SwiftUI

Yung Chin, Yen

October 23, 2020

Outline

1 iOS app 的開發界面: UIKit v.s. SwiftUI

1.1 UIKit 與 SwiftUI 的差異性

1. 系統需求 UIKit 是從 Xcode1 就一直存在的 Framework;而 SwiftUI 則是 2019/6 WWDC 所發表的全新用來繪製 UI 的 Framework。因此,SwiftUI 必須搭配 iOS13+ 和 MacOS10.15+。[1]

2. 底層語言 UIKit 底層仍為 Objecitve-C;而 SwiftUI 則是完完全全用 Swift 打造的 Framework。

3. 語法簡潔度 SwiftUI 產生一顯示文字的元件更精簡潔了。

4. Declarative vs Imperative Programming
   - imperative programming:告訴電腦如何做 (HOW) 來得到我們想要的結果 (WHAT),如 Java, C++, PHP, C#, Swift
   - declarative programming:告訴電腦我們想要的結果 (WHAT),讓電腦決定如何做 (HOW),如 SwiftUI

5. 跨平台 跨平台指的非跨 Android(但希望有那麼一天是可以支援的)。跨平台指的是使用 SwiftUI 所開發的專案,可以同時支援 macOS、watchOS、tvOS 等系統。引用一句 WWDC2019 SwiftUI 演講者所說的一句話。

   Learn once, apply everywhere.

6. Automatic Preview 這是此次 SwiftUI 最大的亮點之一,所謂 Automatic Preview,意思指的是即時預覽,即我們一邊調整程式碼的同時,也可以立即看到修改後的結果。

7. 自動支援進階功能 SwiftUI 本身即支援 Dynamic Type、Dark Mode、Localization 等等。這邊特別來講一下 UIKit 和 SwiftUI 在文字設定上有關於 Dark Mode 的差異,UIKit 若是無特別指定文字的顏色 (意即使用 Default 的選項),在 Light Mode 字體會是白色;相對的在 Dark Mode 即會是白色,這點跟 SwiftUI 沒有特別的差異,但是 SwiftUI 除了 Default 外,還有 Secondary,如果還不喜歡的話,還有第三個選項,就是在 Assets 自行設定 Light Mode 和 Dark Mode 分別要顯示的顏色。

[1]簡介 SwiftUI & 用其建構一簡單的 APP

	UIKit	SwiftUI
系統需求	無限制	iOS 13+ MacOS 10.15+
底層語言	Objective-C	Swift
語法簡潔度	敗	勝
跨平台	無	有
Automatic Preview	無	有
自動支援進階功能	無	Dynamic Type Dark Mode Localization

Figure 1: UIKit 與 SwiftUI 的差異性比較圖

1.2 SwiftUI vs UIKit: Benefits and Drawbacks

1. Drawbacks of SwiftUI [2]
   - It supports only iOS 13 and Xcode 11. By switching to them, you abandon users of older versions of iOS, which is a radical move devoid of concern for the user. But since Apple annually updates its list of supported iOS versions, I think SwiftUI will be used more over the next two years as users install the latest iOS version.
   - It′ s still very young, so there isn′ t much data on Stack Overflow. This means that you can′ t get much help resolving complicated issues.
   - It doesn′ t allow you to examine the view hierarchy in Xcode Previews.
2. Benefits of SwiftUI [2]
   - It′ s easy to learn, and the code is simple and clean.
   - It can be mixed with UIKit using UIHostingController.
   - It allows you to easily manage themes. Developers can easily add dark mode to their apps and set it as the default theme, and users can easily enable dark mode. Besides, it looks awesome.
   - SwiftUI provides mechanisms for reactive programming enthusiasts with BindableObject, ObjectBinding, and the whole Combine framework.

[2]SwiftUI vs UIKit: Benefits and Drawbacks

- It offers Live Preview. This is a very convenient and progressive way to see the results of code execution in real time without having to build. I′ m not sure if it somehow affects the processor. So far, I′ ve noticed a delay provoked by the use of Live Preview, but I think Apple will soon make improvements.

- SwiftUI no longer needs Interface Builder. It was replaced by Canvas, an interactive interface editor. When writing code, the visual part in Canvas is automatically generated, and when you create visual presentation elements, they automatically appear in the code.

- Your application will no longer crash if you forget to update the @IBOutlet association with the variable.

- There′ s no AutoLayout or related problems. Instead, you use things like HStack, VStack, ZStack, Groups, Lists, and more. Unlike AutoLayout, SwiftUI always produces a valid layout. There′ s no such thing as an ambiguous or unsatisfiable layout. SwiftUI replaces storyboards with code, making it easy to create a reusable view and avoid conflicts related with the simultaneous use of one project by the development team.

2 AppDelegate v.s. SceneDelegate

- AppDelegate 原來的職責為負責 App 的生命週期和 UI 生命週期，在 Xcode11 後，AppDelegate 將 UI 的生命週期 (Scene Session) 交給 SceneDelegate。原 Xcode10

- 使用 Swift 為 User Interface 的專案 Launch 的生命週期為 AppDelegate ViewController, 而使用 SwiftUI 為 User Interface 的專案則變成為 AppDelegate SceneDelegate ContentView, 原本應該出現在 AppDelegate 的 applicationWillEnterForeground(_:) 等相關 App 到前、背景等相關的生命週期邏輯也都移至 SceneDelegate 裡了, method 名稱 application 的前綴字也都更改為 scene 了。[3]

2.1 SceneDelegate.swift

```
import UIKit
import SwiftUI

class SceneDelegate: UIResponder, UIWindowSceneDelegate {

    var window: UIWindow?

    func scene(_ scene: UIScene, willConnectTo session:
        UISceneSession, options connectionOptions: UIScene.
        ConnectionOptions) {
        // Use this method to optionally configure and attach the
            UIWindow `window` to the provided UIWindowScene `scene`.
        // If using a storyboard, the `window` property will
            automatically be initialized and attached to the scene.
        // This delegate does not imply the connecting scene or
            session are new (see `application:
            configurationForConnectingSceneSession` instead).

        // Create the SwiftUI view that provides the window contents
            .
        let contentView = ContentView()

        // Use a UIHostingController as window root view controller.
```

[3] SwiftUI 初體驗：建構一個簡單 App　讓你了解 SwiftUI 有多強大！

```
        if let windowScene = scene as? UIWindowScene {
            let window = UIWindow(windowScene: windowScene)
            window.rootViewController = UIHostingController(rootView
                : contentView)
            self.window = window
            window.makeKeyAndVisible()
        }
    }

    func sceneDidDisconnect(_ scene: UIScene) {
        // Called as the scene is being released by the system.
        // This occurs shortly after the scene enters the background
            , or when its session is discarded.
        // Release any resources associated with this scene that can
            be re-created the next time the scene connects.
        // The scene may re-connect later, as its session was not
            neccessarily discarded (see `application:
            didDiscardSceneSessions` instead).
    }

    func sceneDidBecomeActive(_ scene: UIScene) {
        // Called when the scene has moved from an inactive state to
            an active state.
        // Use this method to restart any tasks that were paused (or
            not yet started) when the scene was inactive.
    }

    func sceneWillResignActive(_ scene: UIScene) {
        // Called when the scene will move from an active state to
            an inactive state.
        // This may occur due to temporary interruptions (ex. an
            incoming phone call).
    }

    func sceneWillEnterForeground(_ scene: UIScene) {
        // Called as the scene transitions from the background to
            the foreground.
        // Use this method to undo the changes made on entering the
            background.
    }

    func sceneDidEnterBackground(_ scene: UIScene) {
        // Called as the scene transitions from the foreground to
            the background.
        // Use this method to save data, release shared resources,
            and store enough scene-specific state information
        // to restore the scene back to its current state.
    }
}

struct SceneDelegate_Previews: PreviewProvider {
    static var previews: some View {
        /*@START_MENU_TOKEN@*/Text("Hello, World!")/*
            @END_MENU_TOKEN@*/
    }
}
```

3 UIKit

DEMO

4 SwiftUI

4.1 教學影片

- Your First SwiftUI App (Full Compilation!)
- Lecture 1: Course Logistics and Introduction to SwiftUI: Stanford University CS193p
- Lecture 2: MVVM and the Swift Type System
- SwiftUI by Paul Hudson: Play All
- Understanding the basic structure of a SwiftUI app −WeSplit SwiftUI Tutorial 1/10
- Creating a form −WeSplit SwiftUI Tutorial 2/10
- Pushing new views onto the stack using NavigationLink −Moonshot SwiftUI Tutorial 3/10
- Modifying program state −WeSplit SwiftUI Tutorial 4/10
- Binding state to user interface controls −WeSplit SwiftUI Tutorial 5/10
- SwiftUI - Calculator Demo from Stanford iOS Course Part 1 of 2
- The Calculator (part 1) - Learn Swift UI
- Understanding MVVM Design Pattern: 講的超清楚
- azamsharp SwiftUI Videos
- 系列: Setting up - SwiftUI Starter Project 1/14

4.2 使用 SwiftUI 開啟新專案 [3]

