iOS alapú szoftverfejlesztés - Labor 07

A labor témája

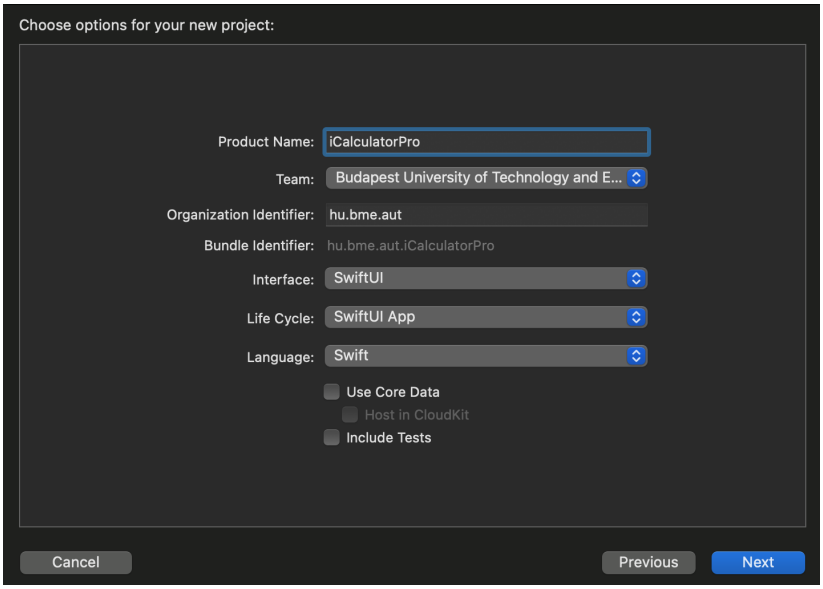
A labor célja az `Adaptive Layout` használatatának a gyakorlása egy névnapos alkalmazás kezdeti képernyőin keresztül.

