

Swift Programming

Lecture 7

Array

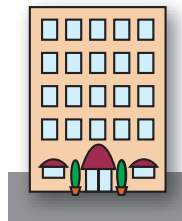
Tatsuo Minohara

配列・リスト

- 配列には 1 つの変数の中に複数のデータ値を入れることができる。



一戸建て(普通の変数)には一世帯



マンション(配列変数)には複数の世帯

- 通常の変数をスカラー変数、リストや配列をベクトル変数と呼ぶこともある。



空の配列の記法

- 基本的には、配列を示す[]で値やオブジェクトを囲むだけ
- 例： `var xlist: [ Int ] = [ ]` // 空の配列が作成される
- 組み込みのArray() 関数を使っても配列を作成することができる。ただし、作る際に、要素の型を指定する必要がある。
- 例： `var xlist: [ Int ] = Array()`
`var nlist = Array<Int>()`



配列の確保と初期値

- 配列については、各要素の初期値を指定しないと、サイズ分確保されない。
 - ▶ 例： `var xlist = [ 1, 2, 4, 9 ]` // 4つのサイズの配列が作られる
 - ▶ 例： `var nlist = Array( [ 12, 23, 34, 45 ] )`
 - ▶ 例： `var xlist = Array( arrayLiteral: 12, 34, 45, 56 )`
 - ▶ 例： `var xlist = Array( 1...10 )` // 各要素に1～10までの値が代入される
- 同一の初期値が設定された配列を作りたい場合は、次のように作る
 - ▶ 例： `let eights = Array(repeating: 0, count: 8)` //サイズは、8となる
 - ▶ 例： `let twenties = Array(repeating: "Z", count: 20)` //サイズは、20となる

配列への初期値代入

- 代入時に各要素の初期値が代入できる

- 配列名 = [初期値, ..., 初期値]

- 例 : **var** a: [Int]

- a = [10, 7, 4, 6, 3]

- 等価 : **var** a = Array(repeating: 0, count: 5)

- a[0]=10; a[1]=7; a[2]=4; a[3]=6; a[4]=3

- 範囲指定を利用して配列を作成することができる

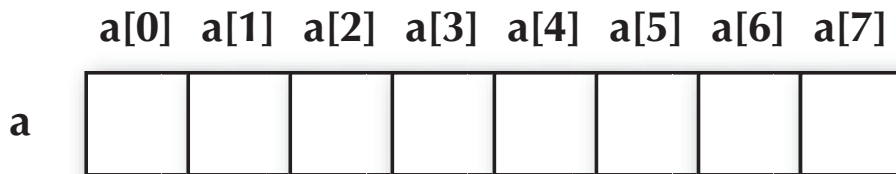
- 例 : **let** nlist = Array(1...10)

- let** mlist = Array(2..<8) // 2から7まで

- let** ylist = Array(stride(from: 3.2, to: 4.0, by: 0.1))

配列の要素へのアクセス

- 確保された配列には、各データを参照するための部屋番号がついている。これをインデックス（指標・添え字）と呼ぶ。



- インデックスは、「0～サイズ-1」の範囲の整数に限られる。範囲外だと実行時エラー（例外）が発生する。



要素への代入・参照

- 配列の要素は、通常の変数のように用いれる。
- 要素を参照するには、次のような書式を用いる（インデックスは整数の式である）。
 - 変数名[インデックス]
 - 例： `c.create_line( x[ 1 ], y[ 1 ], x[ 2 ], y[ 2 ] );`
 - `z = x[ 4 ] * 3`
- 要素に個々に代入するには次のような書式を用いる
 - 変数名[インデックス] = 式
 - 例： `a[ 3 ] = 10`



インデックスの式

- インデックスは整数の式なので、変数や計算するものであっても良い。

- ▶ $x = 3$

- ▶ 例： $a[x] = 5$

- ▶ 例： $a[x+3] = 10$

- 配列の要素を参照する場合は、まずはそのインデックスの式から評価される。

- ▶ 例： $a[a[x+2]] = 20$

- ▶ $x = 3; a[5] = 4 \rightarrow a[a[5]] \Rightarrow a[4] = 20$



配列の要素への代入

- 要素へ代入するとき、代入文の省略形も使える
 - `a[ 4 ] += 67`
 - `a[ 4 ] = a[ 4 ] + 67`

 - `a[ 5 ] += 1`
 - `a[ 5 ] = a[ 5 ] + 1`



配列と繰返し

- 配列は繰返しを用いて操作する。
- 代入を試みる（アプリケーションで）
 - インデックスの値を使う
 - 漸化式を使う
 - 乱数を使う
 - `Int.random(in: a..b)`
 - `[ a, b )`の間の整数を発生させる
 - `Int.random( in: a...b )`
 - `[ a, b ]`の間の整数を発生させる