1. 首先，打開 Xcode，並點擊 Create new Xcode project。在 iOS 之下選擇 Single View App，並為專案命名。
2. 然後在下方勾選 Use SwiftUI 的選項, 如果沒有勾選該選項的話, Xcode 會自動產生 storyboard 檔案 (UIKit)。

3. Xcode 會自動幫你創建一個名為 ContentView.swif 的檔案，Xcode 會在程式碼的右邊呈現一個即時的預覽視窗 (preview), 點選 resume 鈕生成預覽畫面 (會花一點時間)。

1. ContentView.swift

```
1  import SwiftUI
2
3  struct ContentView: View {
4      var body: some View {
5          Text("文字")
6      }
7  }
8
9  struct ContentView_Previews: PreviewProvider {
10     static var previews: some View {
11         ContentView()
12     }
13 }
```

- 第 1 行和 C++ 中的 #include <iostream> 同意，先匯入所需函式庫

- 第 3 行說明有一個 struct 名為 ContentView，這個 ContentView conform(尊循)View 這個 Protocol，這代表必須有一個 some view 或回傳一個 some view

- 在 ContentView 中，有一個叫 body 的變數 (第 4 行)，這個 body 的回傳類型為 some view，some 為 swift 5.1 出現的新 keyword，屬於 opaque 回傳類型，代表它會回傳某些類型為 view 的值，至於實際回傳的是那一種類型的 view，swift 並不太在意

- 第 5 行的最前面省略了一個 return，意思是 body 這個 variable 最後會傳回一個 Text, 即，呈現在 View 上，body 只能回傳一個值，若 view 上面有許多物件，則需包含進一個 container 中，最後回傳這一個 container。

- 第 9 行的 $\text{ContentView}_{\text{Previews}}$ 負責產生預覽畫面。

4.3 Text

1. 改變 Text 的屬性
   - 改變 component 有兩種方式：工具列、code

- Attributes (modifier 的不同順序可能產生不同效果)
  - frame
  - foregroundColor
  - background
  - font
  - padding
  - cornerRadius

(a) SwiftUI Inspector:

  i. on Text object (in preview screen): CMD + click

  ii. select Show SwiftUI Inspector

  iii. change Text, Font, Color

  iv. Monitor the corresponding code changes in code window

  `images/inspector-1.gif`

(b) Inspector frame `images/inspector-2.gif`

(c) code 於 Text("...") 後加上屬性 function 或修改其他屬性
`images/inspector-3.gif`

4.4 VStack

一個以上的物件都要放在 Stack 中，Stack 與 Stack 可相互包含，加入方式有二：

1. 由工具列 drag: Xcode 會自動加入相對的 code `images/vstack.gif`

2. coding

```
import SwiftUI
```

```

struct ContentView: View {
    var body: some View {
        VStack {
            Text("第一行文字")
            Text("第二行文字")
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

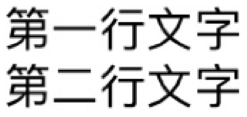


Figure 2: VStack

3. SwiftUI 撰寫原則
   - body 恆為只能 return 一物件。
   - 若有多個物件時，一定得放在 Stack 裡。

4.5 HStack

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        HStack {
            VStack {
                Button("請按我") {
                    print("TEST")
                }
                .frame(width: 60, height: 30, alignment: .center)
                .foregroundColor(.white)
                .background(Color.green)
                Button("別亂按") {
                    print("QQ")
                }
            }
            VStack {
                Text("第一行文字")
                    .frame(width: 100, height: 30, alignment: .
                       center   )
                    .foregroundColor(.white)
                    .background(Color.orange)
                Text("第二行文字")
                    .frame(width: 100, height: 30, alignment: .
                       center)
                    .foregroundColor(.white)
                    .background(Color.red)
            }
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

4.6 ZStack

4.7 Image

影像來源可以是 System Image 或自行下載/編修的影像 (Customized Image)

1. System Image

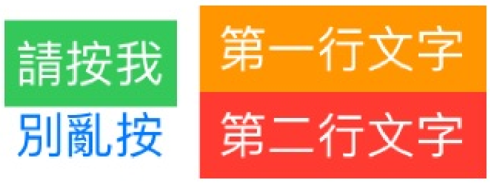


Figure 3: HStack

(a) SF Symbols [4]

(b) 從 iOS 13 開始，Apple 介紹了一個名為 SFSymbols 的新功能。SF Symbols 這功能由 Apple 所設計，當中集合了 1500 多個可以在 App 之中使用的符號。[3]

(c) Download SF Symbols app

(d) code

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        VStack {
            Text("System␣Image")
                .font(.headline)
                .foregroundColor(.orange)
            Image(systemName: "icloud")
            .resizable()
            .scaledToFit()
            .frame(width: 100, height: 80, alignment: .
               center)
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

[4]Find all available images for Image(systemName:) in SwiftUI

(e) Demo

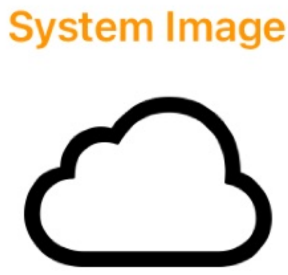


Figure 4: Images-1

2. Customized Image 語法
   (a) Drag image into Project folder Assets.xcassets

   (b) Add following code

```
Image("ImageName") //file name in Assets.xcassets
   .resizable()
   .scaledToFit()
   .frame(width: 200, height: 160, alignment: .center)
```

3. Image Attributes

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        VStack {
            Text("Albert␣Camus")
                .font(.title)
                .foregroundColor(.white)
                .background(Color.orange)
            Image("Albert-Camus")
                .resizable()
                .scaledToFill()
                .frame(width: 200, height: 200, alignment: .
                    center)
                .clipShape(Circle())

        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

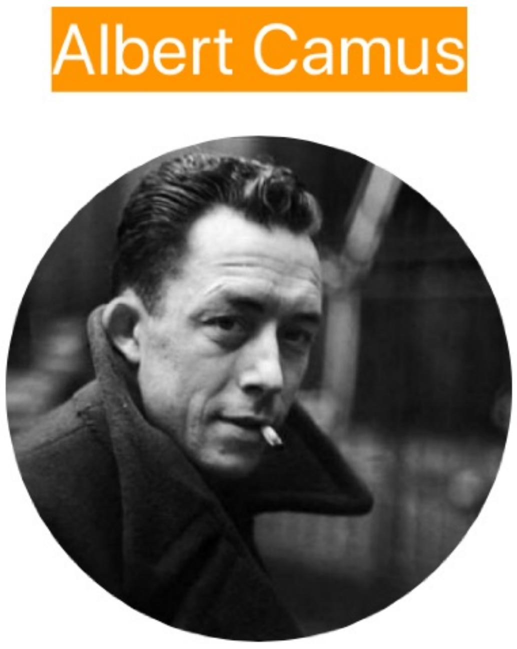


Figure 5: Images-2

4. Using SF Symbols
    - SF Symbols app

    - sfsymbols.com

    - SF Symbols: The benefits and how to use them guide

4.8 Button

1. **語法**

```
//...
Button("Title") {
    //action
}

Button(action: <#T##() -> Void#>, label: <#T##() -> _#>)
// ...
```

2. 將變數加入 View 中

```
struct ContentView: View {
    var title = "Hello SWiftUI"

    var body: some View {
        VStack {
            Text(verbatim: title)
                .padding(4)
                .foregroundColor(.white)
                .background(Color.gray)
        }
    }
}
```

3. 問題：可否於程式中改變 title 的值來改變 View 的顯示內容?
4. @State

```
struct ContentView: View {
    @State var title = "Hello SWiftUI"

    var body: some View {
        VStack {
            Text(verbatim: title)
                .padding(4)
                .foregroundColor(.white)
                .background(Color.gray)
        }
    }
}
```

加了 @State 後，SwiftUI 將在背後另外產生空間儲存 property 的內容，它不再儲存在 ContentView 裡，因此我們可以修改它的內容[5]。以 @State 宣告的 property 有個重要的特性，只要它的內容改變，畫面也會立即更新。它帶來了以下兩個好處:

[5]用狀態設計 SwiftUI 畫面—認識 @State property，binding & Toggle

- 不用另外寫 property 內容改變時更新畫面的程式。
- 不用擔心畫面顯示的內容跟 property 的內容不同步，比方修改了 property，但卻忘了更新畫面。

(a) 問題：在什麼地方去改變 title 的值? –> Button

5. 範例: 按下 Button，改變 Text title

```
struct ContentView: View {
    @State var title = "Hello␣SWiftUI"

    var body: some View {
        VStack {
            Text(verbatim: title)
                .padding(4)
                .foregroundColor(.white)
                .background(Color.gray)
            Button("Click␣Me") {
                self.title = "QQ"
            }
        }
    }
}
```

6. 問題：可否自行輸入要更改的 title 內容? –> TextField + $var

4.9 TextField

1. 語法

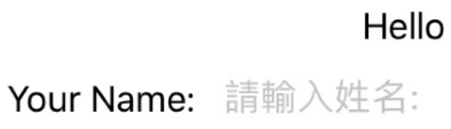
```
@State private var 變數="值"
TextField("提示文字", text: $變數)
```

2. 範例: 於 TextField 輸入資料，顯示於 Text 中

```
import SwiftUI

struct ContentView: View {
    @State private var title = ""

    var body: some View {
        VStack {
            Text(verbatim: "Hello␣"+title)
```

```
                HStack {
                    Text("Your␣Name:␣")
                    TextField("請輸入姓名:", text: $title)
                }
            }
        }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

3. Demo



Hello

Your Name: 請輸入姓名:

Figure 6: Button

4.10 some view

4.11 SubView

1. 自行改寫
2. Extract Subview

4.12 @State v.s. @Bidning

```
struct ContentView: View {
    var isRain = true
    var body: some View {
        VStack {
            Image(systemName: isRain ? "cloud.rain.fill" : "sun.max.
                fill")
                .resizable()
                .frame(width: 100, height: 100)
                Text(isRain ? "我們淋著大雨不知何時才能放晴" : "太陽公公
                    出來了, 他對我呀笑呀笑")

                Button("今天天氣如何␣?") {
                    self.isRain = Bool.random()
                }
            }
        }
}
```

上述程式中第 11 行會產生 Cannot assign to property: self is immutable 的錯誤，因為 Swift 的 view 為 struct 定義, 而 struct 為 value type, 故無法在 computed property 裡 getter & function 中修改其 property.

