Swift Programming Lecture 6

Functions

Tatsuo Minohara

関数を定義する理由

- 例：多重の繰返しで、プログラムの一部分が一体何をやっているのかわからなくなってきた
  - 意味のあるブロックに名前をつけて、外に出して、それを呼び出すようにする。
  - drawSineCurveなど
- 例：少しだけ異なる（例えば一辺の長さあるいは角度が違うだけ）がほぼ同じ処理をしている部分がある。
  - その機能に名前を付けて、異なるデータをパラメータで受け渡して、プログラムを構造化する

関数の２つの局面

- 関数を定義する
  - **func**構文を用いて定義する。
  - **func** drawPaint(c: Canvas) { .... }
- 関数を呼び出す
  - これは、いままでも散々やってきました。
  - print("Hello, Swift Programming")

関数の定義（記述）

```
func 関数名( ) {
        その関数が呼ばれたら行なわせたい内容
}
```

- 行なわせたい内容は、for文などと同様にブロックの形にしてインデントを下げる
- 関数名は、小文字始まりで動詞を使うのが一般的
- 動詞＋名詞→　setRectangle　のように名詞を大文字にする

関数の定義の例

- **func** displayNumber( ) {
  for n **in** 1...10 {
  print(n, terminator: " ")
  }
  print( )
  }

- **func** shortSleep( ) { print("ZZZ…") }

関数の呼出し

関数名( )　　あるいは

オブジェクト名.関数名( )

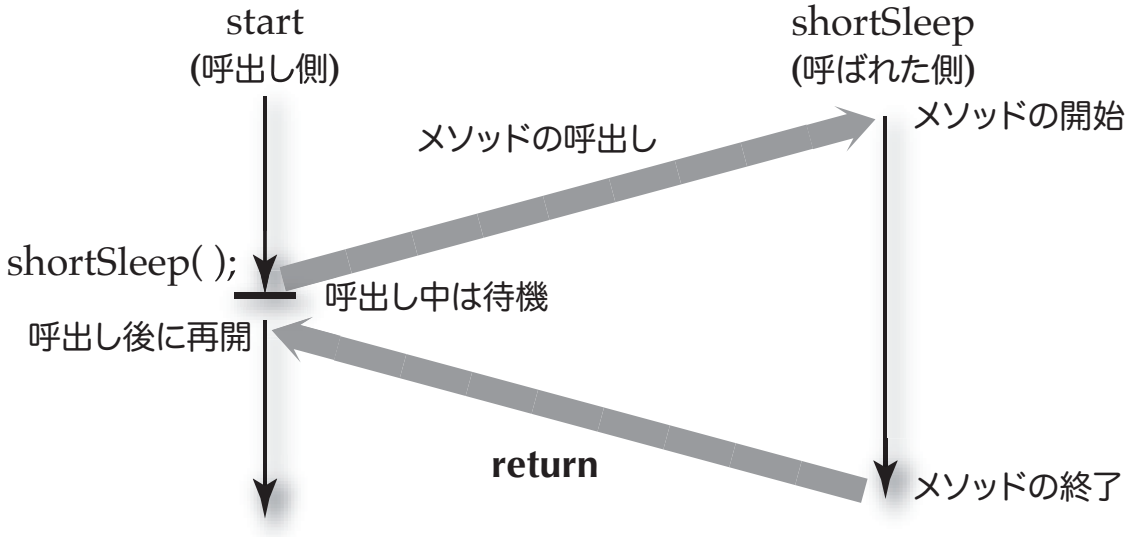
定義された関数であれば、関数名だけで呼び出せる

displayNumber( )

shortSleep( )

関数の呼出しの構造

- shortSleepの場合



引数のある関数の定義

```
func 関数名(  変数名: 型名 [, 変数名]… ) {
        関数で行なわせたい内容
}
```

```
func drawCircle( radius: Int ) {

}
```

関数名

引数を受け取る変数名

この関数ブロック内では*radius*が使える

引数のある関数の呼出し

関数名(引数名: 実引数の式 [, 引数名: 実引数の式]...)

