Swift 入門テキスト

プログラミング教室 KOTONOHA

2019 年 10 月 4 日

目次

第 1 章　超速 Swift プログラミングツアー 2
1.1　はじめに 2
1.2　定数と変数 4
1.3　データ型 7
1.4　基本の演算子 9
1.5　関数 12
1.6　構造体 16
1.7　スコープ 27
1.8　練習問題 28

第 1 章

超速 Swift プログラミングツアー

1.1 はじめに

本章では、これよりプログラミングを学ぶにあたり基礎知識となる概念や基本的な文法を学びます。プログラミング言語の仕様は、その言語の中で閉じていることが多いため、初歩的な概念でもその理解には高度な知識が必要な場合が多々あります。よくできている解説本でも、解説しているところでまだ学習していない内容があったりする場合が少なくありません。このような解説本では学習時に本の後ろと前を行ったり来たりしなければなりません。プログラミングを学習するにあたり、このようなことを避けるため、最初に基本的な概念を本章で解説します。

1.1.1 本書の使い方

本書では例としてコードを記載しています。コードを写しながら学習を進めると理解しやすいでしょう。Mac をお持ちの方は Xcode をインストールいただき、Xcode の playground を起動してコードを書いてください。もし、Mac でなかったり Xcode がダウンロードできない場合は以下のサイトで取り組むことが可能です。

`http://online.swiftplayground.run/`

1.1.2 学習内容

本章での学習内容は以下の通りです。

1. プログラミングとは？
2. 定数と変数
3. データ型
4. 関数
5. 構造体
6. スコープ

1.1.3 プログラミングとは？

プログラミング

プログラミングは端的に言えば「コンピュータの動作指示命令を書くこと」です。例えば、

ソースコード 1.1　初めてのプログラミング

```
1  print(1+1)
```

と書くと、コンピュータは 1+1 を計算し、結果である 2 を画面に出力してくれます。このコードでは 1+1 を計算して画面に出力せよと書いています。計算部分が 1+1 で出力せよが print() です。たった 1 行のコードですが、これも立派なプログラミングです。このようにして、コンピュータへの命令と順序を決めてファイルに書くことをプログラミングと呼

びます。プログラミングにおける命令の書き方の決まりがプログラミング言語です。

プログラミングでできること

　では、プログラミングができるとどのようなことが嬉しいのでしょうか？プログラミングが使われているところについてパッと思いつくものを挙げてみましょう。

- スマホやパソコンのアプリ
- 企業や銀行などの決済システム
- 隕石や小惑星の軌道予測
- 天気予報
- 防犯カメラの画像解析

などなど、そのほかにもたくさんあります。現在プログラミングは至る所で使われ、私たちの生活に欠かせないものとなりました。どんなことでもそうですが、プログラミングの世界はアイディア次第で可能性は無限大です。世界の大企業 G(google)A(amazon)F(facebook)A(apple) はどれもプログラミングとの関わりが深い企業です。このことからもプログラミングが世界においてどれほど重要かをうかがい知ることができます。

1.2　定数と変数

プログラミングでは変数や定数といったものを使い、データを扱います。コンピュータは通常メモリに記録されたデータやそれを電気信号に変換したものしか扱うことができません。したがってプログラミングでは主にコンピュータへのデータの扱い方の命令をすることになります。そこで最初に登場するのが定数と変数です。これらはデータの一時的な保管庫です。もちろん私たちが全ての数を覚えておいて、複雑な処理や演算ができれば良いのですが、もしそれができたらプログラミングは必要ありませんね。それができないので、変数や定数という保管庫を用いて値を保持しておきます。実際に使い方をみてみましょう。下記のプログラムを作成してください。

ソースコード 1.2　定数の簡単な使い方

```
// スラッシュ二つ重ねたものはコメントアウトといって、無視されます。
// これはコメントを残すことで読んだ人が理解するための仕組みです。
// コーディングの際はこの//の部分は無視して進めてください。
// それでは早速コーディングしていきましょう！

// 定数x を宣言し、値 1 を保管庫に入れておく
let x = 1
// print()で画面に表示する
print(x)
```

