SwiftUI-III: Knihovna Combine (úvod)

Programování zařízení Apple (IZA)

Martin Hrubý, FIT VUT, 2025/26

Úvod

Opráštíme historické koncepty a oblékneme je do moderního kabátku.

- Key-Value Observing (KVO) - třída NSObject v Objective-C.
- Notifikační centrum.
- Grand Central Dispatch.
- Future-Promise.

... a budeme tomu říkat *reaktivní programování*.

Úvod: druhy sledovatelných objektů

Klasika:

```
class TRIDA: ObservableObject {
    //
    @Published var name: String = ""
}
```

Novinka:

```
@Observable class TRIDA {
    //
    var name = ""
}
```

Key-Value Observing

Objective-C:

```
@interface MojeTrida: NSObject {
    // uložená property
    @property NSString *jmeno;
}
```

Swift:

```
class MojeTrida {
    //
    var jmeno: String = "..."
}
```

Key-Value Observing

Objective-C:

```
@implementation MojeTrida {
// bude to uložená property v datech
// opak je @dynamic (CoreData)
@synthesize jmeno;

-(void) setJmeno: (NSString *) jmeno {
    // udalost
    [self willChangeValueForKey: "jmeno"]

    // primitivni operace
    _jmeno = jmeno

    // tady NSObject rozesílá zprávu o změně hodnoty
    [self didChangeValueForKey: "jmeno"]
}}
```


Registrovaný observers obdrží svou zprávu o změně `jmeno`

KVO, Observers (zjednodušeně)

```
let ref = MojeTrida()
// ...

ref.addObserver(self, forKey: "jmeno", ...)

// ... zprava dle protokolu NSKeyValueObserver
func udalostNaZmenu(from: Any, key: String, args: ...) {
    //
}
```

- Registrace observingu. (Odhlášení na `deinit`)
- Sync zpráva při změně hodnoty na property `jmeno`.
- Funkční koncept. Možná poněkud pracný.
- Napojení GUI (Cocoa).

Notifikační centrum (abstraktně)

- Singleton (typicky) v aplikaci.
- Komunikační kanály.
- Přihlášení odběru kanálu (observing).
- Poslání zprávy do kanálu (událost + nějaký argument).
- Synchronní rozeslání zprávy odběratelům.

V knihovnách UIKit/AppKit/SwiftUI/Combine pro systémové zprávy bez znalosti o jejich odběratelích.

- Delegátství - pro přímou komunikaci na delegáta.
- NotifCentrum - zprávy pro nejasné delegáty.

Šíření zprávy o změně Modelu app.

Teď však máme Swift/SwiftUI a Combine.

- Deklarativní programování UI.
- Implicitní propojení mezi Model/ViewModel a View.

SwiftUI aplikace se řídí změnami v datech!

Magické propojky:

- `@State` - je **uložená property** struktury typu `View`.
- `@Binding` - je dvojice getter/setter. Víc nic.

`@State` - při změně hodnoty posílá systémovou zprávu "Komusi". Výsledek je přebudování kompletního View aplikace.

- ... víc bližší poznání o věci chybí ... :(

@propertyWrapper (opakování)

Property-wrapper je uložená datová struktura (jsou to vnitřní data), která:

- v sobě zapouzdřuje (wrapper) cílovou datovou proměnnou
- automatizuje poslání `didSet` zprávy Komusi
- automatizuje vytvoření `Binding<Value>` na sebe.

@propertyWrapper (opakování)

```
@propertyWrapper struct State<Value> {
    //
    var wrappedValue: Value {
        //
        didSet {
            // zprava Komusi
        }
    }
    // $hodnota
    var projectedValue: Binding<Value> {
        //
        Binding<Value>(get: { wrappedValue },
                       set: { wrappedValue = $0 })
    }
}
```

@propertyWrapper (opakování)

```
struct CosiMoje: View {
    //
    @State var jmeno: String = "ddd"
    //
    var body: some View {
        // projectedValue
        TextField("", text: $jmeno)
    }
}
```

