

SwiftUI I.: Úvod do SwiftUI

Programování zařízení Apple (IZA)

Martin Hrubý, FIT VUT, 2025/26

Úvod

- SwiftUI ohlášeno na WWDC 2019. Wow-effect.
- Určeno pro iOS 13+, macOS Catalina+.
- Stav vývoje: docela se stabilizuje.

Aktuálně lze vývoj v XCode provádět:

- ve Storyboard stylu,
- ve SwiftUI stylu.

Apple označuje SwiftUI za (vážně míněnou) budoucnost.

SwiftUI je knihovna implementovaná ve Swiftu. **Budeme si ho tam hledat.**

Východiska

Srovnání s UIKit/Storyboard stylem:

- Jiná architektura MVC versus MVVM.
- Ruční programování ViewControllerů versus *deklarativní programování*.
- Zvýšení produktivity práce.
- Narovnání některých stylových "nečistot", které vznikly zavedením Swiftu do UIKitu.

UIKit si probereme zkratkovitě příště - jako "ochutnávku", pro srovnání.

Východiska

Sloučení programování aplikací přes všechny platformy.

- UIKit (iOS), AppKit (macOS), speciální kity pro tvOS, watchOS.
- XCode: aplikace "univerzálního" typu.

S přechodem HW architektury na Apple Silicon (Mac, iOS) máme blíž k binární přenositelnosti aplikací.

Východiska

Historicky všechny frameworky Apple:

- jsou knihovny používané Apple, dané k dispozici,
- středně-úrovňové v abstrakci.
- Apple nestanovuje striktní metodiku programování,
- ... spíš se kouká, jak se s tím vývojáři "popasují".

A je to zejména případ SwiftUI.

Architektura (mobilních) aplikací

- Backend - data, procesy, vnitřní logika aplikace.
- Frontend - "views", UI, ..., ...

U mobilních aplikací nováček typicky začíná přes UI.

- ... toto SwiftUI dokonale ulehčí.

Nutno si vysvětlit rozdíl MVC versus MVVM pro nováčky.

Co je MVC, stručně

- Model, View, Controller. Smalltalk, 70tá léta.

Prakticky:

- View - knihovní objekty. Frontend.
- Model - backend.
- Controller - směska všeho. Místo bujarého bastlení.

Controller:

- je nositelem akce. Sestaví, vlastní a řídí View.
- akce z View (tlačítko) -> akce v Cont -> View.

Co je MVC, hokna

- neustálé předávání dat z Modelu do View.
- opakované programování stejných konstrukcí (prototypy buněk tabulky, ...).

Snaha o automatizaci.

- Key Value Observing (formuláře).
- NS-FetchedResultsController. NSArrayController.

Co je M-V-VM

Presentation model (Martin Fowler) -> MVVM.

- M—Model (**uživatelova data**), V—View, VM—ViewModel.

VM — Explicitní vnitřní stav aplikace/View.

- Drží stav uživatelského rozhraní aplikace.
- Drží data aplikace ve formátu 1:1 pro zobrazení.
- Řekli bychom, je to interní stav všeho, co jsme dřív nazývali Controller+Views.

Kde zmizel Controller? Je tam nějaký kód?

Funkce Controlleru se automatizuje. Deklarativní programování.

Co je M-V-VM

Jasně odděluje Backend-Frontend.

- dynamika aplikace se řídí dynamikou VM.
- View je pouze transkripce VM do viditelné podoby.

Nadsázka:

- Aplikaci ve SwiftUI programují Backend-vývojáři (-> VM).
- ... Frontend-vývojáři obarvují VM do viditelné podoby.
- Obojí se však dá dělat dobře.

(Může být) Naprosto precizně oddělený BackEnd a FrontEnd.