コンソール画面に 1 という文字が表示されたと思います。上記のコード (1.2) では x という名前の定数を宣言し、その定数 x に値 1 を代入しています。print() は print 関数と呼ばれる機能の名称[*1]で、() の中身をコンソール画面に表示しなさいという命令です。つまり、print(x) は x の中身を表示しなさいという命令になります。今、定数 x には値「1」が代入されているので、定数 x の中身である値「1」がコンソールに表示されます。

定数や変数には、プログラミング言語側で予約されているキーワード以外で自由な名前を付与することができます。ソースコードが膨大な場合や複雑な場合は、似たような処理や意味で使用する定数や変数が数多く定義されます。変数や定数名はその名前が重複してはいけないため、複合語を用いて定義する時がしばしばあります。例えば「りんごの数」を意味する変数を定義する場合、これを英語に直して「number of apples」と変数に名前をつけたいですが、スペースで文が区切られてしまうため、この名前をつけることはできません。そこで、スペースを詰めて、単語の先頭を大文字にする命名規則が用いられます。今の場合、変数名は「numberOfApples」となります。この規則により、複合語を扱えるようになり、かつ読みやすさを維持できます。このように、先頭を小文字、それ以降の単語の頭文字を大文字にする命名規則を**キャメルケース**といいます。定数や変数の名付けのセンスはプログラミングにおいてとても重要です。今のうちから気にしておくようにしましょう。

1.2.1　定数

定数は一度中身を決めたら中身の値を変更することができないデータです。後から値を変更されたくないデータに用います。定数は let キーワードを用いて以下のように宣言します。

ソースコード 1.3　書式

```
// 定数の宣言方法
let 定数名 = 値
```

ソースコード (1.3) を見るとわかる通り、定数の宣言は「let」キーワードの後に定数の名前を入力し、=を入力します。その後ろに入力したい値を入力します。以下に、実際の定数宣言の例を示します。

[*1] 関数については後述します。今は print 関数でコンソールに出力が可能であるということをおさえておいてください。

ソースコード 1.4　定数の宣言方法の例

```
// 定数x を宣言し、値 1 を代入します。
let x = 1 // これは整数データとなります。

// 定数y を宣言し、値 3.14を代入します。
let y = 3.14 // これは少数データとなります。

// 定数z を宣言し、値"Hello, world!"を代入します。
let z = "Hello, world!" // これは文字列データとなります。
```

ここで、定数 z に代入した”Hello, world”のはスペースを表します。

1.2.2　変数

定数に続いて、変数の解説をします。変数は定数と異なり、宣言後も値を変更することができるデータです。変数は var キーワードを用いて宣言します。

ソースコード 1.5　書式

```
// 変数の宣言方法
var 変数名 = 値
```

以下に変数の宣言例と簡単な使用例を示します。

ソースコード 1.6　変数の宣言方法の例

```
// 変数a を宣言し、値 1 を代入します。
var a = 1 // これは整数データとなります。

// 変数b を宣言し、値 3.14を代入します。
var b = 3.14 // これは少数データとなります。

// 変数c を宣言し、値"Hello, world!"を代入します。
var c = "Hello, world!" // これは文字列データとなります。

// 以下のコードでコンソールに表示します。
print("最初の値")
print(a, b, c)

// 変数は値を変えることができます。
// すでに変数a, b, c は宣言済みなので var や let なしに書くことができます。
a = 20 // 変数a の値を 20 に変更しました。
b = 2.718 // 変数b の値を 2.718に変更しました。
c = "Hello, Swift!" // 変数c の値を"Hello, Swift!"に変更しました。

print("変更後の値")
print(a, b, c)

// どうですか？値が変わったことだと思います。

// ちなみに以下のコードはエラーとなります。
a = 3.14
// エラーメッセージは以下のような出力となります。
// cannot assign value of type 'Double' to type 'Int'
// これは整数データであるa には少数データを入れられないよというメッセージです。
```

```
// データには「型」という概念が存在し、正しいコードを書くためには
// 型について正しく扱う必要があります。
```

