SwiftUI II.: Aplikační základy

Programování zařízení Apple (IZA)

Martin Hrubý, FIT VUT, 2022/23

Úvod

- Aplikační infrastruktura - AppDelegate.
- Obvyklé konstrukce v aplikacích.
- Observable Object - globální Model v aplikaci.
- Něžný úvod do Combine.

Obvyklé konstrukce - seznam záznamů, editace/zobrazení detailu.

AppDelegate

Opět vidíme to `some`.

```
@main
struct MojeAplikace: App {
    var body: some Scene {
        // init(content: ...)
        WindowGroup {
            // Instanciace hlavního pohledu
            ContentView()
        }
    }
}
```

- `@main` - StoryBoard style "initial view controller"

Celé UI je vypočteno "najednou" a vzniká jeho derivační strom.

AppDelegate - globální data

Chceme z kontextu "app delegáta" vlastnit klíčové singletony.

V případě `struct` by to nedávalo příliš smysl.

Obvyklejší jsou záznamy typu `@environment`.

```
@main
struct MojeAplikace: App {
    //
    static let mujHlavniDatovyObjekt = ToJeOn()

    //
    var body: some Scene {
        // init(content: ...)
        WindowGroup {
            // Instanciace hlavního pohledu
            ContentView()
        }
    }
}
```

AppDelegate - globální data

Singleton data/process model.

```
class MyAppGlobalModel {
    //
    static let shared = MyAppGlobalModel()

    //
    init() {
        // nastartuj procesy
        // otevri sitova/DP spojeni
        // apod
    }
}
```

AppDelegate - globální data

Singleton data/process model.

```
class MyAppGlobalModel {
    //
    static let shared = MyAppGlobalModel()
    // ...
    func startup() {
        //
    }
}
//
@main struct MojeAplikace: App {
    //
    init() {
        DispatchMain.main.async {
            MyAppGlobalModel.shared.startup()
        }
    }
}
```

KeyPath

```
KeyPath<Root,Value>
```

je vyjádření přístupu na property strukturované hodnoty.

```
class Record {
    var name: String
    var age: Int
    func value<X>(kk: KeyPath<Record, X>) -> X {
        //
        return self[keyPath: kk]
    }
}
//
let p = Record(...)

// KeyPath<Record,String>
let value = p.value(kk: \.name)
```

KeyPath

```
//
let p = Record(...)
// je datova hodnota, lze uložit
let keyp = \Record.name

// subscript
p[keyPath: keyp] = "cosikdesi"

// pokud lze z kontextu dovodit Root
p[keyPath: \.name] = "pepa"

// občas vídáme \.self
p[keyPath: \.self] == p
```

Typicky u List:

```
List(nejakePoleStringu, id: \.self) { i in ...}
```

Koncepce Environment hodnot

Environment je jako systémová proměnná, která je k dispozici od nějakého kořene View směrem k sub-views.

- tj, někdo tu proměnnou založí
- pak se automaticky šíří "dolů" stromem "subviews"
- v sub-views ji lze "zhmotnit" do podoby proměnné a přistupovat na ni (pouze pro čtení)

Koncepce Environment hodnot

```
// zavedu pro nej registrovanou vychozi hodnotu
private struct SensitiveKey: EnvironmentKey {
    //
    static var defaultValue = false
}
// a promennou v globalnim prostoru
extension EnvironmentValues {
    //
    var isSensitive: Bool {
        //
        get { self[SensitiveKey.self] }
        set { self[SensitiveKey.self] = newValue }
    }
}
//
@main struct MojeAplikace: App {
    //
    var body: some Scene {
        WindowGroup {
            ContentView()
                .environment(\.isSensitive, true)
        }
    }
}
```

@Environment property wrapper

```
//
struct ContentView: View {
    // zavede READ-ONLY property
    @Environment(\.isSensitive) var isSensitive

    //
    var body: some View {
        //
        VStack {
            //
            if isSensitive {
                Text("je ...")
            } else {
                Text("neni")
            }
        }
    }
}
```

Co když chci dynamiku? Typičtější bude ObservableObject.

