Swift Programming Lecture 5

Loop Control Statements

Tatsuo Minohara

自己参照代入文

- 左辺 = 右辺
  - これは代入文であって、等しいということを示すものではない。
  - 左辺の変数 ← 右辺の評価値

- そのため同じ変数が左辺にも右辺にも出てくる場合がある。
  - $x = x + 1$

自己参照代入文（続き）

- x = x + 1
  - xのそれまで持っていた値が評価され、+1されて、新しいxの値として代入される。
  - 同じ変数が
    - 右辺に出てきたら、それまでの値
    - 左辺に出てきたら、新しく代入される
- x = x /10 * 10
  - xの持つ値を、10で割り切れる数に丸める

自己参照代入文の省略形

- +=　　$x = x + 5 \Rightarrow x \mathrel{+}= 5$
- –=　　$y = y - 5 \Rightarrow y \mathrel{-}= 5$
- *=　　$z = z * (x+5) \Rightarrow z \mathrel{*}= x+5$
- /=　　$w = w / (x-5) \Rightarrow w \mathrel{/}= x-5$
- %=　　$u = u \% 5 \Rightarrow u \mathrel{\%}= 5$
  - =+　　$x =+ 5 \Rightarrow x = +5$ // 単項演算子
  - =-　　$x =- 5 \Rightarrow x = -5$ // 単項演算子

プログラムの状態

- プログラムの状態
  - 変数の値が一定の値にあること
- 変数の値が変われば
  - 状態遷移が起こる

```
public class Transient {
        public void main( String [ ] arg ) {
                int x;

                x = 10;
                x = x + 11;
                x = x - 78;
                x = 32 / 4;
        }
}
```

// x の値がどんどん変わっていく

繰返しを記述する構文

- **while文**
  - どのような言語でもある
- **repeat while文**
  - 1回は繰返しの中を実行する文
- **for文**
  - Swiftでは、リストあるいは範囲指定のオブジェクトに対しての繰返しになる。これに対応するfor文は、Java/JavaScript/C#などでは用意されている

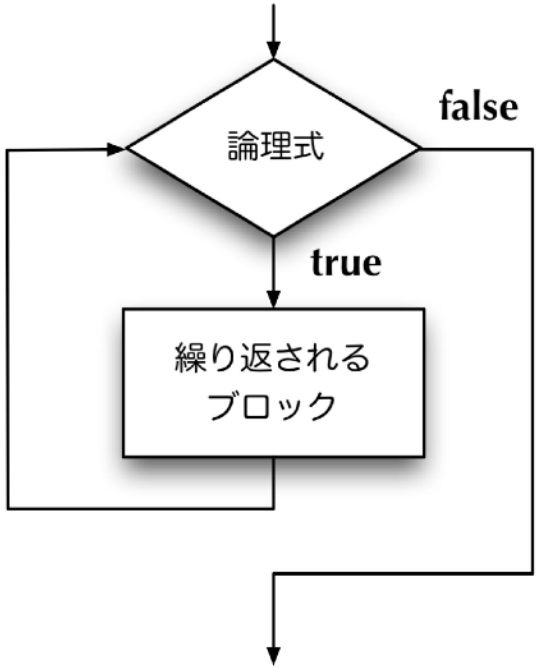
while文による繰返し

```
while 継続条件 {
    繰り返したいこと
}
```

- **while**文の意味
  - 継続条件が満たされている間、実行する
- **while**文を使うには、
  - いつかは継続条件を満たさなくなるように状態遷移させる

while文の動き

- 繰り返すたびに、論理式を評価し、falseになったら、次にいく。



繰返しを作るには

- 繰り返したい部分をブロックでまとめ、インデントする
- 状態遷移をさせる部分をブロックの中に入れる

- **while** 条件式 {
    // 繰り返したい内容
    // 状態遷移させる内容
  }

回数繰返し文

- 繰返しをしている部分がどの範囲か明確になる。

```
count=1
while count <= 10 {
    // 繰り返される内容
    count=count+1
}
```

- 字下げを手動で行なうには、TABキーを使う
- 字下げを戻すのは、deleteキー
- インデントの上げ下げは、⌘+]と⌘+[

状態遷移は変数の値を使う

- 変数の値を使って指折り数えさせることができる（ループ変数と呼ばれる）
  - このときの変数は整数型
- 変数を大きくしたり、小さくしたりして、いずれは終了させる

ループ変数の変化

- 変数の変化の仕方で繰返す回数が決まる

```
var count = 初期値
while  count < 最終値 {
    count = count  +  差分
}
```

