Závěrečné opakování

Programování zařízení Apple, IZA

Martin Hrubý, FIT VUT, 2020, #covid19

Přehled

- Programovací jazyky — Objective-C, Swift.
- Programování iOS aplikací ve Storyboard/MVC.
- Řízení vláken v aplikaci — GCD.
- Databázová podpora aplikací — CoreData, CloudKit, UIDocument, kódování dat.
- SwiftUI v základech. Combine.
- (aplikace pro macOS/tvOS/watchOS)

Objective-C

- Prazáklad veškerého programování v Apple
- Založení bazálních principů:
  - koncept ”properties” (instanční proměnná s nějakým chováním, zapouzdření inst. proměnné), KVC/KVO,
  - koncept protokolů a delegátů,
  - koncept extensions, … ve Swiftu dále nad šablonami:
  - extension Array where Element == String { … }

Interface třídy

```
@interface TRIDA : NSObject
{
    int number;
    NSString *text;
}

-(id) init;
-(id) initWithNumber: (int) c;

+(TRIDA *) create;

-(int) number;
-(void) setNumber: (int) c;

@end
```

```
@interface NSObject(myExtensions)
-(void) sayHello;
@end
```

```
@protocol MyProtocol
-(BOOL) canFly;
@end
```

Implementace rozhraní

```
@implementation TRIDA

-(id) init {
    self = [super init];
    return self;
}

-(id) initWithNumber: (int) c {
    self = [super init];
    if (self != NULL) {
        number = c;
    }
    return self;
}

+(TRIDA *) create
{
    return [[TRIDA alloc] init];
}

-(int) number { return number; }
-(void) setNumber: (int) c { number = c; }

@end
```

Property

- Property je public instanční proměnná, která automatizuje deklaraci setteru/getteru.
- @synthesize, @dynamic
- tečková notace
- opět volání metody!

```
@interface TRIDAB
//
@property int numberb;
@property NSString *textb;
@end

@implementation TRIDAB
@synthesize numberb, textb;
@end

//
TRIDAB *b = ...;
//
b.numberb = 3;
[b setNumberb: 3];
int bb = b.numberb;
```

```
@implementation TRIDAB
/* @synthesize numberb = _numberb; */

-(int) numberb {
     // willAccess, didAccess…
    return _numberb;
}

-(void) setNumberb:(int)v {
    [self willChangeValueForKey: @"numberb"];
    _numberb = v;
    [self didChangeValueForKey: @"numberb"];
}
@end
```

Key Value Coding (KVC)

- Základní protokol implementovaný v NSObject.
- Zavádí abstrakci nad properties.
- Každý objekt je slovník (dictionary, map) typu klíč/hodnota:
  - - (id)valueForKey:(NSString *)key;
  - - (void)setValue:(id)value forKey:(NSString *)key;
  - může implementovat libovolné klíče.

Swift

- Poslední roky probíhá transfer funkcionality z Objective-C/Foundation do standardních knihoven Swiftu.
  - … a změny názvosloví NS-, UI-,
- Swift některé zvyklosti Objective-C nepřebírá.
  - některé posiluje: protokoly, extension
  - některé přidává: generické programování
- Swift stojí na silné typové kontrole!

Třídní metaprotokol

- NSObject není povinná nadtřída. Důsledky:
  - (stále je však k dispozici)
  - responds(to:) — není implicitní,
  - KVC/KVO — není implicitní. Properties tříd nejsou implicitně KVC/KVO.
  - ARC (Autom. Ref. Counting) je implicitní.
- Prefix @objc — @IBAction, KVO, #selector
  - CoreData — NSManagedObject, @NSManaged, KVC

Properties ve Swiftu

- všechno jsou properties (public, private).
- uložené/vypočtené.
  - uložené — tvoří paměťovou složku,
  - vypočtené — definují get {}, set {}
- var/let.
- Zpracování událostí — willSet/didSet.
  - tady lze navázat vlastním KVO. Dodejme, že modernizované KVO se vtělilo do knihovny Combine.

class / struct

- Hodnota — vyjádření informace v programu.
  - je předatelná (volání funkce), uložitelná (var / let / kolekce).
- class / instance — hodnota=reference, refCount
- struct — hodnota=obsah struktury, copy sémant
- struct — nemůže dědit až na protokoly.
- struct — nelze předat ”self”, nelze získat pointer na hodnotu struct.

mutable / immutable

- let A = hodnota, nelze dále A := něco jiného
  - pokud je ”hodnota” typu struct, pak je zcela konstantní.
  - pokud je ”hodnota” reference, pak je ”self” hodnoty mutable.
- var A = hodnota, lze dále A := něco jiného

let a *var* deklarace

```
class TRIDA {
    //
    var muta = 2
    let konsta = 10
}

//
let p = TRIDA()
let k = 1
var v = 2

// lze
v = 10
p.muta = 3

// nelze
k = 10
p = nil
p = TRIDA()
```

```
struct STRUCTA {
    //
    var muta = 2
    let konsta = 10
}

//
let A = STRUCTA()
var B = STRUCTA()

// nelze
A.muta = 3

// lze
B.muta = 3
```

```swift
// self je mutable
class TRIDA {
    //
    var a = 3
    let b = "Ahoj"

    //
    func zmense() {
        //
        self.a = 1234
    }
}
```

```swift
// self je immutable
struct STR {
    //
    var a = 3
    let b = "Ahoj"

    //
    func zmense() {
        // error
        self.a = 1234
    }

    //
    mutating func zmense2() {
        //
        self.a = 5678
    }
}

var a = STR()
let b = STR()
//
a.zmense2() // lze
b.zmense2() // nelze, error
```

Kolekce ve standardu

- Všechny kolekce jsou (šablonové) struktury.
- Důsledky:
  - let a = kolekce, ”a” je zcela immutable
  - var a = kolekce, ”a” je mutable.
- Předává se hodnotou, ovšem copy-on-write sémantikou.
  - Předám někomu data A (on si vytvoří kopii B), změní se A…
  - Je to jako poslat dopis. Adresát ho začne číst a já ho stále edituji

Pojmenovávání argumentů funkcí

- func funkce(label name: Type) -> RetType {}
- func funkce(name: Type) -> RetType {}
- func funkce(_ name: Type) -> RetType {}
- Při volání je nutno explicitně pojmenovat argumenty:
  - buď explicitním ”label”,
  - nebo implicitním jménem ”name”,
  - nebo explicitně NEpojmenovat argument.
  - **argumenty jsou z pohledu funkce konstanty (let)**.

```swift
// argument "a" je pro vnitrni i vnejsi pouziti
func funkceA(a:Int, s:String) -> String {
    //
    return "\(a) je \(s)"
}

// pri volani je NUTNO pojmenovat argumenty
let resultA = funkceA(a: 10, s: "Ahoj")

// labelProA je vnejsi jmeno argumentu,
// "a" je vnitrni jmeno argumentu
func funkceB(labelProA a: Int) -> Int {
    //
    return a + 2;
}

//
let resultB = funkceB(labelProA: 1)

//
func funkceC(_ innerName: Int) -> Int {
    //
    return innerName;
}

print(funkceC(10))
```

Konstrukce hodnoty

- Metody init(…).
- Všechny uložené properties musí mít definovanou počáteční hodnotu + init.
  - init a convenience init.
- Výjimky:
  - Type? — optional, unwrapping — if let, guard let,
  - Type! — optional, není třeba unwrapping

Optional se nám propaguje dál…

- *obj?.a* je zkratka pro *volání getteru, pokud je obj ne-NULL*. Lze libovolně řetězit (*obj?.a?.b?.c*)
- *obj* je typu TRIDA?
- *obj!.a* je Int
- *obj?.a* je Int?
- *obj!.b* je Int?
- *obj!.b!* je Int (a krom toho velké zlo…)

```
//
class TRIDA {
    //
    var a : Int = 10

    //
    var b : Int?
}
//
var obj : TRIDA? // = nil
```

Paměťové modely

- Jaký vztah vyjadřujeme datovým atributem?
  - class XY { var jmeno = hodnota }
  - strong reference — refCount++. ARC.
- Platí výhradně pro uložené properties:
  - weak reference — weak var A: Type?
  - { [weak self] in guard let _self = self … }
  - unowned var B: Type
  - unowned(unsafe) var C: Type

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 + 6}) // provolej urci typ pro $0
provolej { return $0 + 7 }; provolej { $0 + 8 }
```

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))
```

Enum

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

```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)"
        }
    }
}
```

Protokoly

- V základní podobě — API funkcí.
- Někdo protokol implementuje. Delegátství.
- Protokol může definovat vlastní abstraktní metody.
  - Zavádí abstraktní chování nad struct/enum/class.

