5 Attributen @State och @Binding (provavsnitt)

PROV

© Copyright 2022 HB Bohel Smådatorprodukter

PROV

© Copyright 2022 HB Bohel Smådatorprodukter

Attributet @State

Reaktiva användargränssnitt kräver stöd för observerbara värden:

- `@StateObject` för referenser till *observerbara objekt* som vyn själv äger

Attributet `@State` kan användas för *observerbara värden* inom en enskild vy:

- Tvingar fram en uppdatering av användargränssnittet när de ändras
- Lagras i applikationens *heap* (dvs med dynamisk minnesallokering)
- Överlever därmed när ett användargränssnitt skapas på nytt vid uppdatering
- Lämpliga för tillstånd som enbart har med presentationen att göra
- Verksamhetsnära tillstånd ska finnas i modellen – inte i vyn!

Exempel:

- Vilka delar av ett användargränssnitt som ska visas eller döljas
- Vilken flik i en vy med flera flikar som visas för användaren
- Med hur många decimalers noggrannhet ett decimaltal ska visas

```
@State private var numberOfDecimals = 2
```

SwiftUI bygger på reaktiva användargränssnitt. Vi har redan sett hur detta hanteras genom att deklarera beroenden till observerbara data. I föregående kapitel såg vi exempel på hur attributet `@StateObject` kunde användas för att deklarera referenser till observerbara objekt som en vy själv äger, medan `@ObservableObject` (och `@EnvironmentObject`, som vi återkommer till senare) deklarerar referenser till observerbara objekt som en vy inte själv äger.

Det kan emellertid även förekomma data av värdetyper som vi vill göra observerbara, dvs att de tvingar fram en uppdatering av användargränssnittet när de ändras. Medan det är sant att applikationens tillstånd i normala fall hör hemma i modellen i MVVM-arkitekturen, och absolut *inte* i vyn, finns det tillstånd som endast har med vyns presentation att göra. Dessa kan representeras av egenskaper som inte behöver vara tillgängliga för andra delar av en applikation. För dem kan attributet `@State` vara ett lämpligt val.

Exempel på sådana tillstånd är:

- om en del av ett användargränssnitt ska visas eller ej
- vilken flik av en vy med flera flikar som ska visas
- med hur många decimalers noggrannhet ett decimaltal ska visas

De poster av posttyper som ingår i den deklarativa specifikationen av ett användargränssnitt är mycket kortlivade. De skapas i samband med att användargränssnittet ska återges i skärmbilden och försvinner kort därefter, när användargränssnittet är på plats. Dessutom är dessa poster i princip inte ändringsbara. Därför måste en egenskap som ska överleva posternas korta livscykel – och därtill vara ändringsbar – skapas någon annanstans. I applikationens *heap* skapas minne med dynamisk minnesallokering, främst objekt av klasser. Genom att `@State`-märkta egenskaper skapas där vinner vi de önskade fördelarna.

PROV

© Copyright 2022 HB Bohel Smådatorprodukter

Attributet @Binding

Attributet `@Binding` skapar en *bindning* till en observerbar egenskap.

- Bindningen fungerar som en referens (ett alias) till egenskapen.
- Vi skapar en bindning till en egenskap med operatorn $ (dollartecken).

Som ett exempel, betrakta en `@State`-märkt egenskap av typen `Bool`:

```
@State private var encrypted = true
```

En `Toggle`-vy är ett användargränssnittselement med ett av- och ett på-läge.

Första argumentet i initieringsuttrycket är en bindning till en `Bool`-egenskap:

```
init(isOn: Binding<Bool>, label: () -> Label)
```

```
Toggle(isOn: $encrypted) {
    Text("Krypterad kommunikation")
}
```

Attributet `@Binding` skapar en *bindning* till en observerbar egenskap, t ex en `@State`-märkt egenskap eller en egenskap inom ett objekt där referensen till objektet märkts med antingen `@StateObject`, `@ObservedObject` eller `@EnvironmentObject`.

En bindning fungerar som en referens (ett alias, om man så vill) till en annan egenskap, så att egenskapen endast lagras på en plats i applikationen (ett stöd för designprincipen *single source of truth*). Vi skapar en bindning genom att placera operatorn `$` (dollartecken) framför egenskapens namn, t ex `$encrypted`.