Ismerkedés a SwiftUI-jal

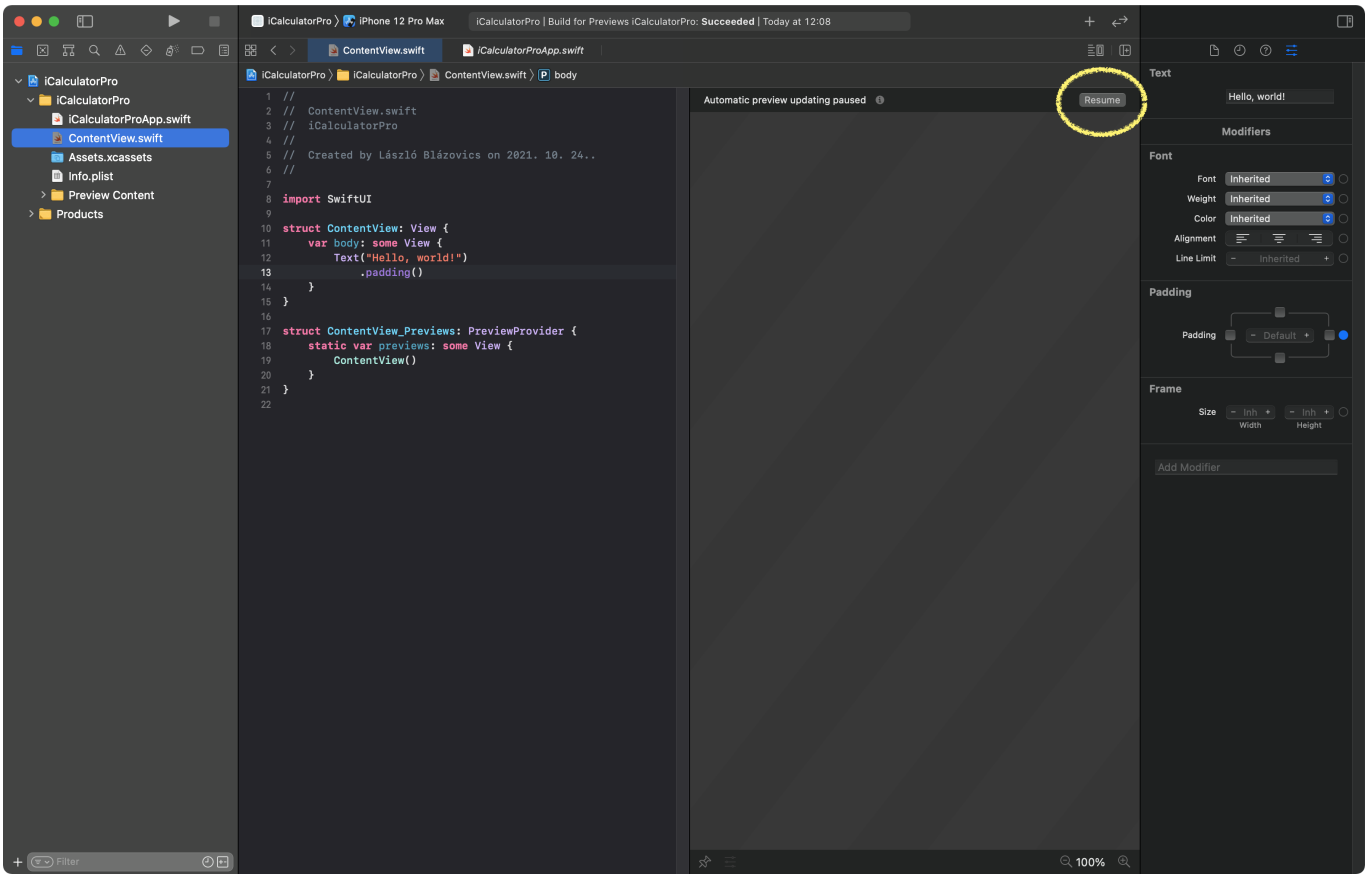
iCalculatorPro

> Hozzunk létre egy `Single View App`ot **iCalculatorPro** névvel a `Developer` könyvtárba!