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()
```

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

```
// 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"])
```

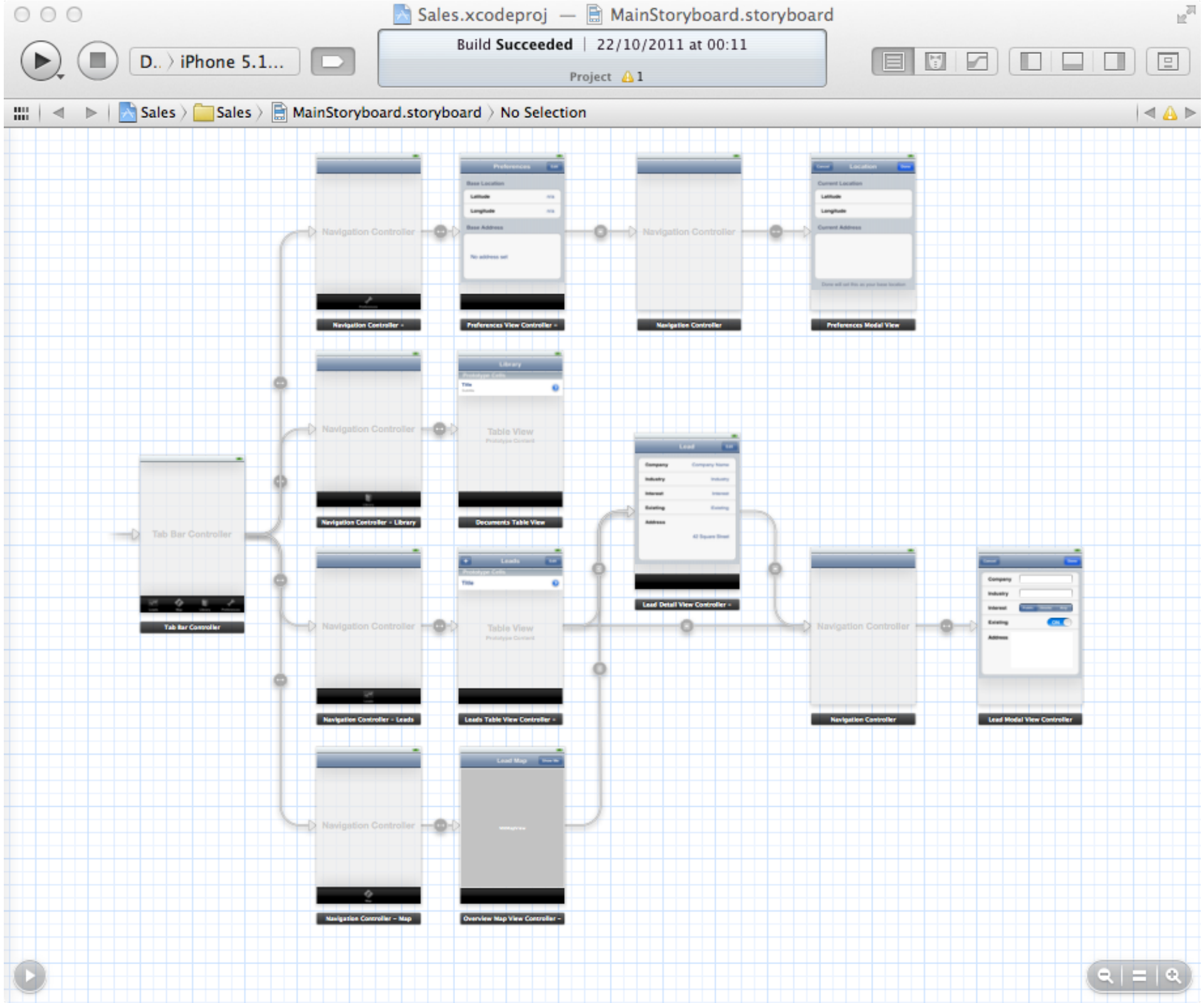
Protokoly pro Extensions

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

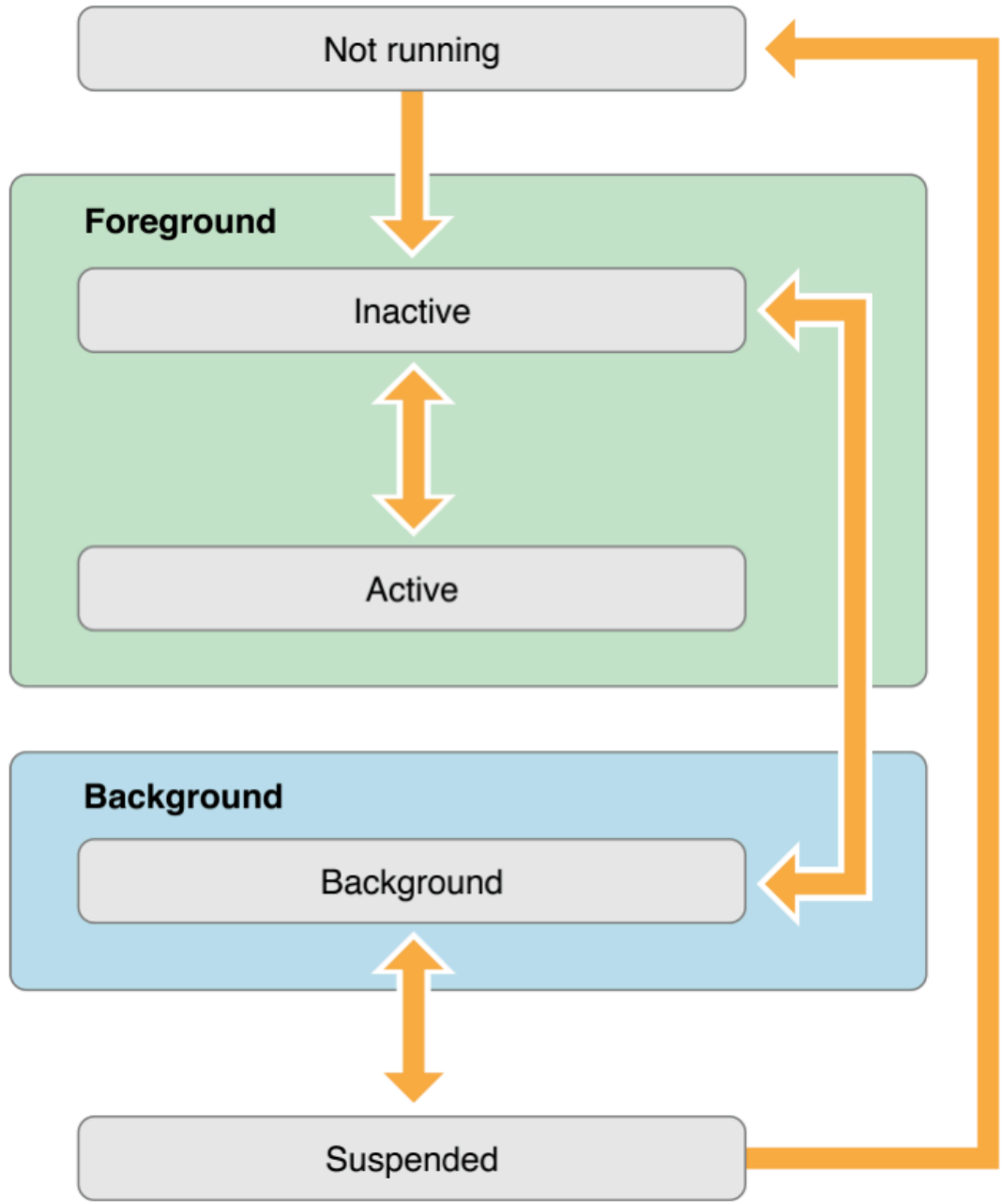
```
// Rozsiruji template-struct Array s asociovanym typem
// Element, kde ovsem Element musi implementovat Comparable
extension Array where Element:Comparable {
    //
    func sorted() -> [Element] {
        // self je [Element]
        // nejaky bubble-sort
    }
}
```

Základy iOS aplikací

- (UI)Application, AppDelegate
- UIWindow, rootViewController
  - StoryBoard, -> Initial VC
- MVC.
- Aplikace jsou přinejmenším VC a Views.
- Storyboard — InterfaceBuilder nad celou aplikací, segue.


Sales.xcodeproj — MainStoryboard.storyboard
Build Succeeded | 22/10/2011 at 00:11
Project 1
D.. iPhone 5.1...
Sales Sales MainStoryboard.storyboard No Selection
Navigation Controller
Preferences
Base Location
Latitude
Longitude
Base Address
No address set
Navigation Controller
Location
Current Location
Latitude
Longitude
Current Address
Navigation Controller
Preferences View Controller
Navigation Controller
Preferences Modal View
Navigation Controller
Library
Table View
Prototype Content
Navigation Controller - Library
Documents Table View
Tab Bar Controller
Tab Bar Controller
Lead
Company
Company Name
Industry
Interest
Existing
Address
42 Square Street
Lead Detail View Controller
Navigation Controller
Navigation Controller
Leads
Table View
Prototype Content
Navigation Controller - Leads
Leads Table View Controller
Navigation Controller
Company
Industry
Interest
Existing
Address
Lead Modal View Controller
Navigation Controller
Load Map
Navigation Controller - Map
Overview Map View Controller

Stavy aplikace



Views

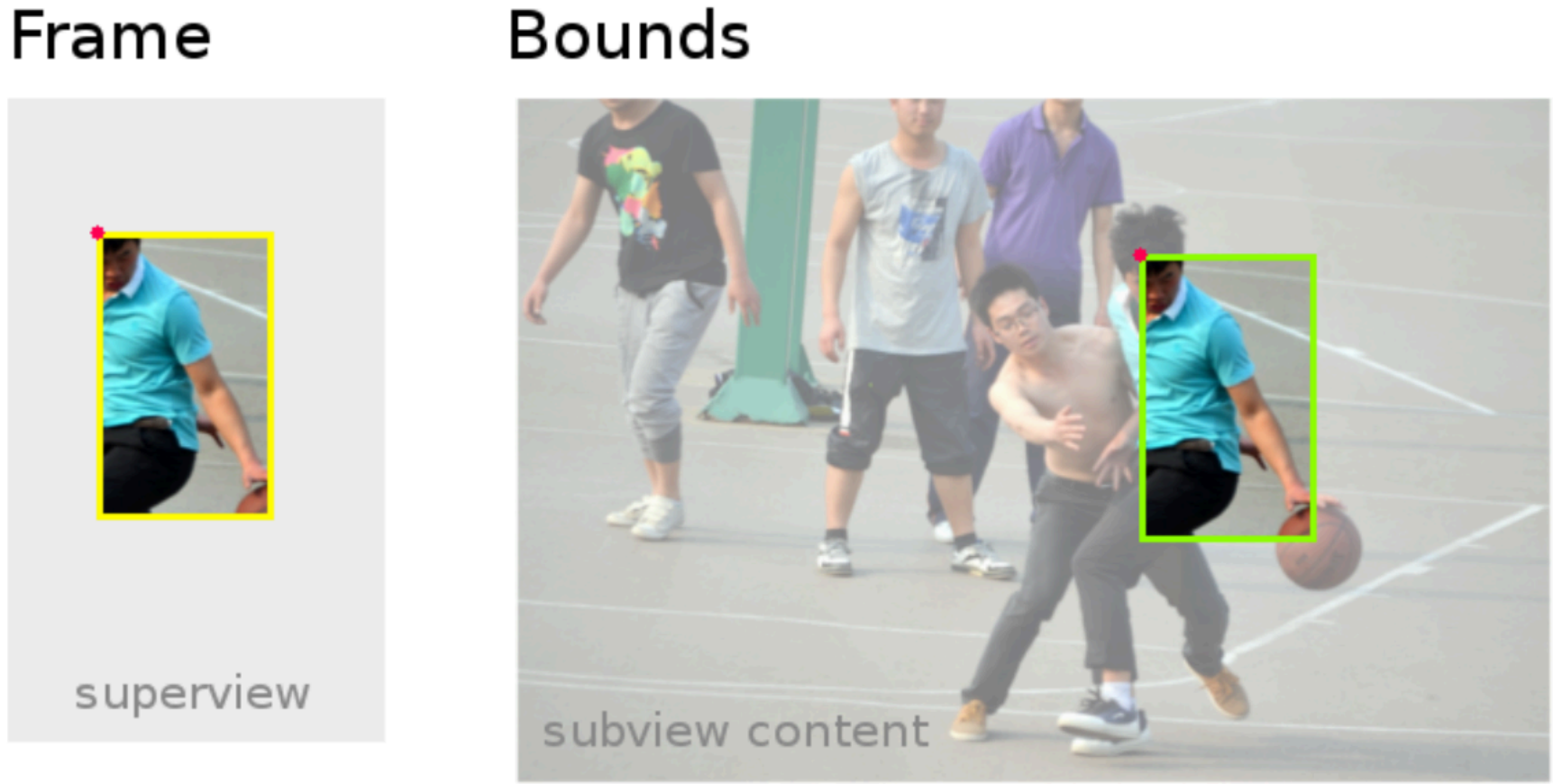
- UIView, UILabel, UISwitch, UITableViewCell,
- Superview, subviews — hierarchie.
- Smyslem je skládat UI z knihovních komponent (versus drawRect:).
- Provádět změny do UI smí pouze *hlavní vlákno*.

