

Swift Programming

Lecture 3

Expression and Assignment
Type Conversion

Tatsuo Minohara



式に何が書けるか

- 式

- ➡ 定数

- ➡ 変数名

- ➡ 式 + 式

- ➡ 式 - 式

- ➡ 式 * 式 ← 乗算

- ➡ 式 / 式 ← 除算

- ➡ 式 % 式 ← 剰余

- ➡ (式)

式の構文

- $4\ 5\ *\ (3\ 4\ +\ 2\ 3)\ /\ (y - 5)$
- 式 $\ast$ (式 $+$ 式) $\ /\$ (式-式)
- 式 $\ast$ (式) $\ /\$ (式)
- 式 $\ast$ 式 $\ /\$ 式
- 式 $\ /\$ 式
- 式
- $\times\ 45x + 65y$



式と評価

- 評価 (Evaluation) とは
 - 単一の値になるまで計算すること
- 式の書式に合っているか
 - 書式に合っていないと文法エラー
- 式の評価の優先順位
 - 演算子には、結合のための優先順位がある
 - 単項の $\pm$ は一番優先される
 - 乗除算の演算子 ($*$ / $\%$)の方が優先される
 - 加減算の演算子 ($+$ -)が優先度低い
 - $()$ で囲むと優先度を高くする



結合性

- 同じ優先順位の演算子は、左から評価されていく（加減乗除などの場合）
 - 左結合性（Left associative）と呼ぶ

$$56 * 34 / 28 * 60 / 30 \% 89$$

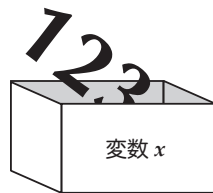
$$\rightarrow ((((56 * 34) / 28) * 60) / 30) \% 89$$

$$83 + 45 - 23 + 38$$

$$\rightarrow ((83 + 45) - 23) + 38$$

変数

- 値を一時的に入れておく箱と思えば良い。



- 参照する前に、値を入れておかなければならない。
 - ➡入っていないと、コンパイル時か実行時にエラーが出る。



変数の名前と代入

- 変数の名前

- 変数の名前に使えるのは、半角の英数字、およびアンダーバー（`_`）
- 名前の先頭の文字は英字でなければならない
- 変数名は、半角の小文字の英字をお願いします
例：

`x y z など`



変数の宣言

- 型指定付きの変数宣言

- 書式： **var** 変数名: 型名

- 例：

- var** x : Int

- var** s : String

- 宣言だけしたときの、変数の初期値としては、以下が設定される

- Int → 0

- Double → 0

- Bool → **false**

- String → **nil**



初期値代入と代入

- 型指定なしの初期値代入

- 初期値代入を伴うときは、型指定がなくても構わない
- **var** 変数名 = 式

例： **var** *x* = 10

- 型指定ありの初期値代入

- **var** 変数名: 型名 = 式

例： **var** *x* : UInt64 = 10

- 代入

- 一度初期値代入、あるいは宣言された変数は、後から代入ができる（ただし、型が異なる値は代入できない）
- 変数名 = 式

例： *x* = 34



変数の参照

- 参照

- ➔ 変数が保持する値に置き換えられる。
- ➔ 変数が保持するオブジェクトが参照される。

- 自己参照代入

- ➔ 元の値を利用して、新しい値が代入される。

$$x = x + 1$$

$$x = -x$$

$$x = x - 20$$

代入演算子としての=

- 右辺と左辺は同一ではない
- 等しいという意味ではなく、右辺を左辺にAssignするもの。

左辺 = 右辺

＊この意味は、「左辺の変数 ← 右辺の評価値」

- なお、左辺の値が評価値として残る（右結合性）Swiftでは書けない！

$$x = y = z = 0 \rightarrow x = (y = (z = 0))$$



定数の初期値代入

- 定数は、初期値代入だけができる、以降代入することはできない
- 定数の初期値代入は、`let`文を使う
 - `let` 変数名 = 式
 - 例： `let value = 123`
- 定数の初期値代入の際も型指定を行なうことができる
 - `let` 変数名 : 型名 = 式
 - 例： `let num : Double = 0`



タプルの代入

- =の両辺にタプルを記述すると一括代入が行なわれる

- 書式：(変数, 変数, ...) = (式, 式, ...)