修正方式為：在 isRain 最前面加上 @State，如

```
// ...
@State var isRain = true
//...
```

4.13 **TODO** onAppear(perform:)

View 秀出時要額外做哪些事情....

4.14 學習資源

- SwiftUI Tutorials from Apple (官方基本教材)
- swiftdoc.org
- Swift Developer Blog
- Hacking with SWIFT

5 Customize UI Components

SwiftUI 提供豐富的 modifier 幫助我們設計客製 UI 元件的樣式，諸如陰影，旋轉等效果皆可透過 modifier 實現，還可以搭配方便的拖曳加入相關程式碼。[6]

5.1 Text

1. Advanced Attributes [6]

```
struct ContentView: View {
    var body: some View {
        Text("Example")
            .font(.title)
            .fontWeight(.bold)
            .foregroundColor(Color.white)
            .padding(4)
            .background(Color.gray)
            .cornerRadius(14.0)
            .rotationEffect(Angle(degrees: 15))
            .rotation3DEffect(Angle(degrees: 30), axis: (x: 10,
                y: 30, z: 30))
            .shadow(radius: 20)
    }
}
```

2. Demo

5.2 Image

1. Advanced Attributes s[7]

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        VStack {
            Text("Albert␣Camus")
                .font(.body)
```

[6]客製 UI 元件樣式的 SwiftUI modifier

[7]SwiftUI 裁切形狀的 clipShape & mask

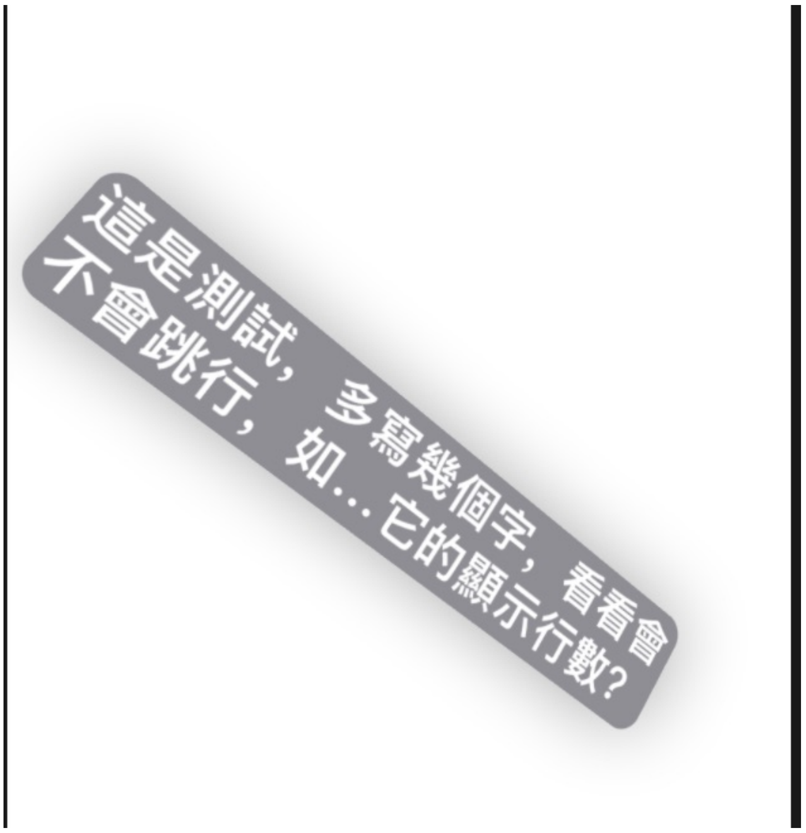

這是測試，多寫幾個字，看看會
不會跳行，如…它的顯示行數?

Figure 7: Text Attributes

```
                .foregroundColor(.white)
                .background(Color.orange)
            Image("Albert-Camus")
                .resizable()
                .scaledToFill()
                .frame(width: 100, height: 100, alignment: .center)
                .clipShape(Circle())
            Image(systemName: "alarm.fill")
                .resizable()
                .scaledToFill()
                .frame(width: 100, height: 100, alignment: .center)
            Image("Albert-Camus")
                .frame(width: 100, height: 100, alignment: .center)
                .mask(Image(systemName: "alarm.fill")
                    .resizable()
                    .scaledToFit())
                .shadow(radius: 20)
```

```
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

2. Demo

5.3 Button

1. Advanced Attributes [8]

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        VStack(spacing: 5.0) {
            Text("Customized␣Button")
                .font(.body)
                .foregroundColor(.white)
                .background(Color.orange)
            Button(action: {
                print("Hello␣button␣tapped!")
            }) {
                Text("HI␣HI")
                    .fontWeight(.bold)
                    .font(.title)
                    .foregroundColor(.purple)
                    .padding()
                    .border(Color.purple, width: 5)
            }
            Button(action: {
                print("Hello␣button␣tapped!")
            }) {
                Text("Press␣me")
```

[8]SwiftUI 小技巧：利用 border 修飾符　輕鬆為按鈕或文本繪製邊框

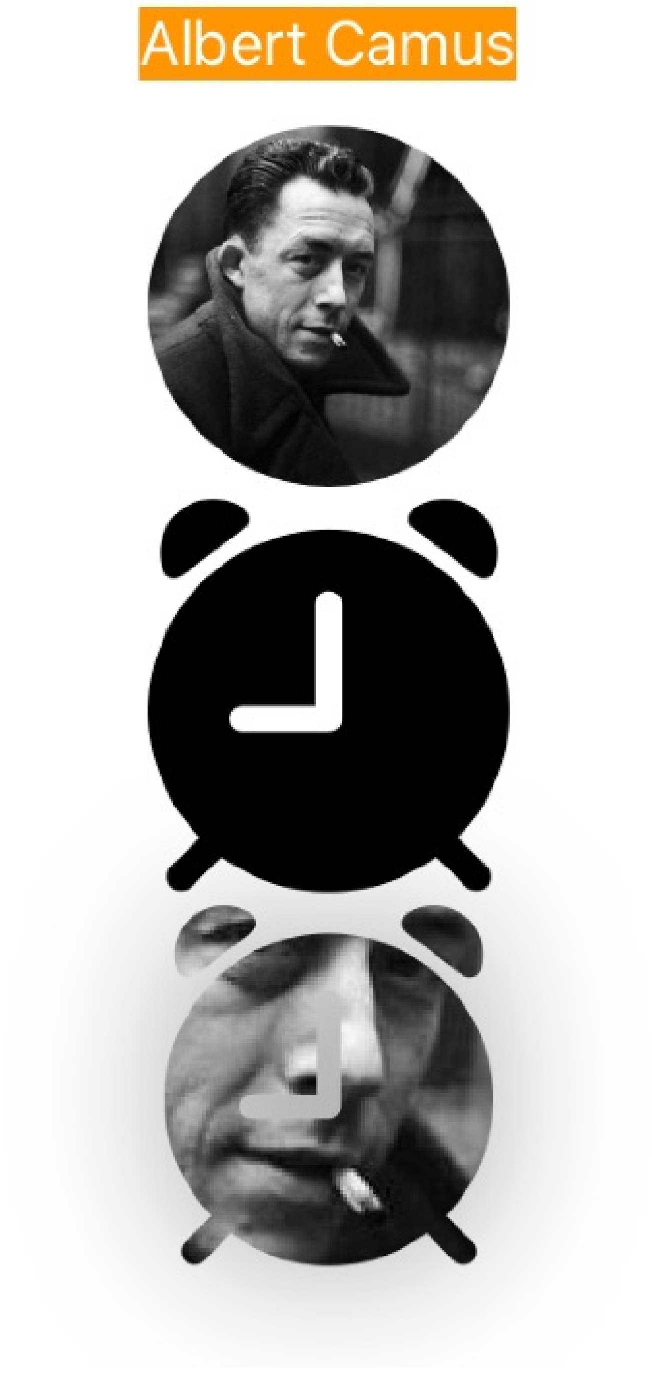


Figure 8: Image Attributes