> Az `Interface` legyen `SwiftUI`, a `Life Cycle` pedig `SwiftUI App`

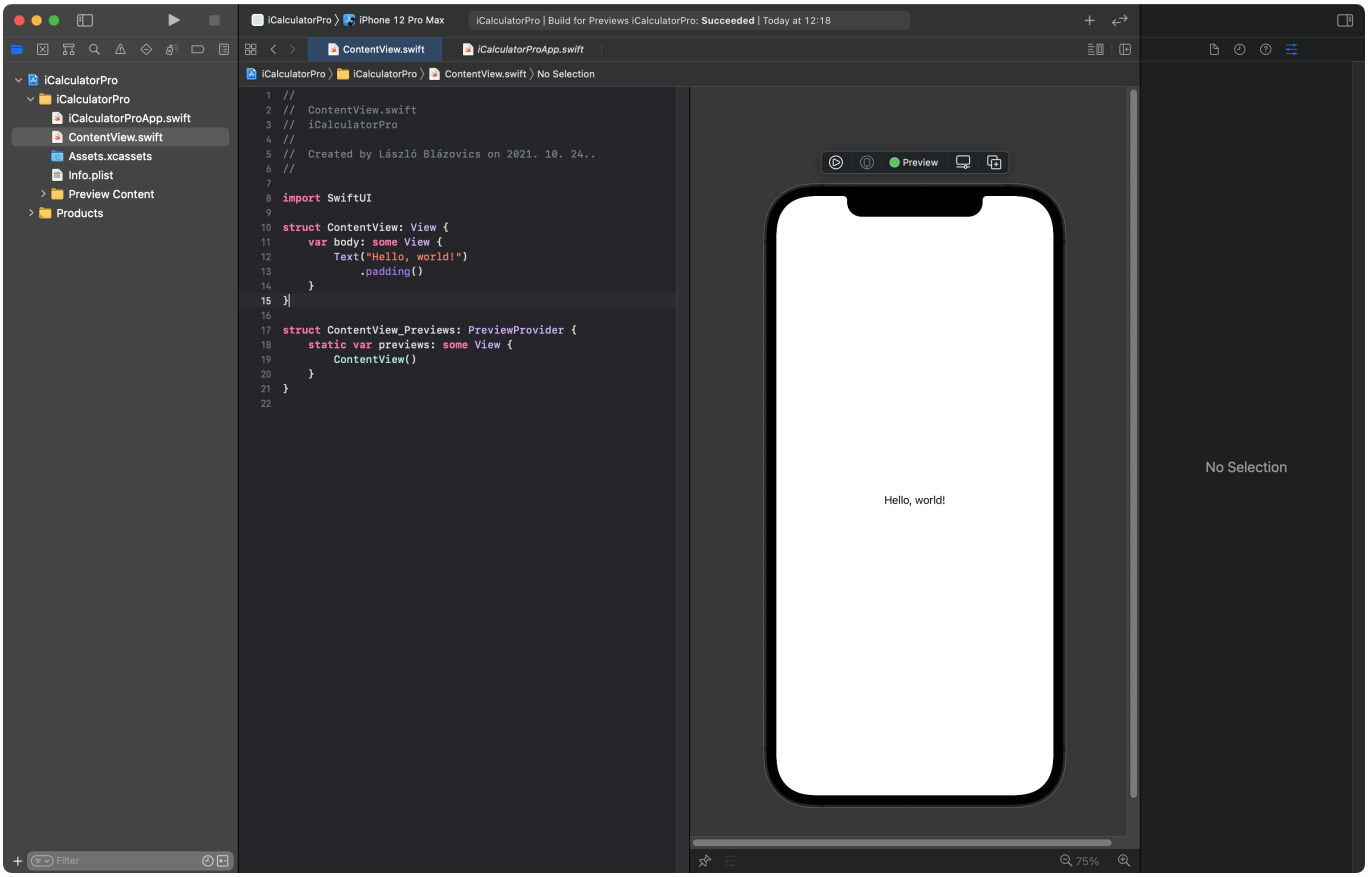


Miután létrejött a projekt a következő látvány fogad minket.



A SwiftUI már nem épít a Storyboard-okra, de a vizuális szerkesztés fontos része maradt a folyamatnak. Ehhez a forráskód nézet mellett automatikusan megnyitott `Preview` nézet lesz a segítségünkre. A `Preview` ablakot UIKit esetében is használhatjuk, de a `Storyboard`-ok miatt ritkán van rá szükség.

> Az automata preview ki van kapcsolva, a `Resume` gombbal kapcsoljuk be!



Látható, hogy megjelent a nézetünk tartlama a kijelzőn. Ha nem rontunk el semmit, ez folyamatosan frissülni fog, ahogy a forráskódot editáljuk.

> Írjuk át a `Text` szövegét `"iCalculator Pro"`-ra!

> Adjunk hozzá egy `modifier`-t, hogy legyen nagyobb a szöveg mérete!

```
Text("iCalculator Pro")
    .font(.largeTitle)
    .padding()
```

> Az első számológépes példához hasonlóan adjunk hozzá a nézethez egy `TextField`-et a már rajta lévő `Text` alá.

```
var body: some View {
    Text("iCalculator Pro")
        .font(.largeTitle)
        .padding()
    TextField("1. operandus", text: nil)
}
```

Több hibát fogunk kapni:

1. A `body` nem egy `View`-val fog visszatérni

> Ágyazzuk be a `Text`-et és a `TextField`-et egy `VStack`-be!

2. A `TextField` önmagában nem tárolja a beírt szöveget, az azt tároló objektumot egy Binding segítségével nekünk kell biztosítani.

> Hozzunk létre a `ContentView`-n belül egy `property`-t `operand1` névvel, ami egy `@State` porperty lesz

```
struct ContentView: View {

    @State private var operand1: String
```

> Most már beállíthatjuk a `TextField` `text` paraméterét

```
TextField("1. operandus", text: $operand1)
```

Ismét hibát kapunk, mert az új `operand1` `property`-nk nincs inicializálva. Ezt ráadásul nem is a `ContentView`-n mutatja az Xcode, hanem a `ContentView_Previews` struct-on, ami azért felelős, hogy a `Preview` működjön és lássuk a tartalmat.

> Adjunk kezdeti értéket az `operand1`-nek.

```
    @State private var operand1: String = ""
```

Összetettebb estekben nyilván inicializálót érdemes készíteni, itt most ez is megteszi.

> Mivel a `Preview` leállt, újra kell indítani, hogy lássuk az eredményt. kattintsunk a `Try Again` gombra!

Látható, hogy már ott is van a TextView-nk is. Ja, hogy nem látszik...

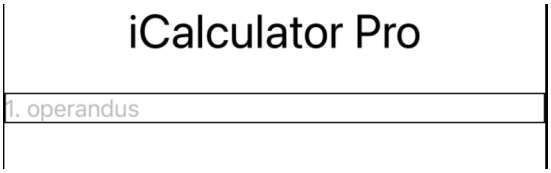
Mivel a `TextField`-ünk semmiféle stílussal nem rendelkezik, állítsunk be neki valami használható megjelenést

> Először is állítsunk be neki egy megfelelő keretet a megfelelő modifier segítségével

Ez iOS 14-ig `.border()` modifier-rel lehetséges.

