2.2 連立方程式

さて、中 2 数学に入ったところで playground でビュー表示することを目的にプログラムを書いてきた。そこまでやるならアプリの形で見たいと思うかもしれないね。アプリとして見たければ Xcode で Project ファイルを作ればよい。それに前節で指摘したように、playground で不具合が起きるようなら Project ファイルで実行しなくてはならない。その場合は Xcode の File メニューから、New ▶ Project... をたどって新しいファイルを作ろう[1]。

Project ファイルはアプリ作成のためのファイルなので、テムプレートを選択するダイアログが表示される。何も凝ったことをするつもりはないので、メニューバーの "iOS" を選択して表示されるアイコンから "App" を選択して Next ボタンを押そう。Product Name:欄は適当な名前を入力し、User Interface:欄は "Storyboard" に変えて Next ボタンを押せば、プロジェクトを保存する場所を選択するダイアログが表示されるので、適当な場所を決めて Create ボタンを押せばよい。

何かたくさんの設定を促すような画面になっても気にしないこと。当面はデフォルトのままプログラムを書くだけだ。画面左側の "Navigator" にフォルダがツリー状に表示されていたら "ViewController" をクリックしてもらいたい（Navigator が見えてなければ、View メニューから、Navigators ▶ Show Project Navigator をたどろう）。中央には見た感じのコードが何行か書かれた画面が見えるはずだ。ここがプログラムを書く領域である。以前書いた playground 用でビュー表示するものを、丸ごとコピー・貼り付ければよい。

ただ前節でも少し触れたように、ふたつ目の `import` 文と最後の 4 行—行末に目印（`// *`）を付けた文—は playground でビュー表示させるためのものであるから、Project ファイルを作るときはいらない。もっとも、これらを書いたままだと Xcode から警告を受けるので削除は必然だけど。

では前置きはこれぐらいにして、ここでは連立方程式を Swift に解いてもらうことにする。と言っても、Swift は私たちが解くような手順で処理するのではない。連立方程式の解の公式を作ってしまえばよいのだ。たとえば

$$※\begin{cases} \boxed{a}x+\boxed{b}y &= \boxed{m} \\ \boxed{c}x+\boxed{d}y &= \boxed{n} \end{cases}$$

という連立方程式は、1 番目の等式には d、2 番目の等式には b を掛けて $\begin{cases} adx+bdy &= md \\ bcx+bdy &= bn \end{cases}$

[1]新規 Projet ファイル作成中に認証のためのダイアログが表示されたら、適切な入力をして先へ進もう。

とすれば、(1 番目) − (2 番目) から $(ad-bc)x = md-bn$ となって、$x = (md-bn)/(ad-bc)$ が求まる。同じようにして $y = (an-cm)/(ad-bc)$ もわかる。つまり

$$\boxed{\begin{cases} ax+by &= m \\ cx+dy &= n \end{cases} \quad \text{の解は、} \quad \begin{cases} x &= \dfrac{dm-bn}{ad-bc} \\ y &= \dfrac{an-cm}{ad-bc} \end{cases}}$$

という公式の出来上がりだ。でも、公式にして活用している人は滅多にいないだろう。正確に覚えるのはちょっと大変だしね。

この場合 Swift にとって連立方程式を解くというのは、係数などを入力したとき結果が表示されることなので、プログラムは単に表示の仕方をどうするか考えるだけになる。数値を入力して結果を表示するのは前回ひな型を作ったばかりだ。では、こんな感じでどうだろう。`//__` となっているところは続けて 1 行で書くところである。改行せず入力してほしい。また、playground 上ではなく Project ファイルを作成した場合は、末尾が `// *` である行の入力は不要だ。