@Environment

```
//
struct ContentView: View {
    @State private var isSensitive = false

    var body: some View {
        VStack {
            //
            Toggle("Sensitive", isOn: $isSensitive)
            //
            PasswordField(password: "123456")
                .environment(\.isSensitive, isSensitive)
        }
    }
}
```

Zavedu `@State var`, která pingne `body` s každou změnou.

Env proměnná se propaguje do vnořeného View.

@Environment

Environment je informace, kterou mi někdo vkládá do kontextu (View) a já ji volitelně můžu použít.

AppDelegate: Životní cyklus aplikace

```
@main struct MojeAplikace: App {
    // stav zivotniho cyklu aplikace je systemovou
    // env promennou
    @Environment(\.scenePhase) private var scenePhase
    //
    var body: some Scene {
        WindowGroup {
            //
            ContentView()
        } // na ktere odchytavam zmeny
        .onChange(of: scenePhase) { phase in
            //
            print(phase)
        }
    }
}
```

Hodnoty:

```
.active, .inactive, .background
```

AppDelegate: Životní cyklus aplikace

```
@main struct MojeAplikace: App {
    //
    @Environment(\.scenePhase) private var scenePhase

    //
    init() {
        // startuju
        DispatchQueue.main.async {
            // startuju svuj globalni app model
            MyAppModel.shared.startup()
        }
    }

    //
    var body: some Scene {
        // ...
    }
}
```

AppDelegate: Životní cyklus aplikace

```
@main struct MojeAplikace: App {
    //
    @Environment(\.scenePhase) private var scenePhase
    //
    var body: some Scene {
        WindowGroup {
            //
            ContentView()
        } // na ktere odchytavam zmeny
        .onChange(of: scenePhase) { phase in
            //
            if phase == .background {
                // jdu na pozadi
            }
        }
    }
}
```

AppDelegate: Životní cyklus aplikace

Lze použít i UIKitovský UIApplicationDelegate! :)

```
class AppDelegate: NSObject, UIApplicationDelegate {
    //
    func application(_ application: UIApplication,
                     didFinishLaunchingWithOptions launchOptions: ...) -> Bool

    {
        return true
    }
}

@main struct ColorsApp: App {
    @UIApplicationDelegateAdaptor(AppDelegate.self) var
    delegate

    //
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

View: stavové události

Podobně lze na každém View.

```
//
struct SomeMyView: View {
    //
    var body: some View {
        //
        Text("hello world app")
        .onAppear {
            //
            print("View startuje")
        }
        .onDisappear {
            //
            print("View konci")
        }
    }
}
```

Demo: předání dat do View

Občas se toto hodí...mimo rámec @State/@Binding/...

```
// nějaký datový objekt (model)
class Record {
    //
    var name: String = "dd"
}
// editace řetězce name nad Record
struct RecordEditor: View {
    // vnitřní @State pro editování hodnoty
    @State var nameEdit: String = ""
    // na tomto objektu
    let record: Record
    //
    var body: some View {
        // edituju do lokální @State
        TextField("...", text: $nameEdit)
            .onAppear { nameEdit = record.name }
            .onDisappear { record.name = nameEdit }
    }
}
```

Demo: předání dat do View

```
struct aRecordEditor: View {
    // vnitřní @State pro editování hodnoty
    @State var nameEdit: String
    // na tomto objektu
    let record: Record
    // explicitni initializator
    init(_ r: Record) {
        //
        self.record = r;
        //nameEdit = r.name
        _nameEdit = State(initialValue: r.name)
    }
    //
    var body: some View {
        // edituju do lokální @State
        TextField("...", text: $nameEdit)
            .onDisappear { record.name = nameEdit }
    }
}
```

Dynamika "view controllerů"

- modální prezentace (sheet)
- malé informační okénko (action sheet)
- navigation view

Už se tomu neříká "view controller", ale stále jsem nenašel lepší název :)

Fakticky se s každou změnou VM (@State, ...) volá rekonstrukce celého View od App::WindowGroup.

Modální prezentace, sheet