- 例： `var x, y: Int`

`(x, y) = ( 100, 200 )`

- 変数が持つ値の交換も行なうことができる

- 例： `( x, y ) = ( y, x )`

- 左辺の変数のところに、_を記述すると捨てられる

- 例： `( x, _ ) = ( 100, 200 )` // 200の値は捨てられる



整数演算

- 整数除算は、小数点以下が切り捨てられる
 - $5/2 \rightarrow 2$
 - $1/8 \rightarrow 0$ 分母の方が大きいと0になる
- どこに整数除算があるか重要
 - $5/2 * 2 \rightarrow 4$
 - $5 * 2/2 \rightarrow 5$
- 剰余算は、余りを計算する（整数のみ）
 - $365 \% 20 \rightarrow 5$
 - $x \% n \rightarrow 0 \sim n-1$ の数しか出てこない

計算結果が0になるときは、割り切れるということ



整数剰余・整数除算

- $x / n * n$

- x と等しいか、 x を超えない最大の数で、 n で割り切れる数が求まる
- 例： $10 / 3 * 3 \rightarrow 9$

- n / m

- $n < m$ の場合は、0になる
- 例： $3 / 4 \rightarrow 0$

- $n \% m$

- $n < m$ の場合は、 n になる
- 例： $4 \% 7 \rightarrow 4$



基数と整数剰余・整数除算

- 各桁に分解できる
 - $3456 \% 10 = 6$
 - $3456 / 10 \% 10 = 5$
 - $3456 / 10 / 10 \% 10 = 4$
 - $3456 / 10 / 10 / 10 \% 10 = 3$
- n進数に関しても同様のことができる
 - $234 \% 7 \Rightarrow 3$
 - $234 / 7 \% 7 \Rightarrow 5$
 - $234 / 7 / 7 \% 7 \Rightarrow 4$

整数除算の計算方法

- $x \% n \rightarrow x - (x / n * n)$
- $x / n \rightarrow (x - x \% n) / n$
- $x / n * n \neq x$ のときがある
 - $9 / 3 * 3 \rightarrow 9$
 - $10 / 3 * 3 \rightarrow 9$
 - $11 / 3 * 3 \rightarrow 9$
 - $12 / 3 * 3 \rightarrow 12$
 - $13 / 3 * 3 \rightarrow 12$
 - $14 / 3 * 3 \rightarrow 12$
 - $15 / 3 * 3 \rightarrow 15$

剰余算の計算

- 負の数の剰余について、Swiftの計算方法は、以下の通り
 - x と y を共に正の数であるとする
 - $x \% -y \rightarrow (x \% y)$, $-x \% y \rightarrow -(x \% y)$, $-x \% -y \rightarrow -(x \% y)$ となる
 - 例：
 $30 \% -8 \rightarrow 6$, $-30 \% 8 \rightarrow -6$, $-30 \% -8 \rightarrow -6$
- 実数の剰余を求めたい場合は、`%`演算子は使えず、`truncatingRemainder(dividingBy: )`を使う
 - 例：`3.0.truncatingRemainder(dividingBy: 0.8 )`



Swiftのインタプリタ

- インタプリタ (n>が出ている) の画面では、実行履歴を使って表示することができる
 - control + p : 1つ前の履歴を出す
 - control + n : 履歴の1つ後に進む (Swiftは使えない)
- インタプリタのその他の編集キー
 - 左右の矢印キー: 挿入カーソルを移動
 - control + a : 挿入カーソルを先頭に移動
 - control + e : 挿入カーソルを最後に移動
 - control + k : 挿入カーソルから後を削除
 - control + d : 挿入カーソルの直前の1文字を削除 (Backspace/Deleteと同じ)



文字列の演算

- 文字列同士の加算
 - ➔ 足し合わされた別の文字列が生成される
 - ➔ "2345" + "6789" → "23456789"
 - ➔ "千里の道も" + "一歩から" → "千里の道も一歩から"
- Stringコンストラクタで繰り返し指定できる
 - String(repeating: 文字列, count: 繰り返し回数)
 - 例 : String(repeating: "ABC", count: 5)
 - ➔ "ABCABCABCABCABC"