Demo VM -> V

```
class MujVM: ObservableObject {  
    //  
    @Published var selectedFilter: NejakyEnum = ...  
    @Published var originalData: [Record] = ...  
    @Published var tobeDisplayed: [RecordVM]  
  
    //  
    static let shared = MujVM()  
}
```

A nějak implementovaná odezva na změnu `selectedFilter`.

- Deklaruji, že `tobeDisplayed` obsahuje vždy VM záznamů platnou v kontextu `selectedFilter`.

Demo VM -> V (schematicky)

```
struct MujV: View {  
  // zajisti prenos udalosti z VM  
  @ObservedObject vm = MujVM.shared  
  
  // View  
  var body: some View {  
    return (!!!) VStack {  
      //  
      Picker($vm.selectedFilter)  
  
      //  
      List(vm.toBeDisplayed) { itm in  
        RecordRowView(record: itm)  
      }  
    }  
  }  
}
```

Co je M-V-VM

Model (interní *data* aplikace):

- data a procesy.

ViewModel (viditelná *data* aplikace):

- vnitřní stav prvků View - jejich "provozní" atributy stavu.
- data určená pro 1:1 zrcadlení do Views.

View

- transformace VM do Views (UIKit?, UIViews?)
- zpětná vazba do VM

Tok dat $M \rightarrow VM$, $VM \leftrightarrow V$

UIKit:

- Controller - "ruční" překládání dat.
- automaticky: Key-Value-Observing

SwiftUI: máme `@State` / `@Binding`, knihovnu Combine.

- `@Published var cosi: typ`
- `Publisher -> ... -> Subscriber`
- zřetězení asynchronních volání
- reaktivní programování

Příklad TextField

Data(VM) - buffer String editovatelné hodnoty.

Provozní atributy(VM):

- umístění kurzoru, volba klávesnice a její stav
- selekce textu, výběr části pro selekci.
- stav našeptávače, autokorekce.

Různé View zpřístupňují své "provozní atributy" a staví je do role ViewModelu.

ViewModifier -> nastavení parametrů/provozních atributů.

SwiftUI demo: TextField

```
struct MainPage: View {
    // dynamicke atributy (VM)
    @State var obsah: String = "ahoj"
    @State var disableAC = false

    // staticke
    let style = PlainTextFieldStyle()

    //
    var body: some View {
        //
        TextField("Napis neco", $obsah)
            .disableAutocorrection(disableAC)
            .textFieldStyle(style)
    }
}
```

Co je cílem ViewModelu

Odděleně popíšu:

- strukturu UI - `var body: some View {...}`
- a stav prvků té struktury

Tím rozdělím:

- referencovanou část - ViewModel
- statickou část - `body`, která má charakter funkce.

Schematická představa V-VM

Uvažujme kód pro hlavní obrazovku aplikace.

```
// data hlavní obrazovky aplikace
class MainPageVM {
    // ... data
}

// popis struktury uživatelského rozhraní
func mainPage(vm: MainPageVM) -> UI pro MainPage {
    // struktura uživatelského rozhraní
}
```

Je to objekt (referencovatelná data) + funkce.

Proto nemůže být překvapením, že deklarace UI se odehrává ve `struct`. Co je vlastně `struct:View` ?

SwiftUI demo

Struct je implicitně immutable. Co je teda @State ?

```
struct MainPage: View {  
    // toto je class-based (ref) datový obsah  
    @State var obsah: String = "ahoj"  
    //  
    var body: some View {  
        //  
        TextField("Napis neco", $obsah)  
    }  
}
```

Takže:

```
let mainPage = MainPage()  
// kopie:  
// kopie.obsah referencuje mainPage.obsah  
let kopie = mainPage
```

SwiftUI: tohle nikdy nedělejte! :)

Zcela proti smyslu SwiftUI, nicméně demonstruje implementaci.

```
//  
func zmena(str: MainPage) { str.obsah = "prepsano" }  
  
//  
struct MainPage: View {  
    // toto je class-based (ref) datový obsah  
    @State var obsah: String = "ahoj"  
    //  
    var body: some View {  
        //  
        VStack {  
            Text(obsah);  
            //  
            Button(action: { zmena(str: self) } ) {  
                Text("Stiskni me")  
            }  
        }  
    }  
}
```

SwiftUI: vrstvy pohledu

- Naučit se naprogramovat UI s nějakou funkcionalitou.
 - přehledová znalost knihovných views
- Pochopit dynamiku v aplikaci SwiftUI.
 - použití magických propojek `@State` apod
- Koncepce aplikace - VM
- Geometrie Views + View-Modifiers.

