Swift App 進階教師專班

AI 機器學習與設備(BLE & IOT)通訊

Day 3 – Bluetooth 通訊 Central App (Part 1)

主辦：新北市政府教育局
日期：2023.07.26(三)
講師：友教有限公司 Michael

Mmmm

ObservedObject

在 SwiftUI 中， @ObservedObject 是一種觀察變化並更新畫面的屬性包裝器，用於觀察外部可觀察物件（Observable Object）的變化來更新畫面內容。這個外部可被觀察的物件必須符合 ObservableObject 的協議 (Protocol)。

接下來要使用 ObservedObject 的方式再實作一次『 首頁 』與『 設定頁面 』的範例，來認識 @ObservedObject 是如何觀察物件變化並更新畫面。

接下來的練習，從 SettingView_Obs_Template 開始，已預先做好『 首頁 』與『 設定頁面 』的程式碼如下：

ContentView　SettingView

```swift
import SwiftUI

//A. 建一個 DataModel 的 class 來儲存資料          步驟A

struct SettingView: View {
    @State var inputName = ""
    //B1. 宣告變數 dataModel                        步驟B
    var body: some View {
        VStack{
            TextField("Your Name", text: $inputName)
                .textFieldStyle(RoundedBorderTextFieldStyle()).padding()
            Button("Save"){
                //B2. 設定按鈕內容
            }
        }
    }
}
```

注意：Template 已將 SettingView 的 Previews 刪除，我們可以從 ContentView 可以連結到 SettingView 來預覽 SettingView()

ContentView　SettingView

```swift
import SwiftUI

struct ContentView: View {                          步驟C
    //C1.做一個 dataModel 的實例
    @State var name = "Michael"
    var body: some View {
        NavigationView {
            VStack {
                //C2. 修改 Text()
                Text("Hello, \(name)")
                .font(.largeTitle).padding()
                //C3. 將 dataModel 做為參數傳入 SettingView
                NavigationLink("Setting View", destination: SettingView())
            }
        }
    }
}
```

使用 @ObservedObject 來觀察 『 設定頁面 』中的資料更新，步驟可分為 A ～ C ：

步驟 A . 建立一個名為 DataModel 的 class 來儲存資料：

DataModel 須符合可觀察物件 (ObservableObject) 的協議，其中包含一個變數 name ，這個變數有自動發布的屬性 @Published，如果 name 的數值發生變化，將會通知給訂閱者，觸發相對應的更新操作。

```
class DataModel: ObservableObject {
    @Published var name = "Michael"
}
```

步驟 B . 修改 SettingView，讓輸入的內容可以更新到 DataModel：

B1 程式碼

接著在 SettingView 的畫面內，宣告一個變數 dataModel ，型別為 DataModel，先不給初始值，在 SettingView 被使用的地方再傳入初始值就可以。

```
var dataModel: DataModel
```

B2 程式碼

設定按鈕內容，當按鈕被按下後，將輸入內容 inputName 儲存至 dataModel.name 。

```
Button("Save"){
    dataModel.name = inputName
}
```

步驟 C . 修改 ContentView ：

C1 程式碼

做一個 dataModel 的實例，使用@ObservedObject 來觀察其變化，並將宣告的 name 變數刪除。

```
@ObservedObject var dataModel = DataModel()
```

C2 程式碼

修改畫面中的 Text() 將 name 改為 dataModel.name。

```
Text("Hello, \(dataModel.name)")
```

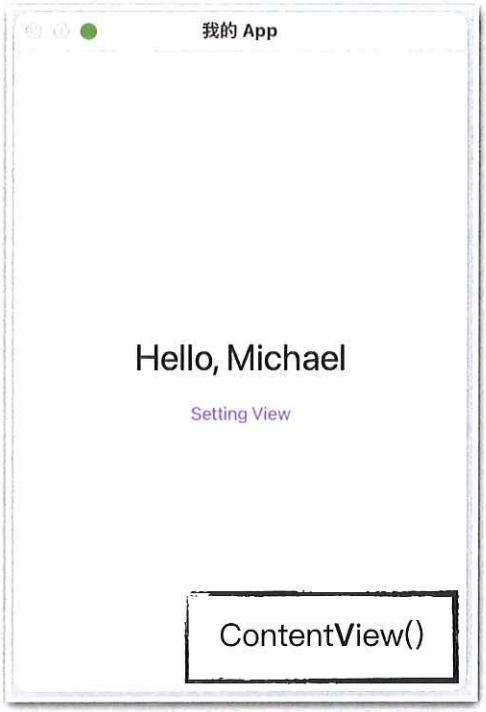
C3 程式碼

修改 SettingView() ，將 dataModel 做為參數傳入 SettingView 。

```
SettingView(dataModel: dataModel)
```

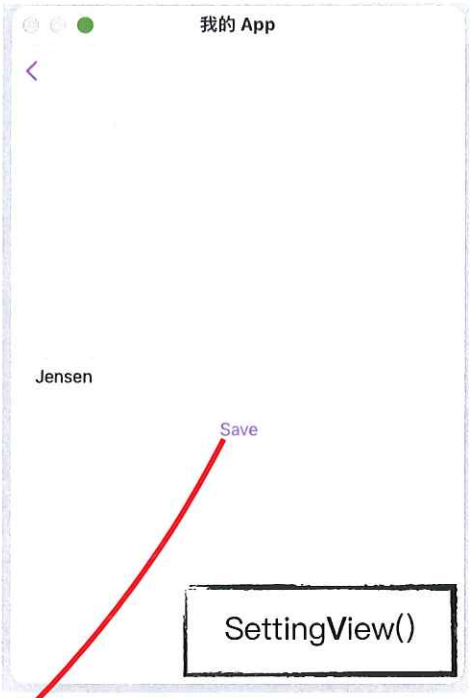
ContentView() 為觀察者，觀察可觀察物件

```
@ObservedObject var dataModel = DataModel()
```



SettingView 使用了可觀察物件

```
var dataModel: DataModel
```



按下 save 之後，
資料會經由 DataModel 更新發佈給 ContentView

DataModel 為可觀察物件
包含 @Published 變數

```
class DataModel: ObservableObject {
    @Published var name = "Michael"
}
```

章節筆記

語法疑問：

教學重點：

反思回饋：

WeatherCenter Delegate App

這邊我們要練習一個 App 專案 WeatherCenter Delegate App，模擬氣象觀測站觀測氣溫資料後，氣象中心自動發布觀測溫度訊息，或是發布高溫/低溫警報。這一個專案運用了這幾堂課所學的觀念，來熟悉 Delegate、 Protocol 與 @ObservedObject 的使用。

開始練習之前，我們將會拿到一個已經定義好 TempObs 這為一個 class ，與一個定義好的 TempObsDelegate 協議 (protocol)：

- TempObs 這個 class 是氣象觀測站在觀測氣溫資料。為了模擬觀測站的氣溫觀測，我們需要產生假的氣溫資料，並且每 10 秒發送一次這個隨機產生的假溫度資料。
- TempObsDelegate 這個 protocol 被已定義完成，當 WeatherCenter 為 TempObs 的 delegate 時需要採用的協議。

這個專案從 WeatherCenter_DelegateApp_Template 開始，完成這個專案，接下來可以分為兩大步驟：

步驟 A. 實作一個 WeatherCenter 的 class 。設定 WeatherCenter 為 TempObs 的 delegate ，並遵循 TempObsDelegate 的協議，此外 WeatherCenter 需要是一個 ObservableObject 物件。

步驟 B. 在 ContentView 內使用觀察物件 @ObservedObject 宣告一個 weatherCenter 為 WeatherCenter() 的實例，並將 weatherCenter 發布的資訊顯示在畫面上。

```swift
import SwiftUI

//步驟A
class WeatherCenter {  //A1.新增 WeatherCenter 遵循的 protocol
    //A2. 宣告兩個 @Published 變數

    //A3. 宣告一個 TempObs() 的實例，使用 init() 初始化

    //A4. 建立 TempObsDelegate 協定內的三個函數。

}

//步驟B
struct ContentView: View {
    var body: some View {
        VStack {

        }
    }
}
```

步驟 A

步驟 B

WeatherCenter Class

步驟A. 實作一個 WeatherCenter 的 class

將步驟A分為 A1 ~ A4，在 WeatherCenter 內依序編寫程式碼：

A1 程式碼

WeatherCenter 遵循 TempObsDelegate 的協議，且為一個 ObservableObject 物件。

```
3   class WeatherCenter: TempObsDelegate, ObservableObject {
```

Type 'WeatherCenter' does not conform to protocol 'TempObsDelegate'

```
4
5   }
```