- 繰返しの回数はCeilを使って求められる

⌈（最終値 - 初期値）／差分　⌉

- ⌈a⌉ …Ceil: aと等しいか、aよりも大きい最小の整数

ループ回数の例

```
m = 2
while m < 21 {
    m = m + 3
}
```

- 回数は、⌈（21 - 2）/ 3 ⌉で、７回

変数の値を使った繰返し

- ループ変数の値を変えていく
- 最終的に、継続条件を満たさなくする
- ループ変数の値を使ってメソッド呼出しのパラメータなどに使える
- 変数をいくつ使うべきかは、表わしたいデータの種類による

ループ変数のトレース

- ループ変数の値の変化を追う
  - 最初の値（初期値）
  - 途中の増分（差分）値
  - 継続条件が終わるときの値（最終値）
- 値の推移を追っていれば繰返しがどう動くかわかる

- 開発環境でのデバッグ方法

繰返しを作るコツ

- ループ変数がいつかは継続条件を満たさないように条件を作る
- ループ変数がどの変数か注意する
- 条件はきつめに設定する

- １回も繰返さない場合もある

繰返しの文法ミス・作成ミス

- **while** 条件式 { } while文と後ろのブロックが切り離される

- 例：１回も繰り返されない

```
var x = 1
while x > 10 {
    x += 1
}
```

- 例：永遠に繰り返される

```
var x = 1
while x <= 10 {
    x -= 1
}
```

束縛のついたwhile文

- while文の条件式の中に、オプショナル束縛のついた代入文を入れることができる
- while文の繰返しブロックの中では、ラップを外す演算子を入れる必要はない（既に外れているらしい）
- 例：

```
print( "入力を下さい：", terminator: "" )
while let line = readLine() {
    print( "入力されたものは、" + line + "です。" )
    print( "入力を下さい：", terminator: "" )
}
```

ブロックと変数のスコープ

- while文などのブロック内で宣言された変数は、そのブロックの実行が終わってしまうと、変数もなくなってしまう

- 例：

```
var  n = 1
while n <= 10 {
    var m = n-1
     n += 1
}
// m はここでは有効ではない
```

repeat while文

- 一度は繰り返したい内容を実行させたいときに使う

```
repeat {
    繰り返される内容
} while  条件式
```

繰返しの例題

- if文と組み合わせて、
  - 12ヶ月分の小の月・大の月を求める
  - 1900年から2100年までの閏年の一覧を出す
- 合同数を求める
- 任意の桁でn進数の文字列に変換する

Swiftのリストと他の言語

- Swiftのリストは、配列とリストの両方の性格を持っている。
  - Pythonのリストとほぼ同様
  - JavaScriptのArrayに似ている
  - Javaだと、ArrayListに該当するが、配列にも該当
  - C/C++だと配列に該当するが、C/C++には、標準ではリストの機能がない
- 複数の値を[ ]で括って持っておくことができる
- それぞれの値は要素と呼ばれるが、Swiftでは要素の型は統一されている必要がある（原則では。Anyを使って異なる要素のリストを作ることも可能）
- 例：[1, 2, 3, 4, 5]

  ["A", "34", "文字列", "αβγ"]

リストと変数

- 変数にも、リストを代入することもできる
  - 例： **var** xlist = [2, 3, 4, 5]

- 各要素を取り出したいときには、インデックス又はスライス（範囲指定）という記法を用いる（詳細は後のスライドで説明する）
- インデックスで１つの要素を取り出したいときは、インデックスは0から始まる
  - 例： **let** xlist = [3, 4, 5, 6]
  - xlist[2]　　// 5が取り出される
  - xlist[0]　　// 3が取り出される

for文とリスト

- **for**文の書式

```
for 変数名 in リスト {
    繰返したい内容
}
```

- 意味
  1. リストの各要素が、先頭から順番に、変数に代入される
  2. その状態で、「繰り返したい内容」が実行される
  3. 最後の要素まで代入されて実行されたら終了

for文の例

```
for n in [ 4, 3, 2, 4, 6 ] {
        print( n, terminator: "  " )
}
```

- 最初に変数nが用意され、最初の要素が代入される（この場合、整数の4）
- print関数が呼ばれ、nの値が表示される
- 最後の要素まで繰返しを続ける

Swiftにおける範囲指定（整数）

- n...m　nからmまでの整数の範囲（nとmも含む）
- 例：0...9　　0 〜 9までの範囲
- n..< m　n〜 m-1までの整数の範囲

  例：　1..<10　1〜9までの範囲
- n...　n以降すべて（リスト配列などの要素のインデックスの場合）

  例：　names[2...]　2番目以降すべて
- ...n　nまですべて（リスト配列などの要素のインデックスの場合）

  例：　names[...8]　最初から8番目まですべて
- ..<n　n-1まですべて（リスト配列などの要素のインデックスの場合）

  例：　names[..<8]　最初から7番目まですべて

範囲指定によるサブリストとfor文

- for文で繰返しの対象をリスト指定するときにも用いることができる
- 使用例：

```
let names = ["Anna", "Alex", "Brian", "Jack"]
for name in names[ 2... ] { print( name ) }
for name in names[ ..<2 ] { print( name ) }
for name in names[ 1...2 ] { print( name ) }
```