Dynamika: šíření událostí skrz Model a ViewModel tak, aby se překreslovalo View.

App: úrovně popisu

```
@main
struct IZA_lec5_2022App: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

Úrovně:

- Aplikace. AppDelegate.
- Scene - typicky jedna scéna.
- Window - typicky jedno okno. (UIWindow, UIScreen)
- View - dynamický obsah okna (UIView).

View, XCode Preview

```
struct ContentView: View {
    var body: some View {
        Text("Hello, world!")
            .padding()
    }
}
// old-school...
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
// nove: makra
#Preview {
    ContentView()
}
```

struct: View

`View` je protokol. `some View`

```
struct MyView: View {  
    //  
    var body: some View {  
        // nejaky obsah  
    }  
}
```

- `MyView` je viditelný obsah, který vyplní nadřazenou plochu.
- tj celou obrazovku
- nebo je to grafická komponenta.

Nedělají se rozdíly mezi instancemi `View`. Všechno jsou `View`.

some ... opaque type

```
protocol MujProtokol {  
    // implementátor určí konkrétní typ MujType  
    associatedtype MujTyp: CustomStringConvertible  
    //  
    var obsah: MujTyp { get }  
}  
//  
struct ImplementaceMujProtokol: MujProtokol {  
    // určení: MujTyp == String  
    let obsah: String  
}  
// kompilátor: co vím s určitostí o MujProtokol?  
func giveItATry(cosi: some MujProtokol) {  
    //  
    print(cosi.obsah.description)  
}
```

struct: View

- elementární views (Text, Textfield, Button, ...)
- kontejnery views (topologie)
 - stack - VStack, HStack, ZStack
 - ScrollView, NavigationView, TabView...
- abstraktní views:
 - EmptyView, Group

Protocol View

```
public protocol View {  
    //  
    associatedtype Body : View  
  
    //  
    @ViewBuilder var body: Self.Body { get }  
}
```

```
struct MyView: View {  
    //  
    var body: some View {  
        // nejaký obsah  
        Text("hello")  
    }  
}
```

struct: View, ViewModel

Referencovaná data + `var body`

```
struct MyPage: View {  
    //  
    @State var nazev: Typ [= initialValue]  
    @Binding var napojeni: Typ  
    //  
    @Environment var env(...)  
    //  
    @StateObject var obj = ...  
    @ObservedObject var obj = ...  
  
    //  
    var body: some View {  
        //  
    }  
}
```

Property wrapper

```
// struct|class
@propertyWrapper struct MW1 {
    // toto je primitivni hodnota
    var wrappedValue: String = ""
    // pripadne nejaka akce
    mutating func necoUdelej() {
        //
        wrappedValue = "hello"
    }
}

//
struct ContentView: View {
    // property wrapper MW1
    @MW1 var muj
    //
    init() {
        // _muj je typu: MW1
        _muj.wrappedValue = "cosi"
        muj = "primo tam"
    }
}
```

Property wrapper

```
// struct|class
@propertyWrapper struct MW2<Element> {
    // toto je primitivni hodnota
    var wrappedValue: Element {
        //
        didSet {
            // nejaka akce -> prekresli
        }
    }
}

struct ContentView: View {
    // MW2.Element == String
    @MW2 var muj: String = ""
    ...
}
```

Magické propojky @cosiKdesi

Budeme mluvit o paměťových prvcích, takže nás zajímá klasika (životní cyklus):

- kdo to vlastní
- jak se to chová
- jak se na to dá dostat z vnějšku (referencovat to).

Budeme odlišovat:

- vlastnictví pocházející ze `struct` -> `@State`
- vlastnictví z objektů -> `@Published`
 - `Observable...`
 - `Bindable...`

Data SwiftUI struktury

`@State` je VM struktury. Jsou to data, která tento View vytváří.