在 WeatherCenter 後方加入 TempObsDelegate 的協議時，會出現一個錯誤，這是因為目前 WeatherCenter 並沒有遵循 TempObsDelegate 的協議。

要讓 WeatherCenter 遵循 TempObsDelegate ，在 WeatherCenter 內需要有三個協議內所規範的函數，如下圖：

（已定義的protocol）

```
protocol TempObsDelegate {
    func highTempAlert(temperature: Double)
    func lowTempAlert(temperature: Double)
    func noTempAlert(temperature: Double)
}
```

我們在 A4 程式碼 部分會**建立**這三個函數的內容。

A2 程式碼

宣告兩個 @Published 變數來**儲存**發布的警報資訊。

一個為字串，**儲存**最新的發布警報訊息 alertString ，另一個為字串陣列，**儲存**警報歷史訊息 historyAlerts。

```
@Published var alertString: String = ""
@Published var historyAlerts: [String] = []
```

A3 程式碼

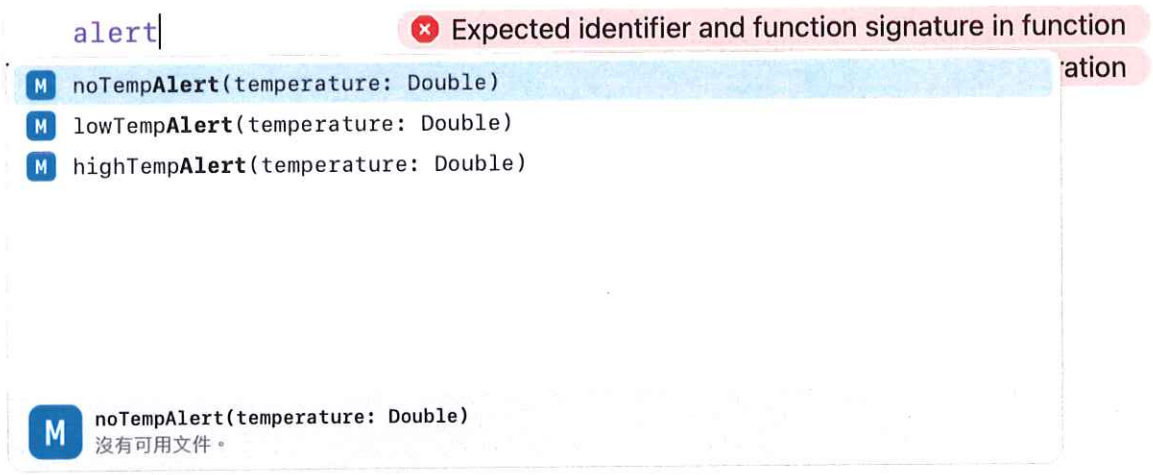
在 WeatherCenter 內宣告一個 TempObs() 的實例 tempObs ，並使用建構子 (initializer) init() 在初始化 WeatherCenter 時，將 WeatherCenter 設定為 tempObs 的 delegate，並開啟 tempObs 的計時器 startTimer()。

```
var tempObs = TempObs()
init(){
    tempObs.delegate = self
    tempObs.startTimer()
}
```

A4 程式碼

在 WeatherCenter 內，**建立** TempObsDelegate 協定內的三個函數。

輸入關鍵字，在選單可以找到 TempObsDelegate 內的三個方法，使用自動帶入功能來加入這三個函數：



加入的函數如下：

```
func noTempAlert(temperature: Double) {

}
func lowTempAlert(temperature: Double) {

}
func highTempAlert(temperature: Double) {

}
```

函數 noTempAlert(temperature:)

這個函數在氣溫正常的狀況下會被呼叫。

氣溫在正常範圍(10~36)時，將目前的氣溫資訊與時間儲存在 alertString 的變數內，並且將 alertString 新增至歷史訊息 historyAlerts 陣列內。

```
func noTempAlert(temperature: Double) {
    alertString = "Temperature: \(temperature)°C \n \(tempObs.timeString)"
    historyAlerts.append(alertString)
}
```

小提示：°C 的 ° ，可在 iPad 鍵盤上長按 0 ，會出現 ° 的選單。若使用鍵盤輸入，可使用 Shift + Option + 8 來輸入 。

在函數被呼叫時，目前氣溫(隨機產生的假資料)會被傳入這個函數做為參數，在函數內要使用這個假資料，可以使用 temperature 這個區域變數（local variable）得到。

alertString 內容包含目前溫度 temperature 與目前時間 tempObs.timeString，變數使用\()轉換後加入字串內，\n 表示在這個字串內換行。

historyAlerts.append(alertString) 使用 .append() 這個方法，讓 historyAlerts 陣列末端新增一個 alertString 元素。

以同樣的方式來設定下方兩個函數。

函數 lowTempAlert(temperature:)

（中央氣象局在氣溫低於 10 度以下會發布低溫警報）

此函數在氣溫低於 10 度時會被呼叫：

```
func lowTempAlert(temperature: Double) {
    alertString = "🥶🥶🥶😰\nLow Temperature Alarm! \n Temperature:
     \(temperature)°C \n \(tempObs.timeString)"
    historyAlerts.append(alertString)
}
```

函數 hightTempAlert(temperature:)

（中央氣象局在氣溫高達 36 度以上會發布高溫資訊）

此函數在氣溫高於 36 度時會被呼叫：

```
func highTempAlert(temperature: Double) {
    alertString = "🔥🔥🔥🥵\nHigh Temperature Alarm! \nTemperature:
     \(temperature)°C \n \(tempObs.timeString)"
    historyAlerts.append(alertString)
}
```

App 畫面與 ObservedObject

步驟 B. 在 ContentView 內設定觀察物件 @ObservedObject

App 畫面設定分為 4 個部分 B1 ~ B4 ：

B1 程式碼

在 ContentView 宣告變數區宣告可觀察物件 weatherCenter ，是一個 WeatherCenter() 的實例：

```
@ObservedObject var weatherCenter = WeatherCenter()
```

B2 程式碼

在 ContentView 的 body 內，放置一個 TabView，之後會在其中放兩個頁面，一個為目前發佈的最新警報訊息，另一個為歷史警報訊息。

```
TabView {

}
```

```
struct ContentView: View {
    @ObservedObject var weatherCenter = WeatherCenter()          B1
    var body: some View {
        TabView {                                                B2
            //最新警報訊息頁面
                                                                 B3
            //歷史警報訊息頁面
                                                                 B4
        }
    }
}
```

語法筆記：

B3 程式碼

最新警報訊息頁面，將放置在一個 VStack{ } 內。

weatherCenter.alertString 可以拿到 weatherCenter 這個物件內中 alertString 的內容。

當 tempObs 還沒有溫度資料時，weatherCenter 的函數不會被呼叫，alertString 此時為空字串，使用 if 來判斷如果為空字串 .isEmpty 時，顯示載入動畫 ProgressView()，否則顯示 Text(weatherCenter.alertString)。

```
VStack {
    if weatherCenter.alertString.isEmpty {
        ProgressView()
    } else {
        Text(weatherCenter.alertString)
    }
}.tabItem {••
    Label("Alert Page", systemImage:  "thermometer.low")
}
```

使用 .tabItem { } 設定分頁標籤，裡面放一個 Label 。

B4 程式碼

歷史警報訊息頁面，將使用 List 來列出所有資料 ：

```
List(weatherCenter.historyAlerts.reversed(), id: \.self) { alert in
    Text(alert)
}.tabItem {••
    Label("History Alert", systemImage: "doc.text")
}
```

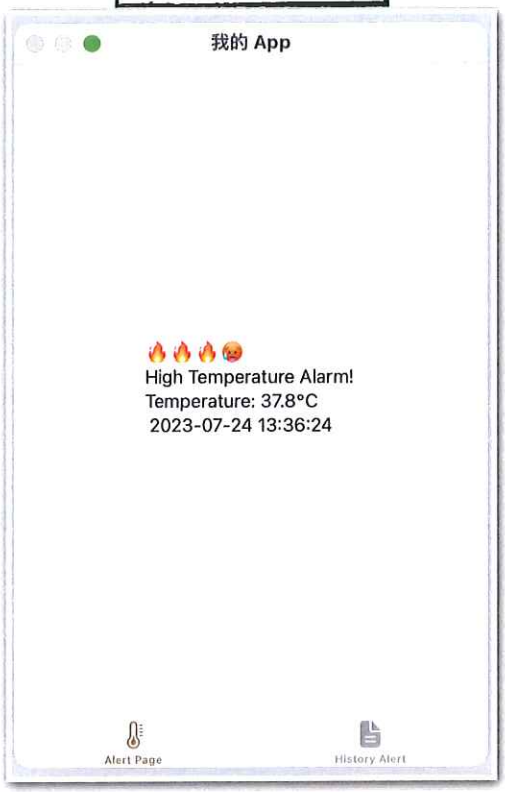
List 使用 closure 的寫法，將 weatherCenter.historyAlerts 每一個元素使用列表呈現，weatherCenter.historyAlerts 加上 .reversed() 後，會反轉整個 historyAlerts 陣列，將最後一個內容放在最上方。

