Swift III.
More advanced… :)

IZA, Martin Hrubý, FIT VUT, 2019
http://perchta.fit.vutbr.cz/vyuka-iza

Témata

- Dodatky k přednášce Swift-I (mh).
- Dodatky k přednášce Swift-II (fk).
- Closures. Escaping…
- Přehled knihovních kolekcí.
- Extensions, Protocols, Generics.
- [1] Skupina objc.io: Advanced Swift

Třída, struktura. Retain-count

```
//
struct STR {
    //
    var a: Int = 2
}

//
class TRIDA {
    //
    var a: Int = 3
}

//
var t : TRIDA? = TRIDA()
var s : STR? = STR()
// CFGetRetainCount(o) - vypise refCount (v Objective-C/Swift se rika
"retain count")
// napise 2: jednou za "t" + jednou za argument volani CFGetRetainCount
print(CFGetRetainCount(t))
```

Test referencovatelné hodnoty

```
// podminenene se vytvori novy "let _t : TRIDA = t", tj.
// 1) _t je konstanta,
// 2) _t dostane obsah prirazenim z "t", tj. navysuje refCount
if let _t = t {
    // napise "3": 1x "t", 1x "_t", 1x funkce CFGetRetainCount
    print(CFGetRetainCount(_t))

    // zapisuju do objektu
    _t.a = 1

    // _t zanika
}

// podminenene se vytvori novy "var _t : TRIDA = t", tj.
// 1) _t je promenna,
// 2) _t dostane obsah prirazenim z "t", tj. navysuje refCount
if var _t = t {
    // napise "3"
    print(CFGetRetainCount(_t))

    // zapisuju do objektu
    _t.a = 1

    // muzu modifikovat _t, ale nedava to zadny smysl
    _t = TRIDA()

    // _t zanika, "t" snizuje refCount
}
```

Test kopírované hodnoty

```
// vznika novy let _s: STR, podminene
// !!! dochazi ke kopii hodnoty
if let _s = s {
    // nelze priradit, nebot _s je konstanta
    //
    // _s.a = 1
}


// vznika novy var _s: STR, podminene
// !!! dochazi ke kopii hodnoty
if var _s = s {
    // lze zapsat, ale zapisuju do struktury,
    // ktera za chvili zanikne
    _s.a = 1
}

// co tedy chceme provest?
// pokud chceme podminene zapsat do atributu, pak nejspis
t?.a = 10
s?.a = 20
```

Property s typem Type!

```
//
class TRIDA_B {
    // ukaze se, ze Int! je ekvivalent Int?, kde se zanedbava testovani
    var nei: Int!
}

//
let p = TRIDA_B()

// slitne
// print(p.nei!)

// lze !!! testovat na nil
if p.nei == nil {
    //
    p.nei = 1
}

// dokonce tohle...
// tj. "nei" je typu Int?, akorat se nevyzaduje explicitni unwrap a dela se implicitne
p.nei = nil

// optional binding, kopiruju hodnotu
if let _nei = p.nei {
    //
    print("Neco tam je \(_nei) ")
} else {
    //
    print("Nei prazdno")
}
```

Dodatky Swift-II

- Tuples — nepojmenované struktury bez přidaných metod.
- Doporučení:
  - Raději tvořit struktury namísto tuples.
  - Důvody pro tuples — datová struktura **pro bezprostřední předání hodnoty** (návrat z funkce) bez dalšího zpracování.
  - Pojmenovávat složky n-tice.
- N-tice jako klíč do Dictionary.

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 hodnototy.

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

Protokoly

- Mírně odlišné pojetí od Objective-C.
- Protokol X je sada hlaviček funkcí. Deklarujeme:
  - Třída implementuje protokol X. Také: struct/enum.
  - Instance třídy X rozumí zprávám protokolu X.
  - Proměnná typu X obsahuje/vyžaduje hodnotu typu implementujícího protokol X.
- Použití: delegátství, typové kontroly, extensions.
  - Nahrazujeme také více-násobnou dědičnost.

