Swiftについての説明資料

目次

太字はObjective-Cにはない概念です

1. 変数・定数定義のやりかた

Swiftで変数を定義するには、var文を使用します。

```
var name = "Tanaka Yuhei" // Swiftの場合

NSString *name = @"Tanaka Yuhei"; // Objective-Cの場合
```

この場合、右辺の型がString型(Objective-CでいうNSString)であるため、nameの型はString型となります。

Objective-Cのように、型を明示的に示して変数定義をすることもできます。

```
var name:String = "Tanaka Yuhei"
var hoge:String
print(hoge)  // エラー(値を使用する前にかならず値を代入しておく必要がある)
```

定数を定義したい場合はlet文を使用します。

```
let name = "Tanaka" // Swiftの場合
name = "Yuhei" // 定数なので再代入できない(エラー)

NSString *const name = "Tanaka"; // Objective-Cの場合
```

2. 配列・辞書定義のやりかた

Swiftになってからだいぶ簡単になったような気がします。

```
let array = ["a", "i", "u", "e", "o"]
let array2:[String] = ["a", "i", "u", "e", "o"] // 型を明示することもできる

let dict = ["a": 1, "b": 2, "c": 3]
let dict2:[String:Int] = ["a": 1, "b": 2, "c": 3] // 型を明示することもできる
```

Objective-Cだとこんな感じです。『@』が目立ちますが、これは参照型(NS~,UI~)に変換するために必要なものです。(1 -> int, @(1) -> NSNumber)。Objective-Cのリストや辞書は参照型しか渡せないので、@で参照型に変換する必要が出てくるわけです。

```
NSArray<NSString*> *array = @[@"a", @"i", @"u", @"e", @"o"];
NSDictionary<NSString*, NSNumber*> *dict = @{@"a":@(1), @"i":@(2),
@"u":@(3)};
```

3. if文とfor文

if文

Swiftでは条件式のカッコを省略することができるようになりました。

```
let v = 10

if (v == 10) {
    print("var == 10")
}
if v == 10 { // 条件式のかっこは省略ができる
    print("var == 10")
}
```

for文

よく使うもの(配列の要素を1つずつ取り出す etc...)だけ載せておきます

```
for i in 0...10 { // iが0から10まで変化
    print(i)
}
for i in 100..<110 { // iが100から109まで変化
    print(i)
}

let a = ["あ","い","う","え","お"]
for i in a { // 配列aの中身を1つずつ読み込み
    print(i) // iはString型
}
for i in 0..<a.count { // a.countはaの要素数
    print(a[i])
}
```

4. クラスを書こう

宣言しよう

Swiftの場合は以下のようにクラスを宣言します

```
class Car {
  // インスタンス変数を用意したりメソッドを実装したりする
}
```

Objective-Cの場合はこんな感じですね。Objective-Cの場合は全てのクラスのスーパクラスとしてNSObjectクラスがあるので、NSObjectクラスを継承することになります。

ヘッダファイル(.h)には他のクラスに公開するメソッドやプロパティを定義しておきます。実装は.mファイルに行います。

```
// ヘッダ部分(.hに記載)
@interface Car: NSObject // NSObjectクラスを継承する
// プロパティの定義や、外部に公開するメソッドの宣言をする
@end

// 実装部分(.mに記載)
@implementation Car
{
  // 外部には公開しないインスタンス変数等の定義をする
}
// メソッドの実装をする
@end
```

メソッドを書いてみよう

Swiftでメソッドを書く場合は以下のようにします。

```
func メソッド名(引数ラベル名:型) -> 戻り値の型 {}
```

```
class Car {
    func run(speed: Int) -> String { // 引数が1つで戻り値がある場合
        return "時速" + speed + "kmで走行中"
    }
    func say() { // 引数も戻り値が無いパターン
        print(self.run(100))
    }
}
```

Objective-Cの場合はこんな感じになります

```
// ヘッダ部分(.hに記載)
@interface Car: NSObject // NSObjectクラスを継承する
-(NSString *)run:(NSInteger)speed;
-(void)say;
@end

// 実装部分(.mに記載)
@implementation Car
-(NSString *)run:(NSInteger)speed {
    return [NSString stringWithFormat:@"%@%ld%@",@"時速",speed,@"kmで走行
中"];
}
-(void)say {
    NSLog(@"%@",[self run:100]);
}
@end
```

