［スマホプログラミング］

# 1 iOS/Swift プログラミング入門

沼田哲史 | 大阪電気通信大学

## Swift 4 と iPhone X の登場

2014 年 6 月，iPhone や Apple Watch といったスマートデバイス用のアプリケーションを効率的に開発するために，Apple 社から新しいプログラミング言語 Swift が発表されました．Swift はそれまで使われていた Objective-C よりも高速に動作し，よりプログラミングしやすい環境として紹介されました．Swift は 2014 年 9 月に正式版が公開されて以降，2015 年 9 月にバージョン 2，2016 年 9 月にバージョン 3，2017 年 9 月にバージョン 4 が公開され，そのたびにユーザの声を反映した改良が加えられてきました．1 年に 1 回のメジャーリリースという安定したスピードです．本稿を書いている 2017 年 11 月現在では，Swift で書いたアプリケーションは，以前に iOS アプリケーションの開発に使われていた Objective-C という言語の 2.6 倍も高速な処理が可能であると Apple が公表しています．

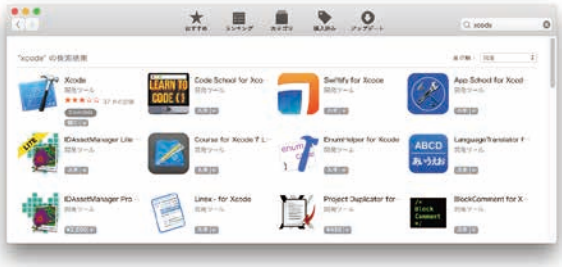
現代的なプログラミング言語の特徴を踏まえて作られた Swift の文法は，アプリケーションの動作を最適化してパフォーマンスを引き上げるために役立っているだけでなく，コーディング時に引き起こされるヒューマンエラーが低減できるようにもデザインされています．本稿では，最新のバージョンである Swift 4 の基本的な文法とその特徴について解説します．Swift を使って安全かつ効率的にプログラミングするポイントを押さえていきましょう．

## Xcode のインストール

Swift を使うためには，開発環境として Xcode というアプリケーションが必要になります．また最新の Xcode をインストールするためには，macOS のバージョンも最新の状態にしておく必要があります．Mac のメニューから，[ アップルマーク ] →[App Store...] を選択して，App Store アプリで OS のアップデートを行います．この記事を執筆している時点では，macOS High Sierra（v10.13.1）が最新版です．そして App Store アプリの右上にある検索フィールドに「Xcode」と入力して検索し，最初に出てくる「Xcode」を選択してインストールします（図 -1）．

## Swift で作る各種プラットフォーム用アプリケーション

Xcode は，Mac アプリ，iOS アプリのみならず，Apple Watch 用の watchOS アプリや Apple TV 用の tvOS アプリを作成するときにも，欠かすことのできないツールです．Xcode は，アプリケーションに必要なソースファイルやユーザインタフェース設計情報のファイル，画像ファイルなどのリソースを編集・管理するところから，アプリケーションをビルドして完成したアプリケーションに電子署名を行うところまで，一貫してアプリケーション開発を



■図 -1　App Store で Xcode を検索してインストール

サポートしてくれるツールです.

Xcode がサポートするすべてのプラットフォームで，基本のプログラミング言語として設定されているのが Swift です. Swift はコンパイラ型のオブジェクト指向言語でありながらも，コンパクトでクリーンなコードが書けるように工夫されており，プラットフォームごとの違いが吸収できるように API が整備されています.

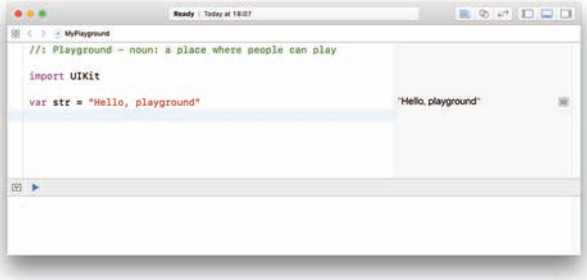
## Swift Playground でインタラクティブに勉強する

Xcode には Swift Playground という機能が搭載されています. Playground 上に Swift のコードを入力すると，インタプリタのように自動的に実行されて，リアルタイムに実行結果が表示されます. 1 行ごとに実行結果が右側に表示されますので，結果を確認しながら着実に勉強が進められるのです.

Xcode を起動して，メニューから [File]-[New]-[Playground...] を選択します. 最新版の Xcode では作成する Playground を 4 種類から選択できるようになっていますが，最も基本的な「Blank」を選択するだけで Swift の勉強を始められます.

作成直後の Playground は図 -2 のようになっています. メインのエディタ上では変数を宣言するプログラムが 1 行書かれており，右側にはそれを評価・実行した結果が「Hello, playground」という文字列になることが表示されています.