Protokol, delegátství

- Komunikace Producent-Konzument. Velmi časté v knihovnách Foundation/Cocoa.
  - alternativně callbacky, closures.
- Je třeba mít jasno ve vlastnictví objektů (kdo koho).
- Protokol je typicky třídního typu (AnyObject).
  - weak var delegate, weak vždy ve spojení s ref hodnotou.
- Optional metody protokolu (Obj-C).

```
//
class Sluzba {
    // weak vzdy! Branime cyklicke referenci.
    weak var delegate: ZpravyPosluchaci?
    //
    func work() {
        // neco delam
        delegate?.praceJeHotova(odSluzby: self)
    }
}
// AnyObject – implementator musi byt class
protocol ZpravyPosluchaci : AnyObject {
    //
    func praceJeHotova(odSluzby: Sluzba);
}
// implementuje protokol. Prekladac vyzaduje splneni impl.
class Posluchac : ZpravyPosluchaci {
    //
    var sluzba: Sluzba?
    //
    func zahajeniPrace() {
        // snad uz sluzba nebezi...
        guard sluzba == nil else { return; }

        //
        sluzba = Sluzba(); sluzba?.delegate = self
        //
        sluzba?.work()
    }
    //
    func praceJeHotova(odSluzby: Sluzba) {
        // prevezmi vysledky
        sluzba = nil
    }
}
```

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
    }
}
```

```
//
struct MojeSTR : Equatable {
    //
    let val: Int
    //
    static func == (lhs: MojeSTR, rhs: MojeSTR) -> Bool {
        //
        return lhs.val == rhs.val
    }
}
//
var p : [MojeSTR] = [MojeSTR(val: 1), MojeSTR(val: 2)]
// porovnavam hodnoty, coz musim explicitne implementovat
if p.contains(where: { $0 == MojeSTR(val: 1)}) {
    //
    print("Je tam")
}
//
extension Array where Element:Equatable {
    //
    func found(_ val: Element) -> Bool {
        //
        for i in self {
            //
            if i == val { return true }
        }
        //
        return false
    }
}
```

Protokol jako zdroj chování

```
// Muzeme protokolem dodat i implementovane funkce
protocol ForALL {
    //
    var name: String { get }
}
// ... ovsem az v extension...(proc???)
extension ForALL {
    // muze si sahnout na self, je to INSTANCNI metoda
    // budoucich struct/class/enum
    func describeMe() {
        //
        print("Jsem hodnota \(self.name)")
    }
}
//
struct STRF : ForALL {
    //
    var name: String { return "Struct-STRF" }
}
//
STRF().describeMe()
```

Protokolové programování

```
// 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()
```

Obj-C protokoly

```
// Protokol typu Objective-C, ktery:
// 1) musi byt implementovan jedine tridou
// 2) musiByt() je povinna
// 3) jeVolitelna() je opt, pouze v Obj-C !
@objc protocol OldFashion : AnyObject {
    // swiftove pojata metoda
    func musiByt();
    // obj-ceckovsky pojata metoda
    @objc optional func jeVolitelna();
}
//
class Implementator: OldFashion {
    //
    func musiByt() {}
}
// objekt...
let i = Implementator()
// upcast na "nekoho, kdo impl OldFashion"
(i as OldFashion).musiByt()
// podminene volani...
(i as OldFashion).jeVolitelna?()
```

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")
}
```

Protokol jako datový typ

- *AnyObject* — protokol implicitně implementovaný všemi třídami.
- *Any* — protokol … všemi hodnotami.

```
//
protocol Entity {}
protocol Producent: Entity {}
protocol Consument: Entity {}

// Implementatori
class ProdImpl : Producent {}
struct ConsImpl : Consument {}

// pole hodnot implementujicich Entity
let p : [Entity] = [ProdImpl(), ConsImpl()]
let all: [AnyObject] = [ProdImpl()]
let allall: [Any] = [ProdImpl(), ConsImpl(), "ahoj", 3.14]
```