範囲指定とArray

- 範囲指定によって、作られる羅列は、Array関数（Arrayクラスのオブジェクトのコンストラクタ）によって、リストに変換することができる
- 例：

  Array(1...7) ⇒ [1, 2, 3, 4, 5, 6, 7]

- これを用いて、リストとしても利用することが可能になる
- 例：

```
var numlist = Array( 1..<8 )
print( numlist[ 1 ] )   // 2が表示される
```

範囲指定とパターンマッチ演算子

- ~= 演算子（パターンマッチ演算子）と呼ばれると範囲指定の中に数が入っているかどうかの論理値を得ることができる
- 書式：
  - 範囲指定 ~= 式
  - 例：　0..<8 ~= 8　// falseとなる

    10...37 ~= 25 // trueとなる

    3.5...4.8 ~= 3.7 // trueとなる（実数でも使用可能）

    ..<10 ~= 12 // falseとなる（片側指定でも可能）

    10... ~= 12 // trueとなる

contains関数

- 範囲の中に、その数が入っているかどうかは、~=演算子以外に、contains関数を利用することもできる
- 書式：範囲指定.contains(値) → 論理値が返される
- 使用例：

```
let range = ...5
range.contains(7)   // false
range.contains(4)   // true
range.contains(-1)  // true
```

for文と範囲指定

- **for**文の**in**の後には、範囲指定することが可能になる
- 書式は、**for** 変数 **in** 範囲指定 { 繰り返されること }
- 例：

```
for n in 0..<12 { print( n ) } // 0〜11まで表示
for n in 1...9 { print( n ) } // 1〜9まで表示
```

ループ変数を参照しない場合、省略が可能

- ループ変数を繰返しの中で使用しない場合、省略することもできる。この場合は、_（アンダーバー）が用いられる。
- 使用例：

```
for _ in 1...10 {  print( "ABC" )  } // 10回繰返し

let base = 3
let power = 12
var answer = 1
for _ in 1...power {  answer *= base }
print( "\(base) to the power of \(power) is \(answer)" )
```

リストのインデックスでアクセスしたい

- 組込み関数のcount属性がリストの長さを返してくれる
- count属性と範囲指定を組み合わせる
- 例：

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

総和・階乗を求める

- 総和

```
var summ = 0
for  i in 1...10 {
        summ = summ + i
        print( summ ) }
```

- 階乗

```
var factorial = 1
for  i in 1...10 {
        factorial = factorial * i
        print( factorial ) }
```

総和を求める

- ループ変数の値の変化に注目
- 足し合わされる変数*summ*の変化にも注目する

for文のwhere句

- for文のin句の後にwhere句を指定することが可能
  - 書式：**for** 変数 **in** 範囲指定 **where** 条件式 { }
- where句で指定された条件を満たす繰返ししか実行しない
- 例：

```
for n in 1..64 where n % 3 == 0 || n % 4 == 0 {
    print( n )
}
```

範囲指定をする関数

- Pythonのrange関数に似たstride関数が用意されている。
- 書式： stride(from: 開始値, through: 終了値, by: 間隔) あるいは

  stride(from: 開始値, to: 終了値, by: 間隔) // 終了値は含まれない
  - from:から開始、through:になるまで、by:で指定された間隔で範囲を指定できる
- 使用例：

```
stride( from: 3, through: 9, by: 3 )    // 3, 6, 9
stride( from: 4, through: 15, by: 2 ) // 4, 6, 8, 10, 12, 14
stride( from: 4, through: -4, by: -2 )  // 4, 2, 0, -2, -4
```

for文とstride関数

- for文のin句のところで、stride関数を範囲指定として使うことが可能
- 使用例：

```
let interval = 5
let minutes = 60
for tick in stride( from: 0, to: minutes, by: interval ) {
    print( tick )
}
for countdown in stride( from:10, to: 0, by: -1 ) {
    print( countdown )
}
```

動く設計

- 初期値と継続条件の設定の仕方による

➡ 1 回も実行されない

```
for n in stride( from: 3, to: 0, by: 10 ) {
    print( n )
}
```