```
class ViewController: UIViewController {

    @IBOutlet var textik : UILabel?

    func inicializuj() {
        textik?.text = "Napis hello";
    }
```

Geometrie View

- *Frame* — definuje umístění *view* v jeho *superview*.
- *Bounds* — definuje velikost *view* a jeho lokální souřadný systém.
- *Frame* a *bounds* nemusí mít stejné velikosti.
  - *CGRect*(x: y: width: height:). Origin. Size.
- Souřadnice (0, 0) — levý horní roh, (macOS).
- Pozor: jednotkou souřadnice NENÍ 1 pixel.

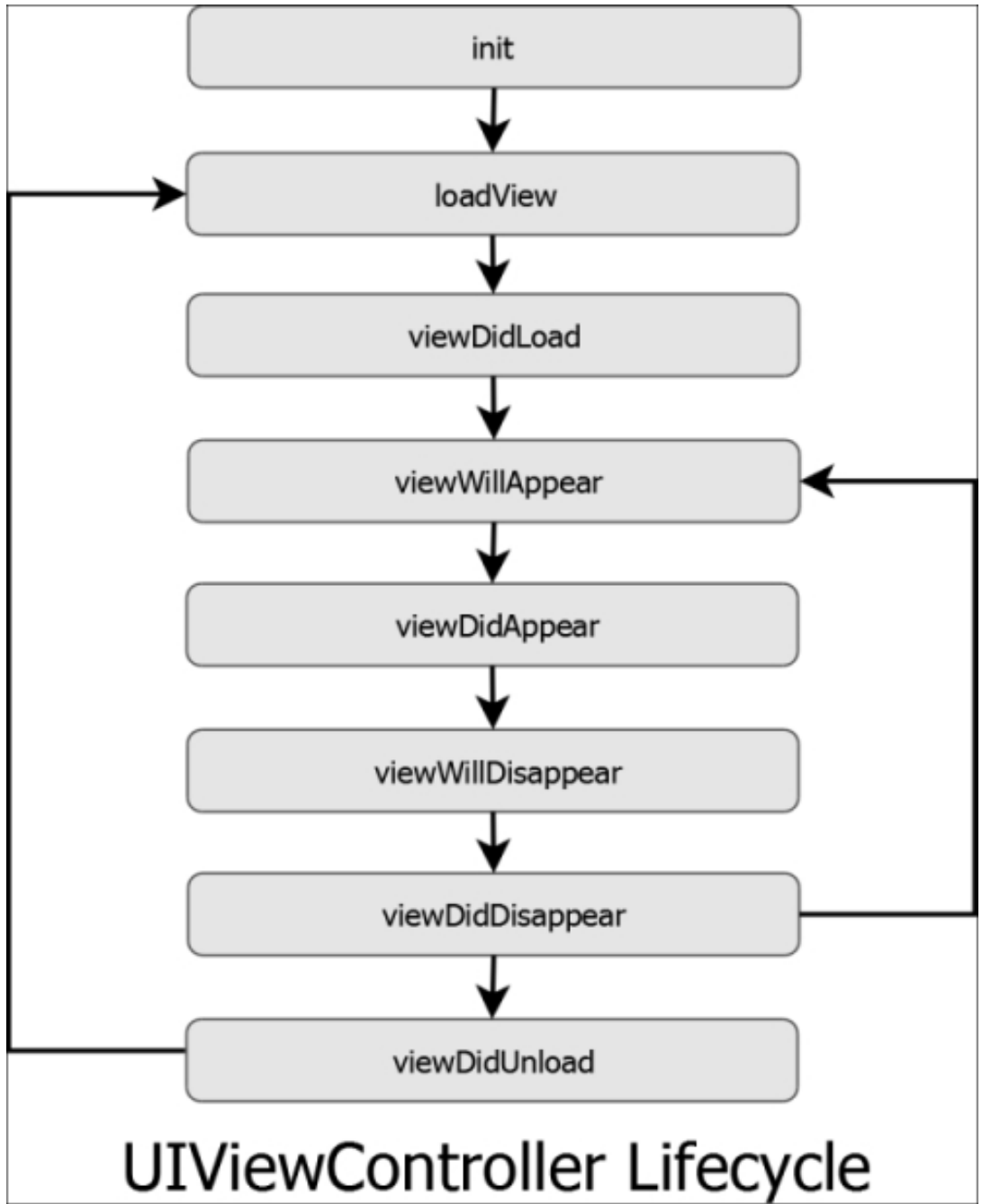
```
Frame
    origin = (40, 60)
    width = 80
    height = 130