drawCircle(radius: 34 * x)

まずこの部分が計算される

先ほどの*radius*に代入される

仮引数と実引数

- 仮引数（仮パラメータ）

  ➡関数の定義で宣言されている変数のこと

- 実引数（実パラメータ）

  ➡関数を呼び出すの式のこと。関数を呼び出すときに、まず式が評価されて、定数値（あるいはオブジェクトを指す値）になってから、仮引数に代入されて、該当の関数が呼び出される。

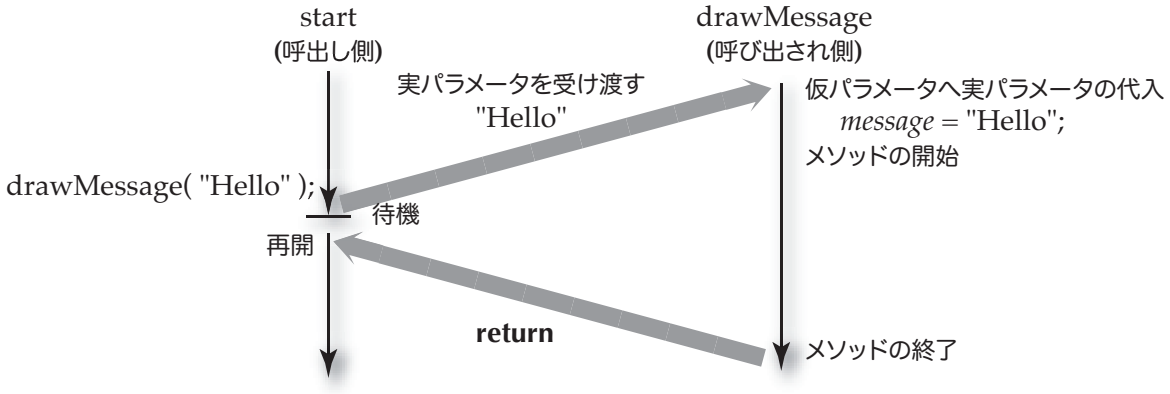
引数のある関数の例

- 改行しないで表示を行なうdrawMessageの例

```
// drawMessageの定義
func drawMessage( message: String ) {
    print( "Message is ", message, terminator: "" )
}

// drawMessageを呼び出す
drawMessage( message: "Hello" )
```

引数のある関数の呼出しの構造

- drawMessageの場合



複数の引数

- 複数の仮引数があるときは、該当する実引数が仮引数に代入される

```
func  displayPower( base: Int, exp: Int ) {
     var result: Int = 1
   if exp == 0 { print( result ) }
   else {
      for _ in 1...exp {
         result *= base
      }
      print( result )
   }
}

displayPower( base:12, exp:3 )   // 12がbaseに、3がexpに代入される
```

順番での仮引数への代入（Swift特有）

- 関数の定義で、順番に実引数を仮引数に代入することができるように、仮引数名を省略したい場合に、省略の記号（アンダーバー：_）を付けることが可能
- 例：

```
func displayPerson(_ name: String, _ gender: String ) {
  print( "Name:", name, " Gender:", gender )
}
```

- 引数名のラベルを省略する場合は、順番に引数を指定することで、その仮引数に代入することができる。
- 例：

```
displayPerson( "John", "male" )
displayPerson( "Mary", "female" )
displayPerson( "Susanna", "female" )
```

ラベルと仮引数の変数を別々に用いる（Swift独自）

- 仮引数の変数名と別にラベルをつけることが可能である
  - 書式：**func** 関数名(ラベル名　仮変数名: 型名) { 関数の本体 }
  - 呼出しの書式：関数名(ラベル名: 実引数)
- 例：

```
// 定義

func printPerson(person: String, from hometown: String) {
    print( "Name: \(person)  come from \(hometown)." )
}
// 呼出し

printPerson( person: "Bill", from: "Cupertino" )
```