```
struct MojeCosi: View {  
    //  
    @State var jmeno: String = "pepa"  
    @State var login: String  
  
    //  
    var body: some View {  
        //  
        VStack {  
            Text("Jmeno: \(jmeno)")  
            Text("Login: \(login)")  
        }  
    }  
}
```

`MojeCosi` je kvazi-vlastníkem `@State` dat.

@State

Životní cyklus `@State` proměnných řídí SwiftUI.

- hodí se pro dočasná pomocná data

Referencovatelnost:

- prostřednictvím `Binding<Value>`
- ... ovšem výhradně do dalších Views.
- ... předávat je jako referenci do Modelu je perverzní ;)

@Binding - sdílení dat

Obousměrný přístup na nějakou datovou položku.

```
//  
@propertyWrapper struct Binding<Value> {  
    //  
    let _get: ()->(Value)  
    let _set: (Value)->()  
  
    //  
    var wrappedValue: Value {  
        //  
        get { _get() }  
        set { _set(newValue) }  
    }  
}
```

@Binding - sdílení dat

```
func akce(s: Binding<String>) {  
    //  
    print(s.wrappedValue)  
  
    //  
    s.wrappedValue = "cosi"  
}
```

Pokud byste skutečně chtěli někde mimo `struct ...:View` modifikovat její `@State var` .

`Binding<Value>` je get/set rozhraní na někde umístěnou property typu `Value` .

@Binding - sdílení dat

```
struct MojeEdit: View {
    @Binding var buf: String
    var body: some View {
        // return TextField($buf)
        TextField($buf).neco().necoJineho()...
    }
}
//
struct AppMain: View {
    //
    @State jmeno: String = "pepa"
    // jaký je návratový typ funkce body?
    var body: some View {
        // return HStack({...})
        HStack {
            Text("Moje jmeno); Spacer()
            MojeEdit(buf: $jmeno)
        }
    }
}
```

operátor \$

```
@State var moje: String = "cosi kdesi"
```

pak `$moje` je symbol pro `moje.projectedValue` :

```
@propertyWrapper struct State<Value> {  
    // ...  
    var projectedValue: Binding<Value> {  
        Binding<Value>(  
            get: { self.wrappedValue }  
            set: { self.wrappedValue = $0 }  
        )  
    }  
}
```

operátor \$

```
struct Person { var name: String; let age: Int }  
//  
struct EditName: View {  
    //  
    @Binding var name: String  
    //  
    var body: some View {  
        // nestačilo by name? _name?  
        TextField("name", text: $name)  
    }  
}  
//  
struct MyView: View {  
    //  
    @State var person = Person(name: "...", age: 123)  
    //  
    var body: some View {  
        //  
        EditName($person.name)  
    }  
}
```

SwiftUI: komponenty

Textový obsah se zadanými atributy.

```
Text("nejaky text")
```

Textový vstup (+atributy)

```
// $obsah – Binding<String>  
TextField("label", text: $obsah)
```

Tlačítko (množství podob)

```
Button(action: {...}) {  
    // obsah  
}
```

SwiftUI: Text + ViewModifiers

```
struct ContentView: View {  
    //  
    var body: some View {  
        //  
        Text("Ahoj")  
            .font(.largeTitle)  
            .padding()  
            .background(Color.yellow)  
            .cornerRadius(20)  
    }  
}
```



Ahoj

Zobecnění často používaných komponent, ať program ve SwiftUI nevypadá jako HTML :)

SwiftUI: topologie Views

- `HStack {}`
- `VStack {}`
- `ZStack {}`

Druh adjustace (alignment).