Bounds
    origin = (280, 70)
    width = 80
    height = 130
```



ViewController

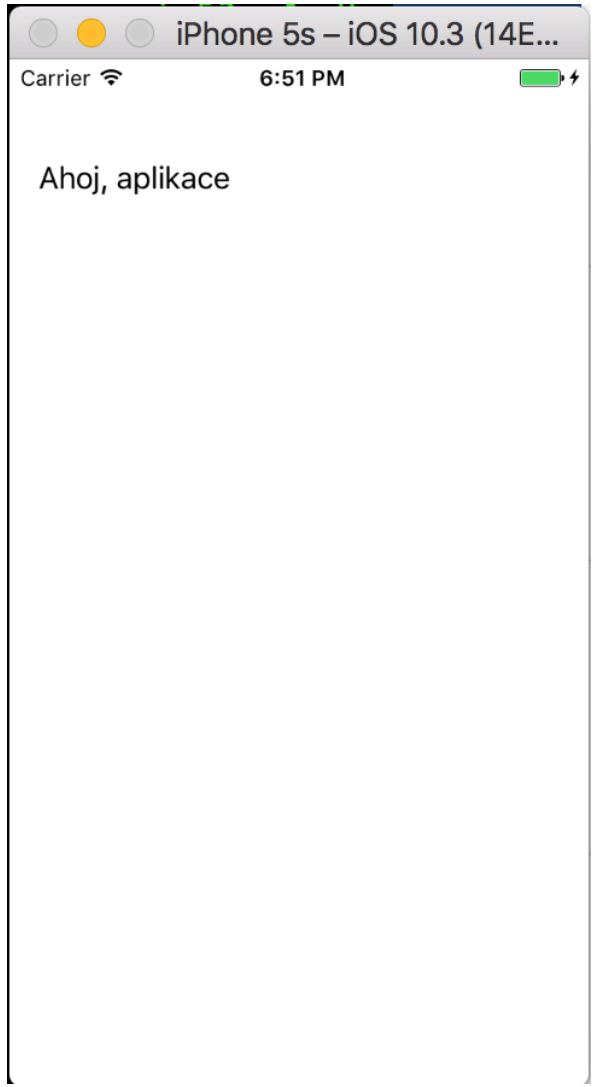
- Vlastní jeden view (ten má subviews).
- Má životní cyklus.
- Je/není aktivní (jeho view je/není viditelný).
- Kontejnerové VC:
  - UINavigationViewController,
  - UITabBarViewController,…
- UITableViewController

Životní cyklus ViewController



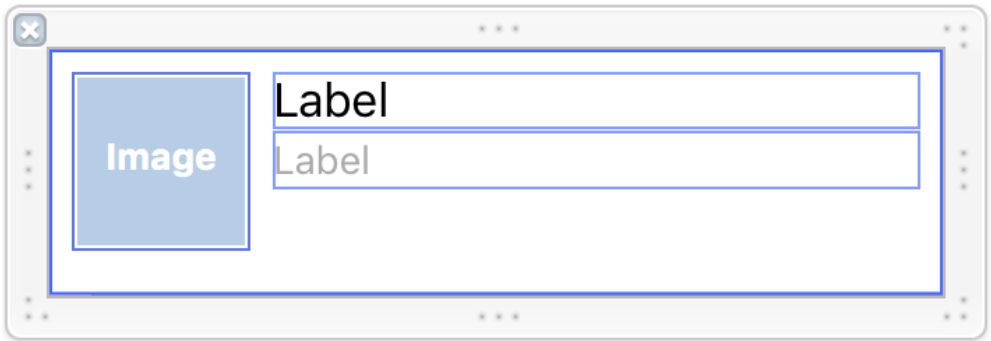
M-V-C, Základní demo

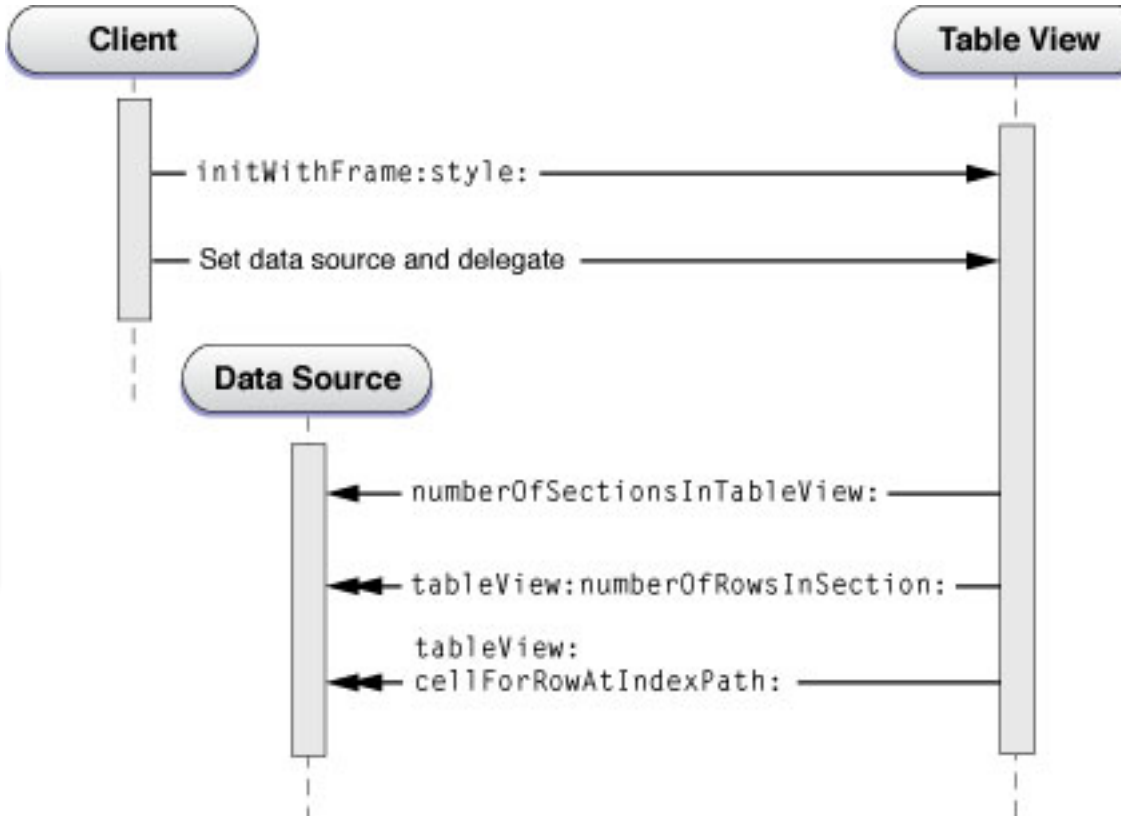
```
class MujModel {
    var obsah : String = "Ahoj, aplikace"
}

class ViewController: UIViewController {

    @IBOutlet var lei : UILabel?

    let mujmodel = MujModel()

    override func viewDidLoad() {
        super.viewDidLoad()