在 List 的 closure 內使用 alert 這個變數拿到每一個weatherCenter.historyAlerts 的元素，並使用 Text(alert) 在列表內呈現。

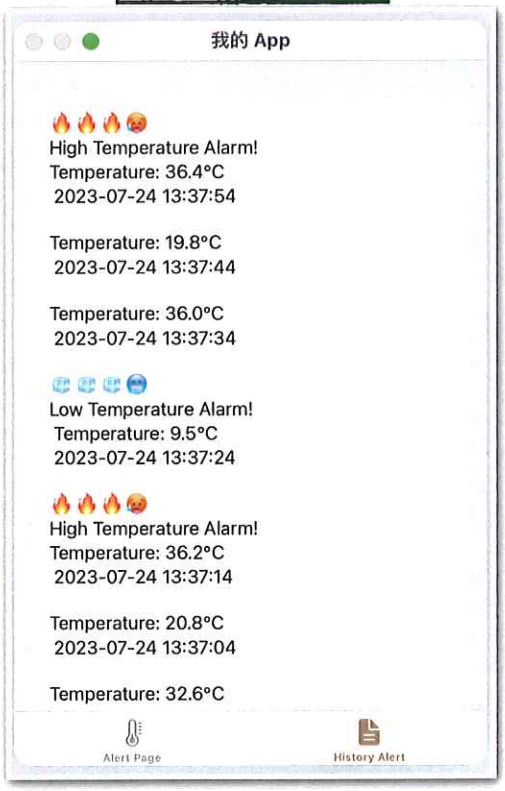
最後使用 .tabItem { } 來設定分頁標籤。

完成的 App 畫面如下：

最新警報訊息



歷史警報訊息



章節筆記

語法疑問：

教學重點：

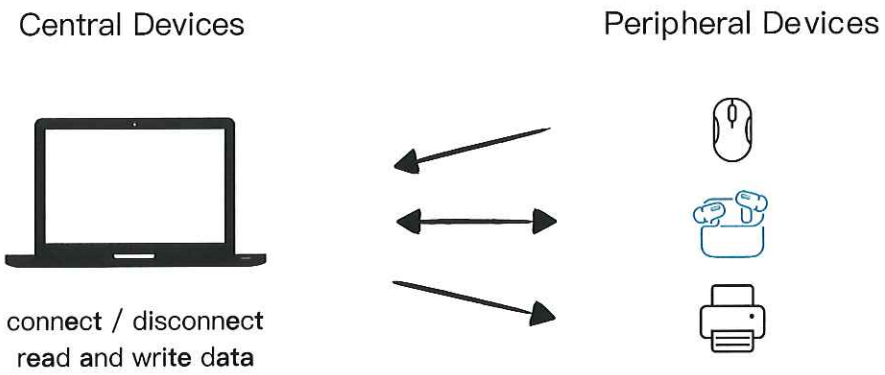
反思回饋：

Bluetooth 基本觀念

藍牙（Bluetooth）是一種無線通信技術，透過電磁波傳輸訊號，讓不同的電子設備能在短距離內彼此通信。這種技術主要用於將資訊在設備間進行傳輸，像是從手機傳輸音樂到無線耳機，或是從電腦傳輸文件到影印機等。

在藍牙系統中，我們會有「中心裝置」（central devices）和「週邊裝置」（peripheral devices）兩種角色。例如，手機將音樂資料傳送至耳機撥出，其中手機是中心裝置 central devices ，藍牙耳機式週邊裝置 peripheral devices。

一個「中心裝置」可以同時與多個「週邊裝置」連線。例如你的筆電可以連接一個藍牙滑鼠來操控電腦，同時可以連接耳機播放音樂，耳機可以控制音樂暫停或播放，還可以用藍牙連線印表機印文件資料。



「中心裝置」通常是擁有更強大處理能力且電源充足的裝置，例如智慧型手機或筆電。它的主要職責是發起與管理連接，也可以同時與多個週邊裝置連接和交換數據。「週邊裝置」通常是較小型且功能較單一的裝置，且常需要低耗能的設計（藍牙低能量 Bluetooth Low Energy, BLE），例如無線耳機、無線滑鼠。它們主要的職責是發送資料或接收從中心裝置發來的數據，通常只與一個中心裝置連線。

資料可以在「中心裝置」與「週邊裝置」之間傳輸與接受，有時候是單向傳輸資料有時候是雙向傳輸資料，取決於你的設備的需求來設計資料傳輸的方式。

下方表格整理了中心裝置與週邊裝置比較：

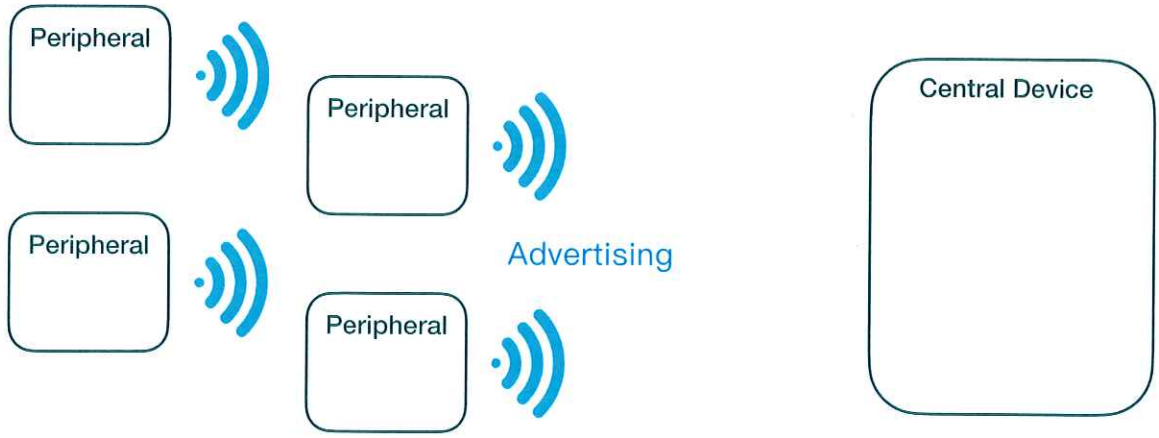
	中心裝置（Central Device）	週邊裝置（Peripheral Device）
功能	負責尋找、連接和管理週邊裝置的連接。	主要回應中心裝置的請求，包括接收和發送數據。
硬體需求	通常需要較大的處理能力和電源，例如手機、筆電、平板電腦。	通常較簡單，電源需求較小，例如藍牙耳機、智慧型手錶或藍牙滑鼠。
雙向傳輸	可以	可以
連接數量	可以同時與多個週邊裝置連接	通常只與一個中心裝置連接

接下來的內容，中心裝置將用 central 表示，週邊裝置將用 peripheral 表示。

Peripheral Device Advertising 週邊裝置廣播

Central 設備要連接一個 peripheral 設備之前，要先找到這個 peripheral 設備。

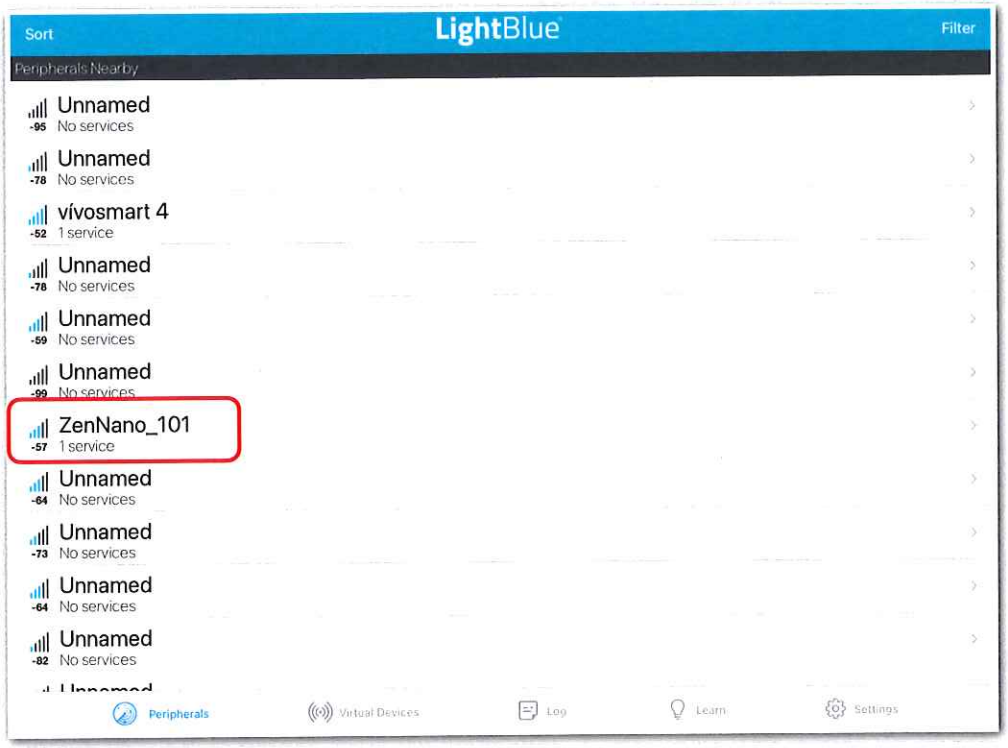
Peripheral 設備在允許被連接的狀態時，peripheral 設備會透過廣播 (Advertising) ，讓其它 Central 設備知道自己的存在。



