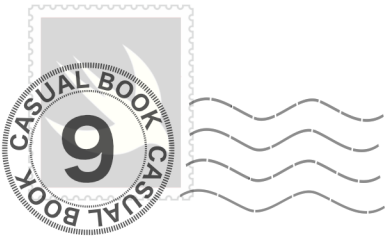

Swift 変数入門

Constants & Variables

FROM
熊谷 友宏

CASUAL BOOK #09

Swift 変数入門

お試し版 (build-05649)

著　熊谷 友宏

はじめに

この度は CASUAL BOOK シリーズを手に取ってくださり、ありがとうございます。このシリーズは、筆者がプログラミングの勉強会を開催していく中で募った『プログラミングの楽しさを伝えたい』気持ちを表現する方法として辿り着いたひとつの形です。技術書という形に代えて、プログラミングの楽しさが勉強会という場を超えて届いてくれたら嬉しく思います。

本書で扱う内容

Swift 言語では変数を `let` と `var` という 2 種類のキーワードを使って定義します。前者なら値を 1 回設定したら変えられない変数を、後者なら何回でも値を変えられる変数を、用途に応じて定義できるようになっています。

これだけでも変数を使うことはできますけれど、そうやって用途を区別する意味や、それが影響してくるところを知っておくとコードをもっと書きやすくなります。そしてこの違いは変数を使うときだけでなく、自分で型を定義するときにも意識する必要が出てきたりもします。ほかにも、定義する場所によって性格がわずかに違ってきたり、値を書き込んだときの振る舞い方を指定できたり。何気なく存在している Swift の変数ですけれど、変数はいろんなところでいろんな姿を見せてくれます。本書はそんな変数を切り口にして Swift 言語の基礎を見つめていく『Swift 言語の入門書』です。

本書は、お試し版です！

本書は『Swift 変数入門』の **執筆途中のお試し版** になります。技術書典 7 での頒布に間に合わせるべく制作していたのですけれど、**原稿が間に合う見込みがなかったため**、仕上がっている第 4 章までをお試し版として頒布することにしました。引き続き “技術書同人誌博覧会 2” までの完成を目指して制作していきますので、それまでの間はこのお試し版をお手元に置き、お楽しみにしていて頂けたら幸いです。

完成した際には、技術書同人誌博覧会 2 での製本版はもちろん、BOOTH 等で販売開始する予定ですので、どうぞよろしくお願いいたします。詳細は次の URL でお知らせしていきます。

頒布書籍の情報はこちらに掲載しています。
https://ez-net.jp/book/

想定環境

- Swift 5.1
- Xcode 11.0

留意事項

本書は著者が独自に調査したものを扱っています。本書に登場するシステムや製品などの名称は一般に各社の商標または登録商標です。本書では ©や ®、™ の表記を省略している場合があります。

目次

第 1 章

変数の要所をおさえる

Swift の変数は値を入れる役割だけに留まらず、さまざまな機能を提供しています。まずはそれらを言語仕様の観点から確認しておきましょう。

ここでは変数の定義のしかたや、それに用意されている機能について見ていきます。Swift 言語を学んで間もない人や基礎的なところから復習したい人でも、変数の機能をひととおり把握できるように網羅的に紹介します。

1.1　変数と定数

変数というのは一般に、**ソースコードで扱うデータを読み書きするのに使う記憶域**のことを言います。

これらを説明するときは、それぞれの価値観によって“値を入れる入れ物”に例えられたり“値に付ける名札”に例えられたりしますけれど、いずれにしても**プログラマーが何らかの方法で、自由に値を割り当てた**ものになっています。そして、変数に値を割り当てることを**“代入”**と表現します。

変数というと“自分で定義して値を入れるもの”を想像しがちに思いますけれど、そのほかの場面でも変数の仕組みが暗黙的に使われます。たとえば、**関数を実行するときにも変数が自動的に作られて**、引数の値や戻り値の受け渡しが行われるようになっています。

変数に対する操作

そんな変数を扱うにあたっては、**宣言・代入・参照**の 3 つを意識しておくと理解がしやすくなりそうです。

Swift 言語で変数を使うときには、その変数がどのようなものであるかをあらかじめ**『宣言』**しておく必要があります。そうして宣言した変数には、それを使う前までに必ず初期値を**『代入』**する必要があります。これについては <1.2> で説明します。このようにして変数を定義することになります。

変数を定義したら、それを使うことができます。変数に代入されている値を読み出して使うことを**『参照』**と表現します。値が設定されていない変数を参照すると、Swift 言語ではコンパイルエラーになります。Swift 言語は特にこの**変数は必ず初期化してから参照することを徹底**しているので、それについては第 3 章で見ていくことにします。

不変性と可変性

Swift 言語は、値に**2 種類の性格**を持たせて扱います。2 種類の性格というのは**“不変性”**と**“可変性”**、すなわち“代入されている値を変更できない性格”と“代入されている値を変更できる性格”の 2 つです。

これらの性格は、値そのものではなく、その値の代入先として使う記憶域によって決まってきます。そんな記憶域の種類は、それを宣言するときに使

うキーワードが“let”であるか“var”であるかで決まります。このとき、let で宣言した記憶域を**“定数”**(Constant) と呼び、var で宣言した記憶域を**“変数”**(Variable) と呼びます。

```
1.  ❗ 値を変更できない "定数" を定義
2.      let value: Int = 0
3.
4.  ❗ 値を変更できる "変数" を定義
5.      var value: Int = 0
```

そんな定数と変数の性格の違いを端的に挙げると**『値を再代入できるかどうか』**になります。それによってどのような違いが現れてくるのか、そしてそれがどんな効果をもたらすのかについては、第 2 章で整理していきます。

1.2　定義と代入

変数や定数の定義と値の代入のしかたを見ておきましょう。たとえば変数を定義したいときには、キーワード var を使って次の書式で記載します。

```
1.  var 変数名: 型名 = 初期値
```

このようにすることで**『型名』**で指定した型の値を扱える変数が**『変数名』**という名前で定義され、最初の値として**『初期値』**が代入されます。

この例では var キーワードを使って定義したので**『変数』**として定義されましたけれど、ここを let キーワードに変えるだけで**『定数』**として定義さ

れるようになります。変数だけに用意されている機能もいくつかある[*1]のですけれど、基本的には **どちらも同様に定義できる** ようになっています。

値を代入するときの書式

定義した変数に値を代入するときは、次の書式で記載します。

```
1.  変数名 = 値
```

こうすることで**『変数名』**という名前で定義しておいた変数に**『値』**で指定した値を代入するという意味になります。

このとき、記号**『=』**で区切った左側を**“左辺”**といい、右側を**“右辺”**といいます。動作のイメージとしては、右辺に書かれた評価式を計算して、その結果として得られた値を左辺で指定した記憶域に保存する流れになります。

型で値を制限することによる利点

変数や定数に代入できる値は、定義のときに指定した型の値か、それに相当する値の場合だけに限られます。

このような宣言時に型を定める仕組みのことを“静的型付け”といいます。この仕組みによって変数で扱う値の種類をコンパイラーが判断できるようになり、**代入できる値が妥当であるかを検出** したり、**値の解釈を間違えて計算結果が不整合になるのを未然に防ぐ** といった効果が期待できます。