文字列への変数の値への埋め込み

- 変数名を指定して、その値を文字列に埋め込むことができる

書式: " \ (変数名) " "\ (式) "

例:

```
var value = 123
```

```
print( "value is \ (value)" )    → value is 123
```

- 数値などから文字列への変換へは、標準的なフォーマットで行なわれる
- 細かいフォーマットしたい場合は、以下のスライドにあるように
String(format: フォーマット, 変数)



プログラム上に現れる英字

- `a` ... 変数名
- `'a'` ... `a`一文字から成る文字列 Swift 4.0ぐらいまで
- `"a"` ... `a`一文字から成る文字列
- `"\(a)"` ... 変数`a`の内容が埋め込まれた文字列
- `0xa` ... 16進数の`a` (=10)



数の文字列への変換

- String()...コンストラクタを使う
 - String(整数) ... 10進数の文字列に変換される
 - String(整数, radix: n) ... n進数の文字列に変換される
 - String(実数) ... 文字列に変換される



文字列のフォーマット演算

- C/C++言語のprintf関数との互換性を考えて、Swiftには文字列のフォーマットを持ったコンストラクタ（後述）がある
- Foundationライブラリにあるので、**import Foundation**が必要
 - `String( format: "フォーマット文字列", 引数, ... )`
 - `String( format: "フォーマット文字列", arguments: [ 引数, ... ] )`
- 詳しくは、以下を参照
 - Apple Developer Cocoa String format Specifier



フォーマット文字列一覧

- %d...10進数として表示する
- %o...8進数として表示する
- %x...16進数として表示する
- %X...16進数として表示する (A-Fを大文字で)
- %f...実数として表示する
- %e...指数形式の実数として表示する
- %c... 1 文字として表示する
- %C...1文字として表示する Unicode (UTF-16) 対応
- %s...文字列として表示する
- %S...文字列として表示する Unicode (UTF-16) 対応
- %@...NSString対応の文字列を表示する

桁指定を行なうフォーマット文字列

- フォーマット文字列には、次のような桁指定が使える
 - %d %nd %0nd 整数用 nは数字
 - %x %nx 整数用16進数 nは数字
 - %f %n.mf 実数用 n,mは数字
 - %e %n.me 実数用、指数表記
 - %s %n.ms 文字列用 n,mは数字
 - -をつける 左寄せになる
 - +をつける 符号がつく（数のみ）



フォーマット例

- `%6d...`6文字分は最低限確保される。足りなければ、左側に空白が詰められる
- `%06d...`6文字分は最低限確保される。足りなければ、左側に0が詰められる
- `%+d...`必ず符号が含まれる（符号で上記の指定の1文字分は消費される）
- `%-8s...`8文字分は確保される。結果は左揃えになる（通常は右揃え）
- `%#o...`かならず0で始まる8進数として表示する
- `%#x...`かならず0xで始まる16進数として表示する
- `%10.3f...`全体で10桁、小数部は四捨五入されて3桁になる
- `%10.3@...`表示部分は10文字分確保されるが、そのうち実際に文字が表示されるのは3文字分だけ



Cのprintfと同じ文字列%sを使うには、

- クロージャの形にする（省略形が使えるので、withCStringのあとの()は、省略できる）
- 文字列変数.withCString () {
 String(format: "%s", \$0)
}
- ただし、日本語が表示されない



文字端末（ターミナル）への表示

- `print()`関数を使う
 - `print( 式 )`
 - `print( 式, 式, ... )` // 各式の間は空白 1 文字で区切る
- 改行や区切り文字の指定が出来る
 - `print( 式 )` // 表示したあと改行
 - `print( 式, terminator: "" )` // 改行せず
 - `print( 式, 式, ..., separator: ":" )` // 区切り文字を変える
 - 改行用の特殊記号として`\n`が使える