Central Device Scanning 中心裝置掃描

Central 設備會透過掃描 （Scan） 的方式來找到可連接的 peripheral 設備：

在 iPad 或手機上我們可以使用 BLE Scanner App 來看到這些裝置，在 Apple Store 有多款 App 可以做到這件事，以下使用 LightBlue App 來示範：



LightBlue App

Peripheral 裝置發送的 Advertising Packets 會提供 Peripheral 裝置的基本資訊可能包括裝置的名稱、裝置的型號等等Discover 在藍牙中是指 Central 裝置發現了 Peripheral 裝置，並識別了 Peripheral。這些基本資訊可能包括裝置的名稱、裝置的型號、以及裝置提供的服務等等。

在上圖中，LightBlue App 作為一個 central 接收 Peripheral 裝置的 Advertising Packets，將其表列於畫面上。

Connection Request 連線請求

在**建立**連線之前， central 裝置會先確認連線的裝置**名**稱正確後，再發送連線請求 Connection Request。


```
if Peripheral.name == "ZenNeno_101"{
    connectRequest()
}
```

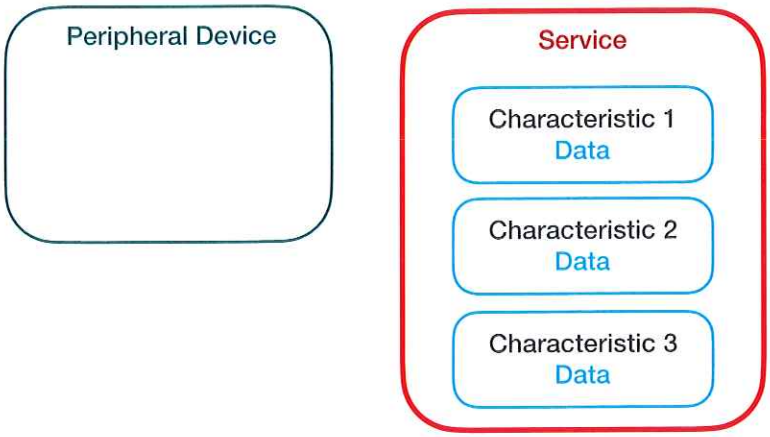
在手機與電腦上與 Peripheral 裝置進行連線時，會在藍牙列表上選擇要**建立**連線的藍牙裝置來**建立**連線，此時手機會確認連線的 Peripheral 裝置**名**稱，再發送連線請求。

當 peripheral 裝置處於廣播（advertising）模式並接受連接請求後，central 裝置才能與其**建立**連接，並進行數據交換。一旦連接**建立**，周邊設備將停止廣播，直**到**該連接被中斷。

Discover Peripheral Services and Characteristics

建立連線後 central 設備會發現（discover）peripheral 設備有哪些服務（services），這些服務會包含一些特徵（characteristics）。這些服務與特徵有各自的 ID，要使用這些資料時，需要設定這些 service ID 與 characteristic ID 才能得**到**資料。

一個 peripheral 可能有多個 services，一個 service 可能有多個 characteristics，characteristics 會有不同的。



在 LightBlue App 中，可**以看到** ZenNano_101 這個裝置有一個 service，這一個 service 有兩個 characteristics。

Read, Write and Notify

每個 services 的 characteristics 都包含著一些屬性（Properties），這些屬性表示可以對這個 characteristics 進行哪些操作。

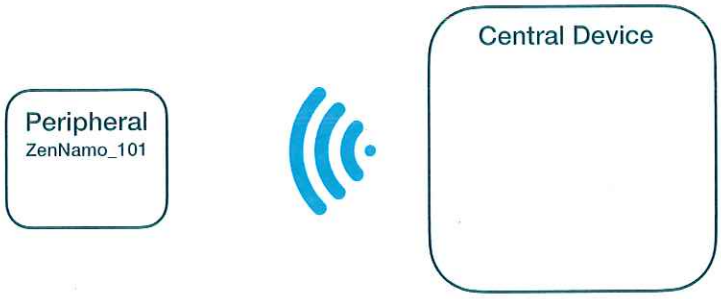
常見的 Property 有：

1. 讀取（Read）：表示 central 設備可以讀取這個 characteristic 的 data。
2. 寫入（Write）：表示 central 設備可以寫入 data 到這個 characteristic。
3. 通知（Notify）：允許 peripheral 設備將這個 characteristic 的新值通知給 central 設備，不需要 central 設備去主動讀取。