プロパティとインスタンス変数

Objective-Cでは、インスタンスの外からインスタンス変数に直接アクセスすることができません。

外部からインスタンス変数にアクセスするために、プロパティという仕組みを使います。プロパティを使用すると、インスタンス変数と、インスタンス変数に対するsetterとgetterが自動的に生成され、インスタンスの外からインスタンス変数にアクセスできるようになります。

インスタンス変数を外部に公開したくない場合は、プロパティを定義しなければよいです(Objective-Cの場合)。

```
// ヘッダ部分(.hに記載)
@interface Car: NSObject // NSObjectクラスを継承する
@property (nonatomic) NSInteger weight;
@end

// 実装部分(.mに記載)
@implementation Car
{
    int length; // インスタンス変数はsetterやgetterがないと外部からのアクセスができ
ません
    // インスタンス変数として_weightが自動的に生成されます
}
@end

// インスタンス生成
Car *ins = [[Car alloc] init];
ins.weight = 2000; // プロパティを定義するとインスタンス.プロパティ名でアクセスできる
ようになります
```

Swiftの場合、インスタンス変数とプロパティは区別されません。もし外部に公開したくないプロパティがあるときは、privateという**アクセス修飾子**があるので使用しましょう。ちなみに**アクセス修飾子はメソッドに対しても適用可能**です。

```
class Car {
    var weight:Int = 100 // Swiftでのプロパティは初期値を持つか初期化される必要があります
    private var length:Int = 1 // インスタンス内部でしかアクセスできません(非公開)
}

let ins = Car() // インスタンス生成
ins.weight = 1
```

ちなみに、Swiftのプロパティには2種類あります。Stored PropertyとComputed Propertyです。

Stored Propertyは`let hoge = 100`のように特定の値を保持しておくために使います。先程Swiftで使用したプロパティはStored Propertyになります。

Computed Propertyは呼ばれたときに都度計算して値を返すプロパティです。実際のコードを示してみます。

```
class Rect { // 四角形クラス
    private var width:Int
    private var height:Int
    var area:Int {
        get { // getterのみ定義
            return self.width * self.height
        }
    }

    init(width:Int, height:Int) {
        self.width = width
        self.height = height
    }
}

let square = Rect(width: 100, height: 40)
print(square.area) // 4000
square.area = 100 // getterしか実装されていないのでエラー(areaプロパティはReadOnlyになっている)
```

クラスを継承してみよう

Swiftでは、クラス定義の際に`class クラス名:継承したいクラス {}`とすることでクラスを継承したクラスを定義できます。

```
class Car {
    var weight:Int = 100 // Swiftでのプロパティは初期値を持つか初期化される必要があります
    func say() {
        print("ぼくは車です")
    }
}

class Benz:Car {
    override func say() { // メソッドをオーバーライドする場合はoverrideをつける
        super.say() // superをつけるとスーパクラスにアクセスできる
        print("ベンツ")
    }
}

let ins = Benz()
ins.say()
```

Objective-Cの場合は@interface部分に記載します。『プロパティとインスタンス変数』の項を参照してください。

5. プロトコルを使いこなそう

Objective-Cにおけるプロトコルは、クラスが実装するべきメソッド定義の集まりです。あくまでメソッド定義なので、実装部分は@implementation部分で行うことになります。

例えばUITableViewDataSourceなんかもプロトコルです。このプロトコルはUITableViewを表示するためのデータを渡すために必要なメソッド群が定義されています。

```
@protocol Hoge
-(void)sayHoge;
@end

@protocol Huga
-(void)sayHuga;
@end

@protocol Piyo <Hoge, Huga> // 複数個のProtocolを継承してProtocolを作ることもできる
-(void)sayPiyo;
@end

@interface Man: NSObject <Hoge, Huga> // 複数個のProtocolを実装することができる
@end

@implementation Man
- (void)sayHoge {
}
- (void)sayHuga {
}
- (void)sayPiyo {
}
@end
```

