

# SwiftUI II.: Aplikační základy

Programování zařízení Apple (IZA)

Martin Hrubý, FIT VUT, 2023/24

# Úvod

- Aplikační infrastruktura - AppDelegate.
- Obvyklé konstrukce v aplikacích.

Napojení Modelu (životní cyklus Modelu):

- Bindable Object.
- Observable Object - globální Model v aplikaci.
- Něžný úvod do Combine.

Obvyklé konstrukce - seznam záznamů, editace/zobrazení detailu.

# Aplikační infrastruktura

# AppDelegate

Opět vidíme to `some` .

```
@main
struct MojeAplikace: App {
    var body: some Scene {
        // init(content: ...)
        WindowGroup {
            // Instanciacce 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.

Časté jsou záznamy typu `@environment`.

```
@main
struct MojeAplikace: App {
    // drzet terminologicky veci "u sebe"
    static let mujHlavniDatovyObjekt = ToJeOn()

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

# AppDelegate - globální data

Singleton data/process model.

Připomínka: životní cyklus `static` dat.

```
class MyAppGlobalModel {  
    //  
    static let shared = MyAppGlobalModel()  
  
    //  
    init() {  
        // nashartuj procesy  
        // otevri sitova/DP spojeni  
        // apod  
    }  
}
```

# AppDelegate - globální data

Singleton data/process model.

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

**Koncepce environment**

# 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 vidáme \.self  
p[keyPath: \.self] == p
```

Typicky u List:

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

# Demo

```
struct AttrView<Root, Value>: View where Value: CustomStringConvertible {  
  //  
  let record: Root  
  let keyPath: KeyPath<Root, Value>  
  //  
  var body: some View {  
    //  
    Text(record[keyPath: keyPath].description)  
  }  
}  
  
struct Ukazka: View {  
  //  
  let a = Datovka()  
  //  
  var body: some View {  
    //  
    AttrView(record: a, keyPath: \.jmeno)  
  }  
}
```

# 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.

Systemové události (je jich hodně) se implementují:

- jako @Environment hodnoty
- zprávami do AppDelegate
- zprávami přes NotificationCenter

# 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 odchyttavam 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 svůj 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 odchyttavam zmeny
        .onChange(of: scenePhase) { phase in
            //
            if phase == .background {
                // jdu na pozadi
                MyAppModel.shared.saveData()
            }
        }
    }
}
```

# 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 rámeček @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
  // explicitní inicializátor
  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"

# Dynamika "view controllerů"

Způsoby přechodu z jednoho pohledu na jiný:

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

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

# Dynamika "view controllerů"

Typicky potřebuju:

- iniciovat přechod na nový pohled
- předat data do něj
- definovat způsob návratu zpátky
- ... a převzetí dat

# 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 `NavigationView`.

```
//
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...1000000000, 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() }
        }
    }
}
```

# NavLink

```
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...

# TabView

```
enum Zalozka {
    case jedna
    case dva
}

struct Aplikace: View {
    //
    @State var selectedTab = Zalozka.jedna
    //
    var body: some View {
        TabView(selection: $selectedTab) {
            PrvniPohled().tag(.jedna).tabItem { ... }
            DruhyPohled().tag(.dva).tabItem { ... }
        }
    }
}
```

# 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
    }
}
```

# Napojení Modelu

- `@State` / `@Binding` - VM pocházející z View. Neměl by překročit rámec View.
- Chceme Model na globální aplikační úrovni.
- Nad Modelem chceme vykonávat procesy.

# Typizovaný model

- Příklad 1:1 (Model-object versus View)
- Příklad 1:N
  - Kolekce záznamů
  - Edituji kolekci (insert, delete, update)
  - Edituji záznam (atributy)

# Připomenutí - struct

```
struct Record { var name: String }
```

```
struct Detail: View {  
    @Binding var record: Record  
    //  
    var body: some View {  
        TextField(..., text: $record.name)  
    }  
}
```

```
struct Main: View {  
    @State var record = Record(name: "asdf")  
    //  
    var body: some View {  
        Detail(record: $record)  
    }  
}
```

# Observable AND ObservableObject

Implementace Modelu mimo rámeček View (@State).