各行の実行結果の右端に表示される四角いボタンを押すと，結果の詳細がプログラムの中に表示されます（図 -3）.




■図 -2　作成直後の Playground

## Swift 4 の基本文法

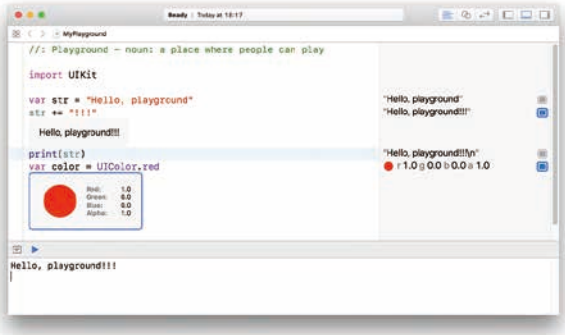
### 変数の宣言

Swift では 2 種類の「変数」が作成できます.「var」キーワードを使って作成した変数は変更可能な「変数」となり，「let」キーワードを使って作成した変数は変更できない「定数」となります.

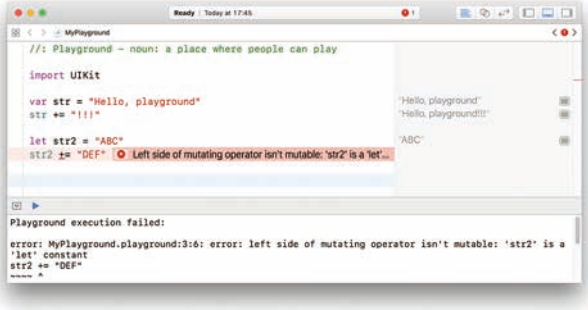
```
// 変更できる変数(変数)
var value1 = 345
value1 += 100  // 変更はオーケー！

// 変更できない変数(定数)
let value2 = 123
value2 += 3  // 変更しようとするとエラー！
```

図 -4 では定数として宣言した str2 に文字列を追加しようとしてエラーが出ていますが，このように定数に対しては変更の操作を行おうとするとエラーが出ます.




■図 -3　実行結果の詳細を表示する




■図 -4　エラー表示

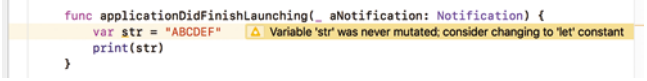
後々変更しない変数に関しては，こうして「let」キーワードを使って変更しないことを明示することで，コンパイラが最適化するヒントを与えることができるのです．単純なことのように思えますが，変数は至るところで出てきますので，これが積み重なるとパフォーマンスに影響が出てきます．Swift では初期のころから「文法によって安全で堅牢なコードを書くためのアフォーダンスを与える」ことを目的として改良が進められてきました．Swift で本番のアプリケーションを書いているときには，コード中で変更されていない変数が見つかった場合には定数に変更するように促すメッセージが現れます（図 -5）．

このことは，人間がソースコードを読む上でも可読性を向上させることにつながります．「クリーンなコードを書くことがクリーンな結果につながる」というのが Swift の設計思想なのです．C 言語などでプログラムを書いているときには変数の値が変化するかということをそれほど強く意識することはありませんが，Swift ではこれを意識するところからプログラミングが始まります．

Swift では，var で宣言した変数でも let で宣言した変数でも，C 言語同様に型付けが行われます．Perl, Ruby, Python といったインタプリタ型のスクリプト言語では，変数を明示的に宣言することなく，次のように途中で異なる型の数値を代入しても問題なく動作します（このコードは Ruby または Python で実行できます）．

```
num = 1234
print(num)
num = "ABC"
print(num)
```

それに対して，Swift では明示的に型を書かなくてもよいものの，初期化時に推論された型に型付けされますので，次のように異なる型の値を代入しようとするとエラーになります．




■図 -5　変更されない変数を定数に変換するように促すメッセージ

```
var num = 1234
num = "ABC"   // エラー!
```

C 言語と同様に明示的に型を指定する場合には，宣言文の変数名の後ろに型名を書きます．Swift で使う主な型名は，Int, Float, Double, String といったものでしょう．配列の型は，[Int] や [String] のように角括弧の中に格納する型名を書いて表します．

```
var num:Int = 1234
```

次の表に，Swift の標準のデータ型を載せます．

表：Swift標準のデータ型

| | |
|---|---|
| Int | 符号付き整数型（32ビットまたは64ビット） |
| UInt | 符号なし整数型（32ビットまたは64ビット） |
| Float | 32ビットの浮動小数点数 |
| Double | 64ビットの浮動小数点数 |
| Bool | 真偽値を表す型（trueまたはfalse） |
| String | 文字列型 |
| Character | 単一の文字を表す型 |