Swiftでのプロトコルは大幅に強化されています。新しくできるようになったことを一覧にしました。

- メソッドだけでなく**プロパティも定義できるようになった**

  めちゃめちゃ強力です。

```
protocol BMI {
    var height:Double {get set} // {get}か{get set}を指定できる
    var weight:Double {get set}
    func calcBMI() -> Double
}

class Man:BMI {
    var height: Double
    var weight: Double

    func calcBMI() -> Double {
        return self.weight / self.height / self.height
    }

    init(height:Double, weight:Double) {
        self.height = height
        self.weight = weight
    }
}

let man = Man(height: 1.92, weight: 158.0)
print(man.calcBMI())
```

- クラスだけでなく**構造体や列挙体にもプロトコルを適用できるようになった**

  Swiftでは構造体や列挙体が大幅にパワーアップしました。

  その結果、これまでクラスでやっていたような処理を構造体に任せるような機会も増えています(構造体については後述します)

  列挙体はエラー処理などでお世話になるかと思います。

```
protocol BMI {
    var height:Double {get set} // {get}か{get set}を指定できる
    var weight:Double {get set}
    func calcBMI() -> Double
}

struct Man:BMI { // 構造体！
    var height: Double
    var weight: Double

    func calcBMI() -> Double { // 構造体もメソッドを持つことができるようになった(後述します)
        return self.weight / self.height / self.height
    }

    init(height:Double, weight:Double) { // 構造体もイニシャライザを持つことができるようになった
        self.height = height
        self.weight = weight
    }
}

let man = Man(height: 1.92, weight: 158.0)
print(man.calcBMI())
```

6. 構造体を書こう

Objective-Cの頃は、構造体ができることに限りがありました。

メモリ管理の関係で、Objective-Cのオブジェクト(NSObjectを継承したクラスのインスタンス)をプロパティに格納することができませんでしたし、メソッドを持つこともできませんでした。イニシャライザを持つこともできなかったので値をセットするのが面倒でもありました。

```
typedef struct BMI { // 構造体定義(struct BMI に BMIという別名をつけている)
    double height;
    double weight;
} BMI;

BMI hakuho;
hakuho.height = 1.92;
hakuho.weight = 158.0;
```

アクセス修飾子をつけてアクセスを制限できることもできないので、いろいろと不安ですね。

Swiftの場合は**クラスとできることが殆ど変わりません**。クラスと構造体の違いが2点あるので示すとこんな感じです。

- 構造体は継承ができないが、クラスはクラスを継承することができる
- **構造体は値型、クラスは参照型**

とくに、構造体が値型になっていることがポイントです。値型と参照型については後述します。

```
protocol BMI {
    var height:Double {get set}
    var weight:Double {get set}
    func calcBMI() -> Double
}

struct StructMan:BMI { // 構造体
    var height: Double
    var weight: Double

    func calcBMI() -> Double {
        return self.weight / self.height / self.height
    }

    init(height:Double, weight:Double) {
        self.height = height
        self.weight = weight
    }
}

class ClassMan:BMI { // クラス
    var height: Double
    var weight: Double

    func calcBMI() -> Double {
        return self.weight / self.height / self.height
    }

    init(height:Double, weight:Double) {
        self.height = height
        self.weight = weight
    }
}

let man1 = StructMan(height: 1.92, weight: 158.0)
var man2 = man1; // man1の値がコピーされている。
// 構造体のプロパティをいじる場合はvarにする必要がある(値型なので)
man2.height = 1.93
man2.weight = 226.0
print(man1.calcBMI(), man2.calcBMI())

let man3 = ClassMan(height: 1.92, weight: 158.0)
let man4 = man3 // man4とman3は同じ参照先(参照型なので参照先を渡している)
man3.height = 1.93
man3.weight = 226.0
print(man4.calcBMI(), man4.calcBMI()) // 同じ値になる
```

7. 型とmutable

値型と参照型

Swiftには、大きく分けて2つの型があります。**値型**と**参照型**です。

	値型	**参照型**
入っているもの	実体	参照(アドレス)

値型には値そのものが、参照型には値が格納された場所の情報が格納されています。

変数を他の変数に代入したり、引数として渡す際には、値型の場合は値自体が、参照型の場合は格納された場所のアドレスがコピーして渡されることになります。

値型と参照型の区分けはSwiftだとこんな感じです。

値型	**参照型**
(Int)	関数
(Double)	クロージャ
(String)	**クラスのインスタンス**
(Set)	
Enum	
Tuple	
構造体	
(Array)	

Swiftの場合、実はIntやDouble、String、Array(配列)は構造体でできています。構造体でできているので値型です。それに対してクラスのインスタンスは参照型になっています。