➡ 1 回も実行されない

```
for n in stride( from: 3, through: 10, by: -4 ) {
    print( n )
}
```

enumerated関数とリスト

- 配列のenumerated関数は、リストに対して適用され、そのリストの要素とインデックスの対（タプル）から構成される新しいリストを返してくれる。
- 例：Array(["A", "B", "C"].enumerated( ))

  →　[(offset: 0, element:"A"), (offset:1, element:"B"), (offset:2, element: "C")]
- enumerate関数を用いて、インデックスと一緒にリストを探索することができるfor文を生成できる
- 例：**for** (*n, value)* **in** ["A", "B", "C"].enumerated( ) {

  print(*n, value*)

  }

zip関数とリスト

- 組込み関数のzip関数は、複数のリストに適用することができ、各リストの先頭から、順番に要素の対（タプル）のリストを生成できる。
- 例：Array(zip(['Kobe', 'Kyoto', 'Osaka'], ['神戸', '京都', '大阪']))

  → [('Kobe', '神戸'), ('Kyoto', '京都'), ('Osaka', '大阪')]
- zip関数とfor文を組み合わせて、複数のリストを先頭から探索するようにすることができる
- 例： **for** (en, jp) **in** zip(['Kobe', 'Kyoto'], ['神戸', '京都']) {

  print(en, jp)

  }

while文とfor文との互換性

- **while**文を**for**文で書き直す場合は、ループ変数に一定の変数を足したり、引いたりしている場合に限られる

  var n=A
  while n < B { 文 n += 1 }

  → **for** n **in** A..<B {文 }

- **for**文を**while**文で書き直す

  for n **in** A..< B { 文 }

  → **var** n= A
  while n < B { 文 n += 1 }

break文

- **break**文に出会うと、一番内側のブロックから脱出する
- ブロックのネストが深い場合は、脱出レベルに対してラベルをつけることもできる
- 入力のガード（既定値以外の入力をさせないようにする）にも**while**文と共に良く用いられる。
- 途中だけ処理をしたい場合に使われる
- **while true** {
  A
  if 終了の条件式 { **break** }
  B
  }
  - A → B → … → B → A → 終了

break文とガード

- 必要以外の値を入力しないようにする

```
var value: Int = -1
while  true {
    value = Int( readLine()! )!  // 入力文字列を整数として解釈
    if value >= 0 {  break }   //  0以上なら脱出
    print( "正の数を入力のこと", terminator: " " )
}
```

- while文を抜けた段階では、0以上の値であることが保証されている

ラベル付き文とbreak

- ブロックのネストが深い場合は、脱出レベルに対してラベルをつけることもできる
- 使用例：

```
import Foundation


elevel:
for n in 1...9 {
    for m in 1...9 {
        print( String( format:"%3d", n*m), terminator: " " )
        if n * m > 50 { print( ) break elevel }
    }
    print( )
}
```

← breakにより、ここに脱出

continue文

- **continue**文は、次の繰返しにいく
- インデントを深くしない場合に使うことが多い

- **while true** {
  if 除外する条件 { **continue** }
  繰り返す処理
  }

ラベル付き文とcontinue

- ブロックのネストが深い場合は、次の継続のレベルに対してラベルをつけることもできる
- 使用例：

```
import Foundation
clevel:
for n in 1...9 {
    for m in 1...9 {
        print( String( format:"%3d", n*m), terminator: " " )
        if n * m > 20 { print( ) continue clevel }
    }
    print()
}
```

→ continueは、外側の繰返しの次の繰返しに進む

do try 文

- 例外（実行時に起こるエラー）を処理するために使われる文
- 書式：

```
do {
        try 式 { ブロック }
        catch パターン { ブロック }
        catch パターン where 条件式 { ブロック }
                :
}
```

- try句の式の後に、ブロックがなくても良い
- catch句では、whereが入っても良い。また、catch句はいくつあっても良い。
- 例：

```
do {
    try vendingMachine.vend(itemNamed: item)
} catch is VendingMachineError {
    print("Invalid selection, out of stock, or not enough money.")
}
```

try式

- try?式は、変数に値を代入するときなどに使われる
  - 使用例：

    ```
    let x = try? someThrowingFunction()
    ```
  - これは以下に等しい

    ```
    let y: Int?
    do { try y = someThrowingFunction() }
    catch { y = nil }
    ```
- try!式は、これ以上エラーを拡げないときに使われる
  - 使用例：loadImageに失敗してもエラーがでない（単にphotoにはnilが代入される）

    ```
    let photo = try! loadImage(atPath: "./Resources/John Appleseed.jpg")
    ```