Int 型や UInt 型はサイズが環境依存で，32 ビットの環境では 32 ビットに，64 ビットの環境では 64 ビットになります．より正確にビット数を指定したい場合には，Int8, Int16, Int32, Int64, UInt8, UInt16, UInt32, UInt64 といったビット数を後ろに付けた型名を使用します．

C 言語ファミリーに代表されるコンパイラ型言語と異なる点として，Swift の変数は初期値を持たないということが挙げられます．たとえば次のように変数を宣言すると，この変数は「初期化されていない状態」になります．

```
var num:Int     // numは未定義
print(num)      // 未定義の値は表示できないのでエラー
```

この状態で print() 関数を使って値を表示しようとすると，初期化されていない変数の値は参照できないということになり，エラーが発生します．C 言語では宣言時に値が代入されない場合は初期値が不

定であり，C# や Java などでは初期値が 0 になりますが，どちらの動作も人間が意図していない結果を引き起こしてバグにつながる危険性がありました．それに対して，このコードがエラーになることは安心感が高いといえます．

次のように型名の後ろに「?」記号を書いて変数を宣言することによって，オプショナルな変数が作成できます．オプショナルな変数は，「値がない」ことを示す「nil」という状態を持てる変数で，宣言時に何も代入しない場合の初期値が nil になります．そのため，次のように書いて，「値が設定されてない」場合に対応するコードを書くことができます．

```
var num:Int?      // numはnil
num = 4
if num != nil {
    print(num)
}
```

C 言語などでこういったことを実現しようとすると，「とりあえずマイナスの値を設定しておく」「とりあえず int 型の最大値を設定しておく」といった方法で表すしかありませんでしたが，Swift ではオプショナルな変数を使うことでより安全に「値が設定されていない」状態を扱うことができます．

細かい点ですが，Swift の if 文には条件文に丸括弧を書く必要がありません．しかしその後ろの実行対象のブロックを表す波括弧は，省略しようとするとエラーになります．C 言語ファミリーの if 文は波括弧を省略した場合には直後の 1 行だけが実行対象となるために，コーディング規約で波括弧を書くことを義務付けている企業がほとんどです．Swift はこれを文法レベルで義務付けることにより，より安全なコードができるようにデザインされているのです．

## リテラル表記と型

Swift では，整数と浮動小数点数に対して，次の表に示すリテラル表記が使えます．リテラル自身にはビット数などの制約はありませんが，明示的な型指定がなければ Int 型または Double 型が優先して使われます．なお，リテラル自身にはビット数の制約がないことから，C 言語の「1UL」や「3.14159f」のようなサフィックスをつけて型を明示するという書き方はありません．

```
// 整数リテラル(型指定がなければInt)
123             一般的な整数123の表記
00123           先頭が0で始まっても10進数の123
0b1101          2進数表記の13
0o20            8進数表記の16
0x7f            16進数表記の127

// 浮動小数点数リテラル(型指定がなければDouble)
3.14159         一般的な小数の表記
12.345e2        指数表現 12.345*10^2 = 1234.5
12.345e-2       指数表現 12.345*10^(-2) = 0.12345
0x8.4p0         16進数表記の8.25
0x8.8p0         16進数表記の8.5
0x8p1           16進数表記の8.0*2^1 = 16
0x8p2           16進数表記の8.0*2^2 = 32
0x8.8p-1        16進数表記の8.5*2^(-1) = 4.25
```

文字列リテラルは，C 言語と同じように，2 個のダブルクォーテーション（"）の間に文字を書いて定義します．

```
let str = "Hello!!"
```

複数行に渡る文字列を定義する場合には，3 個の連続したダブルクォーテーション（"""）を文字列の前後に書いて定義します．

```
let str = """
1行目
2行目
"""
```

コードの見た目を整えるためにインデントを挿入する場合は，各行の先頭に同じ数のスペースを空けます．最後のダブルクォーテーションも同様にインデントするのを忘れないでください．こうすることで，各行の先頭にある空白が無視されます．

```
let str = """
      1行目
      2行目
      """
```

見た目は似ていますが，次のコードでは終わりのダブルクォーテーションをインデントしていないの

で，1 行目と 2 行目の文字列の先頭にあるスペースがインデントではなく意味のある空文字として扱われます．

```
let str = """
     1行目
     2行目
"""
```

文字列の中に変数の値を挿入したい場合には，バックスラッシュ（\）に続いて丸括弧を書き，その「\()」の中に変数名を書きます．

```
var value = 123
let str = "The value is \(value)."
```

「\()」の中には，戻り値のある関数やメソッド呼び出しなども書くことができます．たとえば次のように書くと，"1.4142135623731" という文字列が作成されます．

```
let str = "\(sqrt(2))"
```