Objective-Cだとこんな感じです。配列が参照型になっているところが大きな違いです。C言語由来のintなどが値型ですが、それ以外は基本的に参照型です。Objective-Cにおいては『*』をつけて変数定義を行う変数は参照型です。

値型	**参照型**
int	それ以外
float	
double	
char	
BOOL	
NSInteger	
CGFloat	

mutableとimmutable

変数の定義後に中身を変更できる変数をmutableな変数、できないものをimmutableな変数といいます。

できるだけ、変数はimmutableにしておいたほうがいいです。もし中身が変更可能(mutable)だと意図せぬ変更処理がなされる可能性がありますし、バグが発生した場合にどこで変更処理がなされたか探す必要があります。

では、どれがimmutableな変数で、どれがmutableな変数でしょうか。Swiftの場合は見分けるのが比較的簡単です。

mutable	**immutable**
varで定義した**値型**の変数	**let**で定義した**値型**の変数
それ以外のクラス	プロパティが**全部読み取り専用**になっているクラス

もし他のオブジェクトに対して値を渡したりする際は、副作用が心配です。

```
class ClassMan:BMI { // mutableなクラス
    var height: Double
    var weight: Double

    func calcBMI() -> Double {
        return self.weight / self.height / self.height
    }

    init(height:Double, weight:Double) {
        self.height = height
        self.weight = weight
    }
}

let man3 = ClassMan(height: 1.92, weight: 158.0)
// 固定されているのはインスタンスの参照であって、インスタンスの中身ではない
let man4 = man3 // man4とman3は同じ参照先(参照型なので参照先を渡している)
man3.height = 1.93
man3.weight = 226.0
print(man4.calcBMI(), man4.calcBMI()) // 同じ値になる(man3に与えた変更がman4に波及する)
```

上の例のように、もしmutableなクラスのインスタンスを他の変数にコピーしたとしても、参照先が同じであることが原因の予期せぬ副作用が起こる可能性があります。

なので、immutableなクラスのインスタンスを渡したいですね。例えばObjective-Cの場合はNSStringやNSArrayなどがimmutableです。

値型の変数は**値をコピーして渡す**ため、そもそも副作用が生じず安全です。Swiftでは構造体を多用するプログラミングをするのが望ましいように思います。

もし安全に他のメソッドと値をやり取りしたい場合は、**値型変数を使う**か、**immutableなクラスのインスタンスを使う**ようにしましょう。

8. Optional型

とは？

nilも代入することのできる型です。逆に言うと、SwiftではNon-Optionalな型にnilを代入できないようになっています。

Optional型にするには、Int?のように?をつけます。!をつけるOptional型もあるのですが、あとで説明します。

Objective-CにはOptionalという概念はありません。戻り値がnilになりうるかどうかはメソッドの実装を実際に見ないと分からないというわけです。

	Optional	**Non-Optional**
型の一例	String?, String!	String
nilが代入できるか	できる	できない

```
var a:Int = 2 // Non-OptionalなInt型
a = nil // エラー

var b:Int? // OptionalなInt型(初期値を定義しない場合nilになります)
b = 2 // OK
```

Optional型があって何が嬉しいのか

大きく分けて2つあります。

- 値にnilが含まれ得るかどうかが**型を見ただけで**分かる

  Optional型ならnilが含まれている可能性があります。

  逆にNon-Optionalならnilのことを考慮せずにロジックにだけ集中すればいいわけです。すなわちエンジニアが心配しなければいけないことが減っているわけです。

- nilチェックをうっかり忘れるようなことがなくなる

  Optional型の値はそのままの状態では使用できません。

  使用するためには**必ず**nilチェックをして値を取り出す必要があります。これを**unwrap**といいます。

  Optionalな型の変数に対してunwrapせずに中身にアクセスしようとすると**ビルドエラーになります**。

  Objective-Cの場合、nilチェックを忘れてしまっても**表面上は**動くことが多いです。nilに対してメソッド呼び出しをしてもクラッシュしないためです。

  nilチェックをしない仕様ならともかく、チェックし忘れていたときに気づくのが難しいというわけです。そのため問題が大きくなってからnilチェック忘れに気づくことになり、バグ対応がしんどくなっています。