[2varsEquationView.playground]

```
 1 import UIKit
 2 import PlaygroundSupport // *
 3
 4 class ViewController: UIViewController {
 5
 6     var iTF = [UITextField()]
 7
 8     override func viewDidLoad() {
 9         super.viewDidLoad()
10
11         let labelA = UILabel(frame: //__
                              CGRect(x: 70, y: 20, width: 300, height: 25))
12         labelA.text = "x +             y = "
13         view.addSubview(labelA)
14         let labelB = UILabel(frame: //__
                              CGRect(x: 70, y: 50, width: 300, height: 25))
15         labelB.text = "x +             y =                の解は"
16         view.addSubview(labelB)
17
18         iTF = []
19         iTF += [UITextField(frame: //__
                             CGRect(x:  20, y: 20, width: 50, height: 25))]
20         iTF += [UITextField(frame: //__
                             CGRect(x: 100, y: 20, width: 50, height: 25))]
21         iTF += [UITextField(frame: //__
                             CGRect(x:  20, y: 50, width: 50, height: 25))]
22         iTF += [UITextField(frame: //__
                             CGRect(x: 100, y: 50, width: 50, height: 25))]
```

```
        iTF += [UITextField(frame: //__
                            CGRect(x: 180, y: 20, width: 50, height: 25))]
        iTF += [UITextField(frame: //__
                            CGRect(x: 180, y: 50, width: 50, height: 25))]
        for tf in iTF {
            tf.borderStyle = .roundedRect
            tf.addTarget(self, action: //__
                                #selector(getText), for: .editingChanged)
            view.addSubview(tf)
        }
    }

    let labelC = UILabel(frame: //__
                        CGRect(x: 20, y: 100, width: 300, height: 25))
    @objc func getText() {
        let a = (iTF[0].text! as NSString).integerValue
        let b = (iTF[1].text! as NSString).integerValue
        let c = (iTF[2].text! as NSString).integerValue
        let d = (iTF[3].text! as NSString).integerValue
        let m = (iTF[4].text! as NSString).integerValue
        let n = (iTF[5].text! as NSString).integerValue

        if a*d-b*c != 0 {
            labelC.text = "x = \(Float(d*m-b*n)/Float(a*d-b*c))、//__
                           y = \(Float(a*n-c*m)/Float(a*d-b*c))"
            view.addSubview(labelC)
        }
    }
}

let viewController = ViewController()                  // *
viewController.view.backgroundColor = UIColor.white    // *
PlaygroundPage.current.liveView = viewController       // *
PlaygroundPage.current.needsIndefiniteExecution = true // *
```

妙なプログラムになったものだが、テキストフィールドを配置して、見栄えのためにラベルを付けただけである。ただ部品が多いのでこんなことになっている。もちろん不備も多い。それでも、まずプログラムを実行してみよう。

playground 上に書いたプログラムなら、いままでどおり実行ボタンを押せばよい。Project ファイルに書いたプログラムを実行するときは、Xcode 上部の ▶ ボタンを押すのだが、その前に右横の "TestProject" バーからシミュレートするデバイス（iPhone や iPad）を選択すること。選択肢は多いがどれを選んでも同じようなものである。▶ ボタンを押すとしばらくしてシミュレータが起動してアプリ体験ができる[2]。キーボード操作により画面を縦横にしたり、ジェスチャーも有効

[2]Macintosh の性能、メモリ容量によっては起動まで長時間かかる。

になる。そうは言っても見え方は playground と大差ない[3]。さまざまな操作については各自で調べてもらいたい。

では、プログラムについて説明しよう。プログラムを書く前に考えたのは画面の設計図である。初めに示した連立方程式※において、□に任意の整数を入力したとき、その下に解を表示するようにした。こんな感じだ。

$$\square x + \square y = \square$$
$$\square x + \square y = \square \quad \text{の解は}$$

$x =$（なんたら）、$y =$（かんたら）

そうするとテキストフィールドが 6 個必要なので、たとえば aTF, bTF, . . . , nTF のような変数を用意することも考えられる。それでは各テキストフィールドに属性を与えるとき、6 回同じことを記述しなくてはならないので面倒だ。配列でテキストフィールドを用意すれば for 文の中で 1 回記述すれば済む。そう思って、まず最初に var iTF = [UITextField()] を宣言したのである。[ ] を付けて代入すると Swift は iTF を配列と理解する。

配列宣言した iTF に必要な要素を代入するのは 18 行目以降でしているので、先にそこから見ていこう。まず、iTF = [] は配列に実体（**インスタンス**）を与えるためだ。配列は宣言しただけでは実体がないので、配列 iTF に [] を代入して初期化している。この時点で iTF は要素を持たない配列として定義されたことになる。[] 状態の配列を**空配列**という。

以前、たとえば配列 data に要素 [100] を加えるときは data += [100] と書く、と言ったのを覚えているだろうか。iTF には 6 個のテキストフィールドが必要なので iTF += [UITextField(...)] を繰り返して、iTF が 6 個の要素を持つようにしている。本当は for 文で回したいところだが、テキストフィールドの位置が少しずつ違うので止むを得ない。ただ、27 行目の.addTarget() などは皆同じ属性のため for 文が使える。このための配列宣言であったのだ。25–29 行目における指定の仕方は前回のひな型と同様である。実は、無理をすれば 19–24 行目も for 文を用いて

```
for i in 0...2 {
    for j in 0...1 {
        iTF += [UITextField(frame: //__
                CGRect(x: 20+80*i, y: 20+30*j, width: 50, height: 25))]
    }
}
```

みたいにできるのだけれど、2, 3 回程度の繰り返しに for 文は大袈裟な気がしたんだ。

[3]シミュレート用に選択したデバイスによっては、実行した表示が画面上部のベゼルに隠れてしまう。一時的にはシミュレーションの画面を横にすればよいが、コードの y 座標を書き換えるのが正しい処置である。