```
//
struct ContentView: View {
    //
    @State var zaznamy: [String] = []
    @State var modalIsON = false
    @State var novyNazev: String = ""
    //
    var body: some View {
        //
        VStack {
            Button("Add") { modalIsON = true; }
            List(zaznamy, id: \.self) { i in Text(i) }
        }.sheet(isPresented: $modalIsON,
                onDismiss: { zaznamy.append(novyNazev)} )
        {
            //
            NewZaznamView(ttt: $novyNazev, modalIsOn: $modalIsON)
        }
    }
}
```

Alternativně: `fullScreenCover(...)`

Modální prezentace, sheet

```
struct NewZaznamView: View {
    //
    @Binding var ttt: String
    @Binding var modalIsOn: Bool
    //
    var body: some View {
        //
        VStack {
            //
            TextField("Zadej", text: $ttt)
            Button("OK") { modalIsOn = false }
        }
    }
}
```

Modální prezentace, sheet

```
struct ContentView: View {
    //
    @State var zaznamy: [String] = []
    @State var modalIsON = false
    @State var novyNazev: String = ""
    //
    var body: some View {
        //
        VStack {
            Button("Add") { modalIsON = true; }
            List(zaznamy, id: \.self) { i in Text(i) }
        }.sheet(isPresented: $modalIsON,
                onDismiss: { zaznamy.append(novyNazev)} )
        {
            //
            VStack {
                //
                TextField("Zadej", text: $novyNazev)
                Button("OK") { modalIsON = false }
            }
        }
    }
}
```

Presentation Mode

`NewZaznamView` může být prezentován modálně i NavView.

```
//
struct NewZaznamView: View {
    //
    @Binding var ttt: String

    // systemova promenna pro ovladani UI
    @Environment(\.presentationMode) var presentationMode

    //
    var body: some View {
        //
        VStack {
            //
            TextField("Zadej", text: $ttt)

            Button("OK") {
                presentationMode.wrappedValue.dismiss()
            }
        }
    }
}
```

Multi-sheet (...ing)

```
//
enum ContentSheetKind: String, CaseIterable {
    case newCosi = "Nove"
    case jineCosi = "Jine"
    case atakdal = "Atakdal"
}
```

- Zavedu si kódy (case) pro různé akce.
- `CaseIterable` lze iterovat přes případy.

Multi-sheet (...ing)

```
struct ContentView: View {
    //
    @State var modalIsON = false
    @State var sheetKind: ContentSheetKind = .newCosi
    //
    var body: some View {
        //
        VStack {
            Button("Add") { modalIsON = true; }
            Picker("Zvolsi", selection: $sheetKind) {
                ForEach(ContentSheetKind.allCases, id: \.self) {
                    i in Text(i.rawValue).tag(!!!)
                }
            }
        }
        .sheet(isPresented: $modalIsON) {
            switch sheetKind {
            case .newCosi:
                Text("Udelej cosi")
            case .jineCosi:
                Button("Udelej cosi") {}
            case .atakdal:
                Text("Atkdal")
            }
        }
    }
}
```

Action Sheet

Označeno jako "deprecated" v budoucnosti.

```
//
struct ContentView: View {
    //
    @State var modalIsON = false
    //
    var body: some View {
        //
        VStack {
            //
            Text("neco neco neco")
            //
            Toggle("akce...", isOn: $modalIsON)
        }.confirmationDialog("Are you sure?", isPresented: $modalIsON) {
            //
            Button("Delete", role: .destructive) {}
            Button("Cancel it..", role: .cancel) {}
        }
    }
}
```

Action Sheet

Ukázka **tuplovaného trailing-closure**.