1.2.3 確認問題

1. 定数 hello を宣言してください。初期値は”Hello, world!” とします。
2. print 関数を用いて定数 hello の値をコンソールに表示してください。
3. 変数 greeting を宣言してください。初期値は”Nice to meet you.” とします。
4. print 関数を用いて変数 greeting の値をコンソールに表示してください。
5. 変数 greeting の値を”Have a nice day.” に変更してください。
6. print 関数を用いて変数 greeting の値をコンソールに表示してください。

1.2.4 解答例

```
let hello = "Hello, world!"
print(hello)
var greeting = "Nice to meet you."
print(greeting)
greeting = "Have a nice day."
print(greeting)
```

1.3　データ型

データには「型」と呼ばれる概念が存在し、データの性質ごとにグループ分けしたものを**データ型**といいます。データ型はデータを正しく扱うための仕組みで、不正な演算を防ぐ役割があります。例えば、文字と数字では足し算や掛け算できません。このような処理を防ぐために型が存在します。

以下に代表的な型を示します。

代表的なデータ型

Int　整数を扱うための型です。
Float　単精度の浮動小数を扱うための型です。
Double　倍精度の浮動小数を扱うための型です。
String　文字列を扱うための型です。文字列データはダブルクォート”で囲みます。例：”こんにちは”
Bool　論理値を扱うための型です。値は true か false となります。

実際に定数を宣言して、型のタイプを確認してみましょう。型のタイプの確認には type(of:) 関数を使います。

ソースコード 1.7　データ型を確認するプログラム

```
1 let intNum = 1 // Int 型
2 print(type(of: intNum))
3 let doubleNum = 1.0 // Double 型
4 print(type(of: doubleNum))
5 let str = "こんにちは" // String 型
6 print(type(of: str))
7 let flg = true // Bool 型
8 print(type(of: flg))
```

上記のソースコード 1.7 を実行するとコンソールにそれぞれのデータ型名が表示されたはずです。ここで確認したように変数や定数はそれぞれデータ型を持っています。

1.3.1　型推論

Swift のプログラミングでは、変数や定数の宣言時にデータ型の指定が原則として必要ですが、宣言と同時に値を代入する場合は型を明示しなくても、代入する値を Swift 側で判断して変数や定数にデータ型を自動設定します。これを**型推論** (type inference) といいます。この型推論のおかげで

```
1 let 定数名 = 値
2 var 変数名 = 値
```

という宣言が可能となります。では、データ型を指定して宣言する場合はどのようにしたらよいでしょうか。

1.3.2　データ型を明示して変数または定数を宣言する方法

宣言時点で代入する値が決まっていれば Swift の型推論によってデータ型を明言する必要はありませんでしたが、決まっていない場合はデータ型をあらかじめ明言しておく必要があります。

ソースコード 1.8　データ型を明示した定数および変数の宣言方法

```
1 // 以下の ()内はオプションです。
2 // 型を明示した定数の宣言
3 let 定数名: 型名 (= 初期値)
4
5 // 型を明示した変数の宣言
```

```
var 変数名: 型名 (= 初期値)
```

上記の書式通り、データ型を明示した宣言では let または var キーワードの後に定数名または変数名を入力し、その後ろに:（コロン）と型名を入力します。もし初期値がある場合は従来通り=記号と初期値を入力します。それでは、実際に型名を明示した宣言を行ってみましょう。以下のコード 1.28 を作成してください。

ソースコード 1.9　型名を明示した定数および変数を宣言するプログラム

```
// Int 型を明示して定数 num を宣言する
let num: Int = 7

// String 型を明示して変数 str を宣言する
var str: String = "Nice␣to␣meet␣you."

// Bool 型を明示して変数 flag を宣言する
var flag: Bool = true

// コンソールに表示してみる
// 型名を確認するにはtype(of:変数または定数名)を使います。
print("定数 num の値は", num, "です。")
print("定数 num の型は", type(of: num), "です。")
```