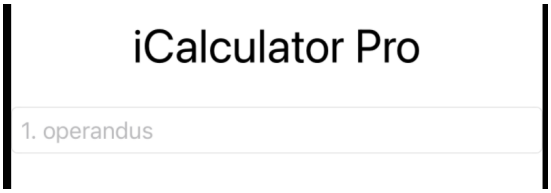
```
    TextField("w", text: $operand1)
        .border(.black)
```

> Frissítsük a Preview-t!



Még mindig nem az igazi. iOS 15-től már használható a `textFieldStyle` modifier-t is, hogy előre definiált stílusa legyen a `TextField`-nek. Állítsuk be ezt.

```
    TextField("w", text: $operand1)
        .textFieldStyle(.roundedBorder)
```

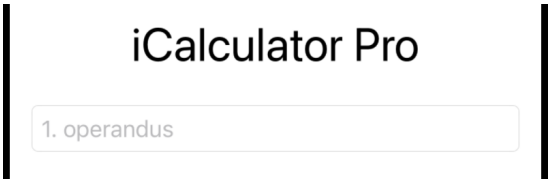


Egy fokkal szebb, de nagyon rálóg a képernyő szélére.

> Használjuk a `.padding` modifier-t, hogy víszintesen legyen némi térköz a képernyő szélei és a `TextField` között.

```
    TextField("w", text: $operand1)
        .textFieldStyle(.roundedBorder)
        .padding(.horizontal)
```

Most már jobban is néz ki.



> Hozzunk létre egy új és egy másik `@State property`-t `operand2` névvel.

```
    @State private var operand1: String = ""
    @State private var operand2: String = ""
```

> Adjunk hozzá a nézetünkhöz egy sima `Text`-et egy második `TextView`-t *2. operandus* placeholderrel.

```
    Text("+")

    TextField("2. operandus", text: $operand2)
        .textFieldStyle(.roundedBorder)
        .padding(.horizontal)
```

Egy modifier-t nem csak az adott néztre, hanem az őt tartalmazóra is rá lehet helyezni.

> Töröljük ki a `.textFieldStyle()` mindkét `TextField` alól és adjuk hozzá az őket tartalmazó `VStack`-hez.

Nem változott semmi, de kicsit tisztább lett a kód.

> Hozzunk létre egy újabb `@State property`-t `result` névvel, hogy az eredményt abban tárolhassuk.

```
    @State private var result: String = "Result"
```

> Ezután adjunk hozzá egy `Button`-t `=` felirattal a Stack-ünkhöz.

A Button-nak két closure is kell, amit `Swift 5.3`-tól a *multiple trailing closures* funkciónak hála, nem kell a init fejlécébe beírnunk.

```

```

```
Button {
    //action
} label: {
    //text
}
```

> A label closure-ben adjunk meg egy `Text`-et a megfelelő felirattal!

> Az action closure-ben pedig számoljuk ki az `operand1` és `operand2` összegét!

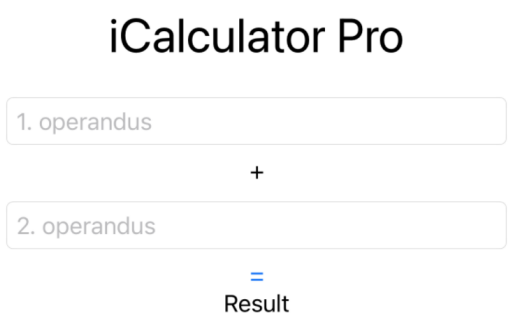
```
Button{
    if let o1 = Float(operand1), let o2 = Float(operand2) {
        self.result = String(o1 + o2)
    }
} label: {
    Text("=")
}
```

Most már csak valahogy meg kellene jeleníteni az eredményt

> Adjunk hozzá egy újabb `Text`-et a Stack-ünkhöz. Ezúttal nem kell a bindig-ot beállítani, elég csak kiíratni.

```
Text("\(result)")
```

Valami ilyesmit kell kapnunk:



Kicsit dolgozzuk át a kinézetét az alkalmazásnak, hogy ne minden középen legyen.