[*1] 変数だけで使える機能には、第 5 章の『初回参照時まで初期値の計算を遅らせる』仕組みや、第 6 章の『値の変化を検出する』仕組みなどがあります。

たとえば、定数 value を Int 型で定義して、それに対して "TEXT" という値を代入しようとすると、コンパイルエラーになります。

```
1.  // Cannot convert value of type 'String'
2.  // to specified type 'Int'
3.
4.  ❗ let value: Int = "TEXT"
```

これは、定数 value は Int 型として宣言されているので整数の値しか代入できないところに、文字列リテラルを代入しているためです。文字列リテラルは既定では String 型として扱われるので、宣言した型に適合しません。

このような型の検査は、値を代入するときだけに限らず、2 つの値を演算するときや、関数の引数に値を渡すときなどに、処理対象の値が適切かどうかを検証するのにも役立ちます。

このように型を使って取り得る値を制限することで、不適切な値の受け渡しをプログラムの実行前に見つけられるという利点だけでなく、どんな値が適切かが明瞭になってソースコードを書く負担も軽減されます。

1.3　型を明記しなくて良い場合

変数や定数を宣言するときに、基本的にはそれが扱う型を明記する必要があるのですけど、前後の文脈からわかる型の記載は省略できます。

たとえば、次のような定数の定義があったとします。

```
let value: Int = 0
```

初期値が 0 の定数 value を Int 型で定義しているのですけれど、初期値に指定している“整数リテラル”は、既定では Int 型として扱われます。

初期値が Int 型であるなら変数も Int 型であることは容易に想像できるので、それが適切な場合には、次のように変数の型の記載を省略できます。

```
✅ let value = 0
```

このような前後の文脈から型を特定する仕組みを**“型推論”**と呼びます。

型推論といえば、ほかにも Swift 言語の整数リテラルは、浮動小数点数を表現する Double 型として扱えることになっています。

既定では Int 型として扱われるため、Double 型として扱うためには型の明記が必要なのですけれど、前後関係から『この文脈では Double 型がふさわしい』と判断されれば、型を明記しなくても Double 型として扱われます。

そんな型推論の挙動について、具体的なコードをいくつか紹介しておきます。

```
✅ 定数宣言での型を省略しない場合（整数リテラルだけ型推論）
    let value: Double = 0

✅ 定数宣言での型と、整数リテラルの型を両方指定（型の省略なし）
```

```
    let value: Double = 0 as Double

✅ 整数リテラルを Double 型として扱うことを明記（定数の型推論）
    let value = 0 as Double
```

ちなみに型推論は、変数や定数の定義ではない場所でも活用されています。たとえば、引数に Double 型の値を受け取る関数があったとき、その引数に整数リテラルを渡すと、それは Double 型として扱われます。

次の例のように、第 1 引数に Double 型を、第 2 引数に Int 型をとる関数があった場合は、それぞれの引数に対して適切な型が推論されます。

```
❗ 関数が、第 1 引数で Double 型を、第 2 引数で Int 型をとるとき、
func subtotal(price: Double, amount: Int) -> Double {

    ⋮
}

✅ 引数に渡したそれぞれの整数リテラルは、
    第 1 引数は Double 型に、第 2 引数は Int 型に推論される

let billing = subtotal(price: 120, amount: 5)
```

宣言と初期化は分離できる

変数や定数の宣言と初期値の代入は別々のタイミングでも行えます。たとえば『変数は定義しておきたいけれど、初期値はまだ決まらない』ときには、定義のときに型情報まで記載して、初期値の代入は書かないでおきます。

```
1.  var 変数名: 型名
```

このようにすると、初期値が代入されていない変数を定義できるので、あとはそれを参照するまでに、初期値を代入するようにします。このように初期化自体を遅らせる方法が Swift 言語にはいくつか用意されているのですけれど、それを含めた詳細については第 5 章で紹介します。

1.4 保存型変数と計算型変数

これまでに紹介してきた変数は、データを保存する記憶域を持っている変数だったのですけれど、それを持たない変数というのも存在します。

Swift 言語では前者を **保存型変数**[*2] と呼び、後者を **計算型変数**[*3] と呼びます。使うときにはどちらも同じ感覚で参照できますけれど、定義方法は異なってきます。計算型変数の定義は、次のような書式になります。

定義方法としては、宣言を `var` キーワードを使って行い、参照時の処理を `get` キーワードで、代入時の処理を `set` キーワードで指定します。

```
1.  var 変数名: 型名 {
2.
3.      get {
4.
5.          ❗ 参照結果としてこの値を返す。
```

[*2] 英語では“Stored Variables”と表現します。
[*3] 英語では“Computed Variables”と表現します。

```
6.            return 値
7.        }
8.
9.        set {
10.
11.           ❗ 代入された値が変数 newValue として渡ってくる。
12.           ⋮
13.       }
14.   }
```

変数と聞くと、変数とは“値を保存するもの”のように感じて、記憶域を持たない変数が不思議なようにも思えますけれど、もう少し広い視野で**“変数はデータを読み書きするのに使うもの”**と捉えると、こういった変数の在り方も自然なものに感じられます。

そんな計算型変数の定義方法や、そこに見られる特徴については、第 7 章で詳しく紹介します。

1.5　値が代入されたときの追加処理

保存型変数にも、変数の参照時や代入時に任意の処理を実行させる手段が用意されています。定義方法は、変数宣言と合わせて `willSet` キーワードや `didSet` キーワードを使用して、追加で処理したい内容を記載します。

```
1.   var 変数名: 型名 = 値 {
2.
3.       willSet {
4.
```

```
 5.          ❗ 値が保存される直前に実行する処理
 6.              ⋮
 7.      }
 8.
 9.      didSet {
10.
11.          ❗ 値が保存された直後に実行する処理
12.              ⋮
13.      }
14.  }
15.
```

この機能の詳しい定義方法や実行タイミング、定義できる場所などのさまざまな特徴については、第 6 章で紹介します。

1.6　プロパティーという言葉が指すもの

変数と同等のものを表現する言葉に **プロパティー** というものがありますけれど、こちらはとりわけ **型に所属する変数を表す言葉** になります。

Swift 言語では、変数とプロパティーとの違いは、型に所属していることを意識しているかどうかの違いくらいで、機能的には同等と捉えておいて大丈夫そうです。型に所属しているかを気にする必要のない場面では、プロパティーも含めて“変数”と表現することもあります。

プロパティーにも `let` で宣言するものと `var` で宣言するものとがあるのですけれど、プロパティーの場合は“定数”や“変数”みたいな言葉の区別は特になく、**両方をひっくるめて“プロパティー”と表現** します。そう

やって区別なく表現することについての所感は <2.7> で綴っています。それでも特に不変性と可変性に着目したいときには**“不変性を持つプロパティー”**[*4] や**“可変性を持つプロパティー”**[*5] みたいに適宜表現します。