        //
        lei?.text = mujmodel.obsah
    }
}
```



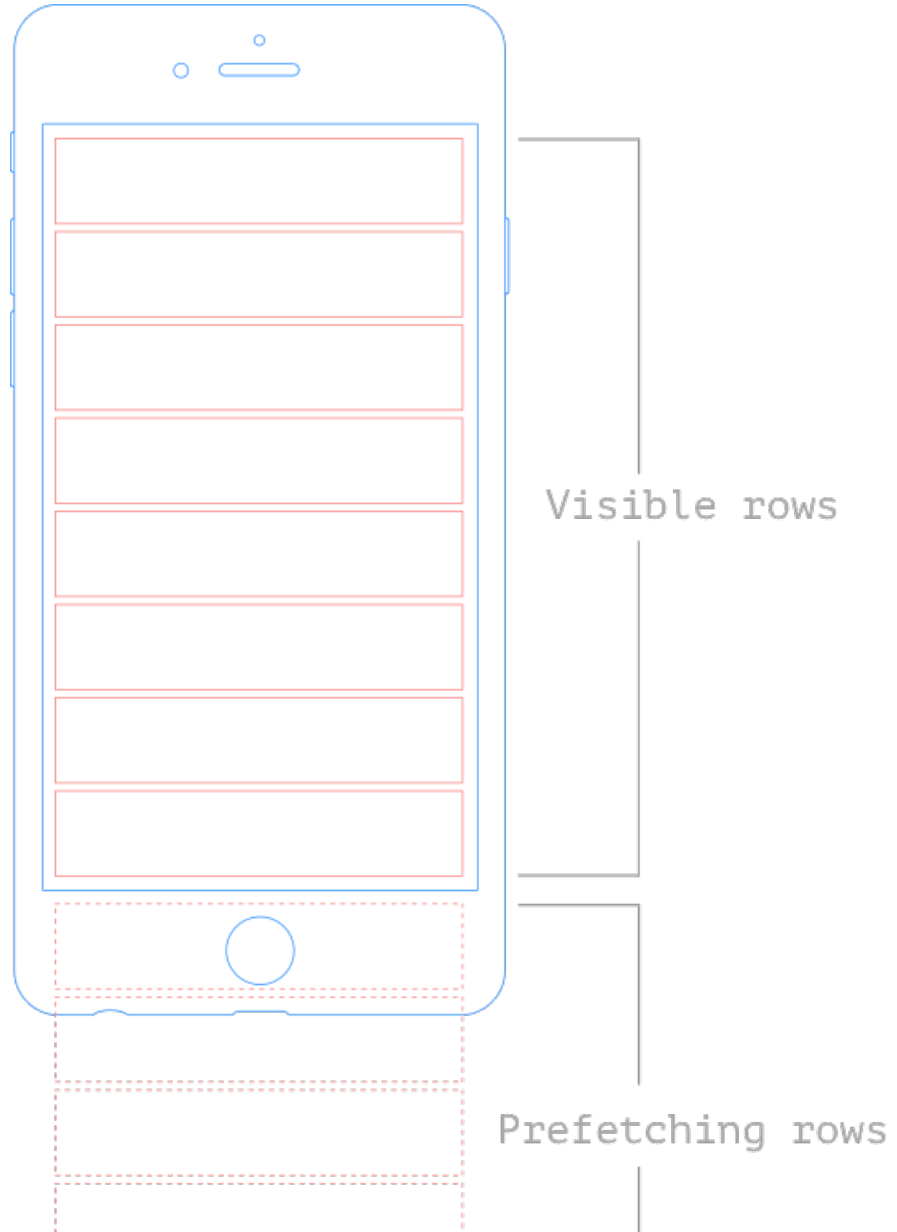
UITableViewController

- TVC zprostředkovává obsah (Model) do View.
- Model zde vystupuje jako ”dataSource” API:
  - počet sekcí, počet řádků v sekci.
  - sestavení TableViewCell pro zadanou souřadnici (IndexPath).





Dynamika přístupu na dataSource



Demo

```
class SimpleTab: UITableViewController {

    var seznam = ["Jeden", "Druhy", "Treti"];

    override func tableView(_ tableView: UITableView,
                numberOfRowsInSection section: Int) -> Int
    {
        return seznam.count
    }

    override func tableView(_ tableView: UITableView,
                cellForRowAt indexPath: IndexPath) -> UITableViewCell
    {
        let cell = tableView.dequeueReusableCell(withIdentifier: "cell",
for: indexPath)

        cell.textLabel?.text = seznam[indexPath.row]

        return cell
    }
}
```

Demo, oddělený dataSource

```
class MyArrayModel: NSObject, UITableViewDataSource {
    let seznam : [String]
    init(withStrings : [String]) {
        self.seznam = withStrings;
        super.init();
    }
    func tableView(_ tableView: UITableView,
                   numberOfRowsInSection section: Int) -> Int
    {
        return seznam.count
    }
    func tableView(_ tableView: UITableView,
                   cellForRowAt indexPath: IndexPath) -> UITableViewCell
    {
        let cell = tableView.dequeueReusableCell(withIdentifier: "cell",
for: indexPath)

        cell.textLabel?.text = seznam[indexPath.row]

        return cell
    }
}
```

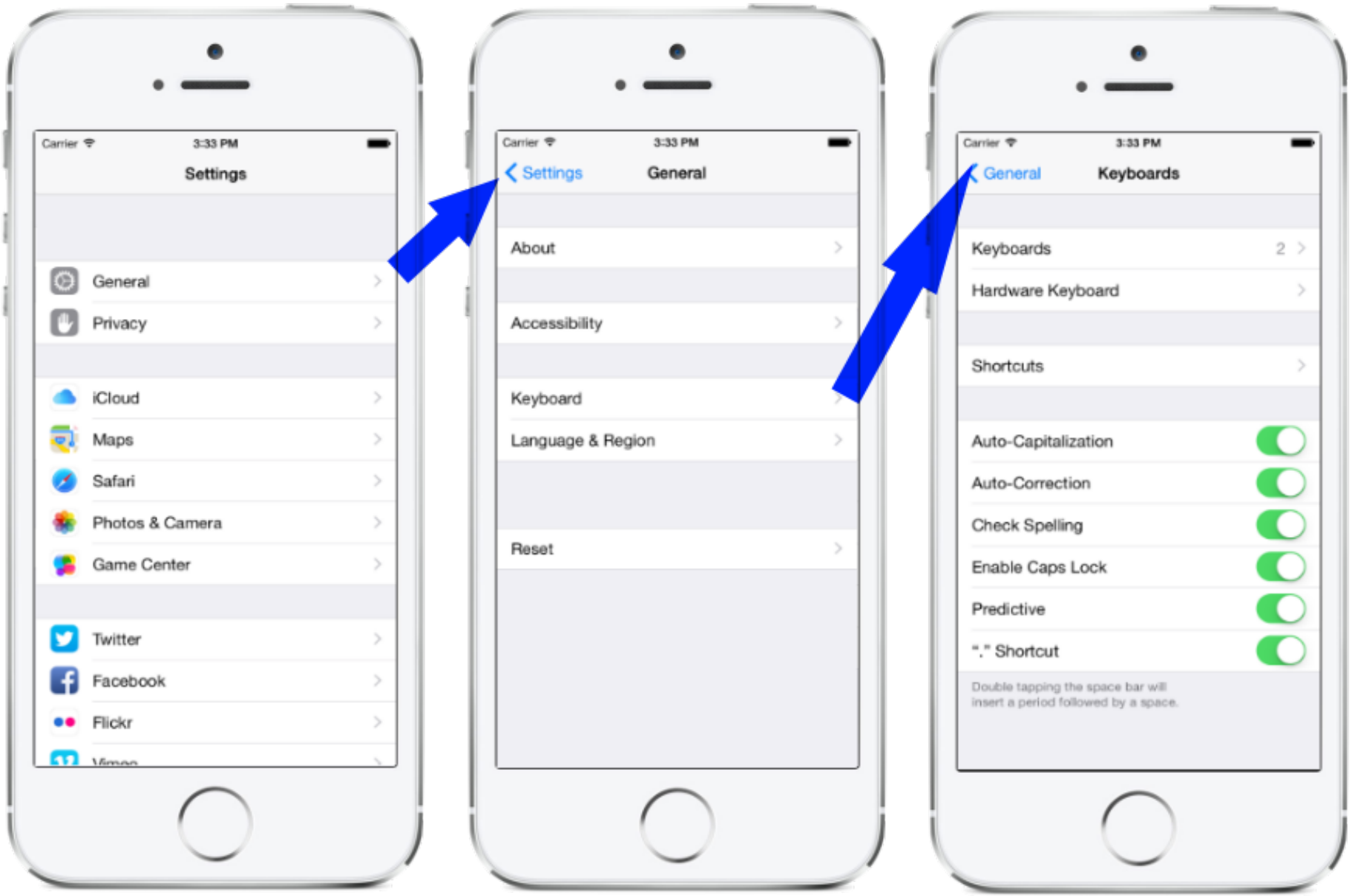
Komunikace mezi VC

- Vzájemné referencování.
- Protokol. Delegátství.
- prepare(for segue:) — předání dat do VC
- předání dat zpět:
  - událost ukončení VC (willMove(to:))
  - modálně — na tlačítko apod

Segue tam…