> Tegyünk be egy `Spacer`-t az `iCalculator Pro` felirat közé

Most címen kívül minden a képernyő aljára került...

Design szempontjából nem túl praktikus, mert a billentyűzet rá talál csúszni a `TextField`-ekre. De ezt érdemes kipróbálni a szimulátort is.

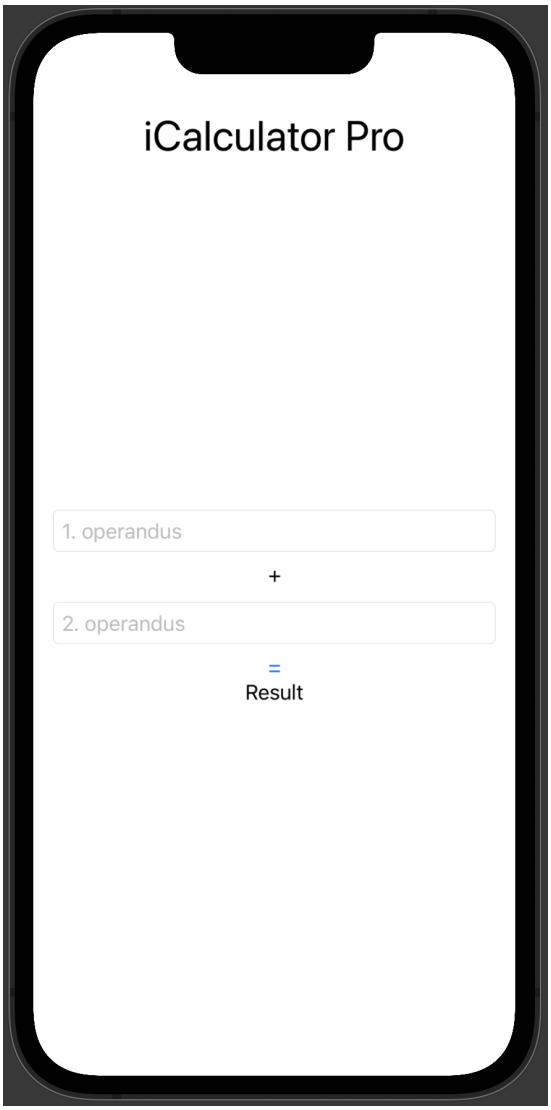
> Válasszuk ki az iPhene 13-t, mint target és futtassuk az alkalmazást.

Megállapíthatjuk, hogy a nézetünk átméreteződik, így a `TextField`-ek is feljebb csúsznak, ha feljön a billenytűzet. És, azt is láthattuk, hogy működik a számológép. A trükk az, hogy a billentyűzet átméretezi a Safe Area-t, amihez a nézetünk hozzá van kötve.

> Tegyünk be egy `Spacer`-t a Stack aljába is, hogy minden középre rendeződjön.

```
Text("\(result)")

Spacer()
```



Alakítsuk egy kicsit a gombunkat, mert se a színe, se a mérete, se a formája nem az igazi.

> Állítsuk be a gomb méretét! Mivel a `Text` az, ami látszik belőle, érdemes ezt átméretezni.

```
Button{
    ...
} label: {
    Text("=")
        .frame(width: 100, height: 30, alignment: .center)
}
```

Ha kipróbáljuk, akkor most már sokkal naygobb területen "érzékeny" a gomb, de a határait még nem látjuk.

> Állítsunk be egy fekete, lekerekített keretet a `Text`-nek!

Ezt `iOS 14`-ig még csak csak az `.overlay()` modifier segítségével lehetett megoldani. Valahogy így:

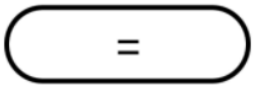
```
Button{
    ...
} label: {
    Text("=")
```

```
            .frame(width: 100, height: 30, alignment: .center)
            .overlay(
                RoundedRectangle(cornerRadius: 20)
                    .stroke(Color.black, lineWidth: 2)
                )
    }
```

> Ha már fekete a keret, akkor a felirat is legyen az!

```
Text("=")
    .frame(width: 100, height: 30, alignment: .center)
    .overlay(
        RoundedRectangle(cornerRadius: 20)
            .stroke(Color.black, lineWidth: 2)
    )
    .foregroundColor(Color.black)
```

Ilyen lett.