省略を指定できる仮引数（Python, Swift等）

- 関数の定義で、引数のところに=をつけて、その引数が省略された場合のデフォルト値を指定しておける
- 例：

```
func displayPerson( name: String = "John Smith", age: Int = 32 ) {
  print( "Name:", name, " age:", age )
}
```

- 省略できる引数の場合でも、順番にその引数に代入することができる。また、実引数を省略しても構わない。
- 例：

```
displayPerson( )
displayPerson( name: "Mary" )
displayPerson( age: 54 )
displayPerson( name: "Susanna", age: 26 )
```

可変の個数の仮引数

- 書式：

  func 関数名(_ 変数名 : 型名...) { }

- 例：

```
func printValue(_ values: Double... ) {
    for value in values { print( value, terminator: " " ) }
    print()
}
```

- 関数内では、可変の個数を受け取った変数は、タプルとして利用することができる。
- 例：

```
print( values )
```

関数の構造化と呼出し

- 一連の細かな作業を１つの機能として定義したい。
- それぞれの細かな作業はそれぞれ既に関数として定義されている。その間を調整したい。
  - それらの関数を呼び出す新たな機能を持つ関数を定義する→ボトムアップの構造化
  - 最初は、その関数を呼び出す。

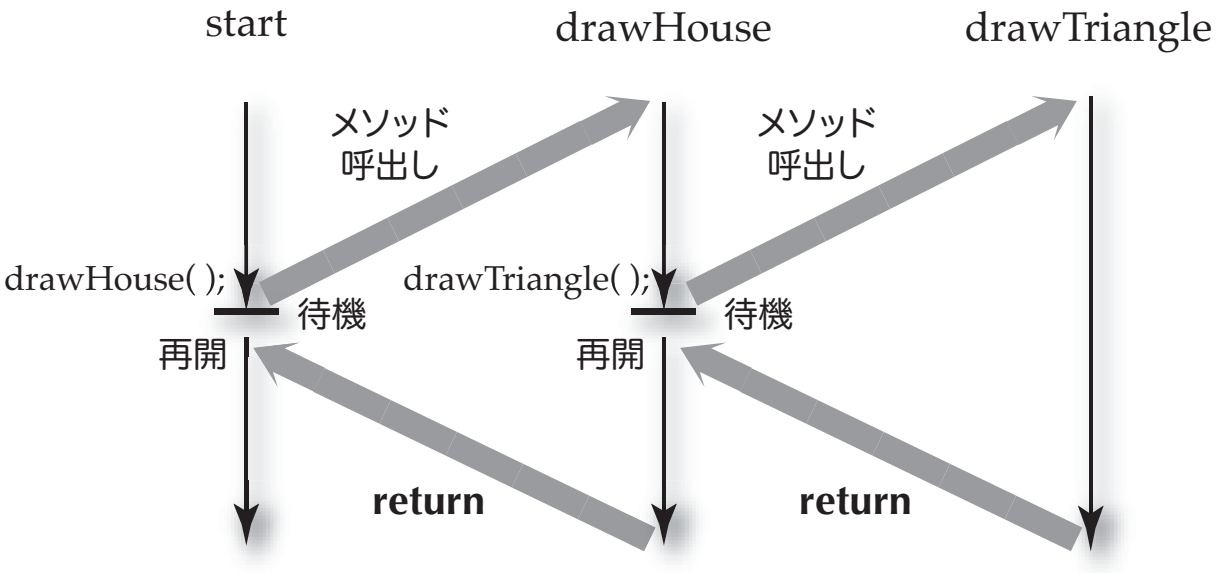
関数の構造化の意義

- 大きな作業を１つの関数として定義する。
- その作業を実現してくれるような関数を更に新たな関数として定義していく

- 段階的詳細化→トップダウンの構造化

多重の関数呼出しの構造

- drawHouseの場合



グローバル変数とローカル変数

- グローバル変数（大域変数）はすべての関数で参照可能
- 関数の中だけで使われるローカル変数（局所変数）は、関数の実行と共に消える
- グローバル変数とローカル変数が同じ名前を使っていると、関数の中では、ローカル変数が優先される（グローバル変数を隠蔽する）
- 同じ名前だった場合、ローカル変数に代入してもグローバル変数の値は書き変わらない

```
var x=20 // Global Variables

func sample( ) {
   var x=30 // Local Variables
}
```