1.3.3　確認問題

これでデータ型の説明は以上です。確認問題で理解度を確認しましょう。

1. String 型を明示して変数 s1 を宣言してください。
2. s1 に値”This is String type variable.”を代入してください。
3. Int 型を明示して変数 i1 を宣言し、値 20 を代入してくださし。
4. Bool 型を明示して変数 f1 を宣言し、値 false を代入してください。
5. Float 型を明示して定数 pi を宣言し、値 3.14 を代入してください。
6. Float 型を明示して定数 radius を宣言し、値 2.1 を代入してください。

1.3.4　解答例

以下は解答例です。

```
var s1: String
s1 = "This␣is␣String␣type␣variable."
var i1: Int = 20
var f1: Bool = false
let pi: Float = 3.14
let radius: Float = 2.1
```

1.4　基本の演算子

プログラミングはデータを扱う方法を書くことです。したがって、データを変換したり、データ同士から新たなデータを生成したりします。こういった操作を**演算**といい、変換や生成の決まりを**演算子**[*2]、対象となるデータを**被演算子**[*3]といいます。ここで、簡単な足し算を考えてみましょう。

$$1 + 2 \tag{1.1}$$

この演算の式で、どれが演算子でどれが被演算子でしょうか。答えは、＋が演算子で、1 と 2 がデータ、つまり被演算子です。

実は、定数や変数の宣言の時に用いた「=(イコール)」も演算子です。「=」は代入演算子といい、変数や定数へ値を代入するためのものです。では、すでに学習した変数の宣言を確認してみましょう。

ソースコード 1.10　変数を宣言するプログラム

```
1 var variable = "This␣is␣a␣variable."
2 print(variable)
```

上記のソースコードにおいて、1 行目の変数宣言部分の変数 variable と文字列値”This is a variable.”が被演算子で=記号が演算子です。この場合、変数 variable に文字列値”This is a variable.”を代入しなさいという意味になります。ソースコードを実行してください。variable に”This is a variable.”という文字列が格納されていることが確認できます。こ

表記	説明
a = b	b の値を a に代入する。

表 1.1　代入演算子の説明。a と b は定数または変数である。

こで、代入演算子 (=) は表 1.1 で説明した通り、代入をする演算子であり、数学の=記号とは異なることである。数学の=記号は左辺と右辺が等しいことを表すが、プログラミングの=は左辺に右辺を代入するという操作です。左辺と右辺が等価であることを示す演算子は別に存在するため、注意が必要です。

次に Int 型、Float 型および Double 型の数データにおける四則演算の演算子について解説します。まずは簡単な例として、3 + 4 を行う簡単なプログラムを組んでみましょう。

ソースコード 1.11　3+4 を行うプログラム

```
1 // 定数three,four の宣言
2 let three = 3 //「=」は代入演算子 定数three に値 3 を代入するという演算
3 let four = 4
4
5 // 足し算の結果を定数sum に代入
6 let sum = three + four // +演算子で足し算を行う
7
8 // 結果を表示
9 print(sum) // 足し算の結果 (7)が表示される
```

数の四則演算にはそれぞれ『+』、『−』、『*』、『/』の記号を用います。表 1.2 にそれぞれの演算子の説明を記載しました。四則演算に慣れるためにソースコード 1.13 を作成しましょう。ソースコード 1.13 では Double 型の定数 q,p を宣言し、それらの間の四則演算をしています。演算の結果は変数 r に代入し、print 関数でコンソールに出力しています。文字列””の中では \ **（バックスラッシュ）定数または変数名**とすることで文字列の中に埋め込むことができます。

