Swift 3 の隅っこをつつく

京都産業大学 荻原剛志

2016.09.14

Swift 3

- オープンソースになって、開発者からの意見を取り入れて言語仕様を改善中。
- かなり変更されてきたが、まだ途中の部分や見直されていない部分がある。
- 例えばこんなことができてしまう部分も。

```
b = a
if a == b {
    print("ここが実行されない")
}
```

関数の引数ラベル

- 関数、イニシャライザの引数ラベルのつけ方が統一された
  - デフォルトで仮引数名がラベルになる
  - 明示的に指定もできる。不要なら "_" を指定

```
init(contentsOfFile: String, encoding: String.Encoding)
    → init(contentsOfFile:encoding:)

func replacingCharacters(in: Range<Index>, with: String)
    → replacingCharacters(in:with:)
```

第１引数のラベルが使われるようになった

Swift発表時から変遷を重ね、結局一番簡単な形に…

関数の引数に var が使えなくなった

- 関数内部で値を変更可能な仮引数を表すために var を使うことができたが、Swift 3 で廃止。

```
func nazarick(var key: String, var value: Int)
```

- inout付き仮引数以外は、定数(let)と同じ扱い
- 例えば次のように記述できるが。。。

```
func nazarick(key: String, value: Int) {
    var key = key, value = value
    // これは極端な例。
}
```

inoutの置き場所が変更に

- 仮引数に付ける inout キーワードが型の一部という認識に。

Swift 2

```
func spell(inout power: Int)
```

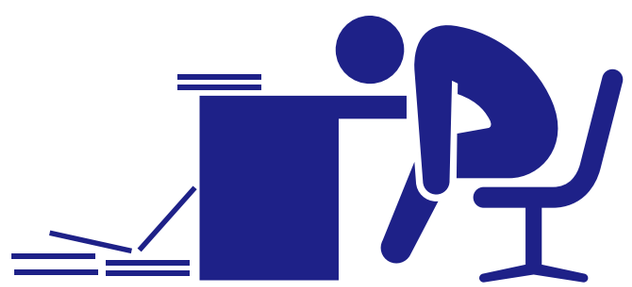
Swift 3

```
func spell(power: inout Int)
```

- ちなみに、現在の実装では：
  - Intなど：関数内で変更されると、呼び出し元の値もその都度変更される（多分、ポインタ渡しで実装）
  - 計算型プロパティなど：呼び出し元の値が変更されるのは、関数の実行が終わった時（書き戻ししている）

関数名が変わった

- Objective-C風の命名法をやめ、オーバーロードの利点を活用した方式に
  - 関数名は簡潔に
  - 各引数のラベルで機能の違いを説明
  - 英語の読解力(!?)が必要に



★ インスタンス自体の値が変更されるかどうかを「-ed」や「-ing」で表す

★ 引数ラベルに「by」、「with」、「to」など前置詞が山盛り

```
mutating func insert(_ : Character, atIndex: Index)
mutating func insertContentsOf<C>(_ : C, at: Index)
```

```
mutating func insert(_ : Character, at: Index)
mutating func insert<C>(contentsOf: C, at: Index)
```

タプル ～比較可能に

- タプルは相互に大小比較できなかった(Swift 2)。
- ジェネリックなグローバル関数が用意され、要素数最大6まで比較可能になった。

```
public func ==<A: Equatable, B: Equatable>
                    (lhs: (A, B), rhs: (A, B)) -> Bool
public func <<A: Comparable, B: Comparable>
                    (lhs: (A, B), rhs: (A, B)) -> Bool
```

!=, > なども同様。要素数6まで

```
(1, "AINZ") == (11/10, "Ainz".uppercased())
(0, 1, "Ooal", 999) > (0, 1, "Gown", 0)
```

大小比較は辞書式順序(第1要素を比較、第2要素を比較、…)