```
                        .fontWeight(.light)
                        .font(.title)
                        .foregroundColor(.green)
                        .padding(5)
                        .overlay(
                            Capsule(style: .continuous)
                                .stroke(Color.green, style:
                                  StrokeStyle(lineWidth: 3, dash:
                                  [10]))
                    )
                }
            }
        }
}
```

2. Demo

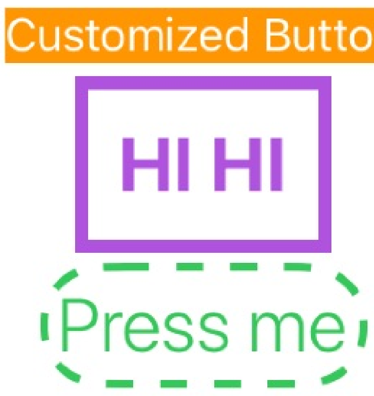


Figure 9: Button Attributes

5.4 Button, Divider

```
//
//  ContentView.swift
//  uitest
//
//  Created by yen yung chin on 2020/7/29.
//  Copyright ľ 2020 Letranger.tw. All rights reserved.
//

import SwiftUI
```

```

struct ContentView: View {
    @State private var a = ""
    @State private var b = ""
    @State private var c = "Ans:"

    var body: some View {
        VStack {
            VStack {
                Divider()
                TextField("Number␣1:␣", text: $b)
                Divider()
                TextField("Number␣2:", text: $a)
                Divider()
                Button(" ") {
                    let one = Int(self.a) ?? 0
                    let two = Int(self.b) ?? 0
                    self.c = "Ans:␣" + String(one + two)
                }
                .frame(width: 40, height: 30, alignment: .center)
                .foregroundColor(.white)
                .background(Color.green)
                .font(.largeTitle)
                Divider()
                Text(verbatim: c)
                    .foregroundColor(.gray)
            }
            .frame(width: 200, height: 160, alignment: .center)
        }
    }
}
```

5.5 background, opacity

```
//
//  ContentView.swift
//  uitest
//
//  Created by yen yung chin on 2020/7/29.
```

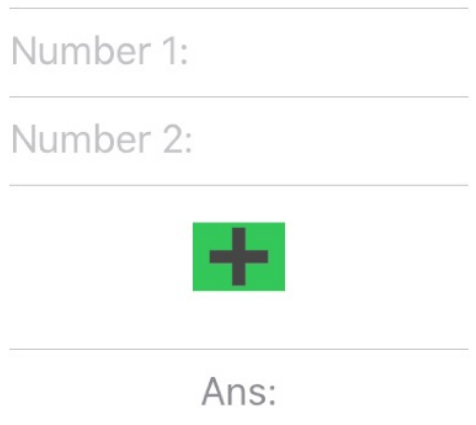


Figure 10: Button

```
//  Copyright Ĩ 2020 Letranger.tw. All rights reserved.
//

import SwiftUI

struct ContentView: View {
    @State private var a = ""
    @State private var b = ""
    @State private var c = "Ans:"

    var body: some View {
        VStack(alignment: .center) {
            Text("計算機")
            Divider()
            TextField("Number 1: ", text: $b)
            Divider()
            TextField("Number 2:", text: $a)
            Divider()
            Button(" ") {
                let one = Int(self.a) ?? 0
                let two = Int(self.b) ?? 0
                self.c = "Ans: " + String(one + two)
            }
            .frame(width: 40, height: 30, alignment: .center)
            .foregroundColor(.white)
            .background(Color.white)
            .font(.largeTitle)
```

```
                Divider()
                Text(verbatim: c)
                    .foregroundColor(.black)


            }
            .padding(60)
            .background(Image("background").resizable().scaledToFill())
            .opacity(0.9)
        }
}
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

6 List

6.1 What is List

```
struct ContentView: View {
    var body: some View {
        List {
            Text("Hello␣world.")
            Text("Hello␣world.")
            Text("Hello␣world.")
        }
    }
}
```

6.2 準備單一 cell 格式

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        HStack {
```

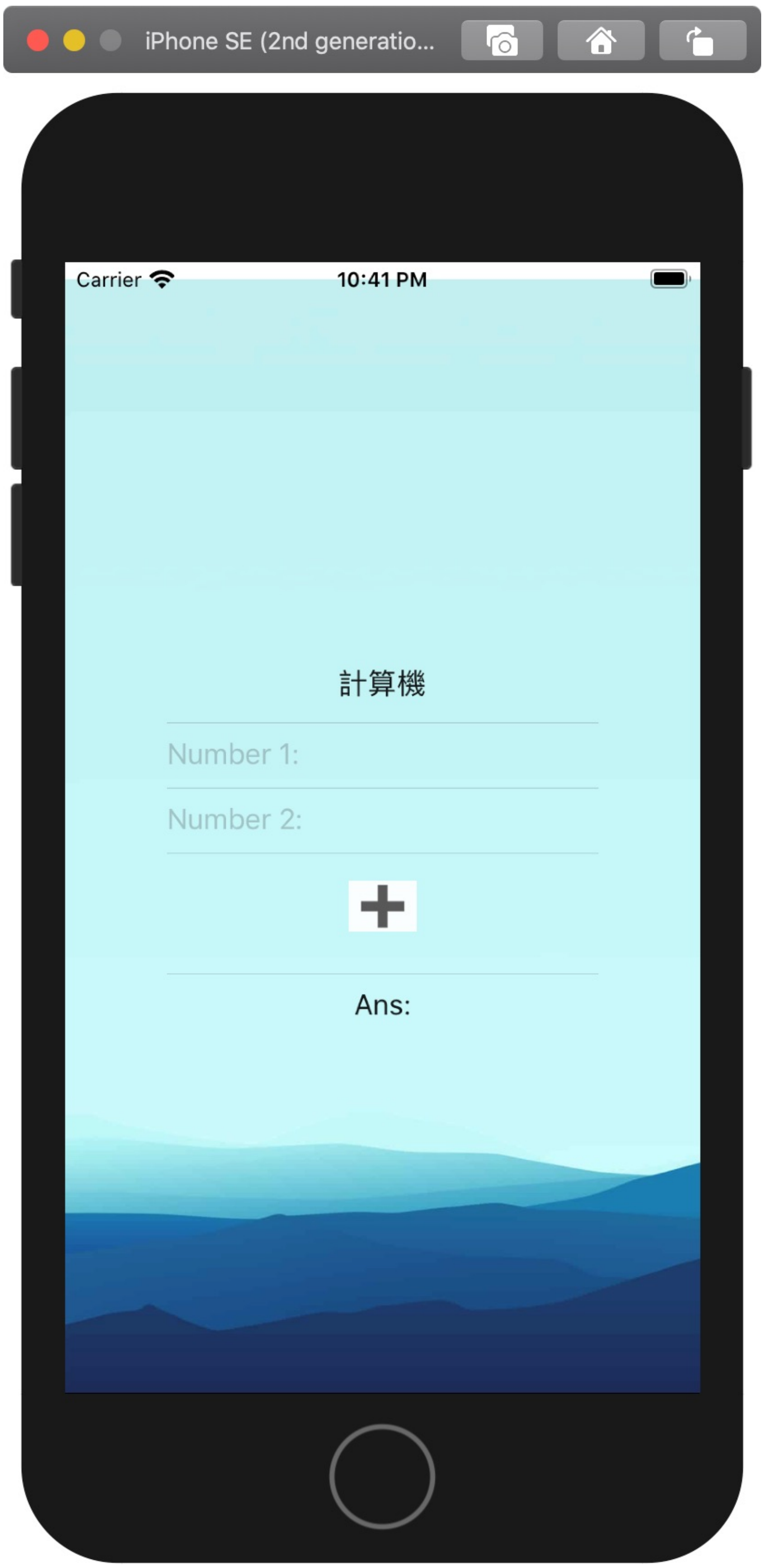


Figure 11: Background