乱数ライブラリ

- Double, Floatなどでも同様に乱数を発生させることができる
- 以下の関数が見える。一様乱数の場合
 - Bool.random() ...trueか、falseかのいずれか
 - Double.random(in: a...b)...a以上b以下の実数
 - Double.random(in: a..**b**).....a以上b未満の実数
- 配列の要素を使って乱数を発生させることも可能（ただし、オプション）
 - 例：**let** array = Array(1...10)

let randomElement = (array.randomElement())!



乱数で初期化された要素を持つ配列

- 1...100の範囲の乱数で要素が初期化された配列を作る

▸ 例：

```
var a = Array( repeating: 0, count: 20 )  
for i in 0..a.count {  
    a[ i ] = Int.random( in: 1...100 )  
}
```



配列のサイズ

- 配列はcountを使えば、配列のサイズが計算される。
 - ➡ 配列名.count
 - ➡ 例：a.count → 配列aのサイズが整数で求まる
- JavaやJavaScriptの配列では、同じように属性として、a.lengthがあるので、この書式を使えば、配列のサイズがどのようなものでも対応できる。
- Pythonでは、len(配列名)組込み関数が利用可能
- C/C++にはないので不便だが、以下の書式で計算できる
 - sizeof(配列名) / 4 を使う(整数が4バイトのとき)



配列への演算

- 配列 + 配列

- 配列の追加になる

- 例： $[1, 2] + [3, 4] \Rightarrow [1, 2, 3, 4]$

- $["A", "B"] + ["C"] \Rightarrow ["A", "B", "C"]$

- 配列.contains(値)

- その値が、配列の要素として含まれていればtrue

- その値が、配列の要素として含まれていなければfalse



Arrayで使える属性・関数

- 配列（インデックスで参照可能）としての特性とリスト（追加・挿入・削除可能）の特性を併せ持つ
 - count ... 個数を返す（整数）
 - isEmpty ... 空かどうか返す（論理値）
 - first ... 先頭の要素を返す
 - last ... 最後の要素を返す
 - append(要素) ... 最後に要素を追加
 - append(contentsOf: 配列) ... 最後に配列を追加
 - insert(要素, at: インデックス) ... インデックス位置に挿入
 - remove(at: インデックス) ... 要素を削除
 - removeLast() ... 最後の要素を削除
 - removeFirst() ... 最初の要素を削除



配列の走査（総和と平均）

- 総和と平均

```
var sum = 0;  
for n in alist {  
    sum += n  
}
```

```
let average = Double( sum )/ alist.count
```



最大値・最小値

- 最大値・最小値

```
var (maxi, mini) = (0, 0)
for (i, ele) in alist.enumerated() {
    if alist[ maxi ] < alist[ i ] { maxi = i }
    if alist[ mini ] > alist[ i ] {mini = i } // miniは不等号が逆
}
```

maxi →最大値のある要素のインデックス

alist[maxi] →最大値



検索とカウント

- 特定の値の場所を検索、何個あるか

```
var target = Int( readLine( "見つけ出す値: " ) ) !
```

```
var alist = [ 5, 2, 565, 222, 111, 344, 22, 99, 348, 22, 2 ]
```

```
var (index, count) = (0, 0)
```

```
for (i, n) in alist.enumerated() { if n == target { count+=1; index = i } }
```

```
if count > 0 {
```

```
    print( target, " は最後に ", index,
```

```
    " で見つかりました ", count, " 個あります ") }
```

```
else {
```

```
    print( target, " はありません。 ") }
```



頻度表

- 誕生月の頻度を求める

```
let birth = [ 1, 3, 5, 2, 8, 11, 4, ... , 6, 7, 12, 4, 2 ]  
var month = Array( repeating: 0, count: 13 )
```

```
for bm in birth {  
    month[ bm ] += 1  
}  
for m in 1...12 {  
    print( m, "月の誕生日の人は", month[ m ], "人いました" )  
}
```



