swiftのデータ型

1. Class

参照型

```
Class Hoge {
   let a
   var b
}
```

2. Struct

値型

```
Struct Hoge {
   let a
   var b
}
```

3. Enum

値型

```
Enum Hoge {
   case a
   case b
}
```

swiftの基本的な組み込み型

1. Int型 → 整数値

2. String型 → 文字列型

3. Charcter → 文字型

4. Bool型 → 真偽値型

5. Float型 → 浮動小数点数型(32bit)

6. Double型 → 浮動小数点数型(64bit)

7. Array型 → 配列型

8. Dictionary<Key, Value>型 → 辞書

9. Range系の型 → 範囲

10. Stride型 → 一定の間隔が空いた範囲

11. Optional型、ImplicitryUnwrappedOptional型 → 値の存在

swiftの制御構文

- if文
  条件にが真になったコードブロックの処理が実行される。

```
if 条件 {
    処理
} else if 条件 {
    処理
} else {
    処理
}
```

- while文
  条件がtrueの限り実行しつづける。

```
while 条件 {
    処理
}
```

- repeate-while文
  必ず一回実行されるwhile文

```
repeate {
    処理
} while 条件
```

- for-in文
  swift3からC言語風な構文

```
for i = 0; i < 1; i++ {

}
```

が廃止されています。

swift3から

```
for 変数名 in 式 (where式) {
    処理
}
```

の形が採用されています。
例)

```
// 0 ~ 10までの値のうち偶数のみを出力する。
for i in 0..<10 where i % 2 = 0 {
    print(i)
}

// hogeを三回表示
// 変数を使わない時はプレースホルダで省略
for _ in 0...2 {
    print("hoge")
}
```

- switch文
  条件にマッチした処理を実行します。swiftではデフォルトでC言語fallthroughしないのでbreakは不要。

```
switch 変数名 {
case  パターン1:
    処理
case  パターン2:
    処理
    fallthrough; // fallthroughを行いたい場合、明示的に記述する。
case  パターン3:
    処理
:
:
case  パターンn:
    処理
default:
    処理
}
```

swiftの関数定義

```
func 関数名(引数: 型) (-> 戻り値) {
    処理
}
```

swiftのアクセス修飾子

1. internal

同一のモジュール内からアクセス可能。デフォルトの修飾子。

2. private

クラスなどの宣言内からのみアクセス可能。

3. fileprivate

同一のファイル内でのみアクセス可能。

4. public

他のモジュールからもアクセス可能。overrideは可能だが、継承ができない。

5. open

publicのアクセス範囲に加えて、継承が可能。

制約の強さ(弱 → 強)
open → public → internal → fileprivate → private

クロージャ

無名関数を変数に代入する

```
let a: (引数: 型) -> (戻り値) =  { (引数: 型)  -> (戻り値) in
    本文
}
```

例)

```
let add: (Int, Int) -> Int = { (x: Int, y: Int) -> Int in
    return x + y
}
```

省略

```
var add: (Int, Int) -> Int
add = { ($0, $1) in
    return $0 + $1
}
```

さらに省略

```
var add: (Int, Int) -> Int
add = { $0 + $1 }
```

値の存在を示す型(Optional, ImplicitlyUnwrappedOptional)

ラップしている型の値が存在しているかどうかを示し、Optionalは値を取り出すためにアンラップ処理を行う必要があります。
ImplicitlyUnwrappedOptionalはアンラップの必要がなく、暗黙的にラップしている型に変換されますが、値がない場合は実行時エラーになります。

```
let a: Int? // Optional<Int>
let b: Int! // ImplicitlyUnwrappedOptional<Int>
```

Optionalのアンラップ

- オプショナルバインディング
  OptionalのWrapped型取り出し、定数に代入します。値がなかった場合はコードブロックの中がスルーされます。

```
let a: Int? = 0
if let b = a {
   print(b)
}
```

- オプショナルチェインニング
  Optionalの後ろに?をつけてWrapped型のメソッドを呼び出します。チェインの中でnilが返った場合は結果がnilになります。

```
let a: a String? = "Hello"
print(a?.length)
```

- guard文
  条件式が真出ないときelse節の中が実行されます。else節の中では現在のスコープから退出する処理を記述しなければなりません。

```
guard 条件式  else {
    処理
    スコープからの退出処理(returnなど)
}
```

- 強制アンラップ
  OptionalのWrapped型取り出します。値がなかった場合は実行時エラーになります。

```
let a: Int? = 0
let b: Int? = 1
a! + b!
```

- ??演算子
  OptionalのWrapped型を強制的に取り出します。値がなかった場合は右辺の値がセットされます。

```
let a: Int? = 0
print(a ?? 0)
```