タプル ～比較可能だけど

- タプル自体は相互に大小比較できる型ではない。
- キーワード付きタプルの場合、キーワードは無視される

⭕ `(name:"上田", age:18) == (city:"上田", route:18)`

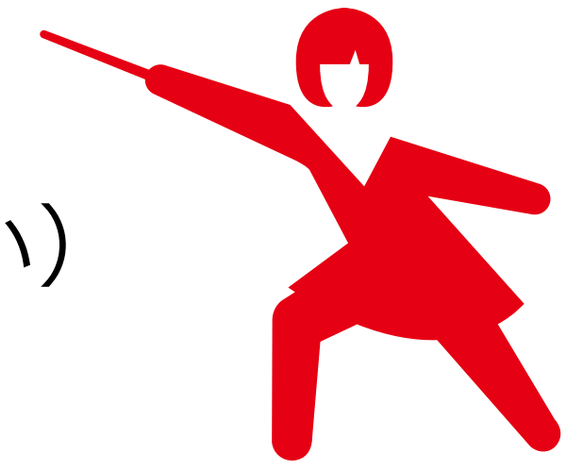
- タプルを含むタプルは比較できない。

❌ `(1, ("Albedo", 1)) == (1, ("Rubedo", 0))`

- タプルを含む配列は、比較関数を与えればソートできる

```
let h = [("ふくからに", 1), ("わすれじの", 2), ("あまつかせ", 3)]
let r = h.sorted(by: <)
// r = [("あまつかせ", 3), ("ふくからに", 1), ("わすれじの", 2)]
```

タプル ～代入とか



- これまでにもあった機能（おさらい）
- 対応位置への代入

```
let (a, b) = (0, "Aiwass")  // a=0, b="Aiwass"
```

- 対応するキーワードへの代入

```
let (x: ax, y: ay) = (x: 0.1, y: 8.0)
let (x: ax, y: ay) = (y: 8.0, x: 0.1)
    // ax=0.1, ay=8.0（順序よりもキーワードを優先）
let (w, h) = (x: 0.1, y: 8.0)  // w=0.1, h=8.0
let (x: cx, y: cy) = (99, 0)   // cx=99, cy=0
```

キーワードがない場合は順番に従う。
キーワードが異なるタプル同士では代入できない。

タプル ～やっちまったな

◆ 次の例はどうか？

```
var a = (price:180, no:32)
var b = (no:0, price:10)
b = a
if a == b {
    print("実行されない")
}else {
    print("こっちが表示さ
}
```

b=(no:32, price:180)

(180,32) == (32, 180)

実は、変数a, b がきっちり型宣言されていれば実行前にチェックできる。
宣言のないタプルのチェックは少々危うい。

似たような話

SE-0111 で指摘された Swift2の動作

```
func attack(atX: Int, y: Int) -> Bool { … }

func meetsBatting(ofHits: Int, forRuns: Int) -> Bool { … }

var predicate : (ofHits: Int, forRuns: Int) -> Bool = meetsBatting
predicate = attack

let r = predicate(ofHits: 1, forRuns: 2) // attack が呼ばれる
```

```
var predicate : (Int, Int) -> Bool = meetsBatting
```

Swift 3 では引数ラベルは型の一部ではない

演算子の優先順位 ～Swift 2

◆ Swift 2 では優先順位が数字で決まっていた

演算子	結合	優先
p++ *p*-- *a*! *a*? *a*.	→	
++*p* --*p* +*a* -*a* ~*a* !*a*	←	
<< >>	×	160
* / % & &*	→	150
+ - | ^ &+ &-	→	140
..< ...	×	135
is as as? as!	×	132
??	←	131
< <= > >= ~= == != === !==	×	130
&&	→	120
||	→	110
con ? *ex*1 : *ex*2	←	100
= *= /= %= += -= <<= >>= &= ^= |=	←	90

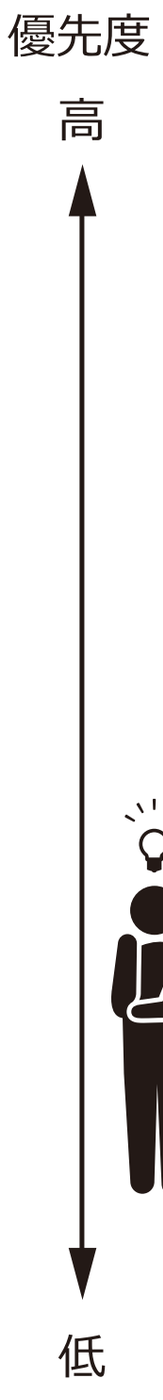
(※)