1.7　型プロパティーを定義する

変数や定数には、型に所属するものがあります。型に所属する変数と定数の両方をあわせて**型プロパティー**[*6] と呼びます。

定義方法は基本的には普通の変数や定数と同じなのですけれど、定義の冒頭に `static` キーワードを添えて定義します。このとき**型プロパティーは所属する型の内側のいちばん最初のスコープに定義する**必要があります。

```
1.  struct SomeType {
2.
3.      static let 変数名: 型名 = 初期値
4.  }
```

型プロパティーを型宣言のいちばん最初のスコープではない場所、たとえば大域変数や関数内の局所変数などで定義すると『`Static properties may only be declared on a type`』というコンパイルエラーになります。スコープや大域変数に関する事柄については第 10 章で説明します。

[*4] 英語では“Constant Properties”と表現されます。
[*5] 英語では“Variable Properties”と表現されます。
[*6] 英語では“Type Properties”と表現します。

再代入と参照の挙動について

型プロパティーへの値の代入操作についても、基本的には通常の変数や定数と同じです。先ほどの例では型プロパティーを `let` で定義したので、初期値を入れた後での値の再代入はできないですけれど、これを `var` で定義すれば、初期化の後でも値を再代入できるようになります。

ただ、データを保存する記憶域や参照時の挙動の感触が、通常の変数と比べて少し違ってきます。それについては第 9 章で説明します。

第 2 章

定数と変数の使いわけ

Swift 言語で扱う『値』は < 1.1 > でも紹介したように、“定数” で扱う場合と “変数” で扱う場合とで異なる性格を示します。ここではこれらの性格の違いと、それによってどんな効果が得られるのかを整理していきます。

2.1　変数の特徴

まずは “変数” の特徴を見ていきましょう。var キーワードで定義する記憶域ですけれど、この主だった特徴は**『値を何回でも代入できる』**ところです。

```
1.  var price = 200
```

こうして変数として定義された記憶域に代入された値は可変性を持つようになり、いつでも別の値を再代入できます。

```
1.  ✅ price = 150
```

このように、変数に割り当てられている値を別の値に書き換えられる性質のことを**“ミュータブル (Mutable)”**と呼びます。

変数という言葉からもイメージしやすい性格ですし、この性質を示すプログラミング言語も多いので扱いやすい印象ですけれど、Swift 言語ではもうひとつの別の性格を持つ“定数”を率先して使っていくことになります。

2.2　定数の特徴

続いて“定数”の特徴について見ていきます。こちらは let で定義する記憶域で、主だった特徴は**『初期値を代入したら変更できない』**ところです。

```
1.  let productName = "Water"
```

このようにして、定数として定義された記憶域に代入された値は不変性を持つようになり、初期化が終わった後は別の値を再代入できなくなります。

```
1.  // "Cannot assign to value:
2.  //    'productName' is a 'let' constant
3.
4.  ❗ productName = "Coffee"
```

このように、変数に割り当てられている値を別の値に書き換えられない性質のことを**“イミュータブル (Immutable)”**と呼びます。

“定数” という言葉を聞いたとき、人それぞれで思い浮かぶ意味合いが異なるかもしれないですけれど、Swift 言語でいう定数は**“初期値は自由に代入できるけれど、それ以降は変わることはない”**という意味での “定まった数” になります。いわゆる円周率 π みたいな普遍的な数を意味しているのではないことを意識しておくと、定数の意味を捉えやすくなりそうです。

2.3　定数が保護する範囲

そんな変数と定数を使い分けていくにあたって、定数が保護する値の範囲をもう少し詳しく見ておきましょう。定数が値を保護する範囲は “参照型に区分される値” と “値型に区分される値” とで異なってくるのがポイントです。

参照型の値の保護範囲

参照型というのは、ひとつのインスタンスを複数の変数や定数で共有する、いわゆるオブジェクト指向プログラミング言語でいうオブジェクト型のことを指します。Swift 言語では**クラス型**と**関数型**が参照型に当たります。

参照型に対して定数が書き換えできなくする範囲は**どのインスタンスを参照しているかの指し示し先だけ**になります。つまり、定数が指しているクラス型や関数型のインスタンス自体が途中のどこかで指し変わらないことだけを保証して、その指し示し先のクラス型のインスタンスが持つプロパティーや、関数型のインスタンス[*7]が内包している変数の値までは保護しません。

[*7] 関数型のインスタンスには、関数やメソッドの名前だけを指定して代入したものや、クロージャーなどがあります。

言い換えると、定数に代入された参照型のインスタンスは『指し示し先が変わらないことが保証され、指し示し先のインスタンスの内部状態は自由に制御できる』ものになります。

つまり、定数で参照型の値を扱うことの効果としては、インスタンスを複数箇所で共有しているときに、すべての箇所で定数を用いて扱っておくと、**どこかの箇所だけいつの間にか他と違う場所を指し示しているみたいなことを予防する効果**が期待できます。

値型の値の保護範囲

値型は、そのインスタンスを“単一の値”として扱うことが想定された型になります。そんな値型の最大の特徴は、インスタンスが内包しているプロパティー全てで１つの値を形成しているところになります。

そのため、値型に対して定数が書き換えできなくする範囲は**値型のインスタンスが内包しているプロパティー全体**に及びます。つまり、定数で扱う値型のインスタンスは、その内容が書き換わることがないことが保証されます。

内容が書き換わることがないとはいっても、値型で内包しているプロパティーが参照型を扱う場合は、それが保持している指し示し先のインスタンスが書き換えられないだけで、指し示しが内包しているプロパティーを自由に読み書きできる点は、参照型の保護範囲として紹介したとおりです。

ちなみに Swift 言語では**構造体**や**列挙型**、**タプル型**といったほとんどのものが値型に当たります。これらはプログラマーが自由に定義できますけれど、そういった型についても、変数に入れるか定数に入れるかで、内包しているプロパティーの保護が適切に働くような言語設計になっています。

そんな制御を実現するのに不可欠なのが `mutating` キーワードなのですけれど、これについては第 12 章で詳しく見ていくことにします。

配列型も、値型

Swift 言語では、配列型 (`Array` 型) や辞書型 (`Dictionary` 型) のような複数の値を束ねて扱う型も “値型” として扱うようになっています。

こういった高度な型は、データサイズが大きくなりがちで、ほかのプログラミング言語では参照型として提供されることが多い気がします。これらのインスタンスを定数として扱ったときの保護範囲は、内部で扱う値全体にまで及ぶところも Swift 言語の大きな特徴と言えます。

2.4　定数がもたらすメリット

定数を使うことによる利点は、なんといっても **値が書き換わらない** ことに尽きます。シンプルな利点ですけれど、この特徴を Swift 言語自体が活かしてくれるのも手伝って、この恩恵はとても大きく現れてきます。

まず、定数を初期化した後に値が書き換わらない性質がもたらす利点から見ていくと、この利点は特に **値型の値を扱ったときに顕著** に現れてきます。定数で扱う値が “値型” の場合、内包しているプロパティーの内容もすべて保護されます。そのため、ひとたび初期化が完了すれば、以降はその定数に代入されている値が内容も含めて変更されないことが約束されます。

予想しない値の変化を防止

これによって、ソースコードを解読しているときに『その定数が今、どんな値になっているか』を想像しなければいけない負担が想像以上に軽減します。また、不用意に同じ名前の定数を使いまわせないことによって**定数名に、広い意味を持つ名前を安易に付けられなくなる**ため、それによって定数名を工夫したり別の関数として分離するなどの対処が迫られ、結果として**ソースコードの読みやすさが、思うよりずっと向上**します。