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 }
```

Closures

- Closure je lambda výraz, který je zabalený do kontextu, tj. referencuje svůj 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

- Knihovny Foundation a Cocoa Touch jsou plné callbacků.
  - Callbacky doplňují klasický režim delegát+protokol.
- Grand Central Dispatch.
  - Programování jako ”Brownův pohyb”.
  - Zapojení různých výpočetních front (dispatch queues).

Extensions

- Pochází už z doby Objective-C.
  - Lze rozšířit *@interface* nějaké třídy. Nelze doplnit instanční proměnné (uložené properties).
  - … nelze zvětšovat objekt (knihovna, aplikace).
  - … Obj-C se příliš nestaralo o private/protected atributy.
  - Moduly programu — extensions třídy. Tematické celky programu. Organizace kódu.
- Lze rozšířit knihovní třídu!
  - Není tedy nutno odvozovat třídy.

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)
```

```swift
//
protocol Protokol1 { var ahoj : String { get } }
protocol Protokol2 { var hodnota: Int { get set }}
//
class TRIDA {
    //
    private var __hodnota: Int = 0
}
// implementace protokolu
extension TRIDA : Protokol1 {
    //
    var ahoj : String { return "Ahoj string" }
}
// implementace protokolu
extension TRIDA : Protokol2 {
    //
    var hodnota : Int {
        //
        get { return __hodnota }
        set { __hodnota = newValue }
    }
}
//
let obj = TRIDA()
let umi2 : Protokol2 = obj
//
print(umi2.hodnota)
```

Extensions nad šablonami

- Jak se jmenuje parametrický typ šablonové deklarace.
- Podmínky na typ (constraint). *where:*
  - *where Element == NejakyTyp*
  - *where Element: NejakyProtokol*

Přehled datových kontejnerů

- Typy kontejnerů (Kolekce):
  - Array<Element> (std::vector),
  - Set<Element> (std::unordered_set),
  - Map/Dictionary<Key, Element> (std::unordered_map).
- Kolekce jsou parametrické struktury!
  - Založeny na sadě protokolů.
- Mutability (let, var).
- Srovnání s Objective-C, NSNumber.

Sekvence a kolekce

- *IteratorProtocol* — Implementátor poskytuje obecnou řadu prvků (obecně nekonečnou).
- *Sequence* — rozsáhlý protokol:
  - Vyčíslitelná posloupnost prvků (!!! ne nutně uložená).
  - Sekvence poskytuje přístup ke svým prvkům přes iterátor.
  - for-in — základní prvek, iterátor. Dále: filter, map, …
- *Collection* — protokol nad *Sequence*.
  - K sekvenčnímu přístupu **přidává index-sekvenční**.

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) }
```

AnyIterator / AnySequence

- Iterátor — dává řadu prvků.
- Sekvence — dává iterátor, implementuje operace nad posloupností.
- Důsledek:
  - Můžeme zbudovat sekvenci nad iterátorem.

Any…

```
// funkce vraci iterator ve forme closure !!!
func generuj(azpo: Int) -> AnyIterator<Int> {
    // tato promenna zustava vazana v closure
    var nowValue = 0
    // vracim iterator zbudovany nad nasledujicim closure
    return AnyIterator {
        //
        guard nowValue < azpo else {
            //
            return nil
        }
        //
        defer { nowValue += 1 }
        //
        return nowValue
    }
}
// konstrukce sekvence nad iteratorem
let sek = AnySequence(generuj(azpo: 100))
// demo...
sek.forEach { print($0) }
```

Poznámky k sekvencím

- Sekvence je abstrakce nad něčím, co má pořadí.
  - a lze vyhodnocovat iterátorem.
- Aplikace:
  - Datový stream: síť, soubor, …
  - Tok událostí.

```
// je sekvence konstruovana nad iteratorem
let loadFile = AnySequence {
    // ... ktery odnekud cte radky...
    return AnyIterator {
        //
        readLine();
    }
}
```

- Důsledek: průchod sekvencí **nemusí** být opakovatelný (stabilní).
- *Lazy* vyhodnocování, *Reaktivní programování*.

Kolekce

- Stabilní sekvence, nedestruktivní průchod.
  - Paměťový prvek v aplikaci (něco se tam ukládá).
- Operace nad kolekcí:
  - Implementuje rozhraní *Sequence*.
  - Subscript — index-sekvenční přístup.
  - Počet prvků.

Array

- Nejčastější kolekce v programech.
- Stylově zcela podle STL, std::vector.
  - Není jako NSArray, NSMutableArray.

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)")
}
```