```
struct ContentView: View {
    //
    @State var modalIsON = false
    //
    var body: some View {
        //
        VStack {
            //
            Text("neco neco neco")
            //
            Toggle("akce...", isOn: $modalIsON)
        }.confirmationDialog("Are you sure?", isPresented: $modalIsON) {
            //
            Button("Delete", role: .destructive) {}
            Button("Cancel it..", role: .cancel) {}
        } message: {
            //
            Text("Fakt si to rozmysli...")
        }
    }
}
```

Action Sheet

```
struct ContentView: View {
    //
    @State var modalIsON = false
    //
    var body: some View {
        //
        VStack {
            //
            Text("neco neco neco")
            //
            Toggle("akce...", isOn: $modalIsON)
        }.alert("Neco se rozbilo", isPresented: $modalIsON) {
            //
        }
    }
}
```

Poznámka: nekonečný List

`List` normálně chce iterovat přes celý obsah.

`LazyVStack` dynamicky vyhodnocuje view.

```
struct ContentView: View {
    //
    var body: some View {
        //
        ScrollView {
            //
            LazyVStack {
                //
                ForEach(0...10000000000, id: \.self) { i in
                    //
                    Text("\(i)")
                }
            }
        }
    }
}
```

Navigation View

- `NavigationView {}`
- `NavigationLink`
- .navigationTitle("neco...")
- ... další nastavení (lišta, titulek, styl)
- lišta na tlačítka `.toolbar { view... }`

Styly:

- DoubleColumnNavigationViewStyle()

Navigation

```
//
struct NecoLevels: View {
    //
    let level: Int

    // systemova promenna pro ovladani UI
    @Environment(\.presentationMode) var presentationMode

    //
    var body: some View {
        //
        List {
            Text("Hello, hloubka \(level)")
            //
            NavigationLink(destination: NecoLevels(level: level+1)) {
                //
                Text("Jdi dal")
            }
            //
            Button("back") { presentationMode.wrappedValue.dismiss() }
        }
    }
}
```

NavigationLink

```
struct ContentView: View {
    //
    var body: some View {
        //
        NavigationView {
            //
            NecoLevels(level: 0)
        }
    }
}
```

NavigationLink

Struktura:

- Label view - jak má "odkaz" vypadat v tomto View
- Destination view - jak má vypadat cílový View

Volitelně:

- isActive: Binding<Bool>
- selection: Binding<T?>

NavigationLink je fakticky **Button** provádějící přechod na nový View.

NavigationLink

```
struct Klasika: View {
    //
    var body: some View {
        //
        NavigationView {
            //
            List(["a", "b", "c"], id: \.self) { t in
                // tlačítko na další obrazovku
                NavigationLink(destination: ...) {
                    // podoba tlačítka
                    Text(t)
                }
            }
        }
    }
}
```

Každý řádek tabulky je tady de fakto tlačítko...

NavigationLink::isActive

```
//
struct ContentView: View {
    //
    @State var detActive = false
    //
    var body: some View {
        //
        NavigationView {
            //
            List {
                //
                Text("hello")

                // link se PROVEDE, pokud detActive==true
                NavigationLink(
                    destination: Text("detail"),
                    isActive: $detActive)
                {
                    // provede detActive := true
                    Text("Tlacitko na detail")
                }

                // ...
                Button("Detail") { detActive = true }
            }
```

Multi-pohled (TabView)

```
struct ContentView: View {
    //
    @State var seleksn: Pohledy?
    //
    var body: some View {
        //
        VStack {
            //
            Picker("Vyber", selection: $seleksn) {
                //
                ForEach(Pohledy.allCases, id: \.self) {
                    //
                    i in Text("\(i.rawValue)").tag(i as Pohledy?)
                }
                //
            }.pickerStyle(SegmentedPickerStyle())

            //
            if let _p = seleksn {
                //
                switch _p {
                case .main:
                    Text("main")
                case .jiny:
                    Text("Jiny")
                case .settings:
                    Text("ddjd")
                }
            }
        }
    }
}
```

Multi-pohled (TabView)

Interně si smím svoje `var body: some View` strukturovat.