小数点以下の桁数を指定するなど，C 言語の printf() 関数のように書式を指定して文字列に変換したい場合には，String クラスの init(format:) というイニシャライザを使います．たとえば次のように書くと，円周率が小数点以下 4 桁で四捨五入された結果が文字列として str 変数に格納されますので，"3.1416" という文字列が作成されます．

```
let str = String(format: "%.4f", Float.pi)
```

文字列リテラルには，絵文字やギリシャ文字などをそのまま書くこともできますが，Unicode のコード番号を直接書くこともできます．次のように「\u{...}」の波括弧の中に 16 進数のコードを書きます．Unicode では 1F300 ～ 1F9FF の範囲に絵文字が含まれますので，たとえば次のようなコードを書くと，女の子の顔の絵文字が使われます（図 -6）．

```
let str = "\u{1f467}"
```

■図 -6 「\u{1f467}」で表される絵文字

■図 -7 「\u{1f467}\u{1f3fb}」で表される絵文字

この Unicode 文字の後ろにセレクタ文字である 1f3fb をつなげて書くと，髪の色や肌の色が変わります（図 -7）．

```
let str = "\u{1f467}\u{1f3fb}"
```

なお，肌や髪の色を変えられるセレクタ文字は 1f3fb ～ 1f3ff の範囲の 5 種類ですので，皆さんも値を変えて結果を確かめてみてください．こういったことが手軽に確認できるのも，Swift Playground の良いところですね．

## 配列と繰り返しの処理

Swift で配列を宣言するには，次のように角括弧 [] の中にコンマで区切って複数の値を並べて書きます．

```
var a = [ 3, 1, 5, 2 ]
a.sort()
a.append(6)
a.insert(4, at: 3)
a.remove(at: 0)
for value in a {
     print(value)
}
```

こうして宣言した配列は，このように sort()，append(_)，insert(_, at:)，remove(at:) といったメソッドを使うことで変更できます．（もちろん let を使って宣言した変数に対してはこれらの変更はできません）ドット記号 (.) を使ってメソッド呼び出しを書く文法は基本的に C# などと同様ですね．そして for-in 文を使うことで，格納された値を 1 つずつ取り出して利用することができます．なお，ここで登場する数字は配列に格納する数値と配列内の位置を表す添字とがありますが，添字として数字を使う場合には「at:」というラベルが使われているので，数値と添字を混同する危険性がありません．

なお，C 言語と同様の 3 項目からなる for 文も初期の Swift には用意されていましたが，分かりにくくバグにつながりやすい文法であるとして，Swift 3 から削除されました．もちろん次のように書くことで，添字を使ったアクセスも可能です．

```
var a = [ 3, 1, 5, 2, 4 ]
for i in 0..<a.endIndex {
      print(a[i])
}
```

しかし特別な理由がない場合には，配列に対して直接 for-in 文を使って値を取り出す方が安全かつ高速な結果が期待できます．なお，逆順にアクセスしたい場合には，次のように配列に対して reversed() メソッドを呼び出した上で for-in 文を使います．

```
var a = [ 3, 1, 5, 2, 4 ]
for value in a.reversed() {
     print(value)
}
```

もちろん reversed() メソッドは配列自体を逆順にソートし直すようなことはせずに，逆順に参照する方法を提供するだけですので，これによって実行速度が悪くなるようなことはありません．これも効率を徹底的に重視した Swift の特徴であると言えます．

Swift では for-in 文や if 文の後ろに，追加で where 節を書くことができます．次のように取り出す値に対して満たさなければならない条件を where 節に書いておくことで，対象となるオブジェクトを絞り込みながら処理を行うことができます．

```
var a = [ 3, 6, 1, 5, 2, 4 ]
for value in a where value%2==0 {
     print(value)     // 6,2,4だけが表示される
}
```

なお，配列というのは上記の例では Int 型のコレクションなのですが，最新の Swift 4 では文字列も Unicode の Character 型のコレクションとなっていますので，次のように文字列に対して for-in 文を書いて各文字にアクセスできますし，先程と同様に reversed() メソッドや where 節を使った絞り込みも利用できます．

```
let str = "Hello"
for c in str {
     print(c)
}
```

次のように，文字列に対して split（separator:）メソッドを使うことで，区切り文字を指定して文字列を分割することができます．

```
let str = "ABC,DEF,GHI"
let a = str.split(separator: ",")
for c in a {
      print(c)
}
```

Swift の文字列は通常 String クラスのオブジェクトとして表されますが，このメソッドで分割された個々の部分文字列は String クラスのオブジェクトではなく，Substring クラスのオブジェクトとして表されます．Substring クラスは直接文字列のデータを格納するためのメモリ領域を確保せず，String オブジェクトへの参照を内部的に保持しつつ抜き出す対象となる個所の範囲を保持することによって，効率的な処理が可能になっています．

