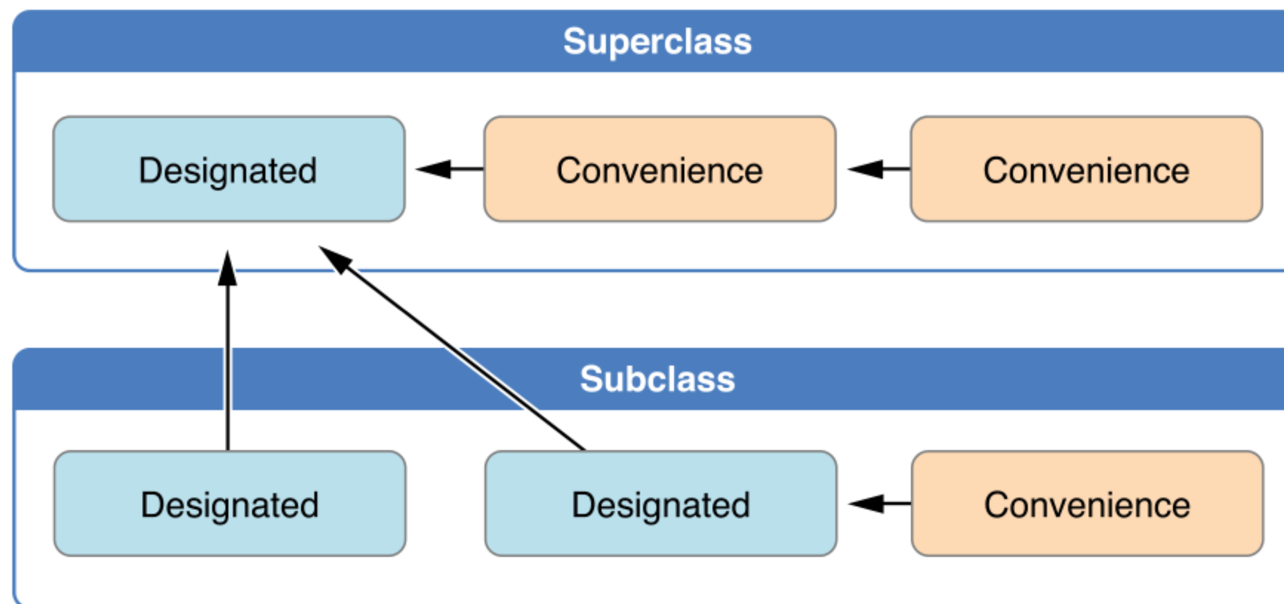
Instanciace třídy, init?

```
//
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
```

Inicializace objektu

- Designated initializer — init(…).
  - DI musí volat DI nadtřídy.
- Convenience initialize — convenience init(…).
  - CI musí volat DI svojí třídy.
  - Dvoufázový process — "řádná" inicializace, "doplňková".



```swift
//
class TridaA {
    //
    var val: Int
    //
    init(val: Int) {
        //
        self.val = val
    }
}
```

```swift
//
class TridaB : TridaA {
    //
    var s = "Ahoj"
    //
    init(val: Int, andS: String) {
        //
        self.s = andS
        super.init(val: val)
        self.bootuj()
    }
    //
    func bootuj() { print("Bootuju") }
    //
    convenience init(doprovodne val: Int,
                        andS: String) {
        //
        self.init(val: val, andS: andS)
        // ted je objekt inicializovany
        // nad "self" muzu provadet dalsi akce
        self.bootuj()
    }
}
//
let p = TridaB(doprovodne: 3, andS: "dkdkdkdk")
```

Model convenience init

```
//
class TridaC : TridaA {
    //
    func bootuj() { print("Bootuju") }

    //
    static func CREATE(val: Int) -> TridaC {
        //
        let s = TridaC(val: val)
        //
        s.bootuj()
        //
        return s
    }
}
//
let t = TridaC.CREATE(val: 3)
```

Přetypování ”as”

- Objektový metaprotokol. Objective-C.
  - Zjistit třídu/typ objektu. Rozumí objekt zprávě?
- Co lze přetypovat?
  - Test typu ”*is*”.
  - Statické přetypování (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, další plány

- MVC:
  - Základy Views, základní ViewControllers.
  - Architektura aplikace. Storyboard.
- Kódování dat. Documents.
- CoreData/CloudKit.
- Vlákna. GCD. Operations.
- SwiftUI. Combine a reaktivní programování.