```
VStack(alignment: .leading, spacing: 20) {  
    //  
    Text("abcdef")  
    Text("abcdefghijklmnopq")  
}
```

HStack

```
struct ContentView: View {
    @State var jmeno: String = ""
    //
    var body: some View {
        //
        HStack {
            //
            Text("Jmeno"); Spacer()
            TextField("zadej jmeno", text: $jmeno)
        }.padding().background(Color.green)
    }
}
```

Stacks

```
VStack {  
    //  
    HStack {  
        //  
        VStack { Text("hello"); Text("jak"); Text("se mas")}  
        Text("Ujde to").font(.largeTitle)  
    }  
    //  
    Text("Tak jo")  
}
```

hello
jak
se mas

Ujde to

Tak jo

Button

- akce - @escaping closure
- podoba tlačítka

```
struct ContentView: View {
    @State var jmeno: String = ""
    // je/neni mutating???
    func akceNaTlacitko() { jmeno = "pepa" }
    //
    var body: some View {
        //
        VStack {
            //
            Text(jmeno)
            // @escaping closure...
            Button(action: { self.akceNaTlacitko() },
                    label: { Text("Zmackni") })
        }
    }
}
```

Button

```
Button(action: akceNaTlacitko) {  
    Text("Zmackni")  
}
```

```
Button("Zmackni") { akceNaTlacitko() }
```

```
Button(action: akceNaTlacitko) {  
    Text("Zmackni")  
        .padding()  
        .background(Color.red)  
        .foregroundColor(Color.yellow)  
        .cornerRadius(10)  
}
```

View-Modifiers: rozsáhlá referenční knihovna,
znovupoužitelnost.

function -> some View

```
func cerveneVelke(label: String) -> some View {  
    Text(label)  
        .padding()  
        .background(Color.red)  
        .foregroundColor(Color.yellow)  
        .cornerRadius(10)  
}
```

```
Button(action: akceNaTlacitko) {  
    //  
    cerveneVelke("Zmackni")  
}
```

function -> some View

```
func redButton(label: String,  
               blk: @escaping ()->()) -> some View  
{  
    // ... Button je struct – syntax  
    Button(action: blk) {  
        //  
        cerveneVelke(label)  
    }  
}
```

```
struct MojeCosi: View {  
    //  
    var body: some View {  
        // ... je func – syntax  
        redButton("asdf") { print("Hello") }  
    }  
}
```

Vlastní styly (pro Views)

```
struct RedButtonStyle: ButtonStyle {  
    //  
    func makeBody(configuration: Configuration) -> some View {  
        //  
        configuration.label  
            .padding()  
            .background(Color.red)  
            .foregroundColor(Color.yellow)  
            .cornerRadius(10)  
    }  
}
```

```
struct ContentView: View {  
    //  
    var body: some View {  
        //  
        Button("Ahoj") {  
            // akce  
        }.buttonStyle(RedButtonStyle())  
    }  
}
```

Jako dedikovaná struktura

```
struct RedButton: View {  
    //  
    let label: String  
    let action: () -> ()  
    //  
    var body: some View {  
        //  
        Button(action: action) {  
            Text(label).padding()  
                .background(Color.red)  
                .foregroundColor(Color.yellow)  
                .cornerRadius(10)  
        }  
    }  
}
```

Jako šabloný View

```
struct RedButton<Label: View>: View {  
    //  
    let label: Label  
    let action: () -> ()  
  
    //  
    init(action: @escaping () -> Void, @ViewBuilder label: () -> Label) {  
        self.label = label()  
        self.action = action  
    }  
  
    //  
    var body: some View {  
        //  
        label.padding()  
            .background(Color.red)  
            .foregroundColor(Color.yellow)  
            .cornerRadius(10)  
    }  
}
```

```
RedButton(action: {}) { Text("Ahoj") }
```

List

Navazuje na systém UITableView a všeho dalšího (delegate, dataSource, Cell-prototype, ...).