I bildens exempel har en vy en `@State`-märkt egenskap med namnet `encrypted` av typen `Bool`. Vyer av typen `Toggle` kan användas för att presentera ett användargränssnittselement som har ett av- och ett på-läge, där tillståndet kan representeras med en egenskap av typen `Bool`.

Undre delen av bilden visar ett initieringsuttryck för en `Toggle`-vy och den motsvarande initieringsmetodens deklaration i typen `Toggle`.

Initieringsuttrycket för en `Toggle`-vy kräver två argument:

- Argumentet med det externa namnet `isOn` är en bindning till en egenskap av typen `Bool` som representerar av- resp på-tillståndet. Bindningen skapar vi med operatorn `$`.
- Argumentet med det externa namnet `label` är en vy som fungerar likt en ledtext för `Toggle`-vyns av- och på-kontroll. Ofta är detta en `Text`-vy, men det skulle t ex även kunna vara en `Image`-vy eller en `Label`-vy (kombination av ikon och text).

PROV

© Copyright 2022 HB Bohel Smådatorprodukter

Att visa ett fast antal decimaler

RPNCalculator.swift (utdrag)

```
class RPNCalculator: ObservableObject {
    @Published private var model = RPNCalculatorModel()

    ...

    func display(decimals: Int) -> String {
        if inNumericEntry {
            return model.numberEntered
        } else {
            let displayValue = model.lastOperand
            if displayValue.isInfinite {
                return "Oändligt värde"
            } else if displayValue.isNaN {
                return "Otillåten operation"
            } else {
                return formatLocalizedNumber(displayValue, decimals: decimals)
            }
        }
    }

    ...
}

func formatLocalizedNumber(_ value: Double, decimals: Int) -> String {
    return String.localizedStringWithFormat("%.\(decimals)f", value)
    // let formatter = NumberFormatter()
    // formatter.maximumFractionDigits = 15
    // return formatter.string(from: value as NSNumber) ?? ""
}
```

Vi kommer nu att komplettera vår kalkylator med möjligheten att påverka hur många decimaler av svaret på en beräkning som ska visas – det är ju inte alltid önskvärt att se ett svar med 15 decimaler!

I ViewModel-klassen `RPNCalculator` behöver vi göra två ändringar:

- Den beräknade egenskapen `display` omformas till en metod `display(decimals:)` som returnerar det som ska synas i visningsfönstret, avrundat till det antal decimaler som argumentet med det externa namnet `decimals` anger:

  `func display(decimals: Int) -> String { … }`

- Den globala funktionen `formatLocalizedNumber(_:)`, som formaterar ett värde av typen `Double` som decimaltal enligt plattformens regionala inställningar byggs ut till funktionen `formatLocalizedNumber(_:decimals:)`, som ska begränsa antalet decimaler i svaret med bibehållen avrundning och regional formatering.

  Vi har valt att inte ta bort – utan endast kommentera ut – funktionens tidigare implementation, med tanke på kommande övningar.

PROV

© Copyright 2022 HB Bohel Smådatorprodukter

En bläddерkontroll för antal decimaler

CalculatorView.swift (utdrag)

```
struct CalculatorView: View {
    @StateObject private var calculator = RPNCalculator()

    @State private var numberOfDecimals = 2

    var body: some View {
        ZStack {
            Color(.systemTeal).opacity(0.4).edgesIgnoringSafeArea(.all)
            VStack {
                HStack { ... }                      // Visningsfönster
                Stepper("Antal decimaler (\(numberOfDecimals))",
                        value: $numberOfDecimals, in: 0...15)
                    .padding([.top, .leading, .trailing], 5)
                    .padding(.bottom, 10)
                Group {
                    HStack { ... }                  // Knapprad 1
                    ...
                    HStack { ... }                  // Knapprad 6
                }
                ...
            }
            .padding()
        }
    }
}
```

Vi behöver därefter komplettera vyn (av typen `CalculatorView`) med en möjlighet att välja med hur många decimaler värdet i kalkylatorns visningsfönster ska visas.