```
//
class MasterVC: UIViewController {
    // datovy obsah
    var obsahModel: String = "Ahoj"

    // akce byla navazana na tlacitko (namisto IBAction)
    override func prepare(for segue: UIStoryboardSegue, sender:
Any?) {
        // test: nazev segue a podmineny downcast na DetailVC
        if segue.identifier == "gotoDetail",
           let _vc = segue.destination as? DetailVC {
            // _vc je ve stavu init (neni jeste "loaded")
            _vc.data = DetailVCData(obsah: obsahModel) {
                // Typ "dt" je zrejmy, [weak self]!
                dt in self.obsahModel = dt.obsah
            }
        }
        /// else????
    }
}
```

Navigation VC



NavigationController

- Tlačítková lišta. Přidání tlačítka (UI Builder, programově).
  - Dynamika tlačítek. *navigationItem.left/right*
- Uživatelský VC je ”embedded” v NVC.
  - *UIViewController* dodává vypočtené property:
  - *navigationItem* — přístup na tlačítkovou lištu
  - *navigationController*: *UINavigationController?*
  - Segue: typ ”Show (e.g. push)”.

Řízení běhu programu

- Obsluha událostí z UI (@IBAction).
- Obsluha událostí od iOS, životní cyklus VC.
- Spouštění procesů (model).
- Asynchronní volání:
  - zahaj nějaký dotaz, zadej callback pro obsluhu zjištěné hodnoty

Vlákna a fronty

- Program se vykonává po kouscích (bloky ve frontách).
- K jednotlivým frontám přistupují dedikovaná vlákna. MainThread + ostatní vlákna.
  - MainQueue — sekvenční fronta.
  - GlobalQueue — obecně paralelní fronta.

MainQueue, MainThread

- MT má významné postavení. Přístup na UI.
- MQ je sekvenční. Kód vykonávaný MT se nemusí zamykat. A -> B -> C -> …
- Problematický je pouze kód vykonávaný jinými vlákny.
  - Přechody MT -> GT -> MT.
- Oddělme důsledně kód pro MT a pro GT.

DispatchQueue, GCD

- main, global
- sync { blok }
- async { blok }
  - func async(blk: @escaping ()->());

Sync / Async

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

DispatchQueue.global().async {
    //
    // otevri soubor
     // sync/async
    DispatchQueue.main.sync {
        //
        // posli zpravu UI, soubor otevren
    }
}
```

Synchronní vkládání

- Vkládající vlákno čeká, tj. je blokováno.

```
class VC: UIViewController {
    //
    override func viewDidLoad() {
        //
        super.viewDidLoad();

        // tady to slitne na EXC_BAD_INSTRUCTION
        DispatchQueue.main.sync {
            //
            print("Hello")
        }
    }
}
```

Operation, OperationQueue

- Operation je closure zabalená do objektu, tj:
  - lze to referencovat — lze tomu poslat zprávu,
  - může to mít vnitřní stav (tj. instanční proměnné)
- Sekvenční/paralelní fronty operací.
  - Lze definovat precedence.
- CloudKit.

Data v aplikaci

- UserDefaults.
- Kódování dat — NSCoding, Codable.
- Přístup na soubory v sandboxu. FileManager.
- UIDocument.
- CoreData.
- CloudKit. KeyValueStorage.

NSCoding demo

```
class City: NSObject, NSCoding
{
    var name: String?
    var id: Int?

    required init?(coder aDecoder: NSCoder)
    {
        self.name = aDecoder.decodeObject(forKey: "name") as? String
        self.id = aDecoder.decodeObject(forKey: "id") as? Int
    }

    func encode(with aCoder: NSCoder)
    {
        aCoder.encode(self.name, forKey: "name")
        aCoder.encode(self.id, forKey: "id")
    }
}
```

NSCoding, demo

```
// ----------------------------------------
// Kodovani
let _mdata = [City("Brno", 1), City("Ostrava", 2)]
// tridni metoda, koduje "_mdata" a vraci Data
let _encoded : Data? = NSKeyedArchiver.archivedData(withRootObject:
_mdata)
// ----------------------------------------
// Dekodovani. Z hlediska prekladace se ted NEVI, CO je vysledny typ
let _decoded = NSKeyedUnarchiver.unarchiveObject(with: _encoded!)
// as?

if let _decoded2 = NSKeyedUnarchiver.unarchiveObject(with: _encoded!)
as? [City] {
    //
    print(_decoded2)
}
```

Protokol Codable

```
// Codable spojuje dva protokoly
typealias Codable = Decodable & Encodable

// Datovy objet implementujici protokol "Codable"
// vsimnete si: NIC VIC se nechce...
struct MFile : Codable {
    //
    let name: String
    let size: Int
    let owner: String = "ja"
    let created: Date
}

//
struct MFolder : Codable {
    //
    let name: String
    let files: [MFile]
}
```

JSON, kódování

```
//
let _f1 = MFile(name: "soubor", size: 1000, created: Date())
let _d1 = MFolder(name: "dic1", files: [_f1])

// try? -- pripadne exception odchyti a vraci nil
// vysledny typ vyrazu je Data?
if let _encoded = try? JSONEncoder().encode(_d1) {
    // vysledek konstrukce je String?
    if let _stringVersion = String(data: _encoded, encoding: .utf8) {
        //
        print("Encoded: \(_stringVersion)")
    }
    // Encoded: {"name":"dic1","files":
[{"size":1000,"created":543077670.01387095,"owner":"ja","name":"soubor"}
]}
}
```