Zavádí uloženou property:

```
// ...
var _jmeno: State<String> = "ddd"
```

`jmeno` je virtualizace pro `_jmeno.wrappedValue`

@propertyWrapper (opakování)

```
struct CosiMoje: View {
    //
    @State var jmeno: String = "ddd"

    // případná konstrukce hodnoty...
    init(withJmeno: String) {
        //
        _jmeno = State(initValue: withJmeno)
    }
}
```

@Binding

Je property wrapper nad zadanou dvojicí getter/setter.

```
@propertyWrapper struct Binding<Value> {
    // uložené closures pro getter/setter
    let get: () -> (Value)
    let set: (Value) -> ()
    //
    var wrappedValue: Value {
        //
        get { get() }
        set { set(newValue) }
    }
    //
    init(get: ..., set: ...) {}
}
```

@State a @Binding -> a dále ...

Jsou "magické propojky" z knihovny SwiftUI.

- Tvoří ViewModel aplikace.
- Na ně navazují `@ObservedObject` a `@StateObject`

Ovšem, jejich činnost je založena na širším konceptu:

- **Publisher-Subscriber**.

`@ObservedObject` a `@StateObject` jsou Subscriber nějakého Publisher, o kterém si dneska povíme.

ObservableObject (protokol)

```
// protokol je typu AnyObject (!)
class MujModel: ObservableObject {
    //
    @Published var jmeno: String = "ddd"
}
```

- `@Published` je property wrapper,
- zapouzdřující objekt `Publisher<Output,Never>`
- (on je to vlastně Subject, ale to si povíme za chvíli)

A `Publisher<Output,Never>` je komponenta, která má Subscribery (tok událostí).

NSObject: KVO versus Publisher

- KVO: rozesílání zpráv o změnách sledované property
- Publisher: toto poskytuje taky, ovšem jsou tam větší rozdíly.

Rozdíly:

- Systém Publisher (Combine) je šablonový a striktně typovaný.
- Publisher se budou řetězit.
  - `publisherXY.map(...).map(...)....`
  - operátory

@Published schematicky

```
@propertyWrapper Published<Output> struct {
    // Subject, CurrentValue
    let publisher: Publisher<Output, Never>

    //
    var wrappedValue: Output {
        //
        willSet {
            // rozesílá se nová hodnota (= událost)
            publisher.send(newValue)
        }
    }
}
```

Rozesílanou událostí je nová-čerstvá hodnota.

- KVO - rozesílá se zpráva o změně.

@Published var

- `@Published var` je jednou z aplikací schematu Publisher-Subscriber
- tj jsou další způsoby použití

Publisher<Output,Failure> schematicky

Je to rozesílač událostí svým Subscribers (observers):

- událost = hodnota typu `Output`
- Systém chyb - `Failure: Error`.
- Registrace odběratele.
- Přijímání zpráv.

Subscription je dynamická (`AnyCancellable`).

- Stream hodnot.
- Ukončen "EOF" nebo chybou.

ObservableObject

```
// protokol je typu AnyObject (!)
class MujModel: ObservableObject {
    //
    @Published var jmeno: String = "ddd"
}
```

- Obsahuje v sobě objekt `objectWillChange`, který je Publisher.
- `ObservableObject` je automaticky Subscriber všech svých `@Published` properties.

Tedy, instance třídy `MujModel` je prvním a implicitním odběratelem zpráv z published property `jmeno`.

ObservableObject

```
// protokol je typu AnyObject (!)
class MujModel: ObservableObject {
    //
    @Published var jmeno: String = "ddd"
}
```

Při události z `jmeno` :

- Publisher ve `jmeno` rozešle notifikaci.
- `MujModel` je mezi odběrateli.
  - generuje událost do `objectWillChange`
  - … který má svoje observers

@ObservedObject a @StateObject

```
struct Obrazovka: View {
    // chová se při změně jako @State
    @ObservedObject model: MujModel

    //
    var body: some View {
        //
        Text(model.jmeno)
    }
}
```

- `@ObservedObject` je registrovaný observer `MujModel`, tj
- konkrétně `MujModel.objectWillChange`
- událost předá Komusi, stejně jako `@State`.