配列の並べ替え

- ランダムにシャッフルする

```
var a = Array( 1...100 )
```

```
for _ in a.count * 3 {  
    let index1 = Int.random( 0..    let index2 = Int.random( 0..    (a[ index1 ], a[ index2 ]) = (a[ index2 ], a[ index1 ])  
}
```



配列のソート

- ソート・ソーティング・整列...順番に並び替えること
- 直接選択法
 - 小さいものを選び、入れ替えをする

```
for i in 0..<a.count-1 {  
  var min = i  
  for j in i+1..<a.count {  
    if ( a[ min ] > a[ j ] ) { min = j }  
  }  
  (a[ i ], a[ min ]) = (a[ min ], a[ i ])  
}
```

直接選択法でのソーティング

直接選択法のソーティングの様子：

19 64 3 30 10 39 20 53

↑ i ↑ min

3 64 19 30 10 39 20 53

↑ i ↑ min

3 10 19 30 64 39 20 53

↑ i, min

3 10 19 30 64 39 20 53

↑ i ↑ min

3 10 19 20 64 39 30 53

↑ i ↑ min

3 10 19 20 30 39 64 53

↑ i, min

3 10 19 20 30 39 64 53

↑ i ↑ min

直接挿入ソートの例

直接挿入法のソーティングの様子：

19 64 3 30 10 39 20 53

↑target

19→64 3 30 10 39 20 53

↑target

3 19 64 30 10 39 20 53

↑target

3 19→30→64 10 39 20 53

↑target

3 10 19 30 64 39 20 53

↑target

3 10 19 30→38→64 20 53

↑target

3 10 19 20 30 38 64 53

↑target



直接插入法（非再帰版）

```
func sisort(_ alist: [Int] ) {  
    for i in 1..<alist.count {  
        let target = alist[ i ]  
        var j = i-1  
        while j >= 0 {  
            if target < alist[ j ] { alist[ j+1 ] = alist[ j ] }  
            else { break }  
            j -= 1  
        }  
        alist[ j+1 ] = target  
    }  
}
```



配列の最大値を再帰で

- 配列のサイズが1個だったら、
 - その要素が最大値であるとして返す
- 配列のサイズが2個以上だったら、
 - サイズを1つ減らして、最大値を求める
 - 求められた最大値と、最後の要素の値を比較して、大きい方を返す



配列のソートを再帰で

- 配列のサイズが1 だったら、
 - ソートされているものとして終わる
- 配列のサイズが2 以上だったら、
 - 最後の1 つ前までをソートする
 - 最後の要素を、順番的に良い場所に挿入する（この部分は、直接挿入ソートと同じ）



直接插入法（再帰版）

```
func recsort(_ alist: [Int], _ n: Int = alist.count-1 ) {  
    if n <= 1 { return }  
    recsort( alist, n-1 )  
    let target = alist[ n ]  
    var j=n-1  
    while j >= 0 {  
        if target < alist[ j ] { alist[ j+1 ] = alist[ j ] }  
        else { break }  
        j -= 1  
    }  
    alist[ j+1 ] = target  
}
```



配列での範囲指定

- 配列[最初...終わり]
 - 部分的な配列が生成される
 - 例：alist[4...7]
- 配列[...終わり]
 - 最初の要素の位置は、0と仮定される
 - 例：alist[...7]
- 配列[最初...]
 - 指定された最初の位置から、最後までになる
 - 例：alist[4...]
- 配列[最初..<終わりの次]
 - 部分的な配列が生成される
 - 例：alist[4..<7]



範囲指定をした代入

- 範囲指定をしたサブリストに、代入をすることができる
- `alist[ 1...3 ] = [ 3, 4, 5 ]`
- 範囲指定したサブリストと、代入するリストの個数があってもなくても可能（自動的に拡張される）
- `alist[ 1...3 ] = [ 2, 4 ]`

配列のコピー

- 配列を別の配列に代入すると、基本的にはコピーが作られる
- 例：

```
var numbers = [1, 2, 3, 4, 5]
var numbersCopy = numbers // 要素はすべてコピーされる

numbers[0] = 100 // 元の配列の要素を変更

print(numbers) // "[100, 2, 3, 4, 5]"が表示される

print(numbersCopy) // Prints "[1, 2, 3, 4, 5]"が表示される
```

- 配列を関数に渡すときもコピーされるので、配列の内容を変えたい場合は、inout指定と、実引数の時は&を付ける

```
func shuffle(_ a: inout [ String ] ) { } // シャッフルする関数

shuffle( &array )
```