関数の中でのグローバル変数の書換え

- 関数の中で、同じ名前の変数をローカルで宣言していない限り、グローバル変数を書き換えることができる

- 例：

```
var x = 20
func  sample( ) {
    x = 30
    var  x = 40   // こちらはグローバル変数を隠蔽するローカル変数
}
sample( )  # 実行後は、xの値は30になる
```

引数で受渡し vs グローバル変数

- 最初に１回だけ設定して、後は参照だけするような情報は、グローバル変数でも良い。

```
// 共通で使うグローバル変数
let win = Window( width=500, height=500 )

// 関数から、winでアクセス
func paintCircle( ) {
    win.draw_circle( 100, 100, 50 )
}
```

inout 仮引数

- 昔のcall by reference（書き換えても良い参照渡し）を保証するためのもの
- 呼出し側の実引数の変数の書換えをする場合、inoutを型名の前につける
- 実引数の変数には、書き換えを示すための&（参照渡しであることを示す記号）をつける
- 例：

```
func changeGlobal(_ a: inout Int) {
    a = 10
    print( "func:", a )
}
var x = 20
changeGlobal( &x )   // & は変数xへの参照を示す
print( x )
```

仮引数の値の変更

- 仮引数の値の変更は、できない
- もし、仮引数の値の変更をしたい場合は、同名のローカル変数を宣言し、それに代入する
- 例：

```
func useParam( x: Int ) {
    print( "before:", x )
    var x = x
    x += 100
    print( "after:", x )
}
```

多層型の関数定義

- 同じ名前で、引数の数、引数の型を変えた関数を定義することができる。別の関数として見做される。
- 例：

```
func printSquare(_ n: Int )  { print( n * n ) }
func printSquare(_ n: Double ) { print( n * n ) }
func printSquare(_ n: Bool ) { print( n && n ) }


printSquare( 12 )
printSquare( 12.3 )
printSquare( true )
```

値を戻す関数

- **return**文を使う。呼出し側に値を戻せる。
  - 関数の中で、
  - **return**　式
  - 例：　**return**　34 * 32

- 関数を定義する際に、戻される型を指定する
  - **func** メソッド名(　仮パラメータ　) -> 戻される型 {　}
- 例：　**func**　square(　x: Int)　-> Int { ..... }

↑戻される型が整数型であることを示す

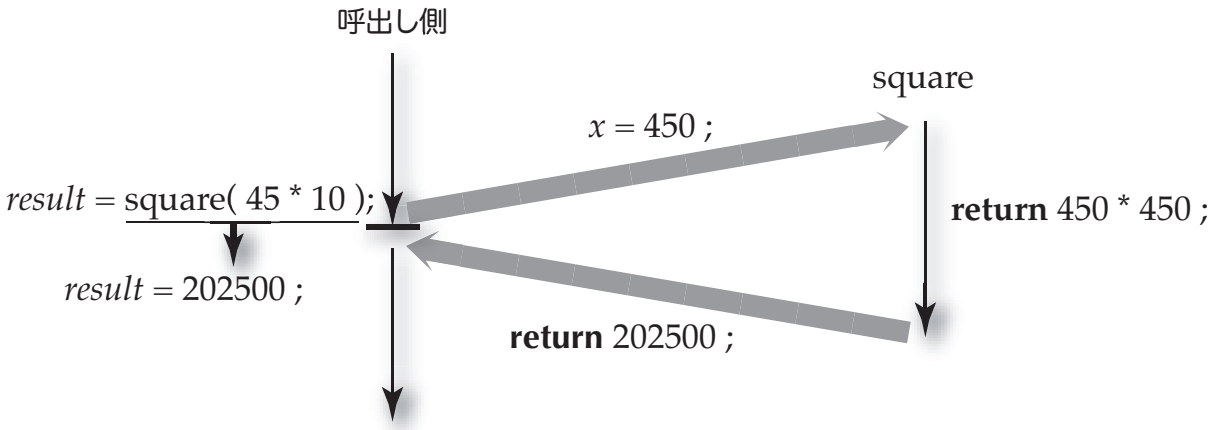
squareとgreater

```
// 与えられた値の２乗を返す関数

func  square (_ x: Int  ) -> Int { return   x * x }


// ２つのうち、大きいほうの値を返す関数

// max( x, y )と同じ

func  greater(_ x: Int, _ y: Int ) -> Int{
      if  x > y { return  x }
      else { return y }
}
func  greater(_ x: Int, _ y: Int )  -> Int { return x > y ? x : y }
```