下圖中為健康監測手錶的 services 所包含的 characteristics，不同的 characteristic 有不同的屬性。

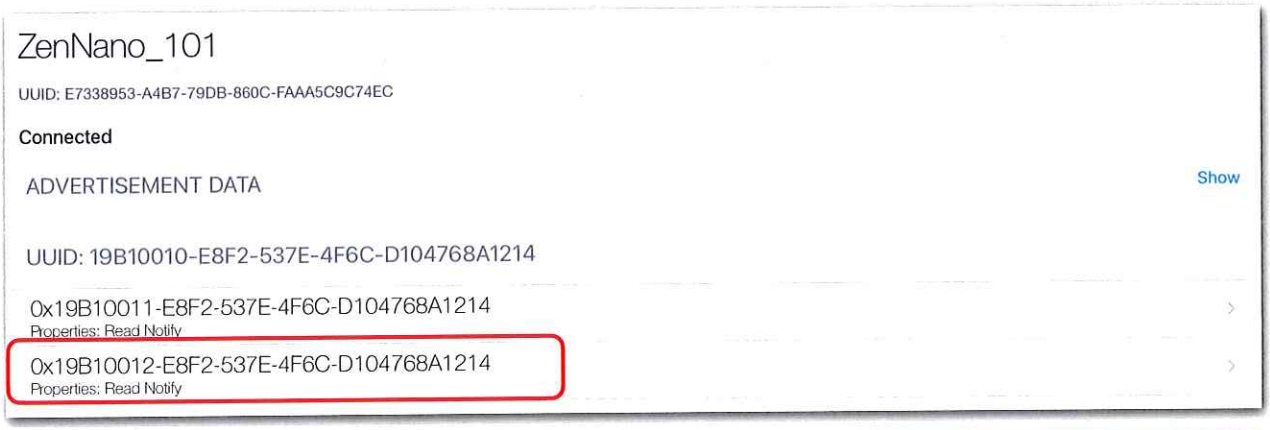
Communicate with Data

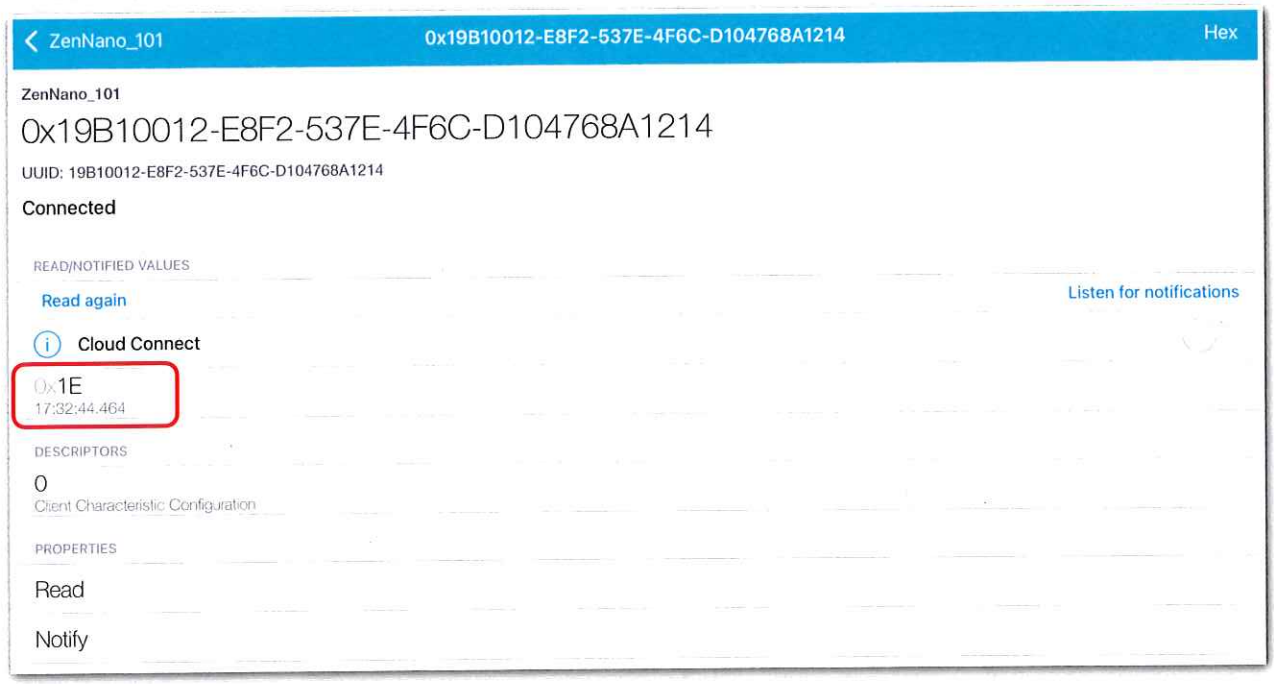
連接 peripheral 設備後，對其資料做**存**取前，必須要先知道並判斷 peripheral 的 services ID 與 characteristic ID 是否符合目前要**存**取的資料，才能進行**讀**取或寫**入**操作。




```
if serviceID == "0xAAoxoxo" {}
if characteristicID == "0xAAxxxxx" {}
```

在 LightBlue App 中，連線 ZenNano_101 後，在 service 底下點選第二個 characteristic ID ，可**以讀**取**到**這一個 characteristic 的數值為 0x1E ：





ZenNano_101 這一個 characteristic 是溫度資料，0x1E 是十六進位的資料呈現方式。

以下是十進位與十六進位的對照表：

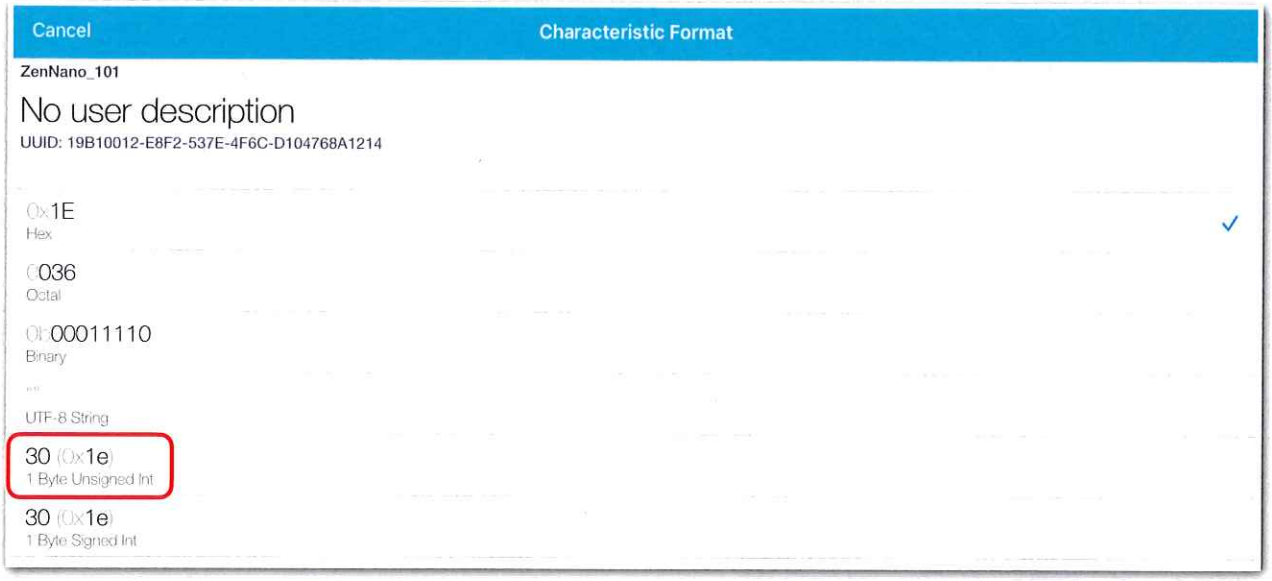
十進位	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
十六進位	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10

0x1E 表示數值只有 1E ，10 轉換為十進位為 16，E 轉會為十進位為 14，1E 的十進位表示為 30 ，正確的計算方式為：$1 \times 16^1 + 14 \times 16^0 = 30$ 。

如果今天十六進位數值顯示0x20 ，轉換為十進位後的值為 $2 \times 16^1 + 0 \times 16^0 = 32$ 。

十六進位制的主要優點是它能夠用較少的位元數來表示較大的數值。例如：十六進位的 "FF" 等於十進位的 $15 \times 16^1 + 15 \times 16^0 = 255$。

在前一張圖的畫面按右上角的 Hex ，可以讓資料用不同格式呈現：



在設備之間的資料傳輸格式，通常是以位元組（bytes）為單位，並且經常使用十六進位（hexadecimal）表示法來描述。這是因為十六進位表示法更為簡潔，可以用更少的字符表示更多的數據。 所以當收到藍牙的資料後，需要進行適當的資料格式轉換，才能得到對使用者來說有意義的數值。

章節筆記

語法疑問：

教學重點：

反思回饋：

BLE Central App – part 1

我們將以 iPad 為 central 設備，寫一個 Bluetooth Central App 來連接 Arduino Nano 33 BLE 這個 peripheral 設備，並讀取 (read) Arduino Nano 33 BLE 發送的資料，並將資料顯示在畫面上。

Arduino Nano 33 BLE 使用 Bluetooth Low Energy (BLE) 的，又稱 Bluetooth Smart，相較於傳統 Bluetooth，能有效降低能源消耗，但傳輸速度較慢，適用於電池容量有限且需長期運作的設備，如健康監測裝置或運動手錶等等。

這個 Nano 33 板子上有加速度感應器、溫度計.....多項感測器，在此專案 Nano 33 中會發送兩種資料，溫度、計步器偵測，其中計步器偵測資料由加速度感應器偵測訊號傳出。

在 LightBlue App 中，我們可以得到 Nano 33 的 services ID 與 characteristics ID ，這些數值在寫 central App 時會需要使用到。