```
List(array, id: \.key) { itm in
    //
    Text(itm.name)
}
```

Foreach:

```
List {
    //
    Text("prvni radek")

    // dalsi radky
    ForEach(array, id: \.key) { itm in ...}
}
```

List

Iterované hodnoty musí být:

- `Identifiable`
- nebo explicitní `id: KeyPath`, vedoucí na `Hashable`

```
struct MojeView: View {  
    //  
    let array = ["Ahoj", "jak", "se", "mas"]  
  
    //  
    var body: some View {  
        //  
        List(array, id: \.self) { itm in  
            Text(itm)  
        }  
    }  
}
```

List je však heterogenní!

Úplně ze všeho nejvíc nejgeniálnější vlastnost `List`.

```
...
List {
    // 1. radek
    Text("Hello")
    // 2. radek
    TextField("cosi", text:$jmeno)
    // 3. radek
    Button("zmackni") { jmeno = "pepa" }
    // 4. radek
    HStack {
        Text("Jmeno"); Spacer(); Text(jmeno)
    }
    // 5. – 105. radek...
    ForEach(0...100, id:\.self) { i in Text("\($i)") }
}
```

View pro strukturování UI

UIKit: kontejnerové ViewControllers.

- UINavigationController/NavigationLink (zásobníkový kontejnerový view).
- TabView.
- sheet (modální prezentace).

UINavigationController/TabView do celkového View vkládají svoje specifické lišty (pro tlačítka, texty, ikonky, ...).

TabView

- různé "obrazovky"
- spodní lišta (ikonka, popisek)

```
// tag hodnoty pro různé stránky v TabView
enum MyAppPages {
    //
    case první, druhá, třetí
}
```

TabView

```
//
struct ContentView: View {
    //
    @State var selPage: MyAppPages = .prvni
    //
    var body: some View {
        //
        TabView(selection: $selPage) {
            //
            PrvniPage().tag(MyAppPages.prvni).tabItem {
                //
                Text("prvni"); Image(systemName: "heart.fill")
            }
            //
            DruhaPage().tag(MyAppPages.druha).tabItem {
                //
                Text("druha"); Image(systemName: "heart.fill")
            }
        }
    }
}
```

NavigationView

- UIKit: UINavigationController
- zásobníkový pohled na "obrazovky"
- push/back
- rootViewController (počáteční obsah)
- Nadpisek obrazovky (title)
- Horní lišta s tlačítky

NavigationView: Detail View

```
//
struct Person {
    //
    let name: String
    let age: Int
}
//
struct DetailView: View {
    //
    let person: Person
    //
    var body: some View {
        //
        VStack {
            //
            Text("Name: \(person.name)")
            Text("Age: \(person.age)")
        }
    }
}
```

NavigationView: List

```
//
struct ContentView: View {
    //
    let array = [Person(name: "Pepa", age: 33),
                 Person(name: "Honza", age: 25)]
    //
    var body: some View {
        //
        NavigationView {
            //
            List(array, id: \Person.name) { itm in
                //
                NavigationLink(destination: DetailView(person: itm))
                {
                    //
                    Text(itm.name)
                }
            }
        }
    }
}
```

NavigationView: lišta

```
//
struct ContentView: View {
    //
    var body: some View {
        //
        NavigationView {
            // ...
            .navigationTitle("Hlavní APP")
            .toolbar {
                //
                HStack {
                    Button(action: {}) {
                        Image(systemName: "plus")
                    }
                }
            }
        }
    }
}
```

SwiftUI: dynamika

Deklarativní programování. Deklaruji UI složené z:

- Textového pole zobrazující `@State var obsah`
- Textového editačního pole `$obsah`

Změna obsah -> šíření události.