```
//
struct ContentView: View {
    //
    @State var selekšn: Pohledy?
    //
    var selectedPohled: some View {
        // tim tomu davam zastřešující datový typ
        Group {
            //
            if let _p = selekšn {
                //
                switch _p {
                case .main:
                    Text("main")
                case .jiny:
                    Text("Jiny")
                case .settings:
                    Text("ddjd")
                }
            } else { EmptyView() }
        }
    }
    ...
```

Multi-pohled (TabView)

Picker - view mnoha tváří.

```
//
struct ContentView: View {
    ....
    var body: some View {
        //
        VStack {
            // šablonový picker => selection: T
            Picker("Vyber", selection: $selekšn) {
                // generuj položky výběru
                ForEach(Pohledy.allCases, id: \.self) { i in
                    // POZOR: datový typ .tag(...)
                    Text("\(i.rawValue)").tag(i as Pohledy?)
                }
                //
            }.pickerStyle(SegmentedPickerStyle())

            // volání funkce...
            selectedPohled
        }
    }
}
```

Případová studie

Aplikace pro správu záznamů XY.

- přidat, odebrat, editovat záznam.
- záznam - implementace jako struct/class.

Přidání nového prvku:

- prosté přidání
- prosté přidání + skok do editace
- editace dočasného záznamu, pak přidání

Studie: struct

`Identifiable` je to pro účely `List/ForEach`. Lze řešit alternativně (`id: Hashable`).

```
//
struct Record: Identifiable {
    // pro ucely List/ForEach
    let id = UUID()

    // jsou var, editovatelnost
    var name: String
    var age: Int
}
```

Studie: struct

```
//
struct ContentView: View {
    // model/viewModel
    @State var lrecords: [Record] = []
    //
    func addNew() {
        //
        lrecords.append(Record(name: "pepa", age: 10))
    }
    //
    var body: some View {
        //
        NavigationView {
            List(lrecords) { l in
                //
                Text(l.name)
            }
            .toolbar {
                // + tlačítko
                HStack { Button(action: addNew) { Image(systemName: "plus")}}
            }
        }
    }
}
```

EditButton - editační mód tabulky

```
struct ContentView: View {
    // model/viewModel
    @State var lrecords: [Record] = []
    //
    var body: some View {
        //
        NavigationView {
            List {
                ForEach(lrecords) { l in
                    //
                    Text(l.name)
                }.onDelete { idxs in
                    // smazání položky v Modelu
                    lrecords.remove(atOffsets: idxs)
                }
            }
            //
            .toolbar {
                // presentationMode
                HStack { EditButton() }
            }
        }
    }
}
```

Detail View

```
struct RecordDetail: View {
    // detail na tomto zaznamu
    let record: Record
    //
    var body: some View {
        //
        Form {
            Section(header: Text("Name")) {
                Text(record.name)
            }
            //
            Section(header: Text("Age")) {
                Text("\(record.age)")
            }
        }
    }
}
```

List view

```
@State var lrecords: [Record] = []
...
List(lrecords) { l in
    //
    NavigationLink(destination: RecordDetail(record: l)) {
        //
        Text(l.name)
    }
}
```

- RecordDetail je `let` nad `Record`
- Detail je tedy jenom pro čtení

Chceme editovatelný detail, tj

- z `RecordDetail` zapisovat do `lrecords`.
- aktualizovat pohled na celkovou tabulku

Editovatelný prvek pole `[Record]`

Jak/možnosti/způsoby:

- Editovat v `RecordDetail`, pak callback do `ContentView`. Najít index s příslušným `.id` a ten přepsat.
- Předávat `Binding<Record>`. Jak to funguje?
- Jak se situace změní, když `Record` bude `class`?

Edit