配列の要素を探索する関数

- 配列.contains(探索値) ... 探索値が含まれていればtrue
- 配列.contains(where: 関数またはクロージャ) ... 満たすものがあればtrue
- 配列.allSatisfy(関数またはクロージャ) ... すべて満たせばtrue
- 配列.first(where: 関数またはクロージャ)? ... 満たす最初の要素を返す
- 配列.firstIndex(of: 探索値)? ... その探索値を満たす最初の要素のインデックス
- 配列.fristIndex(where: 関数またはクロージャ)? ... 満たす最初の要素のインデックス
- 配列.last(where: 関数またはクロージャ)? ... 満たす最後の要素を返す
- 配列.lastIndex(of: 探索値)? ... その探索値を満たす最後の要素のインデックス
- 配列.lastIndex(where: 関数またはクロージャ)? ... 満たす最後の要素のインデックス
- 上記の「関数またはクロージャ」は、要素の型 -> 論理値を満たすもの
- ?がついているものは、オプションで、なければnilが返される



配列の最大値・最小値を返す関数、配列のソート

- 配列.max() ... 最大値を返す
- 配列.min() ... 最小値を返す
- 配列.max(by: 関数またはクロージャ)...適用して、最大値を求める（オプションル）
- 配列.min(by: 関数またはクロージャ)...適用して、最小値を求める（オプションル）
- 配列.sort()...その配列自体をソートする
- 配列.sorted()...ソートされた配列を返す
- 配列.sort(by: 関数またはクロージャ)...その配列自体をソートする
- 配列.sorted(by: 関数またはクロージャ)...ソートされた配列を返す
- 上記の関数またはクロージャの型は、 $\langle T, T \rangle \rightarrow \text{Bool}$ 2つの要素を比較するもの



配列の追加・削除・検索

- 配列.append(value)...最後に要素を追加
- 配列.remove(at:)...該当する値の要素を削除
- 配列.removeSubrange(範囲)...その範囲の要素を削除
- 配列.insert(i, at: value)...i番目に、その値をもつ要素を挿入
- 配列.index(value)...その値を持つ要素のインデックスを返す
- 配列.extend(配列)...配列の結合 (+演算と同じ)
- 配列.copy()...浅いコピーで、コピーの配列を作る
- 配列.clear()...空配列にする



ソート以外の配列の要素の入れ替え

- 配列.reverse()...要素の順番を反転させる
 - 配列.reversed()...要素の順番を反転させた配列を返す
 - 配列.shuffle()...要素の順番をシャッフルする
 - 配列.shuffled()...要素の順番がシャッフルされた配列を返す
 - 配列.partition(by: 関数またはクロージャ)...指定条件を満たす最初の要素のインデックスを返す
 - 配列.swapAt(インデックス, インデックス)...2つの要素を入れ替える
-
- 上記の関数またはクロージャは、1つの引数を取り、論理値を返すもの

‣ 例：

```
var numbers = [30, 10, 20, 30, 30, 60, 10]
let p = numbers.partition(by: { $0 > 30 })
let (first, second) = numbers[..<p], numbers[p...]
// first == [30, 10, 20, 30, 30], second == [60, 40]
```



配列を使った高階関数

- 配列.filter(関数またはクロージャ)
 - 関数は、引数を 1 つ必要とし、論理値を返すもの
 - 返された論理値がtrueの要素だけを持つ配列が生成される
- 配列.map(関数またはクロージャ)
 - 関数は、引数を 1 つ必要とし、何らかの値を返すもの
 - 返された値から構成される配列が生成される
 - 配列が複数ある場合は、関数は、その個数分だけの引数を必要とする
- 配列.reduce(最初の値, 関数またはクロージャ)
 - 関数は、引数を 2 つ必要とし、何らかの値を返すもの（1 つの引数に、演算結果が保存される形になる）
 - 配列の各要素に引数を適用された値が返される