それだけ綴ると“ソースコードを書くときの制約が増えて負担を強いられそう”に感じるかもしれないですけれど、慣れてくるとこれが思いのほか気にならないばかりか、むしろ楽しくソースコードが書ける印象です。

定数で参照型の値を扱う場合も、初期化した後は参照先のインスタンスの指し示し先が変わらないことが約束されるので、ソースコードを解読するときに『適切なインスタンスが参照されているのか』を解読する負担が軽減されるのと、値型のときと同様に不用意に同じ名前の定数を作れないことが功を奏して、ソースコードの読みやすさに繋がる利点が得られます。

2.5　定数の恩恵を得るために

こうした定数の恩恵、とりわけ“不変性”の恩恵を得るには、構造体に代表される“値型”を積極的に活用するようにします。独自に型を作る場合も構造体で定義することで、そのインスタンスを定数で扱ったときに、内包するプロパティーの値を Swift 言語によって保護してもらえるようになります。

独自に定義した型の値を適切に保護してもらうための仕組みが構造体に備わっているのですけれど、それについては第 12 章で見ていきます。

イミュータブルクラスという選択肢

不変性の恩恵を得る方法として、Swift 言語の前身とも言える Objective-C 言語で広く利用されていた**イミュータブルクラス**[*8] という手法もあります。

これは、参照型であるクラス型を定義するときに、それが内包するプロパティーの値をイニシャライザー[*9] で確定させる方法です。つまり**内部の値を初期化の段階で決定し、それ以降は書き換わらない設計にすることで、不変性を約束する**ことができます。

イミュータブルクラスの設計例

たとえば次のようなソースコードは、イミュータブルクラスの実装例です。

```
 1.  class Value {
 2.
 3.      private(set) var rawValue: Int
 4.
 5.      init(_ rawValue: Int) {
 6.
 7.          self.rawValue = rawValue
 8.      }
 9.
10.      var description: String {
11.
12.          return "\(rawValue)"
13.      }
14.  }
```

[*8] 英語では“Immutable Class”と表現します。

[*9] 英語では“Initializer”と表現します。型に `init` という名前で定義されている、その型のインスタンスを生成するときの初期化処理を担当する機能です。

このソースコードの要所としては、プロパティーの値をイニシャライザーで決定して、外側からプロパティーの値に代入する手段を与えないこと、イニシャライザー以外では自身のプロパティーを参照するだけにすること、そのようなあたりに注意して型を設計しているところです。

より確かなイミュータブルクラスを作るためにはさらに、内部で扱うプロパティーの型を値型に限定したり、Foundation フレームワークの `NSCopying` プロトコルを活用したりして、別の変数に再代入したときの **それぞれの値の独立性**[*10] も意識して設計していく必要があります。

値型を積極的に活用する

ただ、実際のところ何かをイミュータブルクラスで表現したいときは、そのインスタンスを “値” として扱いたいときのように思います。

そのようなときは、Swift 言語では **構造体などの値型を使って表現する** のが得策です。そうすることで、あとはそのインスタンスを定数で扱うだけで、その値の不変性と独立性を、Swift 言語が自動的に計らってくれます。

こういった Swift 言語の仕組みを活かすためにも、値を表現する型は値型で定義していくことが大切なことになってきます。構造体を使って設計すれば、イミュータブルクラスのときみたいな不変性や独立性を考慮した独自の制御が不要になるばかりか、それを Swift 言語が確実に担ってくれるため、**より精密な不変性と独立性が保証される** ことになります。

[*10] 値の独立性については、第 4 章で紹介していきます。

2.6　変数の使いどころ

定数のメリットを享受するのを基本としたとき、それでは“変数”はどのような場面で使うことになるのでしょう。

そうして考えてみると文字通り、**『値を何回でも代入できる』**という変数の性格を活用できる場面になりそうです。変数は、最終結果を計算するにあたって途中経過を一時的に積み重ねていきたいときや、その値を状況に応じて変化させたいときといった**“記憶域を同じ目的で使いながら値を更新しないといけない変数”**に照準を当てて使うのが良さそうです。

2.6.1　途中経過を積み重ねていきたいとき

ただ、途中経過を積み重ねていくために“変数”を使う機会は、もしかするとそれほど多くはないかもしれません。

たとえば**『16 進数表記の文字列で構成された配列の各要素を数値に変換して合計を計算する』**場面を想定して、様子を眺めていってみましょう。このとき、数値に変換できない文字列は無視するものとします。

```
1.  let strings = ["10", "b", "NaN", "1d"]
```

これをまずは、変数に途中経過を積み重ねていく方法を使って手続き的に計算してみます。手順としては『初期値が 0 の変数を用意して、そこに 16 進数文字列を数値に変換した値を足し合わせる』という処理の流れです。

```
1.  var sum = 0
2.
3.  for string in strings {
4.
5.      if let value = Int(string, radix: 16) {
6.
7.          sum += value
8.      }
9.  }
```

Swift 言語の機能のおかげで読みやすく書けた気がします。これくらいの規模感であればこれで完成としても良さそうですけれど、せっかくなのでもう少しだけ **定数を活用した方法を模索** してみましょう。

手続き的な雰囲気を感じてみる

定数を活用するということは、処理の手続きを書くというより、ルールを列記していく感覚に近いでしょうか。先ほどの例より、もっと **人間寄りの目線で手順を記していく** ことになります。

それに先立って、先ほどに挙げたソースコードの手順を読んでみると、次のような感じになります。処理の手続きに近い感覚が窺えると思います。

① 変数 sum を 0 で初期化する。
② 16 進数文字列の集まり (strings) から順番に文字列を取得する。
③ 取得した文字列から数値への型変換を試みて ･･･
④ 変換できたら、その値を変数 sum の値に加えていく。
⑤ すべての文字列の集まりを数値に変換して加えたら終了。

人間寄りの目線で書き直す

これをもう少し人間目線での手順に置き換えてみます。単純にひとつに決まるとは限らないですけれど、たとえば次のようにも置き換えられます。

① 16 進数文字列の集まりを、それぞれ数値に変換する。
② 変換できた値の合計を計算する。

こうして置き換えた手順は、最初に挙げられた問題文の『16 進数表記の文字列で構成された配列の各要素を数値に変換して合計を計算する』と同じになりました。ひとまずこれに従って、ソースコードを書き換えてみます。

```
// すべての文字列を数値に変換して、変換できたものだけを残す。
let values = strings.compactMap { Int($0, radix: 16) }

// 変換できた値の合計を計算する。
var sum = 0

for value in values {

    sum += value
}
```

慣れるまでは読みにくいかもしれないので、ゆっくり見ていきましょう。

ここの compactMap メソッドは『すべての要素に関数を適用して、新しい要素の集まりを返す』機能です。ここに『Int($0, radix:16)』を指定して、すべての要素を数値に変換した新しい配列を取得しています。そして compactMap メソッドには『結果として nil が得られた場合はそれを無視する』性質もあり、これによって変換できなかったものは取り除かれます。