Mi lenne, ha megpróbálnánk kitölteni egy színnel? Valamiféle `.background()` modifier-t kellene hívni, de hol? Igazából mindegy; lehet a `Text`-re, az `overlay`-re vagy magára a `label`-re is.

> Állítsunk be kék háttérszínt a gombnak.

```
.background(Color.blue)
```

Nem valami szép... Kilóg a keretből.

Mit rontottunk el? Azt, hogy az `overlay` van lekerekítve, nem maga a gomb.

> Egészítsük ki az előbbi kódunkat. Ezúttal állítsunk be háttérnek egy `RoundedRectangle`-t, aminek kék a színe.

```
.background(RoundedRectangle(cornerRadius: 20).fill(Color.blue))
```

Most már egy fokkal szebb!

> A fekete szöveg a kék háttéren nem az igazi. Állítsuk át fehérre a "szöveget" és a border-t is!

Kész is vagyunk

iOS 15-től szerencsére még könnyebb dolgunk van. Onnantól ugyanis már használhatjuk a `buttonStyle()` és a `buttonBorderShape()` modifiereket.

> Töröljük vagy kommentezzük ki az imént létrehozott gombot `.overlay()`, `.foregroundColor()` és `.background()` modifier-ét.

> Állítsuk be először a `Button`-ön a `.buttonStyle()` modifiert `bordered`-re.

Nem valami szép...

> Állítsuk be a `.buttonBorderShape()`-et `.capsule`-re.

A forma jó, de alig látszik...

> Módosítsuk a `.buttonStyle()`-t `borderedProminent`-re.

Most már szép kék. A prominent beállítás a gomb `tint` color-ját állítja be színnek, ami alapértelmezetten a kék szín.

> Módosítsuk a `Button` színét a `.tint()` modifier-rel.

```
.tint(Color.purple)
```

Önálló feladat

A számológép még csak egy műveletet tud, ezt bővítsük ki! A `+` felirat helyett tegyünk be egy `Picker`-t, amiben ki lehet választani az összeadás mellett a kivonást és a szorzást is.

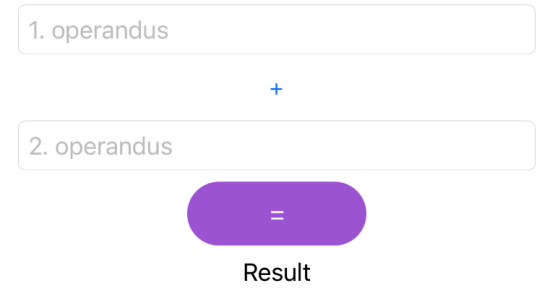
> Ehhez először megint egy `@State property`-t kell létrehozni. Legyen a neve `selectedOperation`.

```
@State private var selectedOperation = 0
```

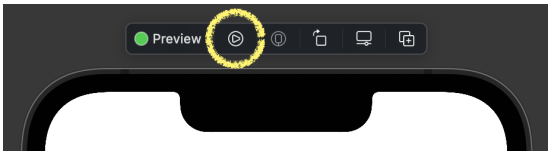
> A `Text("+")` helyett hozzunk létre egy `Picker`-t.

```
Picker("Operation", selection:$selectedOperation)
{
    Text("+").tag(0)
    Text("-").tag(1)
    Text("*").tag(2)
}
```

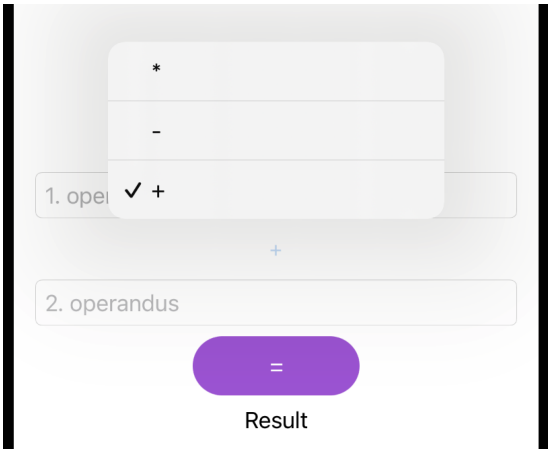
Nem sok látható változás történt.



> Próbáljuk ki az élő `Preview`-t az alábbi képen látható gombra kattintva.



Ha most rákattintunk a `+` jelre, akkor egy felugró ablakot kell látnunk a másik két művelettel.