```
            Image(systemName: "book")
                .resizable()
                .frame(width: 30, height: 30, alignment: .center)
            VStack(alignment: .leading) {
                Text("Artificial␣Intelligence:␣A␣Modern␣Approach")
                    .multilineTextAlignment(.leading)
                    .foregroundColor(Color.green)
                Text("Stuart␣Russell␣and␣Peter␣Norvig")
                    .multilineTextAlignment(.leading)
                    .foregroundColor(Color.orange)
            }
        }
    }
}
```

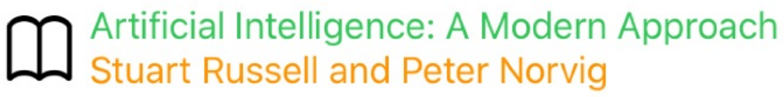
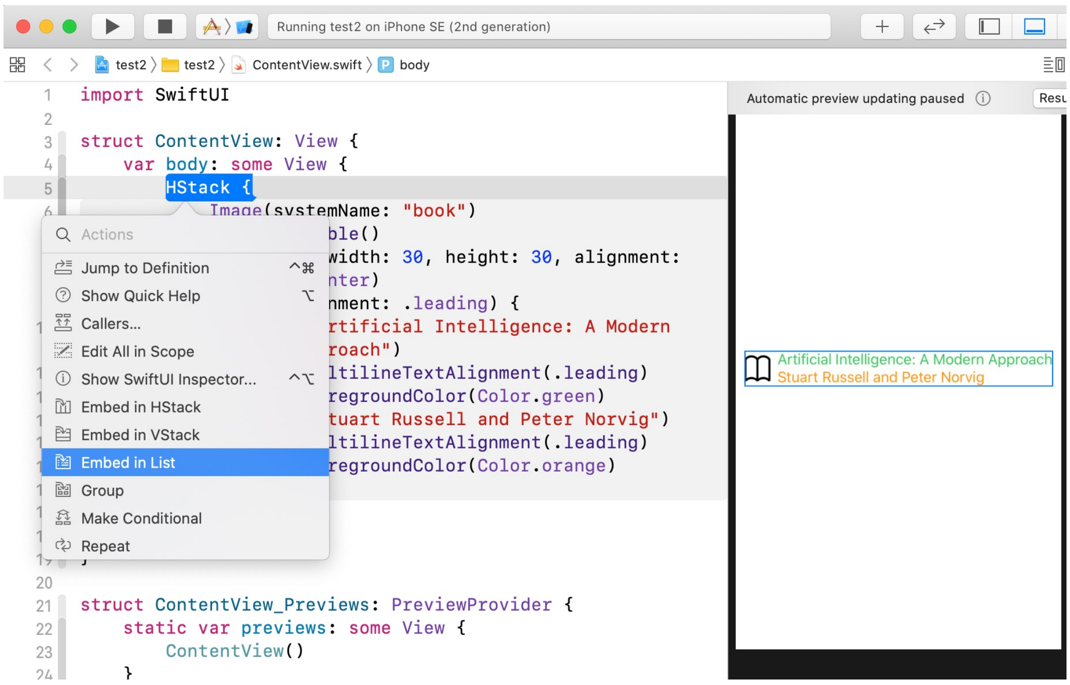

Artificial Intelligence: A Modern Approach
Stuart Russell and Peter Norvig

Figure 12: Single cell

6.3 轉入 List 格式 (靜態 List)

1. 將最外層的 VStack 加入 List 中
2. list 語法

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        List(0 ..< 5) { item in
            Image(systemName: "book")
                .resizable()
                .frame(width: 30, height: 30, alignment: .center)
            VStack(alignment: .leading) {
                Text("Artificial␣Intelligence:␣A␣Modern␣Approach")
                    .multilineTextAlignment(.leading)
                    .foregroundColor(Color.green)
```

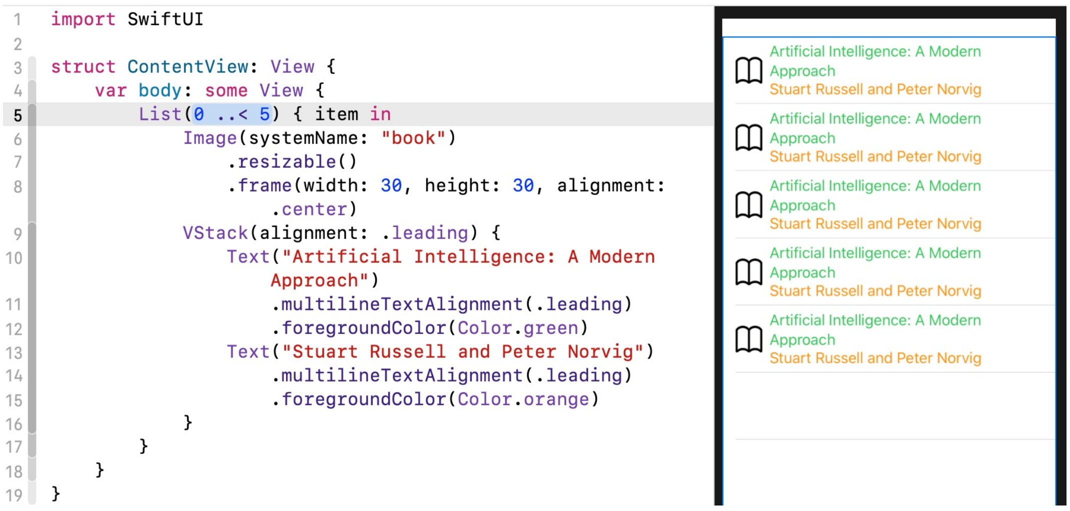

Running test2 on iPhone SE (2nd generation)
test2 〉 test2 〉 ContentView.swift 〉 body
Automatic preview updating paused
Resu
Actions
Jump to Definition ^⌘
Show Quick Help ⌥
Callers...
Edit All in Scope
Show SwiftUI Inspector... ^⌥
Embed in HStack
Embed in VStack
Embed in List
Group
Make Conditional
Repeat
Artificial Intelligence: A Modern Approach
Stuart Russell and Peter Norvig

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        HStack {
            Image(systemName: "book")
                                  ble()
                                  width: 30, height: 30, alignment:
                                  nter)
                                  nment: .leading) {
                                  rtificial Intelligence: A Modern
                                  roach")
                                  ltilineTextAlignment(.leading)
                                  regroundColor(Color.green)
                                  tuart Russell and Peter Norvig")
                                  ltilineTextAlignment(.leading)
                                  regroundColor(Color.orange)

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
```

Figure 13: List-1

```
                Text("Stuart␣Russell␣and␣Peter␣Norvig")
                    .multilineTextAlignment(.leading)
                    .foregroundColor(Color.orange)
            }
        }
    }
}
```

3. 結果


Artificial Intelligence: A Modern Approach
Stuart Russell and Peter Norvig
Artificial Intelligence: A Modern Approach
Stuart Russell and Peter Norvig
Artificial Intelligence: A Modern Approach
Stuart Russell and Peter Norvig
Artificial Intelligence: A Modern Approach
Stuart Russell and Peter Norvig
Artificial Intelligence: A Modern Approach
Stuart Russell and Peter Norvig

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        List(0 ..< 5) { item in
            Image(systemName: "book")
                .resizable()
                .frame(width: 30, height: 30, alignment:
                    .center)
            VStack(alignment: .leading) {
                Text("Artificial Intelligence: A Modern
                    Approach")
                    .multilineTextAlignment(.leading)
                    .foregroundColor(Color.green)
                Text("Stuart Russell and Peter Norvig")
                    .multilineTextAlignment(.leading)
                    .foregroundColor(Color.orange)
            }
        }
    }
}
```

Figure 14: List-2

6.4 建立 list 來源資料 (動態 List) [3]

In order to handle dynamic items, you must first tell SwiftUI how it can identify which item is which. This is done using the Identifiable protocol, which has only one requirement: some sort of id value that SwiftUI can use to see which item is which. [9]

```
import SwiftUI

//建立book struct
struct Book: Identifiable {
    var id = UUID()
    var title: String
    var author: String
    var image: String
}

struct ContentView: View {
    var books = [
      Book(id: UUID(), title: "地獄藍調", author: "李查德", image: "b1"),
      Book(id: UUID(), title: "至死方休", author: "李查德", image: "b2"),
      Book(id: UUID(), title: "一觸即發", author: "李查德", image: "b3"),
      Book(id: UUID(), title: "索命訪客", author: "李查德", image: "b4"),
      Book(id: UUID(), title: "闇夜回聲␣", author: "李查德", image: "b5")]

    //.....
}
```

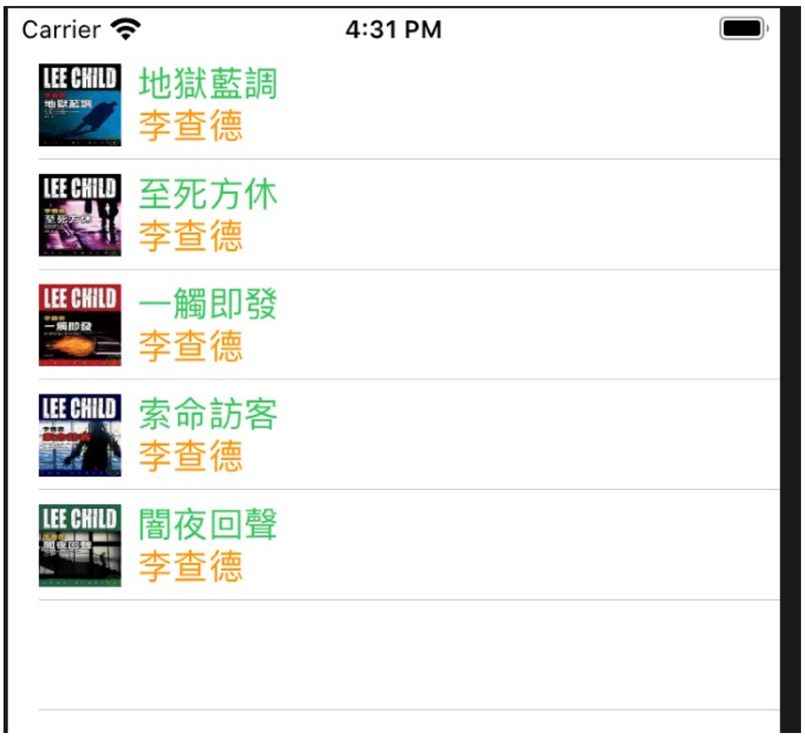
6.5 將資料連結到列表中 [3]

```
import SwiftUI

```

[9]How to create a list of dynamic items

```
    // ....
    var body: some View {
        List(books) { book in
            Image(book.image)
                .resizable()
                .frame(width: 40, height: 40, alignment: .center)
            VStack(alignment: .leading) {
                Text(book.title)
                    .multilineTextAlignment(.leading)
                    .foregroundColor(Color.green)
                Text(book.author)
                    .multilineTextAlignment(.leading)
                    .foregroundColor(Color.orange)
            }
        }
    }
}
```