```
struct Main: View {
    @State var obsah = "ahoj"
    //
    var body: some View {
        //
        List {
            Text(obsah)
            TextField("napis neco", text: $obsah)
        }
    }
}
```

Události nad @State var

```
struct Main: View {  
    // @propertyWrapper  
    @State var obsah = "ahoj"  
    // není mutating!  
    func akce() { obsah = "neco sem napis" }  
    //  
    var body: some View {  
        // ... predavam Binding<String>  
        ExternalView(obsah: $obsah)  
    }  
}
```

Vždycky budeme chtít, aby změna dat "cvrnkla" do View.

Větvení v body

```
struct Moje: View {  
    //  
    @State var pracuje: Bool = false  
    //  
    var body: some View {  
        //  
        VStack {  
            Toggle("Pracuje to: ", $pracuje)  
  
            //  
            if pracuje == true {  
                Text("jo, jede to...")  
            } else { ... }  
        }  
    }  
}
```

@ViewBuilder: jak to funguje?

SwiftUI je implementováno plně prostředky Swiftu.

- `var body: some View` - je funkce a **má návratový typ!**
- SwiftUI je template knihovna.
- Prvky jsou elementární struktury.
- proces budování hodnoty.

Velmi zjednodušeně: s každou změnou VM se zavolá `var body` a přehodnotí podoba UI.

@ViewBuilder: demo

@ViewBuilder je tzv. *function builder*.

```
struct DoubleIt<Content:View>: View {  
    //  
    let content: ()->(Content)  
  
    //  
    init(@ViewBuilder content: @escaping ()->Content) {  
        //  
        self.content = content  
    }  
  
    //  
    var body: some View {  
        //  
        HStack {  
            content(); content()  
        }.background(.green)  
    }  
}
```

@ViewBuilder: demo

Trailing closure je **všude**.

```
//  
struct ContentView: View {  
    //  
    var body: some View {  
        // konstrukce hodnoty DoubleIt(content: ...)   
        DoubleIt {  
            //  
            Text("hello")  
        }  
    }  
}
```

Derivační strom celého uživatelského View.

Jedná velká super-šablonovitá struktura.

Function (result) Builder

```
@resultBuilder struct Builder {  
    // povinna funkce  
    static func buildBlock(_ partialResults: String...) -> String {  
        //  
        return partialResults.reduce("", +)  
    }  
}
```

`abc()` je funkce. Její tělo je předáno jako argument do `@Builder`, který to transformuje na data.

```
@Builder func abc() -> String {  
    "Method: "  
    "ABC"  
    "(asdf)"  
}
```

Views: úvod do geometrie Views

```
//  
@frozen struct Text: Equatable { ... }  
  
// + extensions
```

View-Modifiers:

```
extension View {  
    //  
    public func padding(...) -> some View {  
        // ...  
    }  
}
```

```
// konstrukce hodnoty Text, poslání zprávy padding  
Text("ahoj").padding()
```

Frame, constraints, ...

Systém automatického výpočtu geometrie UIView z constraints.

Struktury: `bounds` a `frame`. Subviews.

- `Text("...").viewModifiers(font, styl, apod) -> Frame`
- `.padding(...)`
- `.frame(...)`
- `.clipped()`, `.clipShape(...)`

GeometryReader

```
GeometryReader { geometry in
  HStack(spacing: 0) {
    Text("Left")
      .font(.largeTitle)
      .foregroundColor(.black)
      .frame(width: geometry.size.width * 0.33)
      .background(Color.yellow)
    Text("Right")
      .font(.largeTitle)
      .foregroundColor(.black)
      .frame(width: geometry.size.width * 0.67)
      .background(Color.orange)
  }
}
.frame(height: 50)
```

Závěr

- Úvodní seznamovací kolo se SwiftUI.
- Více dynamiky (Combine).
- Více praxe.