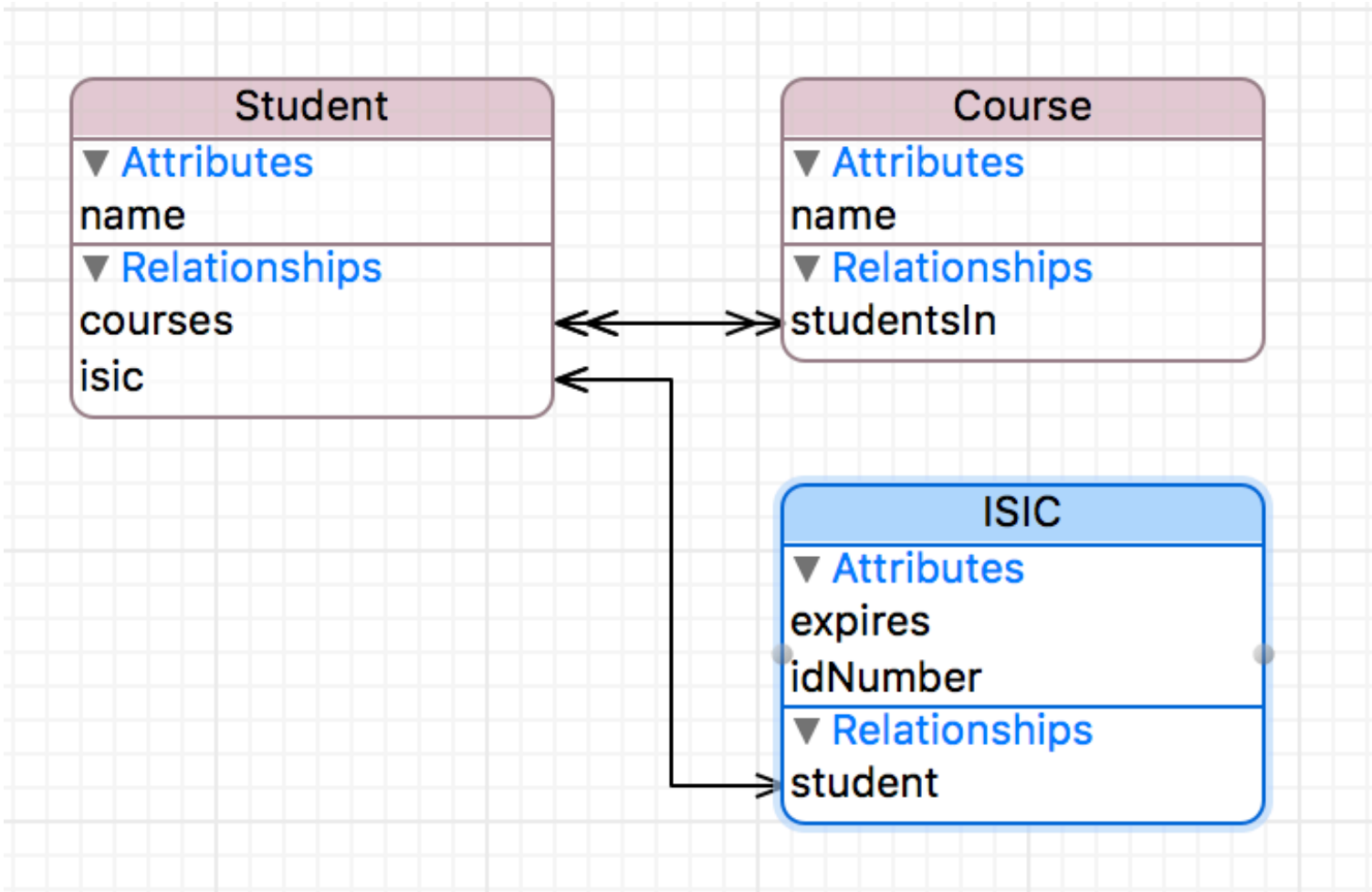
UIDocument

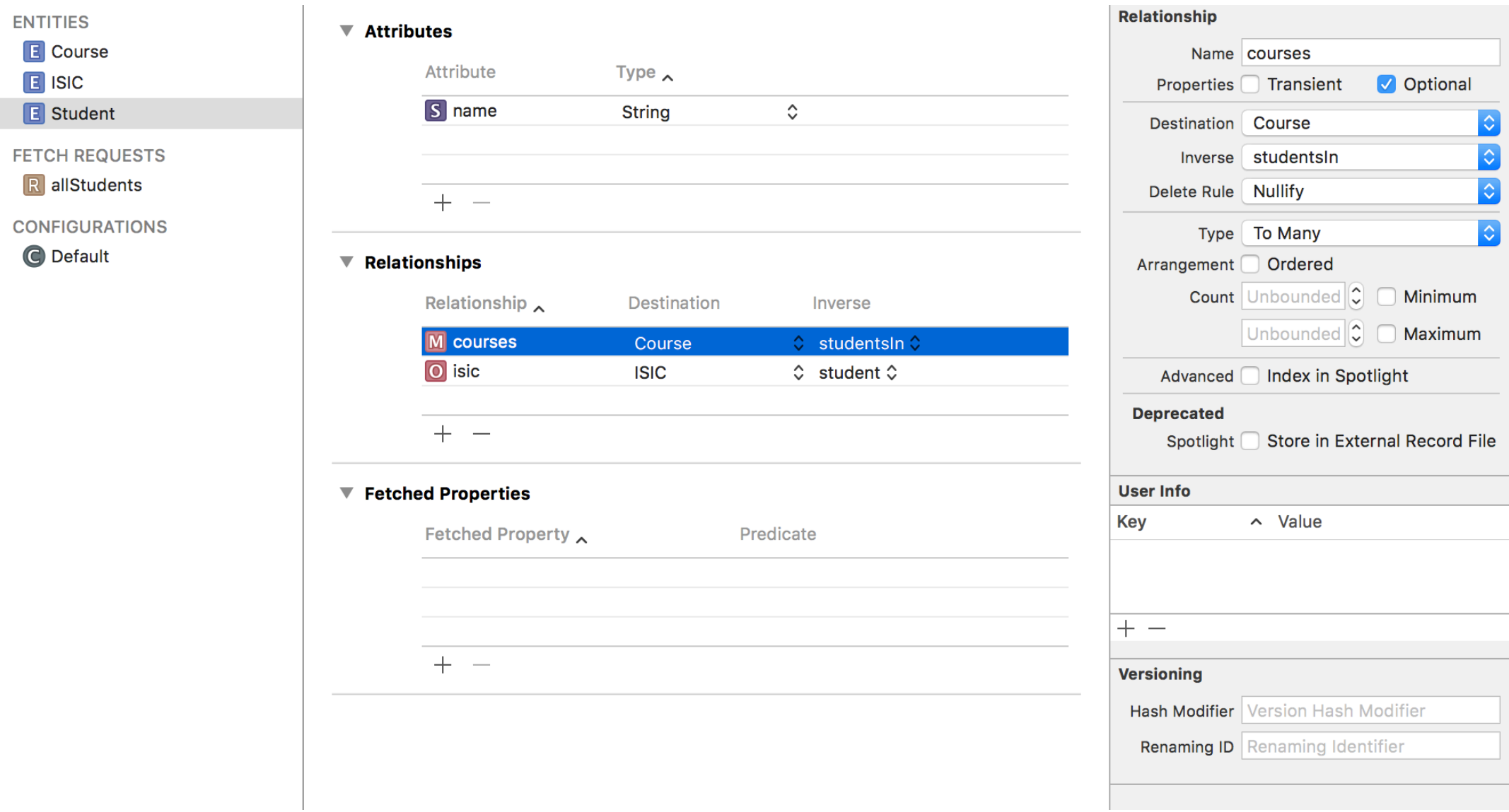
- Třída odvozující z UIDocument.
- Implementuje metody pro kódování a dekódování obsahu.
- Pak metody: open(), save(for creation/update)
  - Jsou asynchronní.
- Synchronizovatelnost cloudem.
  - Ubiquitous…přes FileManager.

CoreData

- Objektový kontejner.
- NSManagedObject — třída pro db entitu.
  - Atributy, reference 1:1, 1:N, M:N (inverzní).
- NSManagedObjectContext — objektová paměť.
  - API pro veškeré DB operace: insert, delete, select (fetch).
- Napojení DB na UI: FRC

XCode DB Model Editor




ENTITIES
Course
ISIC
Student
FETCH REQUESTS
allStudents
CONFIGURATIONS
Default
Attributes
Attribute
Type
name
String
Relationships
Relationship
Destination
Inverse
courses
Course
studentsIn
isic
ISIC
student
Fetched Properties
Fetched Property
Predicate
Relationship
Name
courses
Properties
Transient
Optional
Destination
Course
Inverse
studentsIn
Delete Rule
Nullify
Type
To Many
Arrangement
Ordered
Count
Unbounded
Minimum
Unbounded
Maximum
Advanced
Index in Spotlight
Deprecated
Spotlight
Store in External Record File
User Info
Key
Value
Versioning
Hash Modifier
Version Hash Modifier
Renaming ID
Renaming Identifier

```
class ViewController: UITableViewController {
    // pole objektu pro tabulku
    var content: [Student] = []
    //
    override func viewDidLoad() {
        //
        super.viewDidLoad()
        // fetch se provede az po vykresleni VC
        DispatchQueue.main.async {
            // pristup na MOC referencovany z AppDelegate
            let moc = (UIApplication.shared.delegate as!
AppDelegate).persistentContainer.viewContext
            // konstrukce dotazu -> template, vysledek [Student]
            let fetchr = NSFetchRequest<Student>(entityName: "Student")
            // provedeni dotazu
            if let _results = try? moc.fetch(fetchr) {
                //
                self.content = _results; self.tableView.reloadData()
            }
        }
    }
}
```

```
//
    var FRC: NSFetchedResultsController<Student>!

    //
    func initFRCStudents() {
        // FR
        let _fetchStudents = NSFetchRequest<Student>(entityName:
"Student")
        let _sortByNameUP = NSSortDescriptor(key: "name", ascending:
true)

        // povinna polozka FRC, jinak lehne
        _fetchStudents.sortDescriptors = [ _sortByNameUP ]

        // bez pouziti sections
        FRC = NSFetchedResultsController(fetchRequest: _fetchStudents,
                                         managedObjectContext: MOC,
                                         sectionNameKeyPath: nil,
                                         cacheName: nil)

        // NSFetchedResultsControllerDelegate
        FRC.delegate = self

        // Prvotni nacteni obsahu, inicializace
        try! FRC.performFetch()
    }
```

CoreData a vlákna

- Hlavní CD stack se odehrává v MT (je to forma sdíleného prvku, jako UI).
- Lze definovat další dočasné MOC (parent).