Swift で配列と合わせて覚えておきたいのが，数値の範囲を表す記法です．たとえば「1...10」と書くと，1 以上 10 以下の範囲を表すことができます．範囲指定のための「...」演算子を範囲演算子（Range Operator）と呼びます．この範囲に対して for-in 文を使うと，範囲内のすべての数値に対して処理を行うことができます．たとえば，1, 2, 3, ..., 10 の値を表示するプログラムは次のようになります．

```
for i in 1...10 {
      print(i)
}
```

半開きの範囲演算子「..<」を使って「0..<10」と書くと，0 以上 10 未満の範囲を表すことができます．次のプログラムは，0, 1, 2, ..., 9 の値を表示します．

```
for i in 0..<10 {
      print(i)
}
```

範囲演算子を使うと，特定の範囲内の Unicode 文字を表示するプログラムも，次のように簡単に書けます．

```
// 1f300 1f3ffの範囲の絵文字を表示する
for v in 0x1f300...0x1f3ff {
     let c = Character(UnicodeScalar(v)!)
     print(c)
}
```

範囲には contains() メソッドが用意されていますので，ある数値が特定の範囲内にあるかどうかをチェックすることができます．次のプログラムを実行すると，4.999 は範囲内の数値ですので「YES」が表示されます．範囲に対するチェックは，整数でも小数でも行えます．

```
var value = 4.999
if (1..<5).contains(value) {
      print("YES")
} else {
      print("NO")
}
```

for-in 文で処理をする配列に範囲を付けることで，配列内の特定の範囲の要素に対してだけ処理を行うことができます．次のプログラムでは，添字の範囲が 1 ～ 3 の数値だけを表示しますので，「1」「5」「2」が表示されます．

```
var a = [ 3, 1, 5, 2, 4 ]
for v in a[1...3] {
      print(v)
}
```

配列に付ける範囲は，開始値や終了値を省略することができます．開始値を省略して次のように書くと，先頭から添字が 2 までの要素が表示されますので，「3」「1」「5」が表示されます．

```
var a = [ 3, 1, 5, 2, 4 ]
for v in a[...2] {
      print(v)
}
```

同様に終了値を省略して次のように書くと，添字が 3 の要素から最後の要素までが表示されますので，「2」「4」が表示されます．

```
var a = [ 3, 1, 5, 2, 4 ]
for v in a[3...] {
      print(v)
}
```

## 関数の定義と呼び出し

Swift の変数宣言や if 文，for-in 文は C 言語と似た形ですが，関数の定義とその利用方法は少し異なります．まず関数の実装は次のように書きます．「func」キーワードの後ろに関数名を書き，その後ろの丸括弧の中に引数リストを書きます．それぞれの引数は「引数名 : 型名」の形で書きます．

```
func add(a:Int, b:Int) -> Int {
     return a + b
}
```

こうして定義した Swift の関数は，C 言語の関数と同じように関数名の後ろに丸括弧を書いて呼び出します．ただし，引数の前にコロンを付けて引数名を明示しているのが異なります．Swift の関数は，デフォルトでは引数名を省略できません．

```
let v = add(a:3, b:5)
print(v)
```

関数の宣言において，引数名の前にスペースを空けてもう 1 つ名前を書いておくことで，関数を呼び出す側で書く引数名をカスタマイズできます．これをラベルと言います．関数の実装部では，ラベルではなく実際の名前を使って実装コードを書きます．

```
func add(one a:Int, two b:Int) -> Int {
     return a + b
}
let v = add(one:3, two:5)
```

ラベル名をアンダースコア (_) にしておくと，ラベルを省略して関数を呼び出せるようになります．

```
func add(_ a:Int, _ b:Int) -> Int {
     return a + b
}
let v = add(3, 5)
```

ラベルを省略すべきかどうかは，場合によります．たとえば配列に新しい要素を挿入する insert(_,at:) 関数では，第 2 引数のラベルを「at:」という名前にすることで，第 1 引数の追加対象の要素と第 2 引数の添字との違いを明確にしています．

```
var array = [ 1, 3, 7 ]
array.insert(5, at: 2)
```

「これまでC言語でプログラミングしてきて見慣れないから」というような理由で引数を省略するのではなく，引数の意味を明示した方がよい場所と，簡潔に省略した方がよい場所の違いをしっかりと見極めることが肝心です．ここでは関数呼び出しについてルールを説明しましたが，オブジェクトに対するメソッド呼び出しでもルールは同じです（厳密には初期のSwiftでは関数とメソッドでルールが異なったのですが，現在のSwiftでは同じルールに統一されました）．

## 変数や関数の名前

変数名や関数名には，予約語や演算子に使われている文字以外のUnicode文字を自由に使うことができます．たとえば，次のようなコードはいずれも問題なく認識されます．

```
let value = 123      // OK
let 🐋 = 123         // OK
let π = 3.14159      // OK
```

