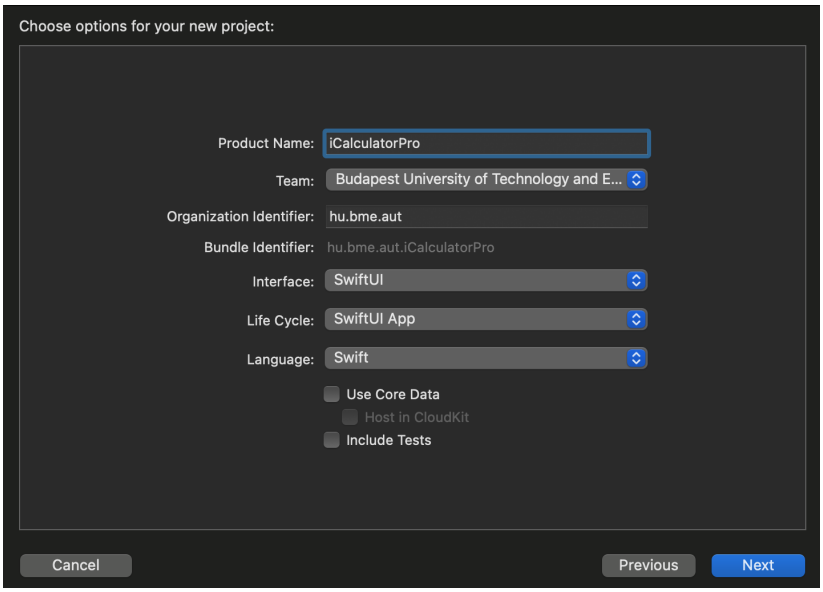
iOS Labor 3 - Ismerkedés a `SwiftUI`-jal

1. feladat: iCalculatorPro

> Nyisd meg az Xcode-ot és a File > New > Project... menüben válaszd az iOS > App sablont.
> Nevezzük el az alkalmazást **iCalculatorPro**-nak!

> Az `Interface` legyen `SwiftUI`



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.

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

> Töröljük ki a `body` computed property tartalmát

> Adjunk hozzá egy `Text`-et **"iCalculator Pro"** szöveggel!

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

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

> 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...

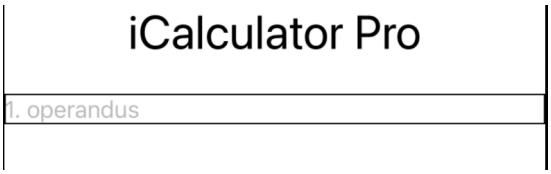
Mivel a `TextField`-ünk semmiféle stílussal nem rendelkezik, állítsunk be neki valami használható megjelenést

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

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

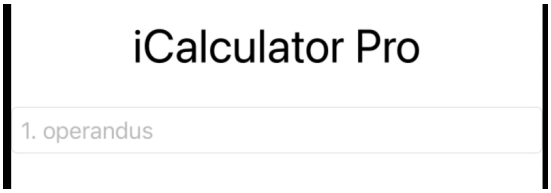
```
    TextField("w", text: $operand1)
        .border(.black)
```