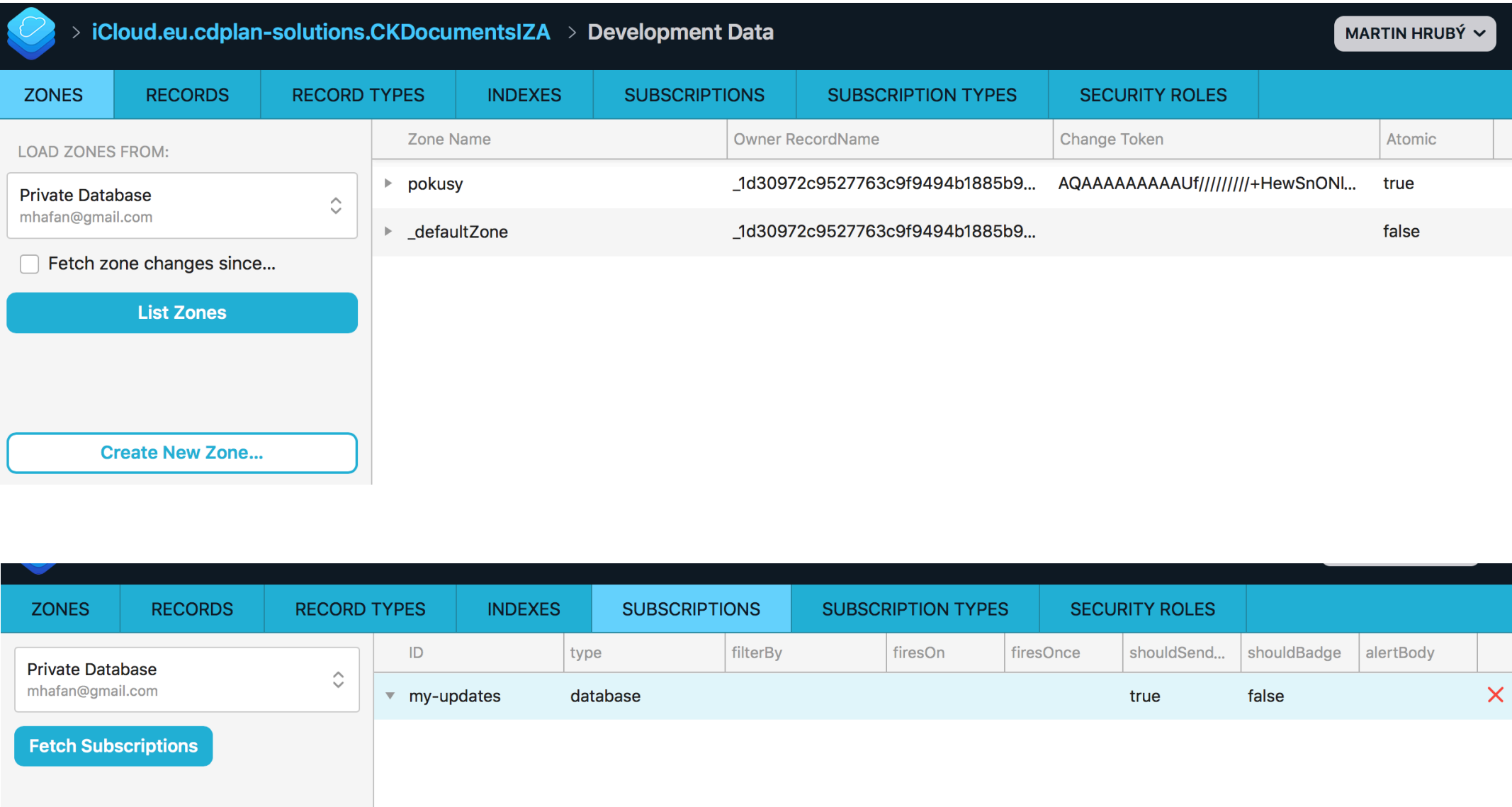
Aktualizace MO na pozadí

```
func onBackground() {
        //
        let _appd = UIApplication.shared.delegate as! AppDelegate
        let _vc = _appd.persistentContainer
        // detsky MOC k hlavnimu MOC, pojede v separatni fronte
        let _nb = _vc.newBackgroundContext()
        // save() provede merge do MOC
        MOC.automaticallyMergesChangesFromParent = true
        // do background vlakna/fronty
        _nb.perform {
            //
            let _fr = NSFetchRequest<Course>(entityName: "Course")
            //
            if let _res = try? _nb.fetch(_fr) {
                //
                for _c in _res {
                    //
                    _c.name = "Zapsano-" + _c.name!
                }
            }
            // proved zmeny do hlavniho MOCu
            try! _nb.save()
        }
    }
```

CloudKit

- Vzdálený objektový kontejner.
- Předpokládá se lokální cache dat.
- Kontejner má public/private/share DB.
- CKRecord — wrapper nad CK záznamem.
- DB operace (async).
- Subscriptions.

Sledování zón



Privátní DB

- Lze provádět operaci rozdílového fetch.
  - Nad explicitní zónou.
  - ServerChangeToken.
  - získávám [CKRecord] modifikovaných záznamů a [CKRecord.Id] smazaných záznamů.
- Vhodné pro synchronizaci CD—CK.

Public DB

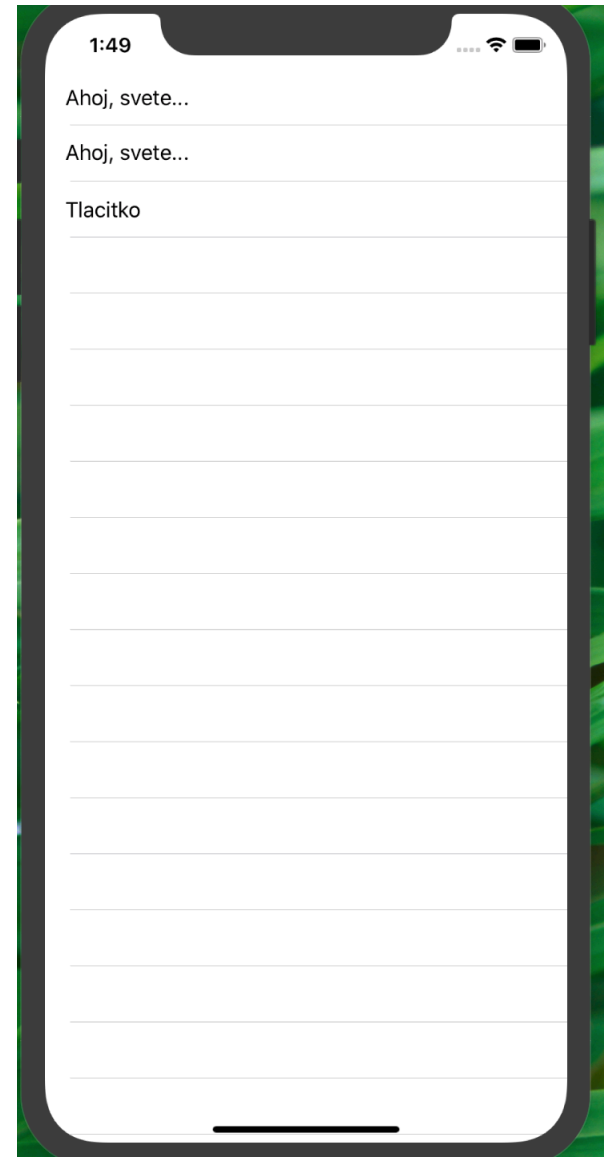
- CKQuery.
- Subscriptions.

SwiftUI+Combine

- Rozsáhlé téma. V detailech až příští akad. rok :)
- M-V-VM.
  - VM — vnitřní stav View.
  - @State, @Binding, @ObservedObject
- Property wrapper.
- some View.

Demo

```
// je to struktura! Dusledky...
struct ContentView: View {
    // vnitrni stav view
    @State var obsah = "Ahoj, svete"
    // podoba view
    var body: some View {
        //
        List {
            Text(obsah)
            TextField("zadej",text: $obsah)

            Button(action: { self.akce() }) {
                Text("Tlacitko")
            }
        }
    }
    // neni "mutating"...proc
    func akce() { obsah = "..." }
}
```



Combine

- Knihovna pro reaktivní programování.
- Publisher—Subscriber.
  - 1) Publisher zasílá do Subscriberu hodnoty / události.
  - 2) Subscriber sděluje Publisherovi, že je připraven získat další hodnotu. Publisher ji začne vyrábět…

Závěr

- Je třeba umět Swift.
- Je třeba umět dělat aplikace v MVC/Storyboard.
- Nadšenci prubnuli SwiftUI.