ただし，可読性の観点からいえば，変数名にギリシャ文字や絵文字を使うのは避けたいところです．たとえば次の絵文字はすべて同一の形状の顔を表す絵文字に，異体字を表すセレクタと呼ばれるUnicode文字を組み合わせて，異なる肌の色を表しています．これらはすべて異なる変数になりますが，混同しやすいことはいうまでもありません．

```
let 👴 = 123
let 👧 = 345
let 👩 = 3.14159
```

通常のアプリケーション開発においては，従来通りアルファベットを使った英語ベースの命名を行うことが推奨されるでしょう．とはいえ，少し取り入れるだけで無機質な見た目のプログラムのコードが華やかになりますので，初心者に向けた講座などに取り入れる価値は大いにあると思います．また，変数名や関数名に日本語が使えますので，プログラムの解説記事を限られたスペースの紙面で書くときにも有用だと思います．

## 列挙型

Swiftで特徴的なのが列挙型です．まず基本的にはC言語と同様の使い方ができますので，次のように書いて，武器の種類を表す変数を用意することができます．「enum」キーワードの後ろに列挙型の名前を書き，波括弧の中に「case 定数名」と書くことで，関連のある定数をひとまとめにして定義できます．

```
enum 武器 {
    case 剣
    case 盾
}
var w1 = 武器.盾
var w2:武器 = .剣
if w1 == .剣 {
    print("剣です")
} else {
    print("剣ではありません")
}
```

列挙型は，列挙型の定数を格納するための変数を用意して使います．変数w1の例のように，列挙型の値を代入すると，その列挙型の定数を格納するための変数として初期化されます．変数w2の宣言では，変数名の後ろにコロンを付けて武器の列挙型を使うことを明示していますので，代入文の右辺には「武器.剣」と書かずに「.剣」と列挙型の名前を省略して定数名を書けます．

C言語のenumは，自動的に加算されていく整数値を割り振って，値の異なる定数を作る機能ですが，Swiftのenum定数には値は割り振られません．Swiftのenumは，if文やswitch文などで，列挙型の変数の値が定数と等しいかどうかが確認できるだけのシンプルな機能です．

しかしSwiftの列挙型でも，型名の後ろにコロンを付けてInt型のenumであることを明示的に指定すれば，C言語のenumと同様に0から順番に定数の値が加算されながら割り振られます．列挙型の変数に対して「.rawValue」と書くことで，定数に対応した値を取得することができます．なお，print()関数に列挙型の値を渡すと，定数の名前がそのまま表示されます．これはデバッグにとても便利な仕様です．

```
enum 武器: Int {
    case 剣
    case 盾
}
var w:武器 =.剣
print(w)                    //「剣」と表示
print(w.rawValue)   //「0」と表示
```

Swift の enum はさらに便利な機能を備えています．たとえば武器として剣を選択したときに，合わせてその剣のパワーを設定したいとします．同様に，盾についてはサイズと重量を設定したいとしましょう．こういったデータをまとめて管理するのは通常はなかなか難しいのですが，Swift の enum にはこれをサポートする Associated Value（値の関連付け）という文法があるのです．次のように，定数名の後ろに丸括弧を書いて，その中に関連付ける値の名前と型名をコロン区切りで書きます．定数ごとに異なる種類・異なる個数の値を関連付けられます．

```
enum 武器 {
    case 剣(パワー:Int)
    case 盾(サイズ:Float, 重量:Float)
}

var w:武器 = .盾(サイズ:50.5 重量:6.5)
w = .剣(パワー:3)
```

こうして関連付けられた値がある列挙型は，先程のように単純に == を使って比較することはできなくなります．次のように if-case 文を使って，剣であるかどうかを比較します．

```
if case .剣 = w {
      print("剣です")
} else {
      print("剣ではない")
}
```

関連付けられた値を取得したい場合には，定数名の後ろに丸括弧を書いて「let」キーワードで定数を宣言し，その中に値を取り出します．取り出す必要のない値はアンダースコア (_) を書いて省略できます．

```
if case .剣(let power) = w {
      print("剣です。パワーは\(power)")
} else if case .盾(let size, _) = w {
      print("盾です。サイズは\(size)")
}
```

## クラスの定義と利用

Swift におけるクラス定義は，C++ や C# に似ています．「class」キーワードに続いてクラス名を書き，波括弧の中に変数や関数を並べて書くことで，クラスの属性やメソッドを定義していきます．

```
class Player {
    var x: Float
    init() {
        x = 0
    }
    func move() {
        x += 10.5
        print(x)
    }
}
```

「init()」はイニシャライザと呼ばれるメソッドで，オブジェクトの生成時に状態を初期化するために呼ばれます．C++ や C# で言うところのコンストラクタですね．イニシャライザの先頭には「func」キーワードは付けません．