För detta kan vi utnyttja en egenskap märkt med attributet `@State`:

```
@State private var numberOfDecimals = 2
```

Denna egenskap kommer alltså – trots sin deklaration inuti en `struct`-typ – att lagras i applikationens heap, och är därmed dels ändringsbar, dels kommer den att överleva när `CalculatorView`-posten försvinner och återuppstår vid en vyuppdatering.

Det finns ett antal olika användargränssnittselement som skulle kunna användas för att låta användaren påverka antalet visade decimaler. Ett bra alternativ är en *bläddерkontroll*, där användaren kan bläddra bland ett begränsat antal alternativ.

I **SwiftUI** representeras bläddерkontroller av vyer av typen `Stepper`. Dessa kan anta olika utseenden beroende på vilken plattform de visas på. Där det finns tillräckligt utrymme visas en förklarande text inom kontrollen, och som första argumentet i initieringsuttrycket visar kan vi t o m använda det aktuella värdet av den egenskap vi påverkar inom texten. Det andra argumentet (`value`) är en bindning till den egenskap som lagrar det valda värdet, medan det tredje argumentet (`in`) är intervallet av tillåtna värden.

Lite oväntat behöver vi göra ytterligare en ändring. När vi lägger till `Stepper`-vyn i den deklarativa beskrivningen av användargränssnittet kommer `VStack`-vyn som den ingår i att ha elva direkt underordnade vyer. Det finns emellertid en maximigräns på tio direkt underordnade vyer, så vi måste skapa en mellannivå som kan omsluta en del av de elva vyerna. En bra lösning är en vy av typen `Group`, som existerar för bl a detta syfte och som inte får någon synbar effekt på användargränssnittet. Vi väljer att placera de sex `HStack`-vyerna för knappraderna inom `Group`-vyn.

PROV

© Copyright 2022 HB Bohel Smådatorprodukter

Fast antal decimaler i visningsfönstret

CalculatorView.swift (utdrag)

```
struct CalculatorView: View {
    @StateObject private var calculator = RPNCalculator()

    @State private var numberOfDecimals = 2

    var body: some View {
        ZStack {
            Color(.systemTeal).opacity(0.4).edgesIgnoringSafeArea(.all)
            VStack {
                HStack {                                    // Visningsfönster
                    Spacer()
                    if calculator.displayInRed {
                        Text(calculator.display(decimals: numberOfDecimals))
                            .font(.title).foregroundColor(Color(.systemRed))
                    } else {
                        Text(calculator.display(decimals: numberOfDecimals))
                            .font(.title)
                    }
                }
                Stepper("Antal decimaler (\(numberOfDecimals))",
                        value: $numberOfDecimals, in: 0...15)
                ...
            }
            .padding()
        }
    }
}

...
```

Eftersom vi har ändrat den beräknade egenskapen `display` i klassen `RPNCalculator` till en metod `display(decimals:)` behöver vi även ändra den deklarativa koden för visningsfönstret.

Vi skiljer i vår lösning (som bygger på kod hämtad från lösningsförslaget till övning 4.2 i föregående kapitel) på om texten ska visas i svart eller rött, men i båda fallen ersätts det tidigare initieringsuttrycket

```
Text(calculator.display)
```

med

```
Text(calculator.display(decimals: numberOfDecimals))
```

Observera att vi varken behöver eller kan skapa en bindning till `numberOfDecimals`, trots att det är en `@State-märkt` egenskap! Vi behöver endast läsa värdet av egenskapen – inte förändra den – och därför är heller inte metoden `display(decimals:)` kapabel att ta emot en bindning som argument.

PROV

© Copyright 2022 HB Bohel Smådatorprodukter

Applikationen med fast antal decimaler

Här visas applikationen med blädderkontrollen mellan visningsfönstret och knappsatsen. Vi har lagt till lite luft runt blädderkontrollen för att de olika delarna av användargränssnittet ska separeras tydligare.

I applikationen i bilden har även lösningsförslaget till övning 4.2 från tidigare kapitel integrerats.

PROV

© Copyright 2022 HB Bohel Smådatorprodukter