(※)「!」「?」「.」は演算子ではない。

演算子の優先順位 ～Swift 3

優先順位は**優先度グループ**間の**半順序関係**で決まる

演算子	優先度グループ	結合
`<<` `>>`	BitwiseShiftPrecedence	×
`*` `/` `%` `&` `&*`	MultiplicationPrecedence	→
`+` `-` `|` `^` `&+` `&-`	AdditionPrecedence	→
`..<` `...`	RangeFormationPrecedence	×
`is` `as` `as?` `as!`	CastingPrecedence	×
`??`	NilCoalescingPrecedence	←
`<` `<=` `>` `>=` `~=` `==` `!=` `===` `!==`	ComparisonPrecedence	×
`&&`	LogicalConjunctionPrecedence	→
`||`	LogicalDisjunctionPrecedence	→
con `?` *ex*1 `:` *ex*2	TernaryPrecedence	←
`=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`	AssignmentPrecedence	←



優先度グループの定義

定義する優先度グループの名称

```
precedencegroup AdditionPrecedence {
  associativity: left
  higherThan: RangeFormationPrecedence
}
```

結合方向（右、左、なし）

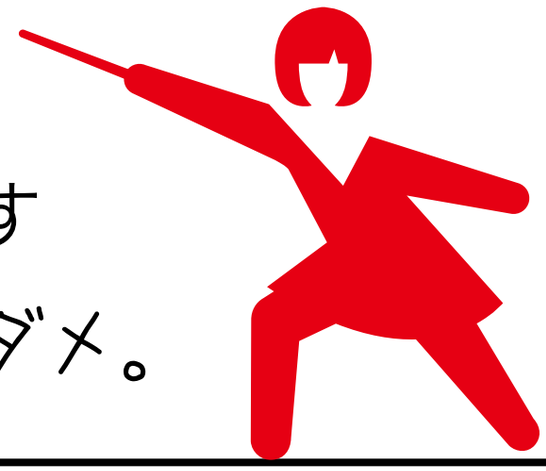
比較対象となる優先度グループの名前

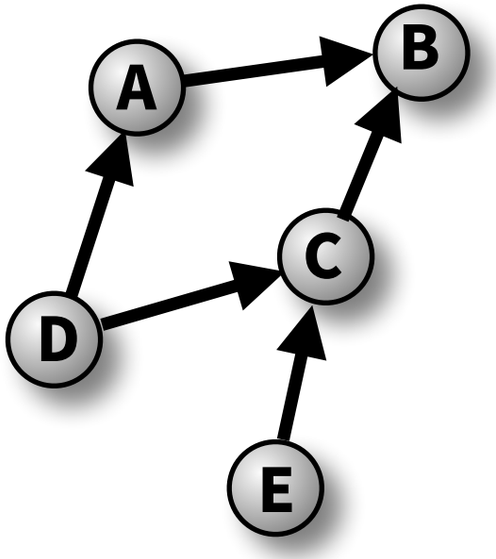
`lowerThan`
というキーワードも使える

- 全順序ではないので、優先度グループ間で順位が定まらない場合もある
- 関係で閉路ができないようにコンパイラがチェック

参考：半順序関係とは

- 数学的な厳密な定義はあるが、簡単に言うと：
  - 要素の間に順序（例えば大小）を定めることができる
  - A≤B かつ B≤C なら A≤C である
    - 関係がループしていてはいけない
  - 要素間で、順序が定まっていないことを許す
- 情報数学でよく出てくるから覚えてないとダメ。





A ≤ B とする。

D ≤ B、E ≤ B だが、

AとC、AとE、DとEには

関係が定義されていない

閉路ができてはダメ