ProjectedValue na @Published

```
struct Obrazovka: View {
    //
    @ObservedObject model: MujModel
}
```

`$model.jmeno` je Binding na hodnotu property `jmeno`

`model.$jmeno` je reference na Publisher z property `jmeno`

Pozn.: `$cosi.kdesi` jsou symbolická virtuální jména generovaná překladačem Swiftu.

- `$` stateVar -> Binding
- `$` publishedVar -> Publisher

.onReceive(blk: ...)

```
struct Obrazovka: View {
    // prosta reference, negeneruje subscription
    let model: MujModel
    //
    @State var modelJmeno: String = ""

    //
    var body: View {
        //
        Text(modelJmeno)
            // subscription přímo na Publisher
            // ve "jmeno"
            .onReceive(model.$jmeno) {
                // v ramci zpravy obdrzim...
                (val:String) in
                // akce
                modelJmeno = val
            }
    }
}
```

@Observable

```
//
@Observable class MujMudel {
    //
    var jmeno = "hehe"
}

//
struct Obrazovka: View {
    // prosta reference
    let model: MujModel
    //
    var body: View {
        // View se stava .onReceiver observerem
        // automaticky
        Text(model.jmeno)
    }
}
```

Sledování @Observable

```
//
@Observable class MujMudel {
    //
    var jmeno = "" {
        //
        didSet { jmenoPublisher.send(jmeno) }
    }

    //
    let jmenoPublisher = CurrentValueSubject<String, Never>("")
}

//
struct Obrazovka: View {
    // prosta reference
    let model: MujModel
    //
    var body: View {
        // stava se .onReceiver observerem
        // automaticky
        Text("cosi").onReceive(model.jmenoPublisher) {
            val in ...
        }
    }
}
```

Sledování @Observable

```
//
@Observable class MujMudel {
    //
    var jmeno = ""
}

//
struct Obrazovka: View {
    // prosta reference
    let model: MujModel
    //
    var body: View {
        // stava se .onReceiver observerem
        // automaticky
        Text("cosi").onChange(model.jmeno) { old, new in

        }
    }
}
```

Sumarizace 1. části

Řekli jsme si o toku událostí o změnách modelu.

- VM -> V (@State)
- M -> VM -> V (@Published, @ObservedObject, ...)

V první části jsme chtěli:

- sledovat proměnnou
- ... a dostávat zprávy o její změně
- ... a případně si vynutit aktualizaci View

Combine

Reaktivní programování = zřetězení async volání.

- Future-promise.
- Publisher - protokol.
- Subscriber - protokol.

Co je vlastně publisher.

Future-promise

- Klasický koncept asynchronního výpočtu.
- Synchronně obdržím "future".
- … a může se asynchronně rozběhnout výpočet,
- … na kterých, když se chci dotázat, tak se s ním synchronizuji.

Future je jako účtenka z automatu v MacDonaldu, která spustí přípravu objednávky a nakonec je směněna za produkt :)

Future-promise symbolicky

Vyjádření návratové hodnoty `Value` nebo hodnoty chyby `Failure`:

```
enum Result<Value,Failure>
```

Vytvořím future:

```
//
func nastartuj() -> Future<String, Never> {
    //
    return Future<String,Never> {
        // ... closure () -> (Res)
        promise in
        // ... vypocet a navratova hodnota
        promise(.success("ahoj"))
    }
}
```

Future-promise symbolicky

Získávám "poukázku na budoucí hodnotu".

```
// synchronne ziskam future
let f = nastartuj()
```

- potenciálně nastartuje výpočet na pozadí
- a když ji chci "poukázku vybrat", pak až teprve potom se na výpočet synchronizuji
- tj případně čekám na výsledek

Pozn.: Implementace v STD C++.

- v Combine je Future především Publisher, tj zahájí se až na připojení Subscribera.