> 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)
```

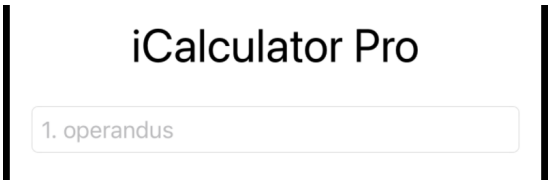


Egy fokkal szebb, de nagyon rálóg a képernyő szélére.

> Használjuk a `.padding` modifier-t, hogy vízintesen 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!

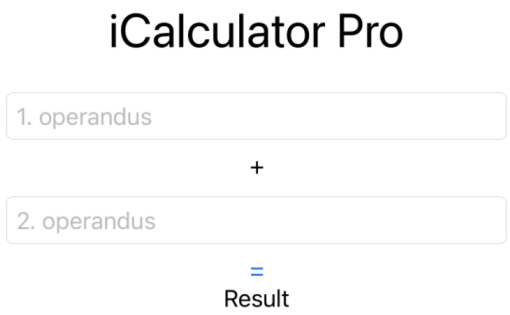
```
Button{
    if let o1 = Float(operand1), let o2 = Float(operand2) {
        self.result = String(o1 + o2)
    }
} label: {
    Text("=")
}
```

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.

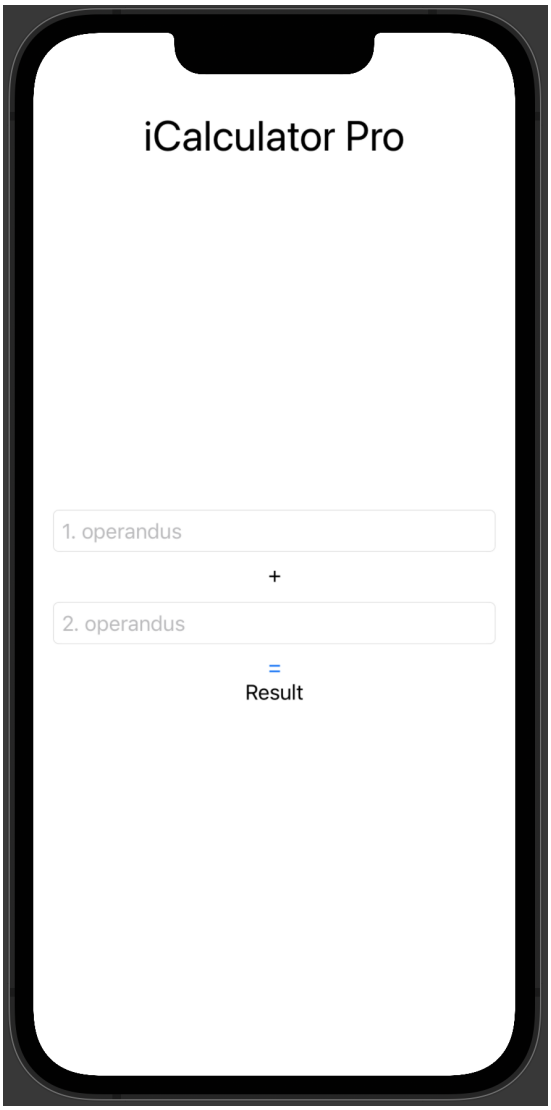
> Válasszuk ki az iPhene 13-t, mint target és futtassuk az alkalmazást.

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. iOS 15-től szerencsére 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)
```

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.

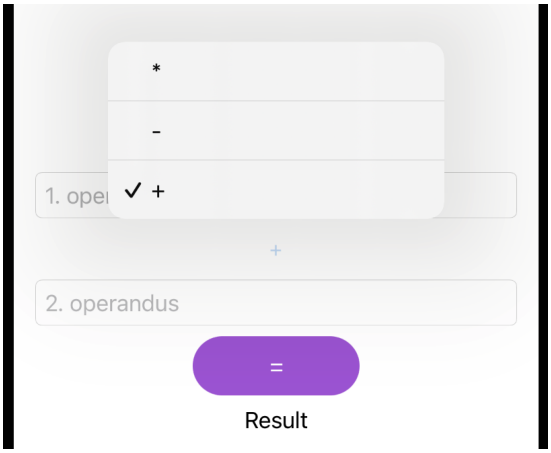
> Ehhez először megint egy `@State property`-t kell létrehozni. Legyen a neve `selectedOperation`.

```
@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)
}
```

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

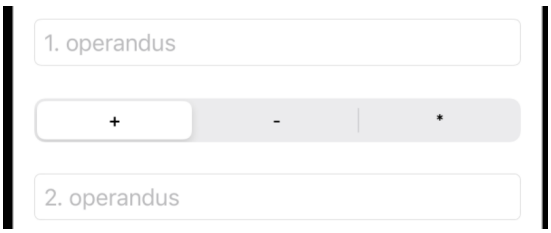


Ez nem annyira ideális, lehetne szebb is.

> 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.