[*2] オペレータともいいます。
[*3] オペランドともいいます。

表記	説明
a + b	a と b の和。
a − b	a と b の差。
a ∗ b	a と b の積。
a / b	a と b の商。

表 1.2　Int、Float および Double 型における四則演算の演算子。a と b は定数または変数。

ソースコード 1.12　簡単な四則演算のプログラミング例

```
1  // 定数p,q を宣言する
2  let p = 5.0 // 小数データ
3  let q = 8.0
4  // 変数r を宣言する
5  var r = p + q // 足し算
6  print("足し算の結果は\(r)") // \変数名 で変数を文字列中に埋め込む
7
8  r = p - q // 引き算
9  print("引き算の結果は\(r)")
10
11 r = p * q // 掛け算
12 print("掛け算の結果は\(r)")
13
14 r = p / q // 割り算
15 print("割り算の結果は\(r)")
```

1.4.1　演算とデータ型に関する注意点

Swift は型に関して厳しいプログラミング言語なので、+、-、*、/の演算結果は演算対象のデータと同じ型になります。試しに Int 型のデータで割り切れない演算を行ってみましょう。

ソースコード 1.13　Int 型で割り切れない演算

```
1  let x: Int = 5
2  let y: Int = 2
3  print(x/y) // 2.5を期待するがInt 型なので小数点以下が無視される
```

上記のコードは、式としては 5/2 を行なっているプログラムです。答えとして 2.5 を期待しますが、Int 型同士の計算なので結果も Int 型となり小数点以下が無いので整数部分の 2 が答えとなります。意図しない答えにならないようにデータ型の選択には注意しましょう。

1.4.2　確認問題

これで基本の演算子の説明は以上です。確認問題で理解度を確認してみましょう。

1. 変数 width を宣言してください。初期値は 12.3 とします。
2. 変数 height を宣言してください。初期値は 45.6 とします。
3. 変数 space を宣言してください。初期値は底辺を width、高さを height とする長方形の面積とします。
4. print 関数で space の値をコンソールに用事してください。
5. 定数 pi を宣言してください。初期値は 3.14 とします。
6. 変数 radius を宣言してください。初期値は 7.89 とします。
7. 変数 space の値を変更してください。代入する値は半径を radius、円周率を pi とする円の面積とします。

8. print 関数で space の値をコンソールに表示してください。

1.4.3 解答例

```
var width = 12.3
var height = 45.6
var space = width * height
print(space)
let pi = 3.14
var radius = 7.89
space = radius * radius * 3.14
print(space)
```

1.5　関数

プログラミング言語には関数という仕組みがあります。関数とは様々な処理をひとまとまりにしたものです。関数はその書式が多岐にわたるため、ここでは代表的な書式を紹介します。

1.5.1　関数とは

プログラミングにおける関数は先述した通り、複数の処理をひとまとまりにして、その処理に名前をつけたものです。また、処理の結果を出力することができます (図 1.1)。関数は多くのプログラミング言語に共通する概念で、同じ処理を繰り返し使用する場面において著しくその利便性を発揮します。また、複数の処理をひとまとめにし、プログラムソースコードの読みやすさやメンテナンス性を高めることができます。