Future-promise demo

```
func dejmi() -> Future<String, Never> {
    //
    return Future<String, Never> {
        // handle na cil,
        // příslib hodnoty Result<String, Never>
        promise in

        // ve vedlejsim vlakne
        DispatchQueue.global().async {
            // předávám v Main Thread
            DispatchQueue.main.async {
                // návratová hodnota async
                promise(.success("hello"))
            }
        }
    }
}
```


Pozn.: není tam `return`. Hodnota se vrací async

Async konvenčně

```
DispatchQueue.global().async {
    // vypocet
    ...

    // predani hodnoty, pripadne:
    // DispatchQueue.main.async {}
    promise(hodnota)
}
```

V čem je rozdíl?

- future je hodnota, kterou držím (je mi to k něčemu?)
- … je to hodnota, na kterou se opakovaně můžu dotázat
- … a vytváří standardní rozhraní (reaktivní)

Kdo ten Future-výpočet spouští?

Když se podíváme znova na tu konstrukci.

```
// synchronne ziskam future
let f = nastartuj()
```

Kde je invokace toho výpočtu Future?

- protokoly Publisher a Subscriber
- celkový koncept reaktivního programování
- back-pressure

```
class MujModel: ObservableObject {
    //
    @Published var lejbl = "cosi"
    // potřebuju někoho referencovat
    var sinkNaFuture: AnyCancellable?
    //
    func udelej() {
        // mám future
        let f = dejmi()

        // registruju observer na future
        sinkNaFuture = f.sink { v in
            //
            self.lejbl = v
        }
    }
}
```

Vlastnictví reference je na straně subscribera.

Future-Promise, smysl

FP je příklad **jednorázového výpočtu** async typu.

- Výpočet.
- Síťový/Databázový dotaz.

`sink` je druh Subscribera. Future je Publisher.

Tady je na místě zkoumat, co jsem touto konstrukcí naprogramoval efektivněji, než konvenčním způsobem :)

dejmi() konvenčně

```
func dejmi(callback: @escaping (String) -> Void) {
    // ve vedlejsim vlakne
    DispatchQueue.global().async {
        // předávám v Main Thread
        DispatchQueue.main.async {
            //
            callback("hello")
        }
    }
}
```

Tohle umíme. Nastartuje výpočet, až dokončí, volá callback.

Co je lepšího na Future?

Přehled výpočtu hodnoty

Synchronně:

```
func dejmi() -> String { return "hello" }
```

Asynchronně konvenčně:

```
func dejmi(callback: @escaping (String)->Void)
{ ... callback("hello") }
```

Asynchronně reaktivně:

```
// Future je druh Publishera
func dejmi() -> Publisher<String, Never>
{ ... }
```

```
class MujVypocet: Operation {
    //
    let zadani: String
    var vysledek: String?
    //
    func notify() {
        //
        DispatchQueue.main.async {
            NotificationCenter.default.post(name: ...,
                                            object: self)
        }
    }
    //
    init(zadani: String) {
        self.zadani = zadani
    }
    //
    override func main() {
        // vypocet
    }
}
```

Smysl?

Smysl je ve standardizaci rozhraní na ASYNC výpočty.

- Rozhraní je `Publisher<Output,Failure>`.
- ...resp `AnyPublisher<Output,Failure>`.
- Publisher je protokol s asociovanými typy.

Na publishera se totiž může napojit subscriber, který:

- ... čte události z publishera,
- události jsou hodnoty typu `Output`, případně chyby,
- události tvoří tok hodnot typu `Output`.

Upstream/Downstream.

Reaktivní programování schematicky

Je zřetězení asynchronních volání.

- `Publisher<Typ1>` ->
- má subscribera, který je zároveň `Publisher<Typ2>`

Tedy, je fakticky mapovací funkcí:

- `(Typ1) -> (Typ2)`

Vytváříme toky událostí:

- P1 -> P2 -> ... -> Subscriber
- ... případné "odbočky" kdekoli na cestě ...

Subscriber

Dva typy Subscriber implementované v Combine:

- Sink (kuchyňský dřez, odtok).
- Assign.

```
// ...
@Published var jmeno: String
```

obsahuje `Publisher<String, Never>`.

```
$jmeno.sink { _jmeno in ... }
```

Je instance Subscribera. Datový typ `AnyCancellable` (protokol).