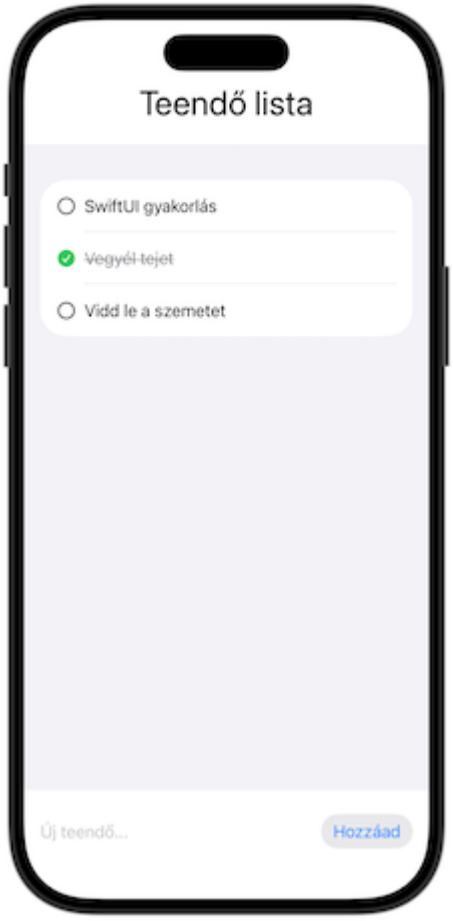
```
Picker("Operation", selection:$selectedOperation)
{
    ForEach(OperationType.allCases){ operation in
        Text(operation.rawValue).tag(operation)
    }
}
```

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

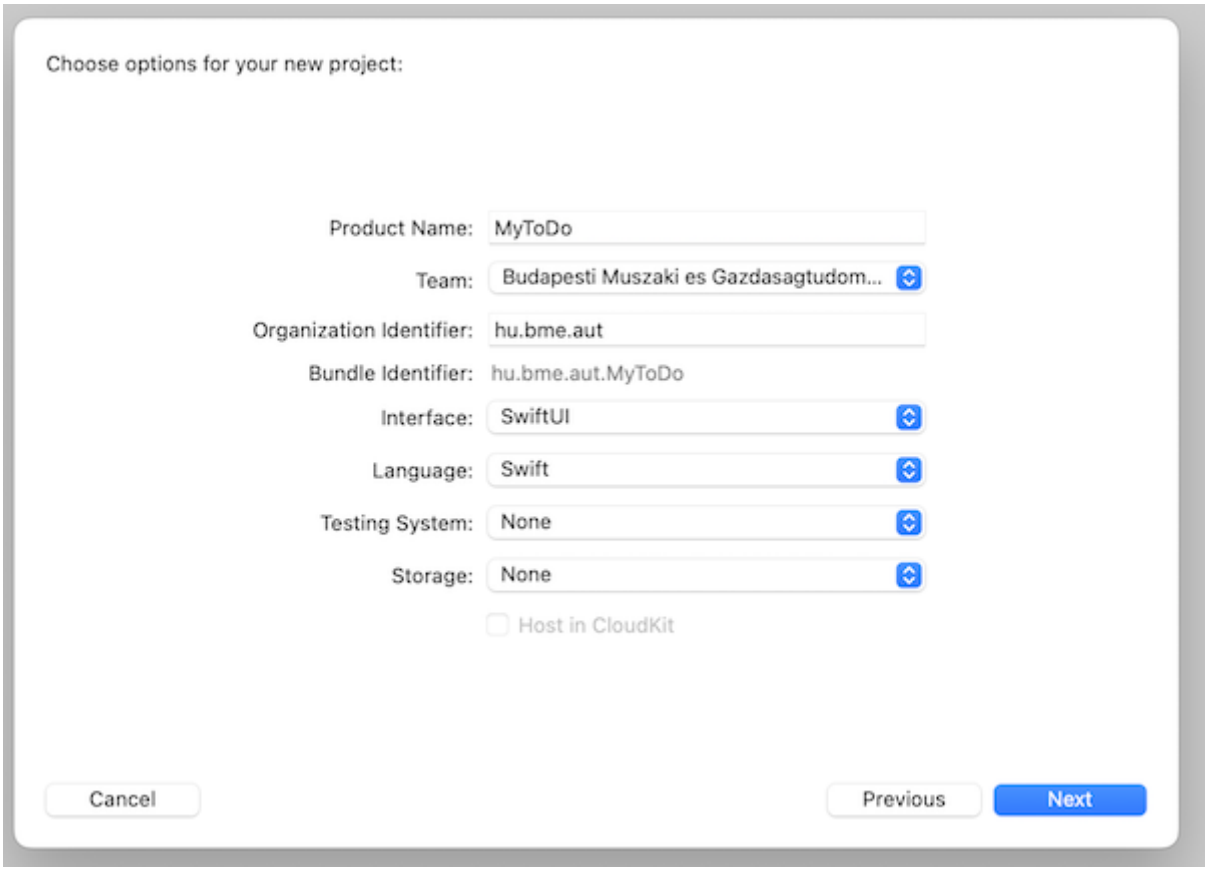
Látható, hogy a UI autómatikusan változik, míg a `Button` eseménykezelőjében a `switch` jelzi, hogy egy ág nincs lefedve.