6.6 結果


Carrier 4:31 PM

地獄藍調
李查德

至死方休
李查德

一觸即發
李查德

索命訪客
李查德

闇夜回聲
李查德

Figure 15: List-3

6.7 為什麼要加入 id 與 Identifiable

- Identifiable: 允許 Array 中有重複值
- id: 明確區分重複值
- UUID: 自動生成 unique 值
- 詳細說明如SwiftUI - Dynamic List & Identifiable

6.8 Reading source

- SwiftUI Basics: Dynamic Lists, HStack VStack, Images with Circle Clipped Stroke Overlays
- Building Lists and Navigation

7 Navigation between Views: Navigation Bar

7.1 Create New Views

1. in same file

```
import SwiftUI

struct BookDetailView: View {
    var body: some View {
        Text("This␣is␣the␣Book␣Detail␣View")
    }
}

struct ContentView: View {
    var body: some View {
        NavigationView {
            VStack {
                NavigationLink(destination: BookDetailView()) {
                    Text("GoToDetail")
                }
            }
            .navigationTitle("Book␣List")

        }
    }
}
```

2. in new file Create a new SwiftUI file to save BookDetailView

7.2 Navigation bar

- 於 body 中最外層的 component 之外加入 NavigationView

- Title: navigationBarTitle(), title 置於 NavigationView {} 內 attach 在 List 上，因為 NavigationView 主要負責於不同的 View 中切換，每個 View 都會有自己的 content 與 Title。

- DisplayMode 有三類：

  1. large: 適用於 top level

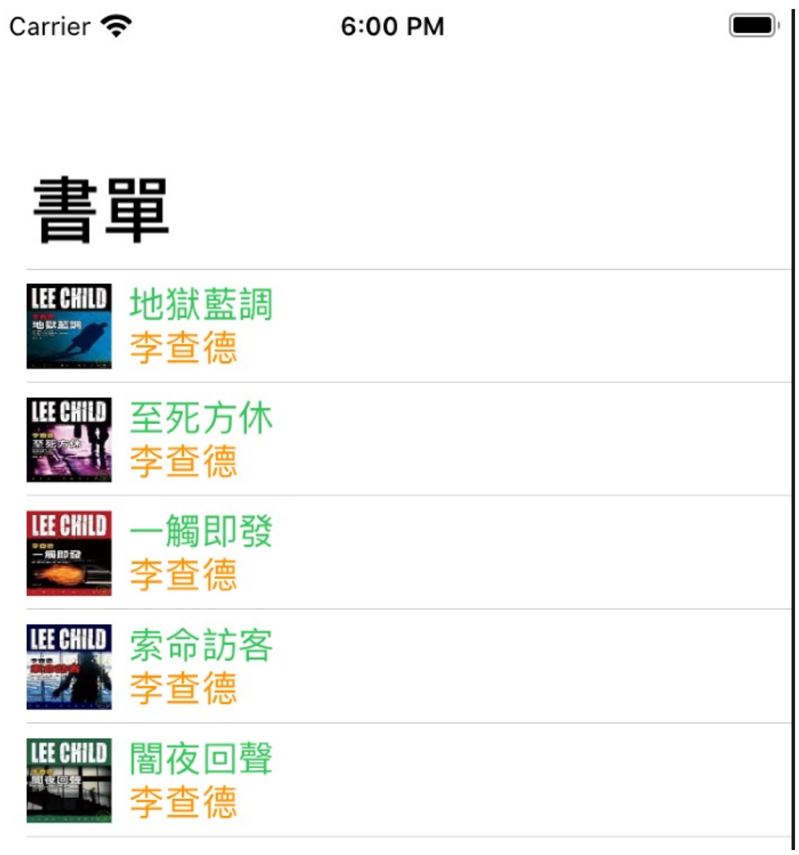
2. inline: 適用於 detail level

3. automatic: 自動判斷有無 parent level

```
import SwiftUI
  ...

struct ContentView: View {
    ....
    var body: some View {
        NavigationView {
            List(books) { book in
                ...
                }
            .navigationBarTitle(Text("書單"))
            .navigationBarTitleDisplayMode(.large)
            }
        }
    }
}
...
```

1. Hide Navigation Bar 使用 Hide 與不設定 BarTitle 的差異在於：後者仍會佔掉 Bar 的空間

```
import SwiftUI
  ...

struct ContentView: View {
    ....
    var body: some View {
        NavigationView {
            List(books) { book in
                ...
                }
            .navigationBarTitle(Text("書單"))
            .navigationBarTitleDisplayMode(.large)
            .navigationBarHidden(true)
            }
```



書單

地獄藍調
李查德

至死方休
李查德

一觸即發
李查德

索命訪客
李查德

闇夜回聲
李查德

Figure 16: Navigation bar

```
            }
        }
}
...
```

(a) NavigationLink

i. Create

ii. 語法:

```
NavigationLink(<title: StringProtocol, destination:)
```

```
<stdin>:1:22: error: expected ',' separator
NavigationLink(<title: StringProtocol, destination:)
                     ^
                     ,
<stdin>:1:22: error: expected expression in list of expressions
NavigationLink(<title: StringProtocol, destination:)
                     ^
<stdin>:1:52: error: expected expression in list of expressions
NavigationLink(<title: StringProtocol, destination:)
                                                   ^
<stdin>:1:1: error: cannot find 'NavigationLink' in scope
```

```
NavigationLink(<title: StringProtocol, destination:)
^~~~~~~~~~~~~~
<stdin>:1:16: error: '<' is not a prefix unary operator
NavigationLink(<title: StringProtocol, destination:)
               ^
<stdin>:1:17: error: cannot find 'title' in scope
NavigationLink(<title: StringProtocol, destination:)
                ^~~~~
```

iii. **方式**

A. Single

```
struct ContentView: View {
    var body: some View {
        NavigationView {
            VStack {
                NavigationLink("JumpToSecond",
                    destination: SecondView())
                Text("Hello,␣world!")
                    .padding()
            }
            .navigationTitle("Book␣List")
            .navigationBarTitleDisplayMode(.large)
        }
    }
}
```

B. attach link to other object(Text in this example)

```
struct ContentView: View {
    var body: some View {
        NavigationView {
            VStack {
                NavigationLink(destination: SecondView())
                    {
                    Text("TextLink")
                        .padding()
                }
            }
            .navigationTitle("Book␣List")
```

```
                .navigationBarTitleDisplayMode(.large)
            }
        }
}
```

若是將 link attach 至 image，則要加上 renderingMode，否則會看不到圖，例：

```
struct ContentView: View {
    var body: some View {
        NavigationView {
            VStack {
                NavigationLink(destination: SecondView())
                    {
                    Image(systemName: "myImage")
                        .renderingMode(.original)
                }
            }
            .navigationTitle("Book␣List")
        }
    }
}
```

(b) Reading

- The Complete Guide to NavigationView in SwiftUI

8 Sharing Data between Views

Three ways for sharing data in SwiftUI[10]

[10]iOS 13 SwiftUI Tutorial: Interactively Transition and Share Data between Views with SwiftUI

@State	@ObservedObject	@EnvironmentObject
Simple properties like *String* or *Int*	Can be shared across views	Similar to @ObservedObject
Belongs to a specific view	More complex properties (e.g. custom type)	Possiblity to make it available to all views through the application itself
Never used outside that view	External reference type that has to be managed (Create an instance of the class, create its own properties, ...)	If one view changes the model all views update
	Class should conform to *ObservableObject*	
	@Published property wrapper used to mark properties that should force a view to refresh	

9 Navigation between View: Tabbed View

9.1 Create subView

```
// create StoreView, AboutView, NewsView first

struct ContentView: View {
    @ObservedObject var user = User()

    init() {
        UITabBarItem.appearance().setTitleTextAttributes([.font:
           UIFont.systemFont(ofSize: 16) ], for: .normal)
    }

    var body: some View {
        TabView {
            StoreView().tabItem {
                Image(systemName: "cart.fill.badge.plus")
                Text("購買")
            }
            AboutView().tabItem {
                Image(systemName: "person.3")
```

```
                    Text("關於")
                }
                NewsView().tabItem {
                    Image(systemName: "message")
                    Text("消息")
                }
            }.accentColor(.pink)      }
}
```

9.2 Change tabView font size

```
    init() {
        UITabBarItem.appearance().setTitleTextAttributes([.font:
            UIFont.systemFont(ofSize: 14) ], for: .normal)
    }
```