その後の合計を計算する部分ですけれど、今のところは“変数”をそのまま使っていますし、書き換える前とそんなに印象は変わっていません。それでもここの役割としては『値を全て足し合わせる』程度までスリムになっているので、“ここで何をしているのか”は幾分は把握しやすくなっています。

合計の計算部分を書き直す

この“合計の計算部分”を、もう少し宣言的になるように書き換えてみます。

その上で役に立つのが `reduce` メソッドで、これは『すべての要素を辿りながら、単一の値を計算する』機能を持っています。初期値と関数を渡してあげると、あとは初期値と最初の要素を関数で処理して、得られた結果と次の要素を処理して … というように、最後まで順番に処理してくれます。

これを使えば“初期値 0 に対して要素を順番に足していく”処理が簡単に書けるので、先ほどのソースコードをこれを使って書き直します。

```
1.  let values = strings.compactMap { Int($0, radix: 16) }
2.  let sum = values.reduce(0) { $0 + $1 }
```

これで、すべての 16 進数文字列を数値に変換した合計を計算できました。このソースコードに書かれている手順は、次のように読み解けます。

① 文字列の集まりである `strings` から、
16 進数として数値に変換できたものを `values` とする。
② `values` の数値すべてを足し合わせたものを `sum` とする。

他所の処理が影響する可能性を減らす

こうして書き換えたソースコードは、短くなって把握しやすくなったことも効果として大切なところと思うのですけれど、そのほかにも“別の場所に記載したソースコードが影響する可能性”や“処理の手続きの流れを間違える可能性”を極力排除できているところも大事なポイントです。

定数を使っているため、values の値はいつでも『strings を 16 進数と見立てて変換できたもの』以外の何物にもなりませんし、sum の値はそんな『values の合計』以外の何物にもなりません。それに加えて“すべての値を順次処理する”という手順が reduce という言葉で抽象化されていて、手順の踏み方を間違えてしまう可能性も排除できています。

実現したいことを言葉に落とし込む

このように、手続きをソースコードに落とし込むよりは“実現したいことを言葉に落とし込むのを心がける”ようにして、ソースコードを組み立てていくことが Swift 言語では大事になってくるように感じます。それを心がけると自然と“変数”の出番がなくなって、“定数”の恩恵が得られてきます。

2.6.2　同じ目的で使いながら値を更新したいとき

それでは、どのような場面で“変数”が活きてくるのでしょう。変数ならではの性格が活きる場面としては**プロパティー**が挙げられるように思います。

プロパティーについては <1.6> で紹介しましたけれど、“型に所属する変数および定数”の総称です。そんなプロパティーは、その型から生成されたインスタンスが“今、どのような状態か”を記憶する役割を担っています。

2.7　変数とプロパティー

たとえば、次のような『イベントの現在の来場者数 (activeVisitors) と会場の収容可能人数 (maxCapacity)』を表現する型があったとします。

```
1.  class Event {
2.
3.      var activeVisitors: Int
4.      let maxCapacity: Int
5.  }
```

このとき“現在の来場者数”は時々刻々と変わります。このように**用途は同じまま、その内容が時間とともに変化し得る場合**に“変数”は最適です。

今回の例で難しいのは“最大許容人数”なのですけれど、基本的には会場が決まった時点で確定するので“定数”を使って表現しても良いように思いますけれど、中には状況に応じて増加できる会場もあったり、入場制限で減少する場合もあるかもしれません。そう考えるとこちらも“変数”で定義するのが妥当かもしれないですけれど、それは現実的な可能性を忠実に再現するというよりも、可能性を考慮に入れて『どこまでを許容・想定して型を設計するか』の問題になってくるのでしょう。

いずれにしても、ある対象の現在状態を表現する際に“変数”と“定数”が活用され、特にその**状態が状況によって変化するときに変数は最適**です。

2.8　値と状態

こうしてみたとき、**定数は値を表現**していて、**変数は状態を表現**していると捉えることもできそうです。

定数は、純粋に『それがどんな値であるか』を表現していて、それ以上の表現力はありません。それによって意図したものを普遍的に指し示すことが可能です。この特徴は、安定したプログラムを作る上での強みになります。

そんな**値に、時間が加わると状態**になります。状態はその時々によって刻々と変化していく性質を持つことになりますけれど、そういった性質を表現するときに、変数が持っている性格は最適です。

そしてそんな状態の中から“今の状態”を切り出したとき、それは時間という概念をなくして“値”になります。

値と状態の相互変換

そういうふうにして Swift 言語では、**刻々と変化する状態の中から値を切り出して使う**みたいな作りになっているように感じます。それを支えているのが変数と定数であり、値型の性格、そして代入時の挙動に現れています。

代入時の挙動については第 4 章でも説明しますけれど、値と状態が相互に交換されながら適切にやりとりされる様子を少し紹介しておきます。

たとえば、温度計をプログラムで表現したとします。

温度については、今回は `Celsius` 型として定義することにします。**温度そのものは、具体的な程度を表す値なので“値型”で定義**しました。

```
1.  struct Celsius {
2.
3.      var degree: Double
4.  }
```

温度計を表現する Thermometer 型は、温度を測る機能の集合体なのでクラス型で定義することにします。温度計が示す温度を表現するのに使うプロパティー temprature を定義します。**計測温度は刻々と変化するので、var キーワードを使って“可変性を持つプロパティー”として定義**します。

```
1.  class Thermometer {
2.
3.      var temperature: Celsius {
4.
5.          return …
6.      }
7.  }
```

この温度計を次のように室内に設置して、これを使えば室温を取得できるようにしてあると仮定して、温度を参照したときの様子を眺めてみます。

```
1.  class Room {
2.
3.      let thermometer = Thermometer()
4.  }
```

変化する状態と、スナップショットとしての値

たとえば、ある会議室 (meetingRoom) があったとき、その温度計の温度を参照することで、そのときの室温を取得できます。

```
let templature = meetingRoom.thermometer.temperature
```

こうして温度を取得した後も、室温は刻々と変化していきます。たとえば、温度計を見たときには 30.0℃ だったとしても、その次の瞬間には室温はもう 30.1℃ に変わっているかもしれません。それでも **参照したときの温度が、変化していく室温を追従していかないのは大切** な性質です。

たとえば、少し愚直なソースコードになりますけれど、次のような 1 時間おきに温度を計測するスクリプトを書いたとします。

```
var temperature1 = meetingRoom.thermometer.temperature

sleep(3600)
var temperature2 = meetingRoom.thermometer.temperature

sleep(3600)
var temperature3 = meetingRoom.thermometer.temperature

⋮

sleep(3600)
var temperature9 = meetingRoom.thermometer.temperature
```

このときもし**参照で取得した値がプロパティーの状態変化を追従してしまったとすると**、最初に取得した temperature1 の値が temperature9 で値を取得した頃には変わってしまっています。そうなると、たとえば“1 時間ごとの室温の変化を計測” する目的で記録を取っていたとしたとき、計測結果は『温度変化なし』という**間違った結果**になってしまいます。

ちなみに参照型が主体だった Objective-C 言語では、そういった事態を避けるために**複製を取ることを明示的に指定**したりしていました。

```
1.  KMGCelsius *temperature
2.      = [meetingRoom.thermometer.temperature copy];
```