Knihovna Combine a `ObservableObject` .

Revize do makra (!!!) `@Observable` .

# Revize - struct -> ObservableObject

```
class Record: ObservableObject {  
    @Published var name: String  
}
```

```
struct Detail: View {  
    @ObservedObject var record: Record  
    //  
    var body: some View {  
        TextField(..., text: $record.name)  
    }  
}
```

```
struct Main: View {  
    @StateObject var record = Record(name: "asdf")  
    //  
    var body: some View {  
        Detail(record: record)  
    }  
}
```

# Observable makro

- datový objekt (třída) - @Observable
- držení reference na objekt - @Bindable
- vlastnictví reference - @State
  - pokud je životní cyklus objektu totožný s daným View

Observable (zatím) vnímejme jako

- provedení sémantiky `struct` do `class`,
- možnost referencovat datový záznam (objekt).

Pokryje všechny "obyčejné" potřeby Modelu.

# Observable - @State vlastnictví

```
@Observable class Record { var name: String }
```

```
struct Detail: View {  
    @Bindable var record: Record  
    //  
    var body: some View {  
        TextField(..., text: $record.name)  
    }  
}
```

```
struct Main: View {  
    @State var record = Record(name: "asdf")  
    //  
    var body: some View {  
        Detail(record: record)  
    }  
}
```

# Observable - jiné vlastnictví

```
@Observable class Record {  
    var name: String = ""  
    static let shared = Record()  
}
```

```
struct Detail: View {  
    @Bindable var record = Record.shared  
    //  
    var body: some View {  
        TextField(..., text: $record.name)  
    }  
}
```

# Observable - pole Observable objektů

```
@State var pole: [Record] = []
```

```
@Observable class ListOfRecords {  
    var list: [Record] = []  
}
```

```
struct ListView: View {  
    @State var list = ListOfRecords()  
}
```

```
struct ListView: View {  
    @Bindable var list = ListOfRecords.nejakyShared  
}
```

# Observable - jiné vlastnictví

```
@Observable class Record {  
    var name: String = ""  
    static let shared = Record()  
}
```

Funguje i tohle. View samotné se stane observerem.

```
struct Detail: View {  
    let record = Record.shared  
    //  
    var body: some View {  
        TextField(..., text: $record.name)  
    }  
}
```

# Observable - kdy ocením

Když chci pracovat s objekty, tj s jejich unikátní existencí v paměti (a referencovat z různých míst).

```
@Observable class Database {  
    var originalVsechObjektu: [Record] = []  
    var filtrovanePodleX: [Record] = []  
    var filtrovanePodleY: [Record] = []  
  
    func update() {  
        //  
        filtrovanePodleX = originalVsechObjektu.filter(...)  
        filtrovanePodleY = originalVsechObjektu.filter(...)  
    }  
}
```

# 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 událostí o změnách:

- `@State` -> `@Binding`
- `@Observable` -> `@Bindable`
- Publisher, Subscriber, Subscription

Základ reaktivního programování (knihovna Combine).

# Case Study: Model

```
//  
struct Record {}  
  
// už nedělám lokální @State, ale globální MODEL  
class RecordModel: ObservableObject {  
    // zdroj dat  
    @Published var lrecords: [Record] = []  
  
    //  
    func addNew() {  
        //  
        lrecords.append(Record(...))  
    }  
}
```

Data aplikace (Model/ViewModel) jsou zde.

# Vlastnictví ObservableObject

Pouze, pokud je životní cyklus `RecordModel` spojen s View.

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

```
struct ContentView: View {  
    // Reference na nějak jinak vlastněnou hodnotu  
    @ObservedObject var model = RecordModel.shared  
}
```

# 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
}
```

# 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 {  
    // 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 {
    // odchyeni 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:

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