```
struct RecordDetailClbk: View {
    // detail na tomto zaznamu
    let record: Record
    let action: (Record)->()
    @State var rName: String = ""
    //
    var body: some View {
        //
        Form {
            Section(header: Text("Name")) {
                //
                TextField("", text: $rName)
            }
            .onAppear { rName = record.name; }
            .onDisappear {
                //
                action(Record(id: record.id, name: rName,
                              age: record.age))
            }
        }
    }
}
```

Zpětný zápis do `lrecords`

Ze strany `ContentView` nechť se `action` napojí na:

```
...
@State var lrecords: [Record] = []
...
func rewr(_ r: Record) {
    //
    if let _i = lrecords.firstIndex(where: { $0.id == r.id }) {
        //
        lrecords[_i] = r
    }
}
```

Detail: `Binding<Record>`

```
struct RecordDetail: View {
    // detail na tomto zaznamu
    @Binding var record: Record
    //
    var body: some View {
        //
        TextField("", text: $record.name)
    }
}
```

Iteruju přes `Binding<Record>`. Novinka ve SwiftUI.

```
List($lrecords) { $l in
    //
    NavigationLink(destination: RecordDetail(record: $l)) {
        //
        Text(l.name)
    }
}
```

Detail: `Binding<Record>`

Old-school style :)

```
List {
    // iteruju pres INDEXy !!!
    ForEach(0..<lrecords.count, id: \.self) { i in
        // Binding na i-ty prvek pole read-write
        let _p = Binding<Record>(
                    get: { lrecords[i] },
                    set: { lrecords[i] = $0 })

        // ...
        NavigationLink(destination: RecordDetail(record: _p)) {
            //
            Text(lrecords[i].name)
        }
    }
}
```

Přechod na `class Record`

`struct Record` -> `class Record`

Proč bych měl chtít datový záznam typu objekt:

- v CoreData to tak je (OO-DB engine)
- víc pohledů na model aplikace (víc TabView-s)
- filtrování, uspořádávání dat

Motivace: Chceme automatizovat (tj uskutečnit) přenos dat a změn dat (událostí) do Views.

- v UIKitu to šlo nějak zbastlit
- ve SwiftUI se to dá udělat pouze řádným způsobem :)

Přechod na `class Record`

`struct Record` -> `class Record`

- Přenos hodnoty na Detail-view a zpátky mohl vést k dojmu, že se vše zlepší přechodem na ref hodnoty.
- Je to naprosto naopak.
- SwiftUI je struct-friendly a object-enemy :)

Především je třeba začít řádně modelovat model/viewmodel ;)

Externí model - ObservableObject

- `ObservableObject` - protokol pro `class` typu Model.
- `StateObject` - vlastnictví Model objektu.
- `ObservedObject` - referencování Model objektu.
- `@Published` - publikovaný atribut Observable objektu.

Rozdíl bude v šíření údálostí o změnách:

- `@State` -> `@Binding`
- zápis do binding se propaguje do state-var (to je smysl)
- případná modifikace state-var se NEpropaguje do binding.
- šíření události se děje znovu volání `var body`

@State, @Binding revisited

```
struct DetView: View {
    @Binding var obsah: String
    //
    var body: some View {
        //
        TextField("", text: $obsah)
    }
}
//
struct Main: View {
    // modifikace obsahMain se dale siri pres View
    @State var obsahMain: String
    //
    var body: some View {
        DetView($obsahMain)
    }
}
```

Case Study: Model

```
// už nedělám lokální @State, ale globální MODEL
class RecordModel: ObservableObject {
    // neco jako @State, ale vyrazne MOCNEJSI :)
    @Published var lrecords: [Record] = []

    //
    func addNew() {
        //
        lrecords.append(Record(id: UUID(),
            name: "pepa", age: 10))
    }
}
```

Data aplikace (Model/ViewModel) jsou zde.