値を戻す関数の呼出し

- 変数に代入する式の中で呼び出される。
- まず実パラメータが評価され、呼び出された後に代入される。

➡例： *result* = square(45 * 10)



値を戻す関数の多重化呼出し

- 他のパラメータ付き関数の呼出しの際に、実パラメータの中で呼び出される

- 例: *result* = square(greater(34, 56))

  ➡まず、34, 56の２つパラメータでgreaterが呼び出される。

  ➡返ってきた値を実パラメータとして（この場合は56）、squareが呼び出される

  ➡返ってきた値が*result*に代入される（3136）

複数の値を戻す関数

- Pythonのように、複数の値を戻すこともできる

例：

```
func multiReply( ) -> (Int, Int, Int) {
    return (12, 13, 14)
}

var (a, b, c) = multiReply()
```

複数の値を返す関数の呼出し

- 受け取る側でも、複数の変数を用意し、カンマで区切って代入する

```
func getMaxMin(_ a: Int, _ b: Int ) {
    return (max( a, b ), min( a, b ) )
}

var (more, lesser) = getMaxMin( 56, 89 )
```

値を返す関数の多層型の関数定義

- 同じ名前で、引数の数、引数の型を変えた関数を定義することができる。別の関数として見做される。
- 例：

```
func square(_ n: Int ) -> Int { return n * n }
func square(_ n: Double ) -> Double { return n * n }
func square(_ n: Bool ) -> Bool { return n && n }


print( square( 12 ) )
print( square( 12.3 ) )
print( square( true ) )
```

約数と素数

- 約数を求めて表示する関数displayDivisorsを作る
  - 繰返しで、その数まで割り切れる数があったら、表示するようなもの
- 約数の個数を求める関数countDivisorsを作る
- 素数であるかどうかを判定する関数isPrimeを作る
  - 素数とは、１とその数でしか割りきれない数のこと
  - 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, …

完全数とピタゴラス数、そして合同数

- 完全数…その数を除く約数の和が、その数と等しい
- 例： $6 = 1 + 2 + 3$　6の約数は1, 2, 3, 6

- ピタゴラス数…$a^2 + b^2 = c^2$ を満たす自然数の組 (a, b, c)のこと
- 例：$3^2 + 4^2 = 5^2$　なので　(3, 4, 5)

- 合同数…ピタゴラス数のa, bを使って、$n = a * b / 2$の数

- 完全数とピタゴラス数、合同数を表示する関数を作成してみる

合同数

- 直角三角形の面積となるような数
- 以下を満たすようなnが合同数である（https://ja.wikipedia.org/wiki/合同数）

$$a^2 + b^2 = c^2$$

$$\frac{ab}{2} = n$$

- 整数の合同数を求めてみる

- 一部の合同数は、以下のように求めることもできる。p を奇数の素数（奇素数）とする。
  - p を 8 で割ったあまりが 3 のとき、p は合同数ではなく、2p は合同数である。
  - p を 8 で割ったあまりが 5 のとき、p は合同数である。
  - p を 8 で割ったあまりが 7 のとき、p と 2p は合同数である。