文字端末（ターミナル）からの入力

- `readLine()`関数を使う
 - 文字列変数 = `readLine( )`
 - 文字列変数 = `readLine( strippingNewLine: false )`
 - 標準値は、**true**
- オプショナル型（後述）なので、入力値を扱うためにラップを外す必要がある
 - 例： `var line: String? = readLine( )`
 - `print( line! )` // オプショナル型を外す



readLine()による文字列入力

- 終了のためには、文字列入力後、returnを押す
または、いきなりControl+Dを押す（入力なし）
- インタープリタから直接readLine()を呼び出すと、終了するのが
難しい（ReturnやControl+Dで終了してくれないため）
 - インタープリタで、readLine()して終了できなくなったら、
Control+Zで前面のジョブをサスペンド（一時停止）させる
 - シェル上で、kill -9 %1として強制終了させる
 - jobsで、終了したかどうか確認ができる



文字列から数値への変換

- 整数（オプション型）
 - `Int( 文字列 )` ... 10進数の文字列として解釈
 - `Int( 文字列, radix: 基数 )` ... n進数の文字列として解釈
- 実数（オプション型）
 - `Float( 文字列 )` ... 単精度実数の文字列として解釈
 - `Double( 文字列 )` ... 倍精度実数の文字列として解釈



オプション型

- nil値（値を取らなくても良い型）を含めて認めるときに、指定する型
- 変数にサランラップ（風呂敷）が掛かった状態になっている
 - 書式：型名?
Optional<型名>
 - 例： `var s : String?`
`var nn: Optional<Int>`
- 実際に値を取り出すときは、サランラップ（風呂敷）を取り外す必要がある
 - 書式：変数名!
 - 例： `s!`

オプション型への代入

- 代入は、その変数のまま行える。また、オプション型の変数については、`nil`を代入することが可能である

▶ 例: `var n: Int?`

```
n = 567    // 具体的な値を代入
```

```
n = nil    // 値を保持していないことを示すnil値を代入
```

- 参照の際は、ラップを取り外す演算子`!`を付ける必要がある
- ただし、`nil`値を保持していた場合は、実行時エラーになる

▶ 例: `var n: Int? = 567`

```
print(n)    // Optional( 567 )が表示される  print(n as Any)が望ましい
```

```
print(n!)    // 567が表示される
```

```
n = nil      // オプション型には、nilを代入できる
```

```
print(n)     // nilが表示される
```

```
print(n!)    // 実行時エラーになる
```



オプション型が返される場合

- データの型変換ができないとnilが返される場合がある
 - ▶ 例：`Int( "HELLO" )` // nilが返される
`Double( "" )` // nilが返される
- ターミナルからの入力で何も入力されない場合、nilが返される
 - ▶ 例：`readLine()`でいきなりControl+Dを押す // nilが返される



オプション型と??演算子

- オプション型の変数に値が代入されていない場合は、??演算子の後に記された値（デフォルト値）が代わりに用いられる。
- 書式：
 - オプション型の変数 ?? デフォルト値の式
- 使用例：

```
let nickName: String? = nil
```

```
let fullName: String = "John Appleseed"
```

```
let informalGreeting = "Hi \(nickName ?? fullName)"
```



論理型への型変換

- 文字列型の値から論理値へ
 - `Bool( 文字列 )` → オプショナル型になる
 - `Bool( "true" ), Bool( "false" )` → `Bool?: true, Bool?: false` この2つの文字列のみ
 - `Bool( "aaa" )` → `nil`値になる（それ以外の文字列すべて）
- 数値型の値から論理値へ
 - **import** Foundationが必要
 - `Bool( truncating: 数値 )`
 - `Bool( truncating: 0.0 ), Bool( truncating: 0 )` → `false`になる
 - `Bool( truncating: それ以外の値 )` → `true`になる