こうして定義したクラスのオブジェクトを作るときには，クラス名の後ろに丸括弧を付けて次のように書きます．C++ や C# のように「new」演算子は使いません．作成したオブジェクトに対してメソッド呼び出しするときには，ドット (.) を付けてメソッド名を書き，丸括弧の中に引数を書きます．

```
var player = Player()  // Playerクラスのオブジェクトが生成される
player.move()              // move()メソッドが実行され、xに10.5が追加される
```

派生クラスを用意するときには，クラス名の後ろにコロンを書いて，その後ろに派生元となるクラスの名前を書きます．既存のメソッドの動作を上書きする時には，「func」キーワードの前に「override」キーワードを追加する必要があります．上書きするメソッド実装の中から親クラスのメソッドを呼び出したい場合には，「super」というキーワードを使って親クラスを参照します．

```
class Player2: Player {
    var y: Float = 0.0
    override func move() {
        super.move()
        y += 0.5
        print(y)
    }
}
```

なお，Swift にも C＋＋ や C# などと似たアクセス制御のための「public」や「private」といったキーワードがあります．これらのキーワードをクラスやメンバ変数やメンバ関数に対して付加してアクセスレベルを制御できるのですが，Swift では変数や関数に対するクラス間でのアクセス保護というよりも，ライブラリやフレームワークといったレベルでクラスを公開するときの仕組みとしてアクセス制御の仕組みが用意されています．そのため，基礎の文法について解説するこの記事では扱いません．

## クラスに getter と setter を追加する

クラスには，変数としての実体を持たないプロパティを追加して，そのプロパティに独自の getter と setter を用意することができます．getter と setter を使うことで，クラス内の変数に単純にアクセスしているような書き方をするだけで，実際の変数に格納する前に値をチェックしたり，計算結果に基づく値をセットしたりすることができます．

クラスにプロパティと getter や setter を追加するには，次のようなコードを書きます．

```
class Person {
    var firstName = ""
    var lastName = ""

    var fullName: String {
        get {
            return firstName + " " + lastName
        }
        set(str) {
            let a = str.split(separator: " ")
            firstName = String(a[0])
            lastName = String(a[1])
        }
    }
}
```

ここでは人を表す Person クラスを定義して，下の名前を表す firstName と名字を表す lastName 変数を用意しています．そして firstName と lastName を結合した文字列を取得するために，fullName というプロパティを用意しています．

プロパティは，このように通常の変数と同様に「var」キーワードを使って宣言しますが，後ろに波括弧を書いて，その中に「get」キーワードから始まるブロックを書くことで getter を，「set」キーワードから始まるブロックを書くことで setter を用意します．

Person クラスを用意して，firstName と lastName の値をそれぞれセットしたあと，fullName プロパティを参照してみましょう．fullName プロパティの getter が呼ばれて，firstName と lastName が空白文字で結合された値が取得できていることが確認できます．

```
var person = Person()
person.firstName = "Satoshi"
person.lastName = "Numata"
print(person.fullName)
```

また，fullName プロパティに対して文字列をセットしてみましょう．すると fullName プロパティの setter が呼ばれて，split（separator:）メソッドで空白文字を区切り文字として 2 つの文字列に分割したあと，最初の文字列を firstName にセットし，2 つ目の文字列を lastName にセットしていることが確認できます．

```
person.fullName = "Osamu Dazai"
print(person.firstName)
print(person.lastName)
```

もちろんメソッドを定義しても同じことはできますが，このように代入の文法を使って簡潔に書けることは，ソースコードが煩雑にならずに見やすいように保つ上で，重要なことだといえます．

なお，getter だけのプロパティを用意する場合には，「get」キーワードを省略して，次のようにプロパティの実装を書くことができます．

```
class Person {
    var firstName = ""
    var lastName = ""

    var fullName: String {
        return firstName + " " + lastName
    }
}
```

## 構造体の定義と利用

Swift ではC言語のようなポインタがなく，クラスから作られたオブジェクトは，代入や引数として値を渡す操作において，すべて参照渡しとなります．次のようなコードを書くと，変数 v1 も変数 v2 も同じオブジェクトを参照しているため，v1.x の値を変更したことが v2 の値にも影響しています．

```
class Vec2 {
    var x:Int
    var y:Int
    init(_ x_:Int, _ y_:Int) {
          x = x_
          y = y_
    }
}

var v1 = Vec2(3, 5)
var v2 = v1
v1.x += 10
print("v1: \(v1.x),\(v1.y)")  // "v1: 13,5"と表示
print("v2: \(v2.x),\(v2.y)")  // "v2: 13,5"と表示
```