これでテキストフィールドは用意できた。でも、このままでは単に係数を入力するための枠が表示されるだけなので、連立方程式っぽく見せるには x や y を表示させなくてはならない。それは `UILabel()` で行う。11–16 行目に書かれた `labelA` と `labelB` がそうである。代入はテキストフィールドと同じく、配置する位置を指定して初期化する。表示する文字は属性`.text` により代入すればよい。`.addSubview()` で配置するのもテキストフィールドと同じだ。

さあ、ここからが入力された係数を用いて連立方程式を解く処理である。`#selector()` が指定する関数 `getText` が、テキストフィールドから文字を読み取り `a` から `n` までの変数に代入する処理を受け持つ。そして公式を利用して解の表示まで行う。解の表示にはラベルが必要なので、`@objc func getText()` の前、32 行目に `labelC` として代入した。playground で実行すれば期待通りの動作を確認できると思う。あまり洗練されているとは言えないけれど、`getText()` が行なっている処理は見てのとおりだ。

とくにテキストフィールドから得た値を逐一 `let a =` , `let b =` , `let c =` , ... と代入するところなんか素人丸出しだろう。ここは、たとえば

```
var k = [Int]()
for n in 0...5 { k[n] = (iTF[n].text! as NSString).integerValue }
```

なんて書けば、変数を `a` から `n` までの 6 文字も使う必要がないと感じるかもしれない。でも、そうすると `a*d-b*c` などの計算は `k[0]*k[3]-k[1]*k[2]` とすることになる。まあ、それほど式は複雑になっちゃいないし、スッキリしていいかもね。プログラミングを自分一人で楽しむならコードの量を削るのはパズルみたいでよいけれど、過ぎたるは猶（なお）及ばざるが如しということもある。自分のいまの立場を考えて最適な書き方を工夫しよう。

ex. 1 [2varsEquationView.playground] には、`labelA`, `labelB` や `iTF` がほぼ同じ形式で書かれている。`for` 文を用いればコード量を減らせる。試してみよう。

ex. 2 関数 `getText` が `if` 文で `a*d-b*c != 0` のチェックをしている理由を考えてみよう。

Swift がプログラムを実行するときは、まず `ViewController` クラスから実行する。その際、いの一番に読まれるのが関数 `viewDidLoad` である。前回ひな型を作ったときもそうだったが、今回もプログラムを `func viewDidLoad() { }`内に書いたのはそのためだ

でも関数 `viewDidLoad` は私たちが新たに作ったものではない。あらがじめ Swift が作っている。私たちはそれにただ乗りさせてもらっているのだ。そのことはキーワード `override` が示している。純粋に私たちが新たに作るのは`@objc func getText()` である。でも、これは前回ひな型として作ったので、そいつを少し変更したまでだ。

ここで、“func 関数名 ()” に関する重要な性質を述べておこう。関数内で利用する変数 labelC が getText() からはみ出ている理由は、前節で説明したとおり、中に書くと getText() が呼び出されるたびに labelC が初期化されることだと言った。それは、つまり呼び出しのたびに新たなラベルが作られることになり、ビューがラベルで溢（あふ）れかえってしまうからだが、関数の中に書くか外に書くかは雲泥の差がある。それは超重要な性質

『関数内で定義する変数はその関数内でのみ有効、関数外で定義する変数は他の関数でも有効』

があるからである。

このことを実際に確認するために、次の見本を実行してみよう。

```
let str = "globalA"
func print1() {
    print(str)
}
func print2() {
    let str = "localA"
    print(str)
}
func print3() {
    print(str)
}
print1() // globalA が出力される
print2() // localA が出力される
print3() // ex.3
```

print1() は変数 str を出力するだけ、print2() は変数 str を初期化して str を出力するだけの関数である。実行結果は、print1() は globalA を、print2() は localA を出力する。関数 print1 内で str を初期化していないのに globalA が出力されたのは、関数の定義に先立って str が初期化されていたからである。この場合の変数 str は**グローバル変数**といい、以降の関数で使用することができる。

なのに、print2() を実行して localA が出力されたのは、str が関数内で初期化されていたからである。この場合の変数 str は**ローカル変数**といい、その関数 print2 内でのみ使用することができる。放射線のように外に漏れて影響を及ぼすことはないのだ。

ex. 3 print3() を実行すると globalA と localA のどちらが出力されるだろうか。考えてみよう。

そのような事情で関数 print2 内の変数名に、1 行目と同じ str を使えたのだ。同じ str でも実際は異なる変数で扱われている。でも紛らわしいので、たとえ外に影響を与えなくても変数名は違うほうがよいかもしれない。

以上の知識を持ってあらためてプログラムを見れば、変数 `iTF` を一等最初に書いたことに合点がいくはずだ。`iTF` は `viewDidLoad()` でも `getText()` でも利用する変数だからである。

かくして、ひな型を元に今回のプログラムが出来上がった。はじめのうちは元々あるプログラムに肉付けしながら作成するものである。この調子でどんどんコードを書いていこう。