9.3 change tabView lable color

```
.accentColor(.pink)
```

9.4 Reading Resources

Tabbed View SwiaftUI - TabBar Tutorial & Basic Customization - Xcode 11 - 2019

10 Passing data between Views

10.1 Passing parameter

Single way, Just pass variable from A to B.

```
struct BookDetailView: View {
    var operation: String
    var body: some View {
        Text("已為你\(operation)這本書")
    }
}

struct ContentView: View {
    var body: some View {
        NavigationView {
            VStack {
                NavigationLink(destination: BookDetailView(operation
                    : "借閱")) { Text("Loan")
                }
                NavigationLink(destination: BookDetailView(operation
                    : "續借")) { Text("Renew")
                }
                NavigationLink(destination: BookDetailView(operation
                    : "歸還")) { Text("Return")
                }
            }
            .navigationTitle("Book␣List")

        }
    }
}
```

10.2 with @State and @Binding

if you have view A with data that view B will also be using, you can pass data by creating a @State in view A and declaring the same variable with @Binding declaration in view B[11]

[11]Go to a new view using SwiftUI

```
struct ViewA : View {
    @State var myItems: [Items]
    var body: some View {
        NavigationView {
            VStack {
                NavigationButton(destination: ViewB(items: $myItems)
                    ) {
                    Text("Go␣To␣ViewB")
                }
            }
        }
    }
}
```

```
struct ViewB : View {
    @Binding var myItems: [Items] //連結回到parent viewl 的myItems
    var body: some View {
        NavigationView {
            List{
                ForEach(myItems.identified(by: \.self)) {
                    Text($0.itemName)
                }
            }.navigationBarTitle(Text("My␣Items"))
        }
    }
}
```

10.3 with @ObservableObject, @EnvironmentObject and @ObservedObject

```
import SwiftUI

class User: ObservableObject {
    @Published var score = 0
}

struct BuyBookView: View {
    @EnvironmentObject var user: User

    var body: some View {
```

```
11            Button("加購!␣數量\(self.user.score)") {
12                self.user.score += 1
13            }
14        }
15 }
16
17 struct ContentView: View {
18        @ObservedObject var user = User()
19
20        var body: some View {
21            NavigationView {
22                VStack {
23                    Text("mount:␣\(user.score)")
24                    NavigationLink(destination: BuyBookView() ){
25                        Text("Show␣amount,␣now")
26                    }
27                }
28                .navigationTitle("Book␣List")
29            }
30            .environmentObject(user)
31        }
32 }
```

第 30 行的 modifier attach 於 NavigationView，所以所有以 Navigation 連接的 View 均可共享此變數。若將此 modifier attach 在 BuyBookView() 後 (第 24 行)，則只有這個 View 可存取此一變數。

11 TODO MVVM

以 "推薦書單" 的 APP 為例:

- Model: 包含書名、作者、出版社....，而實際的資料來源可能是雲端資料庫 (Firebase)、Web API、本機資料庫 (Core data)。

```
struct Book {
    let title: String
    let author: String
    let dateReleased: String
    let publisher: String
    let isFavorite: Bool
}
```

- View: 在 APP 畫面上呈現 Model 中資料的元件，如 Text, Image, Button, List. . . ..
- ViewModel: 將 Model 中的資料取出, 供 View 呈現, 或是接受 View 輸入的資料, 存回 Model。以 "推薦書單 APP" 為例，其 ViewModel 可能包含如下 struct:

```
struct BookDetailViewModel {
    var book: Book

    var isFavorite: Bool

    init(book: Book) {
        self.book = book
        self.isFavorite = false
    }

    var title: String {
        return self.book.title
    }

    var author: String {
        return self.book.author
    }

    var dateReleased: String {
        return self.book.dateReleased
    }

    var publisher: String {
        return self.book.publisher
    }
}
```

如果在 View 上有一個 Favorite Button，則當 user 點了 Favorite 後，ViewModel 應負責將 struct 中的 isFavorite 改存 True，並回存至 Model 中。Model 的資料只能透過 ViewModel 來新增刪除，View 無法直接染指。
Model 與 UI 完全無關，單純用來儲存資料，ViewModel 為 Model 與 View 溝通的橋樑。

11.1 View+ViewModel

- SwiftUI Tip Calculator Using MVVM Design Pattern
- Understanding MVVM Design Pattern: 講的超清楚

11.2 DEMO

- Video: MVVM SwiftUI - Model View ViewModel Pattern - Getting Started
- GitHub: `https://github.com/rebeloper/SwiftUIMVVM.git`

11.3 DICE DEMO

12 **TODO** Protocols

對任何程式開發來說，減少重覆的 code，把權責明確分開，讓 code 維護性變好，是非常重要的課題。而在現今的軟體開發模式中，有許多方法可以做到這點，最為人所知的一個模式，就是利用繼承 (Inheritance)，把會重覆利用的部份放在母類別，讓其它子類別去繼承。另外一種做法，則是利用 Composition Pattern，將功能做成組件分出來，讓需要的模組去組合取用。[12]

> A protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. The protocol can then be adopted by a class, structure, or enumeration to provide an actual implementation of those requirements. Any type that satisfies the requirements of a protocol is said to conform to that protocol. [13]

協定提供類型可以做的資訊，Classes 和 structs 則提供物件的資訊，協定則提供物件將會執行的動作。[14]

> 協定是 Swift 一個重要的特性，它會定義出為了完成某項任務或功能所需的方法、屬性，但是本身不會實作這些任務跟功能，而僅僅只是表達出該任務或功能的名稱。協定為方法、屬性、以及其他特定的任務需求或功能定義藍圖。協定可被 class、struct、或

[12]利用 Protocol Extension 減少重覆的 Code 大大增強 Code 的維護性

[13]Protocols

[14]Swift 開發指南：Protocols 與 Protocol Extensions 的使用心法

enum 類型採納以提供所需功能的具體實現。滿足了協定中需求的任意類型都叫做遵循了該協定。

除了指定遵循類型必須實現的要求外，你可以擴展一個協定以實現其中的一些需求或實現一個符合類型的可以利用的附加功能。[15]

12.1 範例

1. 版本 1 本例中有兩個 struct: Song, Album 以及一個 class 用來播放 Song 或 Album，原本的 Player 要為不同的 struct 寫不同的 func，而且程式碼大多重複。

```
import Cocoa
import AVKit

struct Song {
    var name: String
    var album: Album
    var audioURL: URL
    var isLiked: Bool
}

struct Album {
    var name: String
    var imageURL: URL
    var audioURL: URL
    var isLiked: Bool
}

class Player {
    private let avPlayer = AVPlayer()

    func play(_ song: Song) {
        let item = AVPlayerItem(url: song.audioURL)
        avPlayer.replaceCurrentItem(with: item)
        avPlayer.play()
    }

```

[15] Day-29 Swift 語法 (25) - 協定 Protocol

```
    func play(_ album: Album) {
        let item = AVPlayerItem(url: album.audioURL)
        avPlayer.replaceCurrentItem(with: item)
        avPlayer.play()
    }
}
```

2. 版本 2 宣告一個 protocol，定義 audioURL 變數 (read only)，然後令兩個 struct 皆遵循該 protocol(方式有二)，如此，原本的 Player class 中的 play func 就能只寫一次。

```
import Cocoa
import AVKit

protocol Playable {
    var audioURL: URL { get }
}

struct Song: Playable {
    var name: String
    var album: Album
    var audioURL: URL
    var isLiked: Bool
}

struct Album {
    var name: String
    var imageURL: URL
    var audioURL: URL
    var isLiked: Bool
}

extension Album: Playable {}
class Player {
    private let avPlayer = AVPlayer()

    func play(_ resource: Playable) {
        let item = AVPlayerItem(url: resource.audioURL)
        avPlayer.replaceCurrentItem(with: item)
        avPlayer.play()
    }
}
```

3. 版本 3 原本 protocol 的真正意思其實只是在確定 audioURL 是否能正確轉換成 Audio，所以其實將 protocol name 由 Playable 改為 AudioURLConvertable 會更貼近事實。

```
import Cocoa
import AVKit

protocol AudioURLConvertable {
    var audioURL: URL { get }
}

struct Song: AudioURLConvertable {
    var name: String
    var album: Album
    var audioURL: URL
    var isLiked: Bool
}

struct Album: AudioURLConvertable {
    var name: String
    var imageURL: URL
    var audioURL: URL
    var isLiked: Bool
}

class Player {
    private let avPlayer = AVPlayer()

    func play(_ resource: AudioURLConvertable) {
        let item = AVPlayerItem(url: resource.audioURL)
        avPlayer.replaceCurrentItem(with: item)
        avPlayer.play()
    }
}
```

12.2 mutating

protocol 除了可以提供傳回值型態的彈性，也可以用來變更 class/struct 中的屬性。如：

```
import Cocoa
import AVKit

protocol Likeable {
    mutating func markAsLiked()
}

struct Song {
    var name: String
    var album: Album
    var audioURL: URL
    var isLiked: Bool
}

struct Album {
    var name: String
    var imageURL: URL
    var audioURL: URL
    var isLiked: Bool
}

extension Song: Likeable {
    mutating func markAsLiked() {
        isLiked = true
    }
}
```

可以在不改變原 struct Album 的情況下，藉由 extension 來擴充 Song，使其遵循 Likeable protocol，提供變供屬性 isLiked 的值，* 這在擴充 API 功能時特別有用 *。

12.3 擴充 protocol

除了擴充現有 struct，protocol 也可以用來擴充 protocol，如：

```
import Cocoa
import AVKit

protocol Likeable {
    var isLiked: Bool {get set}
}

extension Likeable {
    mutating func markAsLiked() {
        isLiked = true
    }
}

struct Song {
    var name: String
    var album: Album
    var audioURL: URL
    var isLiked: Bool
}

struct Album {
    var name: String
    var imageURL: URL
    var audioURL: URL
    var isLiked: Bool
}

extension Song: Likeable {}
extension Album: Likeable {}
```

13 **TODO** Web API and JSON

- iOS Swift Tutorial: Use Web APIs and JSON Data with Swift 5

Develop in MVVM

14 Property Wrappers

All of these @Something statements are property wrappers. A property wrapper is actually a struct. These structs encapsulate some “template” behavior applied to the vars they wrap.