これに対して，クラスを定義するときに使う「class」キーワードを「struct」キーワードに変更すると，代入や引数として値を渡す操作において，すべてコピー渡しとなります．そのため，先程と同様に変数 v1 の x の値を変更しても，その値の変更が変数 v2 に影響を及ぼすことはありません．

```
struct Vec2 {
    var x:Int
    var y:Int
    init(_ x_:Int, _ y_:Int) {
          x = x_
          y = y_
    }
}

var v1 = Vec2(3, 5)
var v2 = v1
v1.x += 10
print("v1: \(v1.x),\(v1.y)")  // "v1: 13,5"と表示
print("v2: \(v2.x),\(v2.y)")  // "v2: 13,5"と表示
```

なお，クラスと構造体の違いとして，初期化されていない変数が定義されている場合に，構造体にはイニシャライザを用意しなくても，作成時にラベル付きで必要な値を引数として書いておけば初期化できるということが挙げられます．クラスは初期化されていない変数が定義されている場合には，イニシャライザを用意しなければいけません．さきほどの構造体の例は，次のように書き換えられます．

```
struct Vec2 {
    var x:Int
    var y:Int
}

var v1 = Vec2(x: 3, y: 5)
```

なお，引数の並び順は，構造体に定義されている変数と同じ順番で書かれている必要があります．Vec2 では「x」「y」の順番に変数が定義されているため，次のように「y」「x」の順番で値を渡して Vec2 構造体を初期化しようとするとエラーになります．

```
var v1 = Vec2(y: 3, x: 5)    // エラー!
```

構造体にもクラスと同様にメソッドを用意することができますが，構造体では値を変更するメソッドはデフォルトでは書けないようになっています．値を変更するメソッドを書く場合には，そのメソッドに「mutating」キーワードを追加しなければいけません．

```
struct Vec2 {
    var x:Int
    var y:Int

    // 値を変更しないメソッドはそのまま書ける
    func sqrMagnitude() -> Int {
        return x * x + y * y
    }

    // 値を変更するメソッドは「mutating」キーワードが必要
    mutating func update() {
        x += 10
        y += 10
    }
}
```

構造体はクラスと違って，親の構造体から派生させた親子関係のある構造体を定義することができません．

## iPad で Swift を勉強する

iPad には，Apple が公式で提供している Swift を勉強するための「Swift Playgrounds」というアプリがあります（図 -8）．Swift Playgrounds は App Store で検索して，無料でインストールできます．このアプリでは初心者に向けて，プレイグラウンドと呼ばれるさまざまなステージが用意されおり，各プレイグラウンドにはゲーム仕立てのゴールが設定されています（図 -9）．ゴールを達成するためにさまざまなプログラムのコードの断片を組み合わせて書くことで，Swift の基本的な文法を勉強して，Swift を応用したアプリ制作の方法まで幅広く学ぶことができます．

プログラミング教育に活用できる内容も豊富に用意されていて，カメラを通して見る拡張現実を活用したアプリが書けるプレイグラウンド（図 -10）や，Bluetooth で通信可能なレゴやドローンといった実際のハードウェアを動かすことができるプレイグラウンド（図 -11）なども用意されています．iPad をお持ちの方はダウンロードして，充実した内容を確認してください．

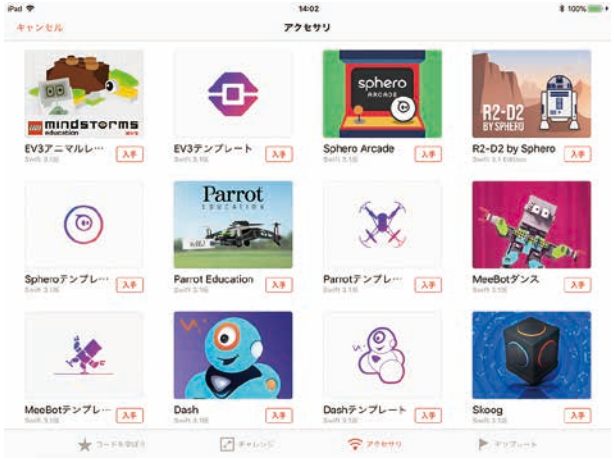
（2017 年 10 月 29 日受付）

**沼田哲史（正会員）** numata@osakac.ac.jp
1978 年 1 月生．2005 年大阪大学大学院情報科学研究科にて博士（情報科学）取得．同年より大阪電気通信大学総合情報学部デジタルゲーム学科講師．「作りながら覚える iOS プログラミング」（SB クリエイティブ）ほか iOS プログラミングの著書多数．


■図 -8　Swift Playgrounds のアイコン


■図 -9　トップ画面に並ぶ各種のプレイグラウンド


■図 -10　拡張現実のプレイグラウンド




■図 -11　追加可能なデバイス操作のプレイグラウンド