2. Feladat: Egyszerű ToDo Alkalmazás

Ebben a feladatban egy működő teendőlista alkalmazást fogunk felépíteni. Az alkalmazás képes lesz új feladatokat felvenni, a meglévőket elvégzettnek jelölni és törölni őket.



> Nyisd meg az Xcode-ot és a *File > New > Project...* menüben válaszd az *iOS > App* sablont. Nevezd el a projektet MyToDo-nak, és győződj meg róla, hogy az Interface opció SwiftUI-ra van állítva.



A Model

A teendők tárolásához készítünk egy egységes struktúrát.

> Hozzunk létre egy új, üres Swift fájlt a projektben (File > New > File... > Swift File). Nevezd el `TodoItem.swift`-nek.

> Ebben az új fájlban definiálj egy struct-ot TodoItem néven, ami megvalósítja a az Identifiable protokollt.

Az Xcode jelezni fogja, hogy szükséged van egy id tulajdonságra.

> Adj hozzá egyet konstans property-t, ami legyen `UUID` típusú.

```
let id = UUID()
```

Ez minden teendőnek ad egy egyedi azonosítót, ami majd a lista kezelésekor hasznos lesz.

> Ha az Xcode problémázna, importáljuk be a `Foundation` frameworköt

```
import Foundation
```

> Adj hozzá két további tulajdonságot is:
>
> - Egy `title` nevű `String` típusú változót a teendő szövegének,
> - Egy `isCompleted` nevű `Bool` típusú változót, ami számon tartja, hogy elvégeztük-e már a feladatot.

> Minden új teendő befejezetlenként induljon.

A ContentView

> Most térjünk át a `ContentView.swift` fájlra, ahol a felhasználói felületet építjük fel.

> A `ContentView` struct-on belül, de a `body` property előtt, hozz létre két állapotváltozót az `@State` **property wrapper** segítségével.

Az első tárolja majd a teendőink listáját:

```
@State private var tasks: [TodoItem] = []
```

A fejlesztés kezdeti szakaszában érdemes nem üres többel indítani, hogy legyen mit megjeleníteni. Így módosítsuk a korábban írt kódot.

```

```

```
@State private var tasks: [TodoItem] = [
        TodoItem(title: "SwiftUI gyakorlás"),
        TodoItem(title: "Vegyél tejet", isCompleted: true),
        TodoItem(title: "Vidd le a szemetet")
    ]
```

A második állapotváltozó a felhasználói beviteli mező szövegét fogja tárolni:

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

> Cseréld le a body teljes tartalmát egy `VStack`-re. Ez a konténer fogja függőlegesen elrendezni a fejlécet, a listát és a a beviteli mezőt.

> A `VStack`-en belül hozz létre egy `List`-et.

> A `List`-en belül használj egy `ForEach` ciklust, ami végigmegy a `tasks` tömbödön.

```
List {
    ForEach($tasks) { $task in
        ...
    }
}
```

> A `ForEach` minden `task` eleméhez hozz létre egy `HStack`-et, hogy a *pipa* és a *szöveg* egymás mellett jelenjen meg

> A `HStack`-en belül az első elem egy `Image` legyen. Az `Image(systemName:)` inicializálót használd.

> Egy feltétellel addjuk meg a kép nevét: ha a `task.isCompleted` igaz, akkor a kép neve legyen **"checkmark.circle.fill"**, egyébként pedig **"circle"**.

```
Image(systemName: task.isCompleted ? "checkmark.circle.fill" : "circle")
```

> Ha szeretnénk, hogy más színű is legyen a pipa, akkor adjuk hozzá az `Image`-hez egy `.foregroundColor` modifier-t, amiben a szín a `task.isCompleted`-től függően `.green`, vagy `primary`.