2進数と完全数

- Wikipediaの「完全数」の項目参照
- 完全数6, 28などを2進数で表わしてみると、110, 11100というように、1の前後に1と0が同数つくような形になっている
- この1...1の部分は、2^p-1ということでメルセンヌ数と呼ばれている。10進数になおすと、$11_{(2)}=3=2^2-1$ とか $111_{(2)}=7=2^3-1$ など。
- また、10....の部分は、2^{p-1} で表わされる。
- つまり、完全数の候補は、$\left(2^p-1\right)\left(2^{p-1}\right)$ で表わされることになる。
- メルセンヌ数が素数であった場合（メルセンヌ素数と呼ばれる）、この完全数の候補は、完全数であることが知られている

素因数分解

- 与えられた数nを素数の積として分解する
- 2〜n/2までの間で、素数かどうかを判定する
- 素数だったら、nをその素数で割り切れる間は、割り続ける
- nが1になったら終了

再帰呼出し

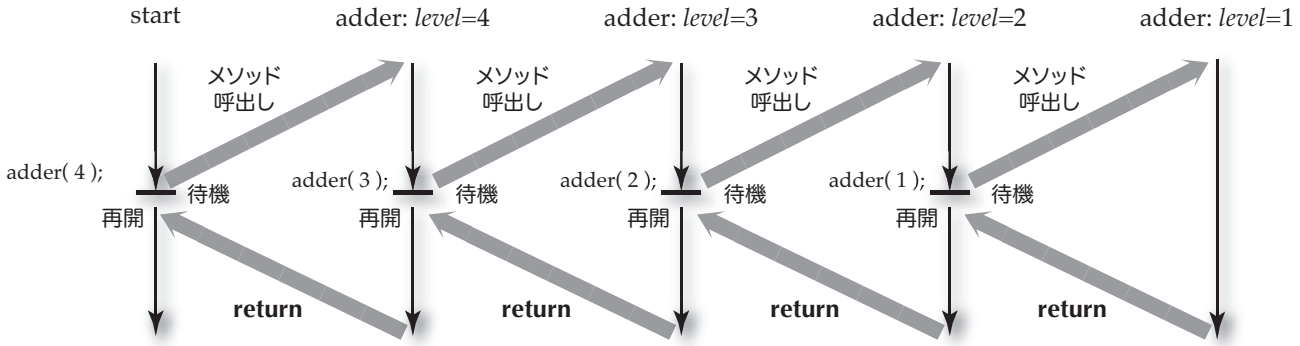
- 関数の中から、その関数自身を呼び出す
- 引数を利用する関数で実現することができる
- 次のような方法でプログラミングする
  - １．基底レベルでの処理を記述
    - そのレベルでは、もう再帰呼出しをしない
  - ２．それ以上のレベルでの処理を記述
    - 再帰呼出しおよび、その前後の処理を記述
- 基底レベルがないとプログラムが止まらなくなる

値を戻す関数と再帰

- 階乗を計算するfactorial
- $n! = 1 \times 2 \times 3 \times \ldots \times n-1 \times n$
  - n == 1のときは、1を返す
  - n > 1のときは、n * factorial(n-1)を返す

- 総和を計算するsummation
- $\Sigma_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n-1 + n$
  - n == 0のときは、0を返す
  - n > 0 のときは、n + summation(n-1)を返す

再帰呼出しの過程


start
adder: level=4
adder: level=3
adder: level=2
adder: level=1
メソッド
呼出し
adder(4);
adder(3);
adder(2);
adder(1);
待機
再開
return

ユークリッドの互除法

- 最大公約数を求める方法
  - ２つの数のうち、大きい方と小さい方に分ける
  - 大きい方の数を小さい方の数で割り切れたら、小さい方の数が求める答え
  - 割り切れなかったら、次の大きい方の数に小さい方の数を代入し、小さい方の数には、「大きい方の数%小さい方の数」を代入して、上記の判定を繰り返す