その点、Swiftはnilチェックを強制させるのでうっかりミスをなくすことができ、問題が小さいうちに処理することが可能です。

Unwrap

Optional型に対して足し算することや、Optional型クラスのメソッドを実行することはそのままではできません。**Unwrap**という処理をしてはじめて、Non-Optionalな型と同じような処理をすることができます。

Unwrapする方法は何種類かあります

- Forced Unwrap

  文字通り強制的にUnwrapします。**中身がnilのときはクラッシュ**してしまうので、**中身がnilではないことが保証されているとき以外は使用するべきではありません。**

```
var a:Int?
let b = a! + 1 // a!でaをUnwrapしている(aはnilなのでクラッシュ)

var c:Int?
c = 10
let d = c! + 5 // dは15になる
```

- !のついたOptional型(Implicitly unwrapped optional)を使う

  !のついたOptional型は、その値を使用する際に自動的にUnwrapされます(!をつけなくてもForced Unwrapされているような感じです)

  使う段階でnilが入っていないことが保証されているようなシチュエーションで使うことになります。例えばIBOutlet等で使用される事が多いです。

```
var a:Int!
a = 100
let b = a + 1 // bは101になる
```

- guard letでUnwrap

```
var a:Int? = 100 // a は Optional
guard let aa = a else {return} // aがnullだったときreturnする
print(aa) // aa は Non-Optional

var b:Int? = 200
var c:Int? = 300

guard let d = b, e = c else {return} // b or cがnilだった場合returnする
print(d+e) // 500
```

- If letでUnwrap(Optional Binding)

```
var a:Int? = 100
if let a = a {
  print(a) // ここでの a はif文のブロック内スコープ( a はnon optional)
  // アンラップ後の変数名も同じにできる(シャドーイングといいます)
}
else {
  // aがnilだったときの処理を書く
}
print(a) // a は optional
```

- Nil Coalescing Operator

  `let c = a ?? b`のように使用します。aがnilでない場合はc=a、nilのときはc=bになります。

```
var str: String?
let a = str ?? "none" // strがnilのときは"none" -> a == "none"
```

Unwrapしないで(?)Optional型を扱う方法もあります。

- Optional Chaining

  Optional型をUnwrapしなくても、Optional型クラスのプロパティやメソッドにアクセスすることができます。

  そのためには、Optional型変数のあとに?をつけてあげます。もしOptional型変数にnilが入っていた場合はnilを返し、そうでない場合はプロパティやメソッドをOptional型として返してくれます。

  実際にコードを見たほうが早いと思うので、見てみましょう。

```
class Hoge {
  let huga = 10
}

var a:Hoge? // a == nil
print(a?.huga) // aはnilなのでnil

a = Hoge()
print(a?.huga) // Optional(10)

if let b = a?.fuga {
  print(b) // unwrapしたので10
}
```

9. クロージャ

とは？

関数自体をインスタンス化する機能です。

？？？となったかもしれないので例を見ていきましょう。

```
let say = {(name:String) -> Void in
    print(name)
}
say("tanaka") // tanaka
```

このサンプルコードだとありがたみが全くわからないですね。少し実践的な例を示してみたいと思います。

どこで使う？

主に、**処理自体**を他のメソッドに渡したいときに使います。

例えば、なにか処理が完了したときにやりたい処理(コールバック処理)を渡したりするときに使います。

```
func hello(name:String, completed:((String) -> Void)) {
    print("hello")
    // 処理完了
    completed(name) // クロージャで渡した処理を実行する
}

let say = {(name:String) -> Void in
    print(name)
}

hello(name:"yuhei", completed:say) // "hello" "yuhei"
```

書き方

`{(引数名:型) -> 戻り値の型(戻り値がない場合はVoid) in 処理}`

```
let test1 = {() -> Void in
    print("a")
}
test1() // "a"

let plus = {(x:Int, y:Int) -> Int in
    return x+y
}
print(plus(2, 3)) // 5
```

10. エラー処理

エラーを発生させる

throw文でエラーを発生させることができます。もしメソッド内でエラーを発生させる可能性がある場合はメソッドにthrowsキーワードを引数の直後に追加しましょう。

```
enum NameError: Error { // Errorプロトコルに対応させたNameError列挙体(詳しくは説明しません)
    case invalid(String)
    case empty(String)
}

func outputName(name: String) throws -> String {
    if name == "tanaka" {
        return name
    }
    else if name.isEmpty {
        throw NameError.empty("Empty Name")
    }
    throw NameError.invalid("You are not tanaka") // nameがtanaka以外のときはエラーを返す
}
```