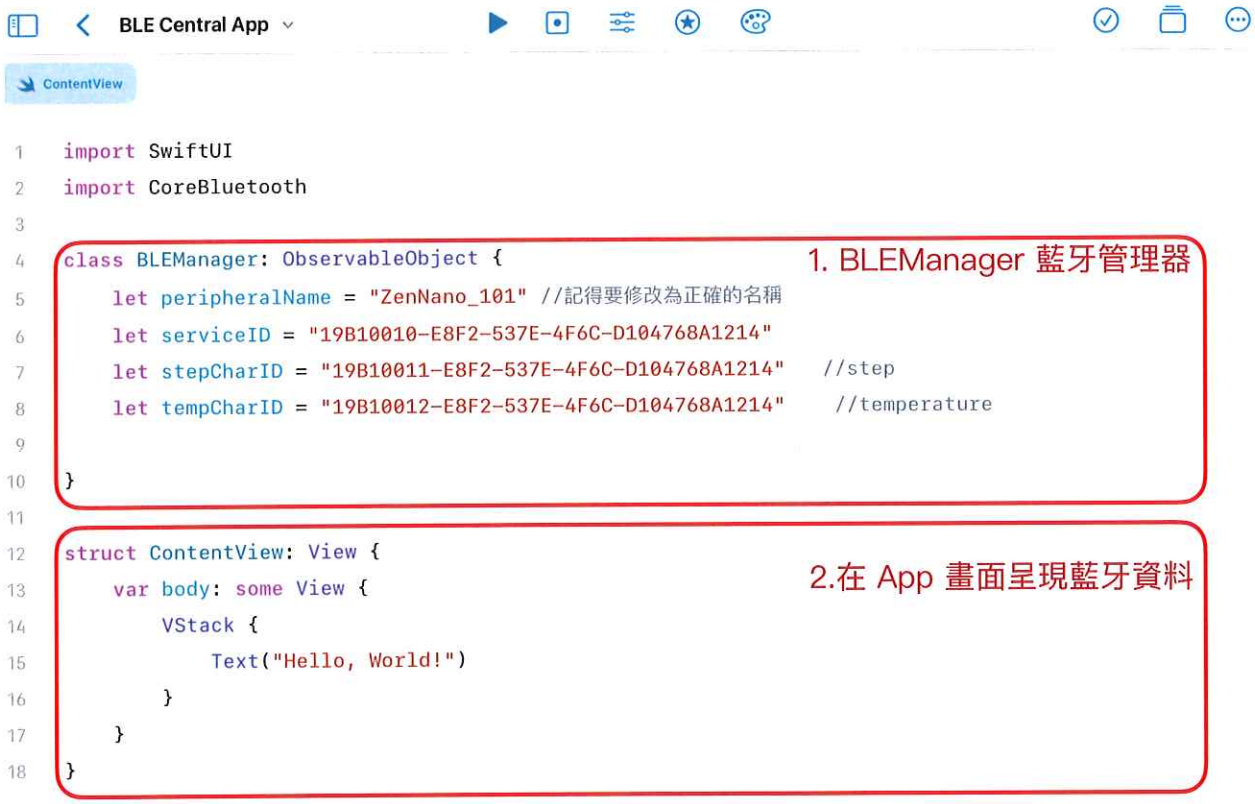
語法筆記：

在開始寫藍牙功能之前，我們已先做一些準備在 BLE Central App_Template 檔案中：

- 開啟一個新的 App 專案，命名為 BLE Central App
- 載入 Swift 的藍牙框架 import CoreBluetooth。
- 新增一個 class 為 BLEManager 作為藍牙管理器，在其中新增四個常數，分別為 Nano 33 的設備名字 (peripheralName)、 services ID (serviceID) 與兩個 characteristics ID (stepCharID、tempCharID)。

從 BLE Central App_Template 開始完成這個專案，可以為兩大部分：

1. BLEManager 藍牙管理器：負責掃描、連接藍牙，確認連接 peripheral設備的 services 和 characteristics ，與接收藍牙的資料.....等工作。
2. ContentView 畫面結構呈現藍牙資料：在 ContentView 的結構中拿到 BLEManager 的資料並顯示在畫面上。




```swift
import SwiftUI
import CoreBluetooth

class BLEManager: ObservableObject {
    let peripheralName = "ZenNano_101" //記得要修改為正確的名稱
    let serviceID = "19B10010-E8F2-537E-4F6C-D104768A1214"
    let stepCharID = "19B10011-E8F2-537E-4F6C-D104768A1214"   //step
    let tempCharID = "19B10012-E8F2-537E-4F6C-D104768A1214"    //temperature

}

struct ContentView: View {
    var body: some View {
        VStack {
            Text("Hello, World!")
        }
    }
}
```

為了能及時呈現藍牙管理器的工作狀態與結果，我們將先建立畫面 ContentView 與 藍牙管理器之間的資料傳輸管道。

App 畫面與 ObservedObject

畫面結構 ContentView 將使用 ObservedObject 來觀察 BLEManager ，BLEManager 也須符合 ObservableObject 。BLEManager 內使用 @Published 修飾符的變數，將被觀察並將改變後的內容呈現在畫面上。

```
4   class BLEManager: ObservableObject {  ①
5
6       let peripheralName = "ZenNano_101" //記得要修改為正確的名稱
7       let serviceID = "19B10010-E8F2-537E-4F6C-D104768A1214"
8       let stepCharID = "19B10011-E8F2-537E-4F6C-D104768A1214"    //step
9       let tempCharID = "19B10012-E8F2-537E-4F6C-D104768A1214"     //temperature
10
11      // @Published 變數
12      @Published var isSwitchedOn = false  ②
13
14  }
15
16  struct ContentView: View {
17      @ObservedObject var bleManager = BLEManager()  ③
18
19      var body: some View {
20          VStack {                                                    ④
21              Text("Bluetooth : " + (bleManager.isSwitchedOn ? "ON" : "OFF"))
22          }
23      }
24  }
```

① 設定 BLEManager 為一個 ObservableObject 物件。

② 在 BLEManager 內，宣告一個布林值 @Published 變數 isSwitchedOn，**儲存** central 的藍牙是否開啟的狀態，初始值為 false。

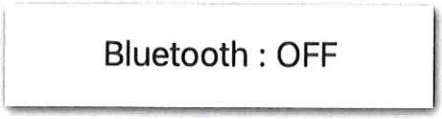
```
@Published var isSwitchedOn = false
```

③ 在 ContentView 宣告一個 bleManager 為 BLEManager() 的實例，並為 bleManager 加上修飾符 @ObservedObject 。

```
@ObservedObject var bleManager = BLEManager()
```

④ 在畫面中加上畫面元件 Text() ，其中使用三元運算子來依照藍牙是否開啟顯示不同的字串。若 bleManager.isSwitchedOn 為 true ，顯示 "Bluetooth：ON " ，否則顯示 "Bluetooth：OFF " 。

```
Text("Bluetooth : " + (bleManager.isSwitchedOn ? "ON" : "OFF"))
```



BLEManager 藍牙管理器

一個 BLE Central App 中，我們會定義一個 class BLEManag er 來負責處理所有藍牙相關的操作，包含搜索和連接藍牙外設裝置（peripherals）。

① 設定 BLEManager 繼承於 NSObject ，且分別符合 CBCentralManagerDelegate 和 CBPeripheralDelegate 兩個協議：

要使用 CBCentralManagerDelegate 與 CBPeripheralDelegate 的 protocol 時，BLEManager 需要繼承 NSObject ，因為 CBCentralManagerDelegate 本身繼承自 NSObjectProtocol。此外 NSObject 必須被放在第一個，因為它是 superclass 。

② 宣告 centralManager 與 peripheral ：
其型別分別為CBCentralManager 與CBPeripheral。

此時第四行程式碼會出現一些錯誤，因為目前 BLEManager 的 class 還缺少一些內容，使其不符合 CBCentralManagerDelegate 與 CBPeripheralDelegate 的協議。

為了要使符合 CBCentralManagerDelegate 與 CBPeripheralDelegate 的協議，並接收到藍牙資料顯示在 App 畫面上，BLEManager 的程式碼可分成 A ～ E 步驟，與 @Published 變數宣告區域，如藍色方框所示：

```
4   class BLEManager: NSObject, ObservableObject, CBCentralManagerDelegate, CBPeripheralDelegate {
    Type 'BLEManager' does not conform to protocol 'CBCentralManagerDelegate'                    ①
5       let peripheralName = "ZenNano_101" //記得要修改為正確的名稱
6       let serviceID = "19B10010-E8F2-537E-4F6C-D104768A1214"
7       let stepCharID = "19B10011-E8F2-537E-4F6C-D104768A1214"    //step
8       let tempCharID = "19B10012-E8F2-537E-4F6C-D104768A1214"     //temperature
9       // @Published 變數
10      @Published var isSwitchedOn = false                                  @Published 變數宣告區
11
12      //宣告 centralManager 與 peripheral
13      var centralManager: CBCentralManager!                                ②
14      var peripheral: CBPeripheral!
15
16      //初始化 centralManager
17
18      //CBCentralManagerDelegate 協議：設定6個函數                          步驟 A ～E
19
20      //CBPeripheralDelegate 協議：設定3個函數
21
22  }
23
24  struct ContentView: View {
25      @ObservedObject var bleManager = BLEManager()
26      var body: some View {
27          VStack(alignment: .leading, spacing: 20) {                       步驟 E
28              Text("Bluetooth : " + (bleManager.isSwitchedOn ? "ON" : "OFF"))
29          }
30      }
31  }
```

步驟A. 初始化 centralManager

步驟**B**. 為了符合 CBCentralManagerDelegate 協議，與 central 設備需要執行的內容为，需要設定6個函數，分別為：

執行順序	呼叫	函數名	功能說明
B1	被動	centralManagerDidUpdateState(_:)	確認 central 藍牙已經開啟。
B2	主動	startScanning()	設定一個 central 設備的掃描函數。
B3	被動	centralManager(_:didDiscover:advertisementData:rssi:)	Central 掃描 peripheral，判斷 peripheral.name 為要連線的裝置名稱後建立連線。
B4	被動	centralManager(_:didConnect:)	成功連線後，發現 peripheral 的 Services。
B5	主動	disconnect()	設定與 peripheral 斷開連線。
B6	被動	centralManager(_:didDisconnectPeripheral:error:)	與 peripheral 斷開連線後要執行的內容。