  - 大きい方の数 % 小さい方の数 （余りを使う方法）
  - 大きい方の数 – 小さい方の数（差を使う方法）
- 余りを使った方が、はやく収束する。

GCDの関数

```
// ２つの数の最大公約数を求める関数（再帰版）
func gcd(_ n: Int, _ m: Int ) ->  Int {
        var (more, less) = ( max( n, m ), min( n, m ) )
        if  more % less == 0 { return less }
        else { return  gcd( less, more % less ) }
}

let value = gcd( 356, 248 )
```

べき乗を求める関数

- 他のプログラミング言語では、べき乗を求める演算子がない。それと合わせるために、べき乗をもとめる関数を作ってみる。
- パラメータは、基になる数と、指数

- **func** power(_ x: Int, _ n: Int) -> Int { # xのn乗を返す

```
  var result = 1
  for _ in 0..<n {  result *= x }
  return  result
}
```

回文を作る

- 最初は、任意の文字を選ぶ
- 前後に同じ文字（任意の文字）を追加していく
- これを再帰で行なう

単位分数

- フィボナッチの強欲算法のアルゴリズムを用いて、分数を単位分数に直して表示する。

$$\frac{x}{y} = \frac{1}{\lceil y/x \rceil} + \frac{x - (y \bmod x)}{y\lceil y/x \rceil}$$

- このアルゴリズムの部分は、再帰を用いて記述する。
- 強欲算法については、Wikipediaの「エジプト式分数」の項を参照。

値を返す関数の呼出しの文字列埋込み

- 文字列中に、\(変数名)で、変数の値を埋め込めるのと同様に、\(関数名(実引数, …))で、値を返す関数を呼び出した結果の値を埋め込むことができる
- 例：
  - print("\(m)と\(n)の積の二乗は\(square(m * n))")
  - print("\(cube(123))")

関数名に別名を付ける

- 関数の引数や戻り値の型を定義して、別名を付けることができる
- 例：

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

var mathFunction: (Int, Int) -> Int = addTwoInts
print("Result: \(mathFunction(2, 3))")    // Prints "Result: 5"
let anotherMathFunction = addTwoInts
```

高階関数（引数としての関数）

- 別の関数のパラメータ（実引数）として関数を渡すもの。

例：

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

func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) {
    print("Result: \(mathFunction(a, b))")
}
printMathResult(addTwoInts, 3, 5)
// Prints "Result: 8"
```

戻り値としての関数

- 関数自体を戻り値にすることができる
- 例：

```
func stepForward(_ value: Int) -> Int { return value + 1 }
func stepBackward(_ value: Int) -> Int { return value - 1}

func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    return backward ? stepBackward : stepForward
}
var currentValue = 3
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)
currentValue = moveNearerToZero( currentValue )
```

無名関数（クロージャ：Closure）

- パラメータを貰って値を返す関数を名前を定義しないで、指定することができる。
- 書式：

  { (仮引数 : 型名) -> 戻り値の型名 **in** 文 }

- 赤色の部分は省略できる

  例：

```
var f1 = { () -> () in print( "Hello" ) } // 省略無し
var f2 = { () -> Void in print( "Hello" ) } // ()とVoidは同じ
var f3 = { () in print( "Hello" ) } // 戻り値の型を省略
var f4 = { print("Hello" ) } // すべて省略
```

- 同じ変数に別のクロージャを再定義することが可能（**let**で代入していない場合）

クロージャの利用例

- 高階関数の引数のところにクロージャを利用してみる
- 例：

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]


let reversedNames = names.sorted(by:
    { (s1: String, s2: String) -> Bool in return s1 > s2 }   )
```

クロージャの省略記法

- sort(by:)高階関数で使われるクロージャは、(String, String) -> Boolの型が推論できるから、以下のように書ける

```
let reversedNames = names.sorted(by: { (s1, s2) in return s1 > s2 }　)
```

- 更に**return**も省略できる

```
let reversedNames = names.sorted(by: { (s1, s2) in s1 > s2 }　)
```

- 更に簡略引数名でも記述することができる（$nは、n番目の引数、0から始まる）