Subscriber: Sink

```
// associatedtype Output
extension Publisher {
    //
    func sink(receiveValue: @escaping ((Self.Output) -> Void))
        -> AnyCancellable
        {
            // vytvori instanci Sink
            // tu učiní subscriberem sebe (self)
            // a vrací referenci na Sink
        }
}
```

Zpráva `sink` poslaná Publisher vrací referenci na vytvořeného Sink. Vlastnictví reference.

Vytvoření Sink I.

```
class Model {
    //
    private var mujSink: AnyCancellable?

    //
    init() {
        //
        self.mujSink = nejakyPublisher.sink { v in
            //
        }
    }
}
```

Vytvoření Sink II.

```
class Model {
    // chci mít více ... je to množina
    private var allMySinks = Set<AnyCancellable>()

    //
    init() {
        //
        nejakyPublisher
            .sink { v in
                //
            } // referenci na Sink posílám store...
            .store(in: &allMySinks)
    }
}
```

Vytvoření Sink III.

Tok hodnot (stream) z Publishera v sobě dále obsahuje:

- případnou chybu (Failure)
- systémové události (typicky EOF - konec toku)

```
$pub.sink {
    // Enum: .finished, .failure(Failure)
    Subscribers.Completion<Never> in
    //
    } receiveValue: { val in
        //
    }
```

Assign Subscriber

```
class Model: ObservableObjected {
    // chci mít více ... je to množina
    private var allMySinks = Set<AnyCancellable>()

    //
    @Published var hodnota: Cosi

    //
    init() {
        //
        nejakyPublisher
            // KeyPath<Root,Value>
            .assign(to: \Model.hodnota, root: self)
            .store(in: &allMySinks)
    }
}
```

Combine: svatá trojice :)

- Publisher
- Subscriber
- Subscription

Subscription referencuje Subscribera a Publishera.

Je implementátorem protokolu `AnyCancellable`.

```
subs.Cancel()
```

Odpojí subscribera, případně i dealokuje Publishera.

Typy publisherů

`Publisher<Output,Failure>` je dán protokolem:

```
public protocol Publisher {
    //
    associatedtype Output
    associatedtype Failure : Error

    //
    func receive<S>(subscriber: S)
        where S : Subscriber,
            Self.Failure == S.Failure,
            Self.Output == S.Input
}
```

Typy publisherů

- Publisher <Output, Failure>
- Subject <Output, Failure>

Subject je typ publishera, do kterého lze "poslat hodnotu".
Subject je roura (pipe) pro události.
Na druhé straně jsou subscribers.

Triviální Publisher:

- `Just<Output>(_ value: Output)` - publikuje konstantu.
- Dynamika sink. Co se odehraje po napojení subscribera.

Čerpání toku směrem OD subscribera.

Back-pressure

- Sice to vypadá, že události tečou "shora dolů".
- Ale je to naopak :)

V rámci protokolu Subscription:

- Publisher registruje subscription
- … a oznámi Subscriberovi vytvořené Subscription
- … ten v odpovědi ohlásí svou připravenost na tok událostí
- publisher to kapacitně vyplní …

Více nechť si nastudují nadšenci (tvorba vlastních Publisher).

Implementovaní Subject

- `PassthroughSubject<Output,Failure>()`.
- `CurrentValueSubject<Output,Failure>(_ value: Output)`

Demo:

```
let Konsta = Just(5)
let CV = CurrentValueSubject<Int,Never>(3)

// obdrží Int(5), pak EOF
let a = Konsta.sink { v in print(v) }
//
let b = CV.sink { v in print(v) }
//
CV.send(1); CV.send(12345)
```

@Published revisited

PropertyWrapper `Published` tedy v sobě obsahuje `PassthroughSubject`.

`ObservableObject` tedy má `objectWillChange` typu `PassthroughSubject<Void,Never>`.

- on je to teda `ObservableObjectPublisher`.