步驟C. 符合 CBPeripheralDelegate 協議，設定3個函數，分別為：

執行順序	呼叫	函數名	功能說明
C1	被動	peripheral(_:didDiscoverServices:)	發現並確認 peripheral 的 Services ID 相符後，發現 peripheral 的 Characteristics。
C2	被動	peripheral(_:didDiscoverCharacteristicsFor:error:)	發現 peripheral 的 Characteristics ID 相符後，設定 Notify Characteristics 的值。
C3	被動	peripheral(_:didUpdateValueFor:error:)	若Characteristics 的值更新後，要做相應的處理。

步驟**D**. **修改** C3 函數，將收**到**的 peripheral 資料做處理後**儲存**，**修改** B6 函數。

步驟E. 將收**到**的資料呈現在 ContentView 畫面內 ，**修改** B6 函數使斷開連**結**後，畫面的步數與溫度資料清空。

在上方表格中，有一些函數的呼叫**欄位**標示為被動，這代表這個函數在特定條件下會被呼叫並執行，這樣的函數叫做回調 (callback) 函數。

*注意：接下來的程式碼編寫，記得利用選單自動帶入。

初始化 centralManager

步驟A 程式碼

在 class BLEManager 內，使用 init() 來初始化 centralManager ，override 關鍵字表示這個方法覆蓋了它在父類別（Superclass）中的內容。super.init() 是呼叫父類別的 init() 方法，確保父類別的所有初始設定都已經完成。

```
override init() {
    super.init()
    centralManager = CBCentralManager(delegate: self, queue: nil)
}
```

建立 centralManager 的實例，為一個 CBCentralManager() 。其中的參數 delegate: self 設定 CBCentralManager() 的 delegate 為目前這一個 class 也就是 BLEManager。另一個參數 queue 設為 nil，nil 表示使用主要的 CPU 資源來運作。

CBCentralManagerDelegate 協議

步驟B 程式碼

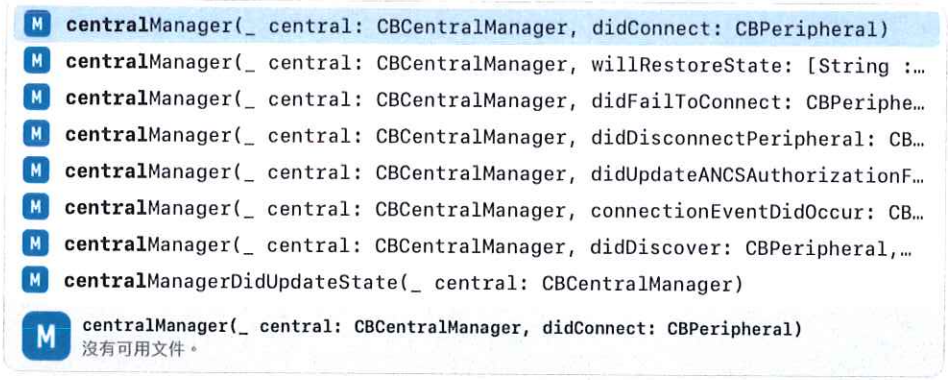
在 class BLEManager 內，為了符合 CBCentralManagerDelegate 協議，與 central 設備需要執行的內容为，需要設定6個函數，分別為：

執行順序	呼叫	函數名	功能說明
B1	被動	centralManagerDidUpdateState(_:)	確認 central 藍牙已經開啟。
B2	主動	startScanning()	設定一個 central 設備的掃描函數。
B3	被動	centralManager(_:didDiscover:advertisementData:rssi:)	Central 掃描 peripheral，判斷 peripheral.name 為要連線的裝置名稱後建立連線。
B4	被動	centralManager(_:didConnect:)	成功連線後，發現 peripheral 的 Services。
B5	主動	disconnect()	設定與 peripheral 斷開連線。
B6	被動	centralManager(_:didDisconnectPeripheral:error:)	與 peripheral 斷開連線後要執行的內容。

B1 程式碼：centralManagerDidUpdateState(_:) 函數

在 class BLEManager 內，使用選單加入 centralManagerDidUpdateState(_:) ：

```
func central
```



加入後的函數：

```
func centralManagerDidUpdateState(_ central: CBCentralManager) {

}
```

加入函數後，在 iPad 上會自動跳出一個視窗，詢問是否允許 BLE Central App 使用藍牙。

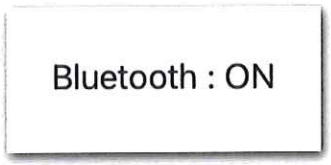
若在 Mac 上編寫程式，要記得從設定開啟 App 的藍牙權限。

在這個函數中要確認 central 的藍牙功能已開啟：

```
func centralManagerDidUpdateState(_ central: CBCentralManager) {
    if central.state == .poweredOn {
        isSwitchedOn = true
    } else {
        isSwitchedOn = false
    }
}
```

若 central.state 是處於 .poweredOn ，要將紀錄藍牙是否開啟的變數 isSwitchedOn **修改**為 true ，否則為 false。

若藍牙有正確開啟，也同意了 App 藍牙權限**存**取，畫面上的藍牙狀態的應該會**改**為 ON。



測試看看你的 iPad 藍牙功能是否有正常開啟？ 如果沒有開啟要如何排除問題？

B2 程式碼：startScanning() 函數

在 class BLEManager 內，設定一個函數為開啟 central 設備的掃描功能：

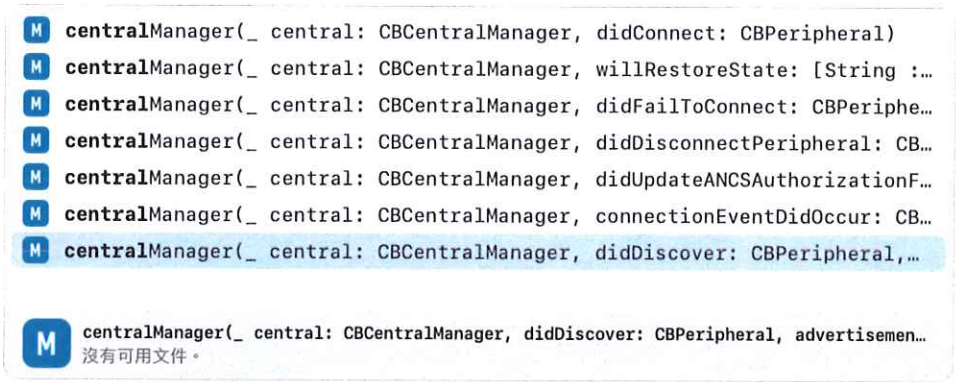
```
func startScanning() {
   centralManager.scanForPeripherals(withServices: nil , options: nil)
}
```

使用 centralManager 的方法 .scanForPeripherals(withServices: nil , options: nil) 來開啟掃描 peripheral 裝置，此時可以不用設定 services 的內容，將其參數設為 nil 。

B3 程式碼：centralManager(_:didDiscover:advertisementData:rssi:) 函數

在 class BLEManager 內設定這個函數，當 central 發現 (didDiscover) 了 peripheral 之後要執行的內容，使用預設選單加入這個函數：

```
func central
```



加入函數後，當 central 發現 peripheral，需進行判斷其裝置是否是要連線的裝置名稱：

```
func centralManager(_ central: CBCentralManager, didDiscover peripheral:
 CBPeripheral, advertisementData: [String : Any], rssi RSSI: NSNumber) {
   if peripheral.name == peripheralName {

   }
}
```

紅色方框中， == 後面 peripheralName 為一開始宣告的內容：

（已宣告的內容） `let peripheralName = "ZenNano_101"`

注意：這邊每個藍牙 peripheral 設備的名稱不一樣，要確實設定要連線的裝置名稱。

確認裝置名稱正確後，要在 if 判斷式中加入以下程式碼，如下：

```
if peripheral.name == peripheralName {
    self.centralManager.stopScan()
    self.peripheral = peripheral
    self.peripheral.delegate = self
    self.centralManager.connect(self.peripheral, options: nil)
}
```