The property wrapper feature adds “syntactic sugar” to make these structs easy to create/use.“ [16]

14.1 Property wrapper syntactic Sugar

```
@Published var dice: Dice = Dice()
```

上述宣告實際同以下 struct

```
struct Published {
    var wrappedValue: Dice
    var projectedValue: Publisher<Dice, Never>
}
```

接下來 Swift **產生以下變數**

```
var _dice: Published = Published(wrappedValue: Dice())
var dice: Dice {
    get { _dice.wrappedValue  }
    set { _dice.wrappedValue = newValue }
}
```

14.2 @State

- Simple properties like String or Int
- Belongs to a specific view
- Never used outside that view
- The wrappedValue is: anything (but almost certainly a value type).

[16] Lecture 9: Data Flow

- What it does: stores the wrappedValue in the heap; when it changes, invalidates the View.
- Projected value (i.e. $): a Binding (to that value in the heap).

14.3 @ObservedObject

- Can b shared across views
- More complex properties (e.g custom type)
- External reference type that has to be managed (Create an instance of the class, create its own properties, ...)
- Class should confrom to ObservableObject
- @Published property wrapper used to mark properties that should force a view to refresh
- The wrappedValue is: anything that implements the OvservableObject protocol (ViewModels basicly).
- What is does: invalidates the View when wrappedValue does objectWillChange.send().
- Projected value (i.e. %): a Binding (to the vars of the wrappedValue (a ViewModel)). You can bind a variable in your View to the variable in your ViewModel with @ObservedObject.

14.4 @EnvironmentObject

- Similar to @ObservedObject
- Possibility to make it available to all views through the application itself
- If one view changes the model all views updatezz
- ThewrappedValue is: ObservvableObject obtained via .environemntObject() sent to the View.

- What is does: invalidates the View when wrappedValue does objectWillChange.send().
- Projected value (i.e. $): a Binding (to the vars of the wrappedValue (a ViewModel)).

14.5 @Binding

- The wrappedValue is: a value that is bound to something else.
- What it does: gets/sets the value of the wrappedValue from some other source.
- What it does: when the bound-to value changes, it invalidates the View.
- Projected value (i.e. $): a Binding (self; i.e. the Binding itself)

14.6 Time to use Binding

Bindings are all about having a single source of the truth (data)!.

- Getting text out of a TextField
- Using a Toggle or other state-modifying UI element
- Finding out which ittem in a NavigationView was chosen.
- Find out whether we're being targeted with a Drag
- Binding our gesture to the .updating function of a gesture.

14.7 Demo: @State v.s. @Bidning

透過 @State 與 @Bidning, ContentView.swift 可以將變數 switchIsOn pass 給 SwitchView.swift, 而後者可以藉由更改變數值來改變 ContentView.swift 的顯示結果。

1. ContentView.swift

```
import SwiftUI

struct ContentView: View {
    @State var switchIsOn = false

    var body: some Vie {
        VStack {
            Text(switchIsOn ? "-_-" : "^_^")
            SwitchView(switchIsOn: $switchIsOn)
        }
    }
}
```

2. SwitchView.swift

```
import SwiftUI

struct SwitchView: View {
    @Binding var swtichIsOn: Bool

    var body: some View {
        Toggle(isOn: $switchIsOn, label: {
            Text(switchIsOn ? "ON" : "OFF")
        })
    }
}
```

14.8 Demo: ObservaleObject

1. UserStats.swift

```
import Foundation
import SwiftUI
import Combine

class UserStats: ObservableObject {
    var objectWillChange = ObservableObjectPublisher()

    var score = 0 {
        willSet {
            self.objectWillChange.send()
        }
    }
}
```

2. ScoreView.swift

```
import SwiftUI

struct ScoreView: View {
    @ObservedObject var userStats = UserStats()

    var body: some View {
        VStack {
            Text("\self.userStats.score")
            Button(action: {self.userStats.score += 1}
                   , label: {
                       Text("Add␣Point")
                   })
        }
    }
}
```

14.9 Demo: EnvironmentObject

1. UserSettings.swift (ObservableObject)

```
import SwiftUI

class UserSettings: ObservableObject {
    @Published var name = ""
    // use this ObservableObject as an environment object
}
```

2. SceneDeleate.swift add new var in SceneDelegate

```
//....

var settings: UserSettings()
func scene(.......) {
    //....
    //let tabbedView = TabbedView()
    let tabbedView = TabbedView().environemntObject(settings)
}
//....
```

and now the settings is universally available throughtout the tabbed views

3. UserSettingsView.swift

```
import SwiftUI

struct UserSettingsView: View {
    @EnvironmentObject var settings: UserSettings
    var body: some View {
        VStack {
            Text("My anme: \(settings.name)")
            EditView()
        }
    }
}
```

4. EditView.swift

```
import SwiftUI

struct EditView: View {
    @EnvironmentObject var settings: UserSettings
    var body: some Veiw {
        TextField("Type in your name:", text: $settings.name)
    }
}
```

14.10 進階閱讀

- SwiftUI Reactive Intro - Understanding State and Binding in SwiftUI in Xcode 11 (2019)
- 用狀態設計 SwiftUI 畫面—認識 @State property，binding & Toggle

15 **TODO** some

Adding the keyword some in front of a return type indicates that the return type is opaque. [17]

15.1 Generics

1. 問題 Generics 允許開發者在不同類型中複用你的程式碼，用來解決下列問題：

```
func swapInts(_ a: inout Int, _ b: inout Int) {
    let temporaryB = b
    b = a
    a = temporaryB
}

var num1 = 10
var num2 = 20

swapInts(&num1, &num2)
print(num1)     // 20
print(num2)     // 10
```

   但若想交換字串，則要寫成

```
func swapStrings(_ a: inout String, _ b: inout String) {
    let temporaryB = b
    b = a
    a = temporaryB
}
```

   可以發現除了參數之外，重複的 code 實在太多

2. 解決方案 將固定型態的參數轉為 Generic type

```
import Cocoa

func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let temporaryA = a
```

[17]What' s this "some" in SwiftUI?

```
    a = b
    b = temporaryA
}


var num1 = 10
var num2 = 20

swapTwoValues(&num1, &num2)
print(num1)    // 20
print(num2)    // 10
```

另一個例子為 Stack 的實作：

原本只能儲存 Int 的 Stack 如下，若要儲存字串則要再另行定義。

```
struct IntStack {
    var items = [Int]()
    mutating func push(_ item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
}
```

改為 Generic type 後可動態變更為整數 stack 或字串 stack，如：

```
struct Stack<Element> {
    var items = [Element]()
    mutating func push(_ item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
}

var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")
stackOfStrings.push("cuatro")
// the stack now contains 4 strings
```

15.2 opaque

帶有不透明（opaque）**返回類型的函數或方法，將會隱藏其返回值的類型**[18]

[18]Swift **程式語言**—Opaque Types

16 TODO UserDefaults

Ways to make data “persist” in iOS:

- Filesystem: FileManager
- SQL database: CoreData
- Cloud: ClodKit, Firebase
- UserDefualts

我們可以將之視為 persistent dictionary。UserDefaults **可以儲存** Property List **類型的資料**。Property List is not a protocol or a struct or anything tangible or Swift-like. It is any combination of String, Int, Bool, Floating point, Date, Array or Dictionary.

A powerful way to do this is using the Codable protocol inf Swift. Codable converts structs into Data objects.

16.1 Using userDefaults

```
let defaults = UserDefaults.standard
```

16.2 Storing Data

```
defaults.set(object, forKey: "SomeKey")
defautls.setDouble(37.5, forKey: "MyDouble")
```

16.3 Retrieving Data

```
let i: Int = defaults.integer(forKey: "MyInt")
let u: URL? = defaults.url(forKey: "MyURL")
let strings: [String]? = defaults.stringArray(forKey: "MyString")
```

17 **TODO** 進階主題

- 利用 Protocol Extension 減少重覆的 Code　大大增強 Code 的維護性
- 精通 Swift：列舉、閉包、泛型、Protocols 和高階函數