このように明示的に記載して制御する方法も明瞭で良いのですけれど、うっかり copy メソッドを書き忘れただけで予期しない不具合に悩まされ、しかもその原因箇所を特定するのが難しいという問題がありました。

そんな問題を Swift 言語では、値型と代入時の挙動によって上手に回避してくれています。このようにして、気付かないところで状態と値の振り替えが行われていて、直感的にプログラムを書いていくことができます。

変化を目的としないものは値

ところで先ほどのスクリプトの例では、参照した値を “変数” に代入していました。これ自体に意味はなく、むしろ “定数” に代入するのが適切です。

というのも、その時点で記録した温度は、将来に変更されることはないためです。時間の概念から切り離された値は、変化することはないため、変化しない “値” として取り扱うのが適切です。逆にいうと、いくら変数で扱って

いても、それが**時間の変化と無関係なら“状態”ではなく“値”**と言えます。それが値であるのなら、値を表現するのに最適な“定数”に代入して扱うことが、安定したプログラムを書いていく上で重要になります。

状態を変化させるとき

また、今回の例にはありませんけれど、プロパティーに値を代入することもあります。これを値と状態の観点で見ると、**現在の状態を値で指定した状態に変える**と言えそうです。値がプロパティーに取り込まれるのではなくて、**プロパティーが示している状態が値と同じになる**という感覚なので、その後で状態が変化したとしても、代入元の値が変化することはありません。

このような性質が、値型のインスタンスを別の変数や定数に代入したときに担保される独立性によって実現されています。

第 3 章

変数は初期化して使う

Swift の変数や定数は、初期化してから使います。当たり前なことを言っているような気もしますけれど、思い返してみると、他の言語では初期化しなくても使えたり、初期値が自動で設定されるのが一般的に思えます。

Swift 言語ではほとんどの場面で、変数や定数に値を設定することをプログラマーに強要します。ここでは Swift 言語がどのように初期化を強要するのかと、それによって得られる効果に焦点を絞って眺めていきます。

3.1　初期化せずに使うとコンパイルエラー

Swift 言語の変数や定数は、必ず初期化しなければいけません。たとえば、次のように定数を初期値を指定せずに定義することはできるのですけれど、それを **初期化しないうちに使おうとするとコンパイルエラー** になります。

```
1.  ✅ let value: Int
2.
3.  // Constant 'value' used before being initialized
4.  ❗ print(value)
```

使う前までに初期値を指定しておけば大丈夫ですけれど、いずれにしても使うまでのどこかで初期化をしておかなければいけなくなっています。

初期化を遅らせられる特徴については第 5 章で見ていくとして、ここでは初期化が義務付けられることで得られる利点について眺めていきます。

3.2　初期化が強要されるメリット

変数や定数を初期化しないと使えないのは、Swift 言語が備える大きな取り柄です。しなければならないと聞くと制約が増えて面倒そうに想像してしまいがちですけれど、初期化が義務付けられることによって **間違って初期化されていない変数を使ってしまう危険性** がなくなることは、初期化コードを書かなければいけない負担を遥かに上回る恩恵があります。

さらに Swift 言語は、本書の <5.1> で説明する初期化を遅らせる仕組みを取り入れて、初期化コードの書きやすさを向上させています。

それに普通は初期化しないで使うことはないので、必ずどこかで初期化するならむしろ初期化が義務付けられているのは好都合です。しかも初期化されていないことはコンパイルエラーで知らせてもらえるため、**ソースコードを書いている段階で初期化のし忘れに気付ける** のは嬉しいポイントです。

3.3　初期値は明示的に指定する

プログラミング言語の中には、初期値が指定されなかったときに既定の値で初期化するものもある[*11] のですけれど、Swift 言語では原則として**初期値はプログラマーがソースコード内に明記**する必要があります。

このルールが思う以上にプログラマーの負担を軽減してくれている印象で、なによりも変数や定数がどんな場合にどの値で初期化されるかを覚える必要がなくなりますし、思い込みから間違った初期値を設定してしまう可能性を摘み取ってくれます。初期値を明示しないといけない決まりによって、プログラマーがうっかり初期化をし忘れたみたいな間違いも検出できます。

こういった小さなミスが根深いバグの原因になることはよくあるので、それを発見したり回避したりできるこの仕組みは、プログラマーがコードを書く負担を軽くしますし、プログラムの安定化にも繋がっていきます。

3.4　初期値の明示が不要な場面

ところで Swift 言語には、初期値を明示しなくても大丈夫な場合があります。それは**オプショナル型を使ったとき**です。

```
1.  var value: Int?
```

たとえばこのように変数を定義したとき、このまま初期値を明記することな

[*11] たとえば Objective-C 言語では、メンバー変数に初期値を指定しなくても、それが定義された段階で 0 に相当する値で初期化されています。

く変数 value を参照できます。これは初期化されていないのではなくて、**オプショナル型は初期化しないと nil で初期化される** 特性があるためです。

プログラマーがオプショナル型を使おうとするときは **値がない場合を想定しているはず** なので、初期値を与えなかったときに自動で nil が代入されたとしても、それ自体が想定外の事態を引き起こす可能性は少ないはずです。

3.5　初期化されていない状態を維持したいとき

それでは、初期値が決まらない場合はどうしたら良いのでしょう。いわゆる“未定義な状態”を表現したい場合ですけれど、目的の変数や定数を参照するまでに初期値が決まるなら、初期化を遅らせる方法[*12]で実現できます。

ただ、未定義な状態を作るのは根深いバグに繋がる可能性があるため、**特別な理由がない限りは初期化されていない状況を許さない** のが肝心のように思います。いつ初期化されるかわからない値の動きを把握するのは難しく、問題が発覚するまで間違いに気付けないので、できることなら初期化されるタイミングがはっきり掴めるソースコードを心がけたいところです。

それでも、プログラムとして“未定義”を表現する必要があるかもしれません。そのような場合は次のような方法を使って実現することになります。

オプショナルで表現する方法

たとえば、オプショナルを使って表現する方法が考えられます。この方法が、ほかのプログラミング言語でいう“未定義”に近い表現になりそうです。

[*12] 初期化を遅らせる方法については、第 5 章で紹介します。

実装方法は簡単で、**変数を定義するときに型名の末尾に『!』を添えて**、暗黙的にアンラップされるオプショナル型として宣言します。オプショナル型の変数なので、初期値を指定しなくても使うことが可能になります。

```
1.  var value: Int!
```

こうして定義した変数は、具体的には**IUO 属性付きの** `Optional` **型**として宣言されて、初期値を指定しなかったときには `nil` が代入されます。

この IUO 属性が付いたオプショナル型の特徴は、普通の型と同じように扱えて、値が `nil` な状態を持つことができ、参照時に値が `nil` だと強制終了されるところです。この性質が**未定義の変数にとてもよく似ている**ので、これを使って未定義な状態を擬似的に実現することができます。未定義な状態であるかを確認したいときには、値が `nil` と一致するかで判定できます。