Operátory

Jsou mapovací funkce `(Typ1)->(Typ2)`:

- upstream: je `Publisher<Typ1>`
- downstream: je `Publisher<Typ2>`

```
// Publisher<Int, Never>
let j = Just(10)

// Publisher<Int> -> Publisher<Int>
let k = j.map { v in v * 2 }

//
let ss = k.sink { print($0) }
```

Striktně typované! Šablonové peklo! :)

Operátory

- map, compactMap, flatMap, filter, removeDuplicates
- decode
- řídicí - receive, debounce, ...
- statistické - min, max, collect, count

Slučování streamů

- Combine, Zip, merge

```
Publishers.Zip($hledej, $napoveda)
    .sink { a, b in
        //
    }
```

Aplikace: síťové dotazy

```
struct Post: Codable {
    let id: Int
    let title: String
    let body: String
    let userId: Int
}

//
let url = URL(string: "...")! //

// řetězení datových typů <Output, Failure>
let reqp = URLSession.shared.dataTaskPublisher(for: url)
    .map { $0.data }
    .decode(type: [Post].self, decoder: JSONDecoder())
    .replaceError(with: [])
    // -> wrapper na AnyPublisher
    .eraseToAnyPublisher()
    .sink(receiveValue: { posts in
        print(posts.count)
    })
```

Aplikace: síťové dotazy

Univerzální funkce na zpracování GET dotazu.

```
func netLoader<T:Decodable>(Record: T.Type,
                            url: URL) -> AnyPublisher<[T], Never>
{
    // Future<...>
    URLSession.shared.dataTaskPublisher(for: url)
        .map { $0.data }
        .decode(type: [T].self, decoder: JSONDecoder())
        .replaceError(with: [])
        .eraseToAnyPublisher()
}
```

Specialita:

```
.eraseToAnyPublisher() -> AnyPublisher<Output, Failure>
```

Model

```
class MujModel: ObservableObject {
    //
    @Published var posts: [Post] = []

    //
    private var _anyc: AnyCancellable?

    //
    init() {
        // AnyPublisher<[], Never>
        _anyc = netLoader(Record: Post.Self, url: ...)
            .sink { v in self.posts = v }
    }
}
```

Znovupoužitelnost `netLoader`.

View

```
struct Obrazovka: View {
    //
    @StateObject var model = MujModel()

    //
    var body: some View {
        //
        List(model.posts) { p in
            ///
        }
    }
}
```

Trápí mě vlákna?

Napojení na GUI

```
class MujModel: ObservableObject {
    // Model, ktery se stava i ViewModelem
    @Published var hledej: String = ""
    @Published var napoveda: [String] = []
    //
    private var _anyc = Set<AnyCancellable>()
    //
    func vypoctiNapovedu(_ z: String)
        -> [String] { return [] }
    //
    init() {
        //
        $hledej
            .debounce(for: 0.5, scheduler: RunLoop.main)
            .removeDuplicates()
            .map { self.vypoctiNapovedu($0) }
            .assign(to: \MujModel.napoveda, on: self)
            .store(in: &_anyc)
    }
}
```

```
struct ContentView: View {
    // VM pro toto View
    @StateObject var model = MujModel()

    //
    var body: some View {
        //
        VStack {
            //
            TextField("hledej", text: $model.hledej)

            //
            List(model.napoveda, id: \.self) { i in
                //
                Text(i)
            }
        }
    }
}
```


Konvenčně?

flatMap

```
$url
    .debounce(for: 0.5, scheduler: RunLoop.main)
    .removeDuplicates()
    .flatMap { dotaz(url: $0) }
    .map { [$0] }
    .assign(to: \MO.results, on: self)
    .store(in: &_anies)
```

Alternativně:

```
    .map { dotaz(url: $0) }
    .switchToLatest()
```

Notifikační centrum

Zmodernizováno do podoby publisherů.

```
NotificationCenter.default
    .publisher(for: .NSCalendarDayChanged)
    .sink { notif in
        //
    }
```

```
// informacni kanal dany jmenem
let NN = Notification.Name("hello-app")

//
NotificationCenter.default.post(NN)
```

Závěr

- Značně rozsáhlé téma.

Příští přednášky:

- D. Procházka: Principy UI.
- (cviko na Combine)
- (AWS Amazon, přehled služeb, Docker, Kubernetes)