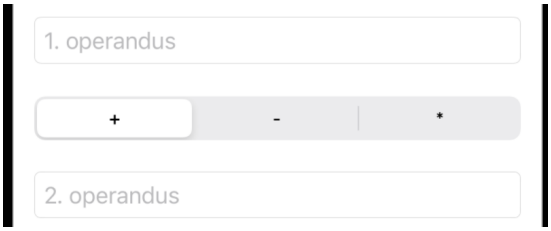


Ez nem annyira ideális, lehetne szebb is.

> Állítsuk át a `Picker` stílusát `segmented`-re és `padding`-et is állítsuk be.

```
Picker("Operation", selection:$selectedOperation)
{
    Text("+").tag(0)
    Text("-").tag(1)
    Text("*").tag(2)
}
    .pickerStyle(.segmented)
    .padding()
```

Máris szebb az eredmény.



Az egyes szegmensek ugyan kiválasztódnak, de jelenleg észben kell tartani, hogy melyik (`tag`) érték, melyik művelethez tartozik, ami nem az igazi.

> Hozzunk létre egy `enum`-ot `OperationType` névvel, ami az egyes műveleteket tartalmazza.

```
enum OperationType {
    case add
    case subtract
    case multiply
}
```

> Írjuk át a `selectedOperation` kezdőértékét. (Thx Type Inference)

```
@State private var selectedOperation = OperationType.add
```

> Cseréljük le a `Picker` `tag` értékeit `Int`-ről `OperationType`-ra.

```
Picker("Operation", selection:$selectedOperation)
{
    Text("+").tag(OperationType.add)
    Text("-").tag(OperationType.subtract)
    Text("*").tag(OperationType.multiply)
}
```

> Bővítsük a ki a `Button` funkcionalitását olymódon, hogy a `selectedOperation` alapján a megfelelő műveletet végezze el.

```
if let o1 = Float(operand1), let o2 = Float(operand2) {
    switch(selectedOperation){
    case .add:
        self.result = String(o1 + o2)
    case .subtract:
        self.result = String(o1 - o2)
    case .multiply:
        self.result = String(o1 * o2)
    }
}
```

> Próbáljuk ki az számológépet!

Még ennél is tovább tudunk meni. A cél. hogy az `enum` értékei alapján generálódjon le a `Picker` tartalma. Ehhez a `ForEach`-et fogjuk segítségül hívni, ami végig fog iterálni az `OperationType`-on.

> Először is bővítsük ki az `OperationType`-ot! Állítsuk be a `rawValue` type-ot `String`-nek és adjunk egyedi értéket is minden elemére.

```
enum OperationType: String {
    case add = "+"
    case subtract = "-"
    case multiply = "*"
}
```

> Hogy a `ForEach` végig tudjon iterálni az `enum`-on, még le kell származni az `Identifiable` protocol-ból. Továbbá, hogy hivatkozni tudjunk az `enum` összes elemére, a `CaseIterable` protokollból is le kell származni. Ezt tegyük is meg!

```
enum OperationType: String, CaseIterable, Identifiable {
    case add = "+"
    case subtract = "-"
    case multiply = "*"

    var id: String { self.rawValue }
}
```

> Végül csupán annyi a dolgunk, hogy lecseréljük a `Picker` törzsét.

```
Picker("Operation", selection:$selectedOperation)
{
    ForEach(OperationType.allCases){ operation in
        Text(operation.rawValue).tag(operation)
    }
}
```

> Adjunk hozzá egy újabb műveletet (pl.: az osztást) az `enum`-hoz!

Látható, hogy a UI autómatikusan változik, míg a `Button` eseménykezelőjében a `switch` jelzi, hogy egy ág nincs lefedve.

Szorgalmi feladat

Korábbi számítások (*`History` nézet*)

> Adjunk hozzá egy `Text`-et a `VStack` aljához, majd hozzunk létre hozzá egy `@State property`-t, amit meg fog jeleníteni.

> A `Text`, hogy görgethető legyen, tegyük bele egy `ScrollView`-ba, aminek a `.frame()` modifier segítségével állítsuk be a magasságát fixen 300-ra!

> Módosítsuk az eredményeket kiszámító kódot oly módon, hogy az aktuális számításról bekerüljön egy sor a `Text`-ba, pl. `13.00 + 13.00 = 26.00`!