安全性の低下に注意

この方法の懸念点は、いわゆる“未定義な状態”だったときにプログラムの実行が強制終了されるところです。

必ず初期化される見込みがあるなら良いのですけれど、そうでなければプログラムの品質を大きく損なう可能性があるので、この方法を使うかどうかは慎重に検討したいところです。いわゆる“未定義な状態”のときに適切な処理がされるコードを書けば良いのですけれど、普通の型と同じように扱えるため**未定義な状態にあるかを判定するのはきっと疎かになりがち**です。

プログラマーが初期化し忘れたことも検出できなくなるため、問題箇所を特定しにくくなる可能性が高まることは覚悟して使いたいところです。

列挙型で表現する方法

もう少し安全に “未定義な状態” を表現する方法としては、列挙型を使う方法が考えられます。または **プログラムの仕様として純粋に “未定義” という状態を表現したい場合** にはきっとこの方法が最適です。

たとえば、商品を表現する型が、使用目的を表現するプロパティーを持っていて、使用用途が想定されていない場合があるとしたときに、使用目的を次のような感じに列挙型で表現できそうです。

```
enum Purpose {

    case industrial
    case medical
    case beauty

    case undefined
}
```

想定される目的候補を列記して、それと合わせて未定義を示す候補を定義します。今回の例では `undefined` という候補でそれを表します。

このようにすれば、論理的に正確な “未定義” という値を定義できるので、それを考慮したプログラミングが可能になります。列挙型なので `switch` キーワードを使って制御していけば、**未定義を想定したコードが書かれていないときにコンパイラーがそれを見つけてくれる** ところも利点です。

この方法の留意点としては、未定義値を示す選択肢 `undefined` が、そのほかの定義値である候補と同列に存在しているところです。他の候補と同等に

扱う必要があるので、**未定義な状態が特殊な場面を想定しているとき**は、論理的なバランスがおかしくなったりして、条件処理が**不必要に複雑になったりする**かもしれません。

任意の値をとる状況での未定義表現

有効なときの値の範囲が広くて列記しきれない場合は、値付きの[*13]列挙型を使うことを検討すると良いかもしれません。

たとえば『整数の計算結果または未定義値』を定義したい場合には、次のように表現できそうです。

```
1.  enum CalculationResult {
2.
3.      case value(Int)
4.      case undefined
5.  }
```

このようにすることで、整数の全範囲を有効な値として表現できますし、未定義な値は `undefined` で表現できます。この方法では有効だったときの具体的な値と未定義な値とが同列ではなくなるので、未定義な状態が特殊な場面だったときの条件処理の論理的なバランスが崩れにくくなります。

[*13] 英語では“Associated Values”と表現します。

オプショナル型もこの方法

ちなみに Swift 言語が標準で持っている Optional 型も、値が関連付けられた列挙型を使う方法で実現されています。

```
1.  enum Optional<Wrapped> {
2.
3.      case none
4.      case some(Wrapped)
5.  }
```

Optional 型の場合は、いわゆる有効な値を some で、そしていわゆる未定義な値を none で表現しています。列挙子の重みとしては先ほどの例で挙げた CalculationResult 型と同じであることが見てとれます。

ただ、実際に使い比べてみるとわかるのですけれど、それぞれの使い勝手はずいぶん違ってきます。これは Swift 言語が Optional 型言語に対してさまざまな言語支援[*14] が用意されているためです。そのほかにも、いわゆる未定義を示す列挙子が “undefined” なのか “none” なのかみたいに、その名前の付け方によっても変わってきます。

IUO 属性付きのオプショナル型を使わなければ、安全性は値付きの列挙型を自分で定義して使うときと同等までは高められるので、未定義値が “値がない” という意味なら積極的に Optional 型を使っていくのも良さそうです。

[*14] オプショナル型をより簡単に扱えるように、たとえば “?” などの糖衣構文や、値がないことを意味する “nil” キーワード、“if let” などの制御構文が提供されています。

第 4 章

値を代入するときの挙動

Swift 言語の特徴に『変数に代入されている値の種類によって、その値を別の変数に再代入したときの挙動が違ってくる』という性質があります。

ここでは、変数に値を代入するときのインスタンスの作られ方から、それを別の変数に複製するときの挙動について眺めていきます。

4.1　インスタンスを代入するとき

変数や定数に値を代入するときは、基本的には **型のインスタンスを作って代入する** という流れで行います。ほとんどの型にはインスタンスを生成するためのイニシャライザーが用意されている[*15] ので、それを使ってインスタンスを作り、それを値として変数や定数に代入します。

[*15] イニシャライザーは `init` という名前で定義されているのが一般的ですけれど、インスタンスを生成するという観点で見れば『列挙型の列挙子もイニシャライザーの一種』と言えます。イニシャライザーに相当するものを持たないものに `Never` 型があります。

```
1.  let tags = Set(["Swift", "Objective-C", "Java"])
```

ほかにも、関数の中で生成されたインスタンスを戻り値として受け取って、それを値として変数や定数に代入するという流れもあります。

```
1.  let tags = languageSet()
```

こういった感じで値を代入していくことになります。どちらの場合も原則的には**『右辺で生成された値を左辺に複製する』**という流れになります。ちなみに左辺への代入処理が終わると、右辺で生成された値は破棄されます。

4.2　リテラルを代入するとき

ただ、変数や定数に値を代入するとき、次のようなシンプルなソースコードを目にする機会も多いと思います。

```
1.  let revision = 9
```

複雑さを感じないソースコードですけれど、この場合も、右辺でイニシャライザーを使ってインスタンスが生成されて、左辺に複製されます。そんな動作を詳しく知るなら、リテラルの初期化の仕組みを知る必要があります。

リテラルは、なんらかの型のインスタンスに暗黙的に変換されます。どんな型に変換されるかは**周囲で指定されている型や、あらかじめ決められた既定の型情報**によって変わってきます。Swift 言語には 7 種類のリテラル[*16]がありますけれど、今回は“整数リテラル”に注目してみましょう。

4.2.1 整数リテラルから変換可能な型

整数リテラルを代入できる型は、**整数リテラルで表現できる型**に限られます。それは `ExpressibleByIntegerLiteral` プロトコルに準拠した型で、たとえば `Int` 型や `Double` 型など[*17]があります。

`ExpressibleByIntegerLiteral` プロトコルでは次のようなイニシャライザーの実装が要求されていて、これを使ってインスタンスを生成します。

```
1.  protocol ExpressibleByIntegerLiteral {
2.
3.      init(integerLiteral value: IntegerLiteralType)
4.  }
```

4.2.2 整数リテラルの変換のされ方

整数リテラルからインスタンスが生成される流れは、まずは整数リテラルが書かれているところの前後関係から <1.3> で紹介した型推論の仕組みを使って、**整数リテラルをどの型として扱うのが適切か**が判断されます。

[*16] Swift 言語には、整数リテラル、浮動小数点数リテラル、文字列リテラル、配列リテラル、辞書リテラル、`nil` リテラルが存在します。

[*17] ほかにも `Numeric` プロトコルに準拠した型など、いくつも定義されていて、`ExpressibleByIntegerLiteral` プロトコルを使って自作もできます。

そして、その型が `ExpressibleByIntegerLiteral` プロトコルに準拠していることを確認して、実装されている `init(integerLiteral:)` が暗黙的に実行されて、インスタンスが生成されます。