> A kép után helyezz el egy `Text` nézetet, ami megjeleníti a `task.title`-t.

> Alkalmazz a `Text`-re egy `.strikethrough()` modifiert.
>
> - Az első paramétere legyen a `task.isCompleted`, ami alapján eldönti, hogy áthúzza-e a szöveget.
> - A második paramétere legyen a `.secondary` Color.

> A `VStack` tetejére tegyél egy `Text`-et **"Teendőim"** felirattal.

> Állítsuk be a szöveg méretét a `.font` modifierrel `.largeTitle`-ra.

A Beviteli Felület Hozzáadása

A felhasználónak szüksége van egy helyre, ahol új teendőket gépelhet be.

> A `VStack`-en belül, a `List` alá, illessz be egy új `HStack`-et.

> Ennek a `HStack`-nek az első eleme egy `TextField` legyen. Az első paramétere legyen a `placeholder` szöveg (pl. **"Új teendő..."**), a második, text paramétere pedig *binding* a `newTodoTitle` propertyhez. Ezt a `$` jel használatával teheted meg: `$newTodoTitle`.

> A `TextField` után, még mindig a `HStack`-en belül, adj hozzá egy `Button`-t. A gomb címkéje legyen a **"Hozzáad"** szöveg. A gomb akciója egyelőre legyen egy üres {} blokk.

```
Button("Hozzáad") {
    addNewTask()
}
```

> A `Button`-hoz adjunk hozzá egy `.buttonStyle` modifiert, amiben a `.bordered` stílust állítjuk be.

> Az esztétika kedvéért adj egy `.padding()` modifiert is a `HStack`-hez is.

Az Új Teendő Hozzáadásának Logikája

Most töltsük fel élettel a *"Hozzáad" gombot*.

> A `ContentView` struct-on **belül**, de a `body`-n **kívül**, hozz létre egy új, `private` metódust `addNewTask()` névvel.

> Ezen a függvényen belül ellenőrzi, hogy a newTodoTitle nem üres-e (`.isEmpty` property).

> Ha nem üres, hozz létre egy új `TodoItem` példányt a `newTodoTitle` értékével.

> Ezt az új példányt add hozzá a `tasks` tömbhöz az `.append()` metódussal.

> Végül állítsd vissza a `newTodoTitle` értékét egy üres stringre (`""`), hogy a beviteli mező kiürüljön.

> Most menj vissza a *"Hozzáad" gombodhoz* a `body`-ban, és az eddig üres akció blokkjába hívd meg az `addNewTask()` metódust.

A Teendők Állapotának Váltása (Befejezett/Befejezetlen)

Tegyük lehetővé, hogy a felhasználó egy koppintással elvégzettnek jelölhessen egy teendőt. A `ForEach` ciklust módosítanod kell, hogy az elemeket bindinggal kezelje.

> Írd át a `ForEach(tasks)`-ot `ForEach($tasks)`-ra, a `task in`-t pedig `$task in`-re.

Ez lehetővé teszi, hogy közvetlenül módosítsd a ciklusban lévő elemeket.

> Keresd meg a `HStack`-et a `ForEach`-en belül (ami egy teendőt jelenít meg). Adj hozzá egy `.onTapGesture` modifiert.

> Az `.onTapGesture` akciójának blokkjában hívd meg a `task.isCompleted.toggle()` metódust.

Ez automatikusan átváltja a logikai értéket az ellenkezőjére. A @State-nek köszönhetően a felület azonnal frissülni fog.

A Törlés Funkció Implementálása

Végül adjuk hozzá a klasszikus *"swipe to delete"* funkciót.

> A `ForEach` ciklushoz adj hozzá egy `.onDelete(perform:)` modifiert.

Ez a modifier egy függvényt vár paraméterként.

> Hozzunk létre egyet! Az `addNewTask` metódushoz hasonlóan, hozz létre egy új, private metódust `deleteTask` névvel, ami egy `offsets` nevű `IndexSet` típusú paramétert vár.

A paraméter megmondja, melyik sor(ok)at akarja a felhasználó törölni.

> A `deleteTask` függvény törzsében mindössze ennyit kell írnod:

```
tasks.remove(atOffsets: offsets)
```

> Végül add meg ezt a metódust a módosítónak: `.onDelete(perform: deleteTask)`.