配列を利用した高階関数（その他）

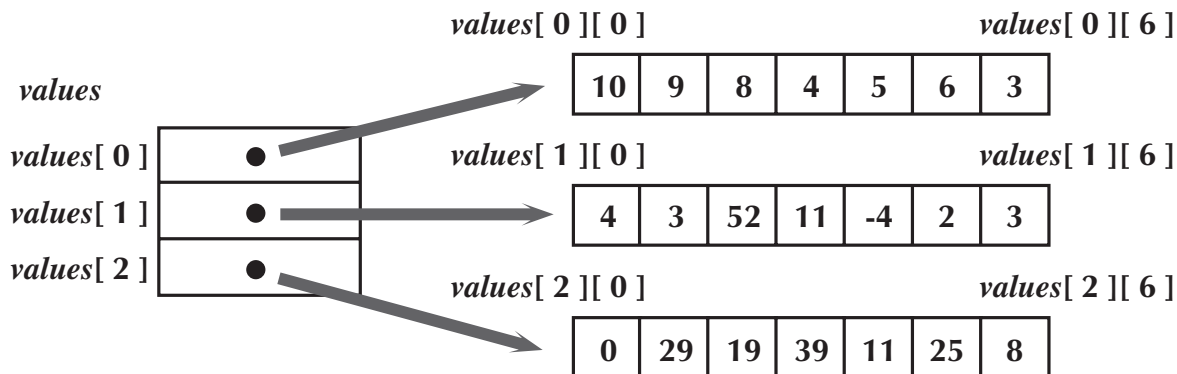
- 配列.forEach(関数またはクロージャ)
 - for文で繰り返すのと同じことを行なう
 - 例：

```
var alist = [ 1, 2, 3, 4, 5 ]
```

```
alist.forEach( { n in print( n ) } )
```


2次元配列

- パズルなどのゲームやデータの統計を行なうには、必要





2次元配列の初期値代入

- 初期値代入の例

```
let values = [  
    [ 10, 9, 8, 4, 5, 6, 3 ],  
    [ 4, 3, 52, 11, -4, 2, 3 ],  
    [ 0, 29, 19, 39, 11, 25, 8 ]  
]
```

- 長さが違う 2 次元配列の例

```
let values = [ [ 10, 20 ], [ 7 ], [ 47, 27, 18 ], [ 8, 2 ] ]
```



2次元配列の宣言

- 空の 2 次元配列の例

```
var values: [[ 要素の型 ]] = [[]]
```

- 2 次元配列

```
var values = Array( repeating: Array( repeating: 要素の初期値, count: 2次元目の個数 ), count: 1次元目の個数 )
```

2次元配列の長さ

- 例題の配列
 - `matrix=Array( repeating: Array( repeating: 0, count: 5 ), count: 10 )`
- 配列名.count ... 1次元目の配列の長さが求まる
 - `matrix.count ⇒ 10`
- 配列名[インデックス].count ... 2次元目の配列の長さ
 - `matrix[ 2 ].count ⇒ 5`
- 2次元目の長さが異なるような場合にも対処する必要



繰返して2次元配列の要素を参照

- C++/Java/C#的な記述の仕方（代入するときなど）

```
for i in 0..<matrix.count {  
  for j in 0..<matrix[ i ].count {  
    matrix[ i ][ j ] = i * j  
  }  
}
```

- Python的な記述の仕方（参照するだけの場合）

```
for rows in matrix {  
  for cell in rows {  
    print( cell )  
  }  
}
```



1次元配列の2次元配列化

- 関数などはないので、自分で関数などを作成する必要がある
- 作成例：

```
func nesting<T>(_ alist: T, _ n: Int ) -> [ T ] {  
    var result: [ T ] = [ ]  
    for i in stride( from: 0, to: alist.count, by: n ) {  
        result.append( alist[ i..i+n ] )  
    }  
    return result  
}
```

2次元配列の1次元化

- 配列.flatMap(関数またはクロージャ)
 - mapを行なった結果のフラットな配列を返す
 - 例 :

```
let numbers = [1, 2, 3, 4]
```

```
let mapped = numbers.map { Array(repeating: $0, count: $0) }  
// [[1], [2, 2], [3, 3, 3], [4, 4, 4, 4]]
```

```
let flatMapped = numbers.flatMap { Array(repeating: $0,  
count: $0) }
```

```
// [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
```




2次元配列の走査

- max, min, sortedのキー関数を使って、比較対象を選ぶようにする
- 利用例：

```
let nestedlist = [ [7, 3], [4, 5], [3, 8], [2, 9] ]  
print( nestedlist.sorted( by: { $0[0] < $1[0] } ) )  
print( nestedlist.max( by: { $0[0] < $1[0] } )! )  
print( nestedlist.min( by: { $0[0] < $1[0] } )! )
```