前後関係から型が判断できない場合 は、そのときに大域スコープで有効になっている型エイリアス `IntegerLiteralType` で指定されている型に変換されます。この型エイリアスが既定では `Int` 型になっているため、**整数リテラルは `Int` 型として解釈** されます。

4.3　別の変数に代入するとき

Swift 言語の代入文の動きを理解する上で大切なことになるのですけれど、インスタンスの種類が **値型か参照型かで代入文の挙動が変化** します。

代入文というのは、いわば『ある変数や定数の値を、別の変数や定数に複製する行動』とも捉えられますけれど、このときのインスタンスの種類で挙動が変わる性質は、値の代入元や代入先の記憶域の種類が“変数”か“定数”かは関係なくて、扱われるインスタンスの種類に影響されます。

この動作の違いを意識できるようになると、値を受け渡した前後でどう変わったかをイメージできるようになりますし、型を定義するときも『構造体型とクラス型のどちらで定義するのが適切か』を判断する材料になったりするので、Swift 言語に慣れたら次に意識を向けておきたいポイントです。

4.3.1　代入文の基本動作

まずは **代入文の挙動** を見ておきましょう。たとえば『変数または定数 `a` に代入されている値を、定数 `b` に再代入する』コードは次のとおりです。

```
1.  let b = a
```

代入文の動作は a や b が“変数”であっても“定数”であっても同じで、右辺にある a に代入されているインスタンスが**複製されて b に代入**されます。

ところでこのとき、代入処理が終わった後には**代入元の a にも値が残されて**います。つまり a と b とが同じ値になるのですけれど、**これらの間の関連性**はどうなっているのでしょう。そんな 2 つの値の関係性は、取り扱われるインスタンスの種類によって変わってきます。

4.3.2　値型を複製するとき

それでは、まずは a に代入されている値が**“値型”**だったときの複製のされ方から見ていきます。このときの a に代入されている値は“値型のインスタンスそのもの”で、これを b に代入すると b には**インスタンスが内包しているプロパティーの内容がすべて複製**されます。

そうなると a と b には『値としてはまったく同じだけれど、独立した別々のインスタンス』が代入されている状態になります。

内容が同じなので同等のものとして扱えますけど、それぞれは独立しているため、**どちらかの内容が書き換えられても、もう一方には影響しない**というところが、値型のインスタンスを再代入したときの大きな特長です。

そんなインスタンスの独立性のおかげで、定数が備える不変性[*18]という性

[*18] 定数の不変性については <2.4> で紹介しています。

格を **より確かなものに** しています。

4.3.3　参照型を複製するとき

次に、値が **“参照型”** だったときの複製のされ方を見てみましょう。こちらの場合は、変数または定数の中に代入されている値は、扱う対象のインスタンスそのものではなくて **インスタンスが置かれている場所**[*19] になります。

これを b に代入したとき、代入先には a に代入されていた **対象のインスタンスが置かれている場所の情報が複製** されます。つまりどちらも同じインスタンスを指している状態になります。

値型を再代入したときとは違って、まったく同じものを共有することになるため、どちらかでインスタンスが持っているプロパティーの内容を変更すると、それを共有しているすべての箇所に影響します。

このように、インスタンスの独立性は持たない代わりに **1 つのインスタンスが持つ状態を、複数箇所で共有できる** という特徴があります。

4.4　独立性がもたらすメリット

代入文で扱うインスタンスが **値型だったときに得られる独立性** は、**スレッドを跨いだプログラムを書くときに大きな恩恵** をもたらします。

スレッドを跨いだプログラムを書いていると、あるスレッドで使用中の値が別のスレッドで書き換えられることで予期しない値で計算したり、値の参照中に別のスレッドが書き換えることで有り得ない値が得られたりします。

[*19] 一般的には “参照先” や “アドレス” などと言われます。

スレッドセーフ

そんな複数のスレッドでの**同時アクセスに起因した問題は、値の独立性が約束されることで防げる**ようになります。値が独立していることで、そもそもどこでどんな値が書き換えられても、その影響が及ぶことはありません。

このような問題が起こると、どこで問題が起こったかを突き止めるのが極めて困難になる場合もあるので、値型の独立性を活用していくことは、プログラムの安定化という観点でとても大きな効果が期待できます。

4.5　複製されるタイミングに注意

ただし、容量がかさみがちな `Data` 型や、利用頻度が高くて高速に処理できることを期待される `Array` 型のような型では、必要になるまで内容の準備を遅らせる“コピーオンライト”の機能によって**“表向きには不変を装い、内部的には初期化後に内容の書き換えが行われる”**ものもあったりします。

このような値を複数のスレッドで扱ったときに、定数の参照と内部的な内容の書き換えが同時に行われて、あり得ない値が得られてしまうことがあります。それを防ぐには**あらかじめ確実に内容の書き換えを完了**[*20]させたり、**排他ロック**[*21]の仕組みを使って、同時アクセスによる値の書き換えが起こらないように注意する必要があります。

[*20] たとえば `Array` 型なら、別のスレッドに渡す値は、渡す前に確実に、内容の複製を終わらせておく方法が考えられます。

[*21] 排他ロックの仕組みには、セマフォ (Semaphore) やミューテックス (Mutex)、`NSLock` 型を使った方法などがあります。

To Be Continued . . .

お試し版をご覧になってくださって、ありがとうございました。

本書は『技術書典 7』での頒布に間に合わせるべく制作していたのですけれど、**原稿が間に合う見込みがなかったため**、仕上がっている第 4 章までをお試し版として頒布することにしました。引き続き『技術書同人誌博覧会 2』までの完成を目指して制作していきますので、それまでの間はこのお試し版をお手元に置き、お楽しみにしていて頂けたら幸いです。

完成まで今しばらくお待ちください

完成した際には、技術書同人誌博覧会 2 での製本版はもちろん、BOOTH 等で販売開始する予定ですので、どうぞよろしくお願いいたします。詳細は次の URL でお知らせしていきますので、どうぞこちらもご覧ください。

頒布書籍の情報はこちらに掲載しています。
https://ez-net.jp/book/

あらためまして、ここまで読んでくださりありがとうございました。
快適な変数の旅をお楽しみください。

著者　熊谷 友宏

幼き頃に MSX-BASIC に巡り会いプログラミング人生が幕明けました。
最近は Swift 言語と勉強会での発表が特にお気に入りです。

プログラミングそのもので遊ぶ楽しさを広く伝えていきたい、
そんな想いの募るこの頃です。

勉強会

カジュアル Swift 勉強会 #cswift　@ 横浜・青葉台
もくもく執筆会☆出張版 #techbook_meetup　@ 首都圏
みんなで Swift 復習会 GO! #minna_de_swift　@ 日本各地

Podcast

熊谷と縉面がプログラミングコードの内から聴こえてくる声に
耳を傾けて楽しむラジオ #mook05

https://kepc.mookmookradio.com

Swift 変数入門

2019 年 09 月 22 日 初版第 1 刷 発行

著　者：熊谷 友宏
編　者：熊谷 牧子

発　行：EZ-NET
連絡先：@es_kumagai

BUILD：#05649