```
let reversedNames = names.sorted(by: { $0 > $1 }　)
```

- 更に比較演算子だけでも記述することができる（わかりにくいので、これは使わないで）

```
let reversedNames = names.sorted(by: >　)
```

trailing（延々と記述する）closure

- クロージャを実引数に取る高階関数の呼出しにおいて、通常の関数のように延々と何行も記述したい場合に使う記法
- 次のような関数定義があった場合

```
func someFunction(closure: () -> Void) {  /* 関数本体 */ }
```

- 次のようにクロージャを使って関数呼出しを記述できる

```
someFunction(closure: { /* クロージャの本体 */ })
```

- また、この呼出しは更に次のように省略して記述できる

```
someFunction( ) { /* クロージャの本体 */ }
```

trailing closureを使ってsort(by:)を呼び出す

- 先ほどのsort(by:)でのクロージャをこの記法を使って呼び出すと次のように記述できる

```
let reversedNames = names.sorted() { $0 > $1 }
```

- 更に、引数がtrailing closureだけの場合は、次のように記述できる

```
let reversedNames = names.sorted { $0 > $1 }
```

- 既に、この記法は、文字列フォーマットのときに出てきている

```
import Foundation
"Hello world".withCString { print( String(format: "%s!", $0) ) }
```

ジェネリック関数

- ほぼ同じことをしているのに、複数の型の変数に対して、いちいち関数を定義したくないときに用いる
- 関数名の後に<T>を付けて記述する。Tは、実引数で与えられた型が代入される。
- 例：

```
func swapper<T>(_ a: inout T, _ b: inout T ) {
    let temp = a
    a = b
    b = temp
}

var (x, y, w, z) = (2.3, 3.4, 12, 56)
swapper( &x, &y )
swapper( &w, &z )
print( x, y, w, z )
```

ジェネリック関数の型に制限を掛ける

- <T: 制限する型あるいはプロトコル> あるいは、
  where T: 制限する型あるいはプロトコルで制限を掛けることができる
- 例：べき乗を返す関数

```
func power<T: Numeric> (_ org: T,_ n: Int) -> T {
    var result : T = org
    if n == 1 { return result }
    for _ in 2...n { result = result * org }
    return result
}
power( 3, 4 )  // 整数で呼び出した
power( 1.2, 5 )  // 実数で呼び出した
```

- 上記の関数の宣言部は、次のようにも記述できる

```
func power<T> (_ org: T,_ n: Int) -> T where T: Numeric
```

制限できるプロトコルの例

- 基本的なもの
  - Numeric … 数値的なもの（乗算が可能）
  - SignedNumeric … 符号の付いた数値的なもの
  - AdditiveArithmetic … 数値的なもの（加減算が可能）
  - Comparable … 比較可能なもの
  - Equatable … 等しいかどうかだけは比較可能なもの
  - Strideable … stride関数が適用可能なもの
- 整数系
  - BinaryInteger, SignedInteger, UnsignedInteger, FixedWidthInteger
- 実数系
  - FloatingPoint, BinaryFloatingPoint

guard文と関数

- guard文は、関数の中で、値が設定されないとき（nil値になるとき）は、returnで終了するために用いられる
- 書式：
    - guard let 変数 = 式 else { returnで終わるブロック }
- 例：

```
func greet(person: [String: String]) {
    guard let name = person["name"] else {
        return
    }
    print("Hello \(name)!")
}

greet( person: [ "name": "Jane" ] )
```

内部関数

- 関数のブロック内部で関数を定義するもの
- グローバル変数と同じような変数の共有・隠蔽が可能になっている
- 例：

```
func sample( ) {
  var x = 10
  func dummy( ) { let x = 20; print( "dummy:", x ) }
  func updater( ) { x = 30; print( "updater:", x ) }
  dummy( ) ; print( "sample:", x )
  updater( ) ; print( "sample:", x )
}
sample( )
```