```
struct ContentView: View {
    // VLASTNICTVI
    @StateObject var model = RecordModel()
```

Case Study: sdílený Model

Víc "obrazovek" nad `lrecords`, pak:

```
extension RecordModel {
    // singleton – definuje i vlastnictvi
    static let shared = RecordModel()
}
```

```
struct Obr1: View {
    // mam referenci, obdrzim udalosti o zmenach
    @ObservedObject var model = RecordModel.shared
}
struct Obr2: View {
    // mam referenci, obdrzim udalosti o zmenach
    @ObservedObject var model = RecordModel.shared
}
```

Case Study: sdílený Model

Víc "obrazovek" nad `lrecords`, pak:

```
struct ContentView: View {
    // = RecordModel()
    @StateObject var model = RecordModel.shared

    //
    var body: some View {
        Obr1(model: model)
    }
}
```

```
struct Obr1: View {
    // mam referenci, obdrzim udalosti o zmenach
    @ObservedObject var model: RecordModel
}
```

Case Study: sdílený Model

Víc "obrazovek" nad `lrecords`, pak:

```
struct ContentView: View {
    //
    var body: some View {
        //
        TabView {
            Obr1(model: ...)
            Obr2(model: ...)
        }
    }
}
```

Co je `ObservableObject`

Je to objekt (instance třídy, class protocol) agregující události z `@Published` atributů.

```
class MujModel: ObservableObject {
    @Published var jmeno = "Lojza"
    @Published var citac: Int = 0

    //
    init() {
        //...
    }
}
```

Později uvidíme, že je to Publisher/Subject událostí.

Někdo se stane observerem ObservableObjectu.

`@Published` atribut

Knihovna **Combine**. Je to `@propertyWrapper` obsahující `Publisher`.

```
class RecordModel: ObservableObject {
    // obsahuje Publisher<[Record],Never>
    @Published var lrecords: [Record] = []
}
```

Lze zachytit událost změny `@Published` hodnoty.

```
.onReceive(model.$lrecords) { _ in }
```

Publisher neformálně

Publisher je objekt, který spravuje kolekci přihlášených observerů (subscribers).

- Rozesílá událost o změně své hodnoty.
- Lze se tedy na něj napojit a odebírat události.

`ObservableObject` je subscriber svých `@Published` atributů.

- je to agregátor událostí z Published atributů
- sám obsahuje systémový publisher `objectWillChange`
- `objectWillChange.send()`

ObservableObject:v1

```
class MyModel: ObservableObject {
    // ne-published atribut
    @Published var cosi: String
}
```

```
struct Cosi: View {
    @StateObject var mm = MyModel()
    var body: some View {
        // sem to pingne
        Text(mm.cosi)
    }
}
```

ObservableObject:v2

```
struct Cosi: View {
    //
    let mm = MyModel()
    @State var necoJineho: String = "dd"

    //
    var body: some View {
        // sem to pingne
        VStack {
            Text(mm.cosi); Text(necoJineho)
        }
        // dostanu zpravu o udalosti
        .onReceive(mm.cosi) {
            // generuju zpravu pro sebe
            necoJineho = "..."
        }
    }
}
```

ObservableObject:v3

```
class MyModel: ObservableObject {
    // ne-published atribut
    var cosi: String {
        // odchyceni zmeny
        didSet {
            // globalni publikovani udalosti
            objectWillChange.send()
        }
    }
}
```

```
struct Cosi: View {
    @StateObject var mm = MyModel()
    var body: some View {
        // sem to pingne
        Text(mm.cosi)
    }
}
```

Závěr

Dále budeme pokračovat (přemýšlím):

- SwiftUI-III., Combine
- Paralelismy, async
- CoreData
- Kódování dat, Dokumenty
- AWS Amazon