Indexování pole, slices

- Apple důrazně radí ne-indexovat pole subscriptem. Jistě jsou výjimky…
  - Slice: pohled na pole (zavádí vlastní indexování). Copy-on-write.
- Korektní jsou: for-in, forEach, filter, map, dropFirst, dropLast, …

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)")
}
```

Map/Dictionary

- Neseřazená kolekce prvků typu klíč-hodnota.
  - Pořadí iterování přes dict. není definováno.
  - Syntaxe typu: [Key:Value].
- Přístup — konstantní časová složitost.
  - Subscript *get* je ”Value?”
- Klíč musí být hodnota implementující Hashable.
  - Int, Double, String …, již jsou hashable.

Hashable, Equatable

- Navrhnout korektní hashovací funkci může být problém.
  - Lze to transformovat na String, Int, …?
- [Key:Value]
  - Preferujme *Key*, který má kopírovací sémantiku.
  - *Key* jako objekt (s proměnlivým obsahem). Co je hash?

[ref:value], Hashable

```
// Aby objekty Customer mohly byt klice Dictionary, pak
// musi trida implementovat ...
class Customer : Hashable, Equatable {
    //
    let id: Int
    let name: String

    // vyzadovano Hashable
    var hashValue: Int { return id; }

    // vyzadovano Equatable
    static func == (l: Customer, r: Customer) -> Bool { return l.id == r.id }

    //
    init(_ id: Int, _ name: String) {
        //
        self.id = id; self.name = name
    }
}

//
var addrs: [Customer:String] = [:]
let c1 = Customer(1, "Petr")
let c2 = Customer(2, "Jan")
//
addrs[c1] = "Bozetechova"
addrs[c2] = "Lidicka"
```

Set<Element>

- Rozdíl Array/Set. std::unordered_set<Element>
- Set je Dictionary bez Value, pouze Key.
  - *Element* tudíž opět *Hashable*.
- Předpokládáme přístup O(1).
- Kolekce **není uspořádáná**.

Set, demo

```
//
extension Array where Element:Hashable {
    //
    func asSet() -> Set<Element> {
        //
        var _out = Set<Element>()

        //
        for i in self { _out.insert(i) }

        //
        return _out
    }
}

//
let nums = [1,2,1,3,1,2,3,1]
let uniqueNums = nums.asSet()

//
print(uniqueNums)
```

Generické programování

- Přetěžování operátorů.
- Šablony se aplikují na struct/class/enum a funkce.
- Lze zadávat omezující podmínky.
- Swift negeneruje klony šablonového kódu (na rozdíl od C++).
  - vtables — šablony se řeší dynamicky.
  - …

Přetěžování operátorů

```
//
struct MValue {
    //
    let v: Int
}

//
func + (l: MValue, r:MValue) -> MValue {
    //
    return MValue(v: l.v + r.v)
}

//
print(MValue(v: 1) + MValue(v: 2))

// T ma by soucasne Numeric a soucasne Comparable
func ~> <T:Numeric & Comparable> (expression: @autoclosure () -> T?, def: T) -> T
{
    //
    guard let _val = expression(), _val >= 0 else { return def }

    //
    return _val
}

//
print( (3 - 4) ~> 10 )
```

Šablonová globální funkce

```
//
protocol ObsahujeNeco {
    //
    var length: Int { get }
}
//
class Type1 : ObsahujeNeco { var length: Int { return 0 } }
class Type2 : ObsahujeNeco { var length: Int { return 10 } }

//
func jinak<T>(val: T) -> Bool where T:ObsahujeNeco { return val.length > 0 }

//
extension ObsahujeNeco {
    //
    func jinak() -> Bool {
        //
        return length > 0
    }
}
```

Závěr

- Běžné programování běžných iOS aplikací vyžaduje výrazně jednodušší jazyk, než jsme si předvedli. Konzervativní programování.
- Dále budeme pokračovat přednáškami o Cocoa Touch, tj. iOS aplikacích.
- MVC — Model—View—Controller.