エラーを処理する

- try,do-catchを使った方法

  do節内にエラーが起こりうる処理を記載します。もしエラーが発生した場合はcatch節に移動します。

  ```
  let name = ""

  do {
      let output = try outputName(name: name)
      print(output)
  }
  catch NameError.invalid {
      print("invalid name")
  }
  catch let error {
      print(error)
  }
  ```

- try?を使った方法

  try?を使用すると例外を無視することができます。この場合はdo-catch文は必要ありません。

  もし例外が発生した場合はnilを返します。

```
let name = ""
let output = try? outputName(name: name)
print(output) // nil
```

- try!を使った方法

  try!を使用してエラーを無視すると、エラーが発生した場合にクラッシュします。

  100%エラーが発生し得ないような状況で使うべきでしょう（そんなシチュエーションがあるかはわかりませんが）

```
let name = ""
let output = try! outputName(name: name) // クラッシュ
print(output)
```

11. Extension

ざっくり言うと**機能拡張**です。extensionでできることは主に2つです。

- 既存のクラス・構造体・列挙体に対する機能拡張

  実際に見てみるのが早いと思うので、コードを示します。

```
class ClassMan:BMI {
    var height: Double
    var weight: Double

    func calcBMI() -> Double {
        return self.weight / self.height / self.height
    }

    init(height:Double, weight:Double) {
        self.height = height
        self.weight = weight
    }
}

extension ClassMan { // ClassManクラスに対して機能拡張
    // var hoge = 100 <- Stored Propertyは格納できない
    func sayHello() {
        print("hello")
    }
}

class SubMan:ClassMan {
}

class SubMan:

let man3 = ClassMan(height: 1.92, weight: 158.0)
man3.sayHello() // "hello"

let man4 = SubMan(height: 1.92, weight: 158.0)
man4.sayHello() // サブクラスでもextensionで追加したメソッドが使用可能です
```

  Extensionでプロトコルに適合させることもできます。

```
extension クラス名: プロトコル名 {}
```

  この機能を活用すると、コードの可読性を高めることができます。

実際の例(おみくじアプリのViewController)を示します。

```
class SecondViewController: UIViewController {

    @IBOutlet var tableView:UITableView!
    var settingsArray:[[String:String]]!

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    override func viewWillAppear(_ animated: Bool) {
    }

    func getUserName() -> String {
    }
}

extension SecondViewController: UITableViewDataSource {
    // TableViewにデータを渡す処理
    func tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
    }
}

extension SecondViewController: UITableViewDelegate {
    // TableViewに何らかのアクションがなされたときの処理
    func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
    }
}
```

機能ごとにextensionが切られているので、Objective-C時代と比べると**可読性が向上**しました。

- プロトコル拡張(Protocol Extension)

  プロトコルに実装をもたせる機能です。ヤバそうな感じしませんか？

  プロトコルは実装を持たないものなはずなのに、どうしてそんなことをするのでしょうか?

  デフォルトの実装を持たせたいからです。プロトコルは非常に便利ですが、実装を持っていません。毎回毎回同じ実装をするような場合は非常に面倒なことになります。そういったときにプロトコル拡張が役に立ちます。

```
protocol Car {
    var speed:Double {get set}
    var weight:Double {get set}
    func run();
}

extension Car { // Protocol ExtensionでCarプロトコルにデフォルト実装を定義
    func run() {
        print("running")
    }
}

struct VolksWagen:Car {
    var speed: Double
    var weight: Double
    init(speed:Double, weight:Double) {
        self.speed = speed
        self.weight = weight
    }
}

struct Benz:Car {
    var speed: Double
    var weight: Double
    func run() { // もしクラスにメソッド定義をしている場合はこっちが優先になる
        print("benz")
    }
    init(speed:Double, weight:Double) {
        self.speed = speed
        self.weight = weight
    }
}

let wagen = VolksWagen(speed: 80, weight: 1000)
wagen.run() // "running" -> デフォルト実装

let benz = Benz(speed: 100, weight: 1600)
benz.run() // "benz" -> クラス側の実装
```