暗黙の型の変換

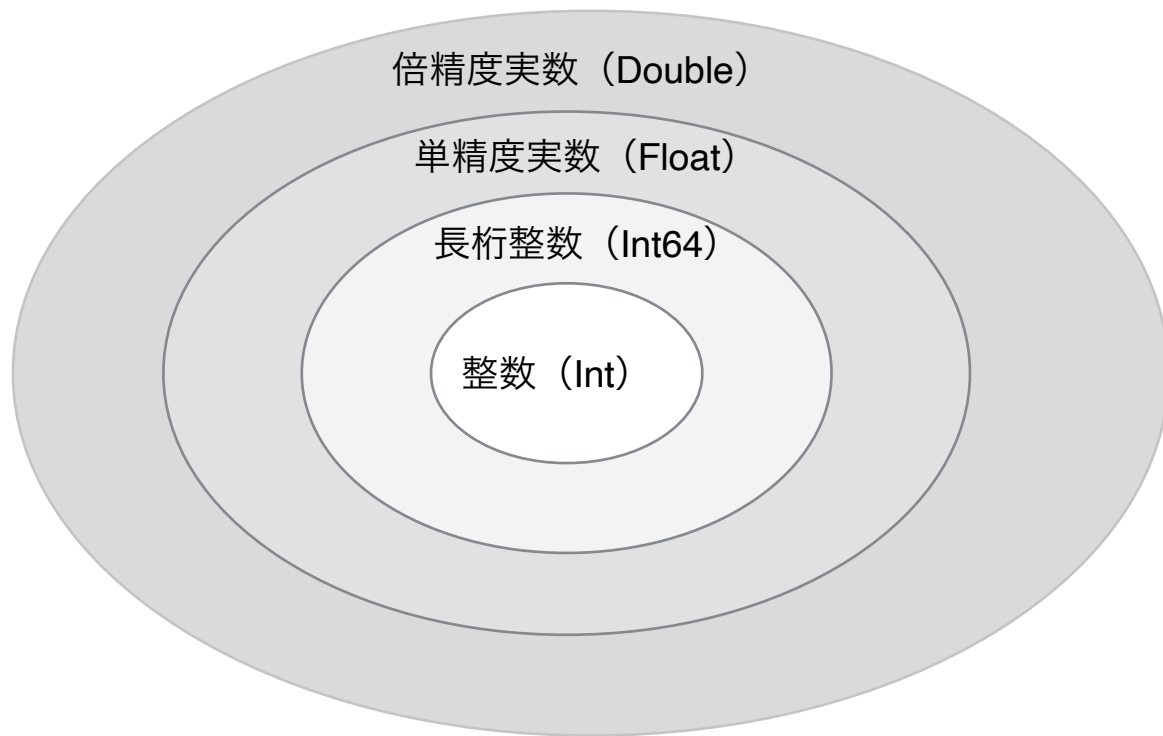
- 暗黙の型変換
 - 実数が式の中に出てくると、式の型が実数へ自動的に変換される
 - 例： $45 * 1.23$
 - 例： $3 / 2.0$
- 明示的な型変換
 - 変換用のコンストラクタ（後述）を使う

数値間の変換

- **Decimal**や**Float80**を使う場合は、以下の指定が必要
 - **import Foundation**
- 整数や他の型（クラス）の変換は、コンストラクタ（後述）を使う（**文字列からの変換と異なり、こちらはオプショナル型ではなく、その数値の型を返す**）
 - **Int(実数) ... 整数に変換される（小数点以下切り捨て）**
 - **Double(整数) ... 実数に変換される・仮数部の有効桁数は17桁程度**
 - **Decimal(実数) ... 多桁の実数（整数）に変換できる・仮数部の有効桁数は最小8桁～最大38桁程度**
 - **Float80(実数) ... IntelのGPU用の80bitサイズの精度実数に変換できる・仮数部の有効桁数は21桁程度**
- 値 **as** 型名
 - 値をより広い範囲の型に変換するとき
 - 例： `56 as Double ... 56.0`に変換される
 - `var x : Int = 56 // 整数変数としてxを宣言`
 - `Double( x ) // 変数のときは、コンストラクタ（後述）を使う方が良い`

型と集合

- 真部分集合になっている





コンストラクタを使った型の変換

- コンストラクタとは、その型（クラス）の値（オブジェクト）を作るときに使われる特殊なメソッドのこと
- 型名と同じになる
 - 例：Int(23.4)
Double(45)
- 内部的には、init(パラメータ)メソッドが、そのクラスで呼ばれている。
 - XcodeのDeveloper Documentationを開く
 - FoundationのNumbers, Data and Basic Valuesを参照