此部分的 self.centralManager 與 self.peripheral ，為一開始宣告的這兩行：

```
(已宣告的內容)   var centralManager: CBCentralManager!
(已宣告的內容)   var peripheral: CBPeripheral!
```

self.centralManager.stopScan() 表示 central 要停止繼續掃描周邊裝置。

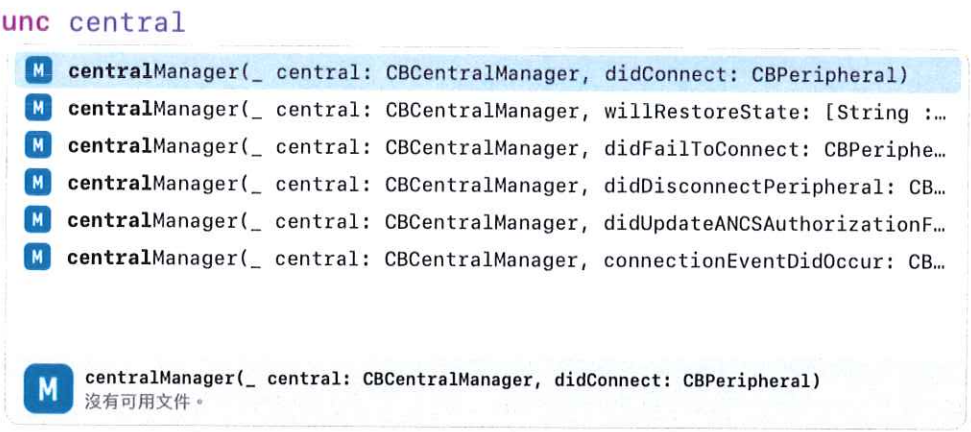
將 self.peripheral 設定為目前掃描到的要做連線的 peripheral 。並將 self.peripheral 的 delegate 設定為目前的 class，也就是 BLEManager。

最後進行 central 與 peripheral 的連線：self.centralManager.connect(self.peripheral, options: nil)

語法筆記：

B4 程式碼：centralManager(_:didConnect:) 函數

在 class BLEManager 內設定這個函數。當 central 與 peripheral 成功建立連線後 (didConnect) 要執行的內容，使用預設選單加入這個函數：



加入的函數如下：

```
func centralManager(_ central: CBCentralManager, didConnect peripheral:
 CBPeripheral) {

}
```

成功連線後，需要一個變數 isConnected 來紀錄連線狀態，且要在畫面的結構觀察這個變數的改變，在畫面上呈現是否連線 peripheral ，所以需要在@Published 變數宣告區宣告一個布林值變數 isConnected 來記錄 peripheral 已經被成功連線。

```
// @Published 變數
@Published var isSwitchedOn = false
@Published var isConnected = false
```

並在函數內加入，成功連線 peripheral 後，要將 isConnected 修改成 true ：

```
func centralManager(_ central: CBCentralManager, didConnect peripheral:
 CBPeripheral) {
    isConnected = true
    self.peripheral.discoverServices([CBUUID(string: serviceID)])
}
```

接下來 central 要執行發現(discover) peripheral 的服務 (services)，在函數內加入 self.peripheral.discoverServices() ，其中 discoverServices(_:) 方法的參數是一個UUIDs 的陣列，代表要尋找的服務的類型，加入的參數內容為一開始宣告的 serviceID。

ContentView 畫面顯示 peripheral 連線狀態

在 ContentView 的 body 內，加入一個畫面元件 Text() 來顯示 peripheral 連線狀態：

```
Text("Connected : " + (bleManager.isConnected ? "\(bleManager.peripheralName)" : ""))
```

"Connected : " 之後使用三元運算子 ? :，如果 bleManager.isConnected 為 true 顯示 bleManager.peripheralName 的內容，否則顯示空字串。

可以在 VStack 後加入參數設定，讓畫面的字串靠書寫方向（.leading）對齊，行距為 20：

```
VStack(alignment: .leading, spacing: 20) {
    Text("Bluetooth : " + (bleManager.isSwitchedOn ? "ON" : "OFF"))
    Text("Connected : " + (bleManager.isConnected ? "\(bleManager.peripheralName)" : ""))
}
```

此時畫面會呈現如下：

Thinking Time：將要連線的藍牙 peripheral 設備接上電源，確認 central 設備藍牙功能已開啟，為什麼這時候沒有連線上 peripheral？

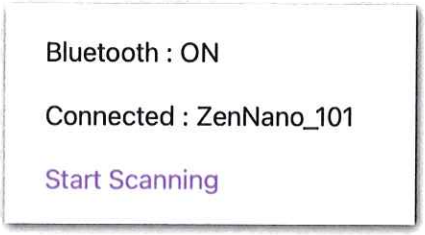
在 ContentView 畫面加入執行掃描的按鈕

目前的 central 設備藍牙功能確實已開啟，但是不是處於掃描(scanning)狀態，所以不會掃描到 peripheral 設備，也不會與 peripheral 做連線設定。

所以我們需要在 ContentView 的 body 中加入一個按鈕，來開啟 central 的掃描功能：

```
Button("Start Scanning") {
    bleManager.startScanning()
}
```

在畫面中按下按鈕，執行 central 設備開始 scanning 周邊的 peripheral 設備，若找到正確名字的 peripheral 就會建立連線，成功連線後畫面上會顯示連線設備的名字：



B5 程式碼：disconnect() 函數

在 class BLEManager 內，設定一個函數來主動斷開與 peripheral 的連線：

```
func disconnect(){
    if let peripheral = self.peripheral {
        centralManager.cancelPeripheralConnection(peripheral)
    }
}
```

在函數內執行 centralManager 的方法 .cancelPeripheralConnection(peripheral) 來取消與 peripheral 的連線。

並在 ContentView 的 body 中加入一個按鈕，來執行 disconnect 的功能：

```
Button("Disconnect"){
    bleManager.disconnect()
}
```

Thinking Time：為什麼這個函數內要用 if let ? 若不使用 if let ，將 disconnect() 定義方式如下方時，會發生什麼問題？

注意：本頁下方內容為提示，答案在下一頁

```
func disconnect(){
    centralManager.cancelPeripheralConnection(peripheral)
}
```

正確的 disconnect() 定義方法如下：

```
func disconnect(){
    if let peripheral = self.peripheral {
        centralManager.cancelPeripheralConnection(peripheral)
    }
}
```

其中 self.peripheral 為一開始宣告的這一個 peripheral 變數：

(已宣告的內容)　`var peripheral: CBPeripheral!`

其內容是在 B3 程式碼 centralManager-didDiscover 的函數內給定：

```
func centralManager(_ central: CBCentralManager, didDiscover
 peripheral: CBPeripheral, advertisementData: [String : Any], rssi
 RSSI: NSNumber) {
    if peripheral.name == peripheralName {
        self.centralManager.stopScan()
        self.peripheral = peripheral
        self.peripheral.delegate = self
        self.centralManager.connect(self.peripheral, options: nil)
    }
}
```

(已定義函數)

在宣告變數時 peripheral 尚未給定內容，所以在尚未建立連線時，peripheral 還沒有初始化，若定義 disconnect() 時，沒有使用 If let 將 peripheral 在尚未給定內容的狀況 排除，就會遇到 App 當機閃退。

所以定義 disconnect() 使用 if let 語法是必須的。

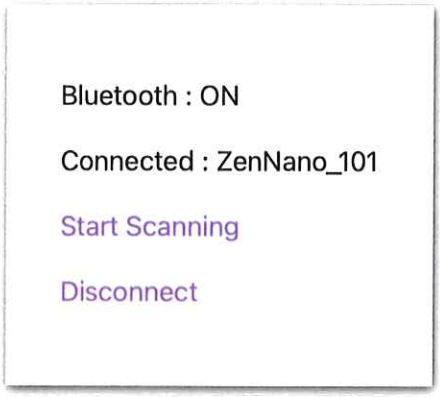
B6 程式碼：centralManager(_:didDisconnectPeripheral:error:)

在 class BLEManager 內，設定當 peripheral 已經斷開連線（didDisconnectPeripheral）後要執行的內容，使用預設選單加入這個函數：：

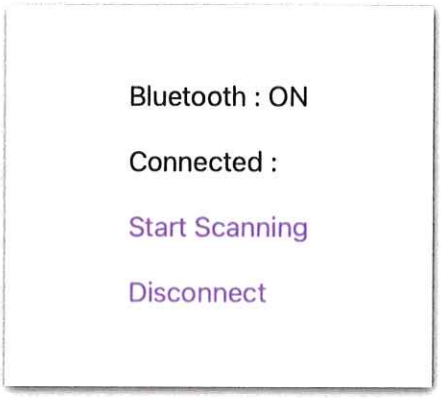
```
func centralManager(_ central: CBCentralManager,
 didDisconnectPeripheral peripheral: CBPeripheral, error: Error?) {
   isConnected = false
}
```

不論是在 App 畫面中，主動按下 Disconnect 按鈕斷開連線，或是 peripheral 設備異常斷開連線，都要將 isConnected 設定為 false 。

按下 Start Scanning



按下 Disconnect



測試一下目前的 App ，是否可以正常連線與斷開連線呢？

章節筆記

語法疑問：

教學重點：

反思回饋：