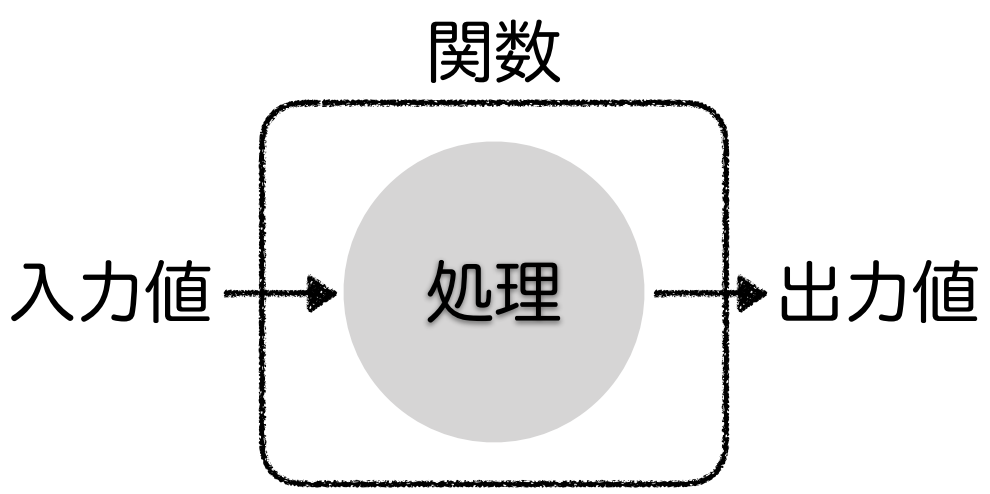

図 1.1　関数のイメージ。関数は入力値を受け取り、処理の後出力値をはき出す。

図 1.1 の通り、関数は入力値を出力値に変換するブラックボックスです。実際のプログラミングでは基本的に入力値を出力値に変換する過程を書いていきます。ただ、必ずしも入力や出力が必要ない場面もあり、処理だけをまとめたい場合もあります。その際は入力値や出力値が無い関数を作ることも可能です[*4]。それでは関数の基本的な書式を解説します。

1.5.2　関数の宣言方法

Swift では関数を以下のように宣言することが可能です。関数に引数が必要ない場合は () の中には何も書きません。

```
1 // 関数の宣言方法
2 func 関数名 ( 引数名: 型名, 引数名: 型名, ... ) -> 戻り値の型名 { // 引数がない場合は ()のなかに何も書
      かない
3     処理
4     return 戻り値
5 }
```

関数は **func** キーワードで宣言を行います。func キーワードの後に関数名を書き、その次に引数名とその型名を書いていきます。変換を表すために**アロー（矢印記号）**をハイフン (-) と大なり記号 (>) を組み合わせて書きます。アローの次に戻り値の型名を書いて、中括弧 { } で括ります。ここまでが関数の基本的な書式です。そして中括弧 { } の中に処理を

[*4] ここでは入力値や出力値が無い場合について解説しません。

書きます。最後に **return** キーワードを書き、その後ろに戻り値を書きます。以上の書式で関数を定義することができます。言葉での説明ではわかりづらいところがあるので、実際にプログラムを作成してみましょう。

ソースコード 1.14　関数の宣言例

```
// 関数の宣言
// 引数を１つ渡して２倍にする関数
func double(x: Int) -> Int{
        return 2 * x
}

// 引数を２つ渡して足し合わせる関数
func sum(x: Int, y: Int) -> Int{
        return x + y
}

// 名前を入力すると挨拶文を返す関数
func greet(name: String) -> String{
        var phrase = "Hello, " + name
        phrase = phrase + ".!"
        return phrase
}
```

ソースコード 1.14 では 3 つの関数を定義しました。次は関数の実行方法を解説します。

1.5.3　関数の実行方法

関数の実行は以下の書式で行います。

```
// 関数の実行方法
関数名（引数名: 値, 引数名: 値,...）
```

宣言した関数名と引数名と引数の値を書き下すことで関数を実行します。関数を実行すると関数自体が戻り値のように振る舞います。なので、関数の処理結果である戻り値を他のデータに渡しておきたい場合は次のようにします。

```
// 関数の実行方法
var 変数名 = 関数名（引数名: 値, 引数名: 値,...）
```

このようにすると実行した関数の戻り値が宣言した変数に格納されます。実際に先ほど宣言した関数を使って確かめてみましょう。先ほどのソースコードに以下のソースコード 1.15 の「ここから追加」コメント以下を追記してください。

ソースコード 1.15　関数の実行例

```
// 関数の宣言
// 引数を１つ渡して２倍にする関数
func double(x: Int) -> Int{
        return 2 * x
}

// 引数を２つ渡して足し合わせる関数
func sum(x: Int, y: Int) -> Int{
        return x + y
}

// 名前を入力すると挨拶文を返す関数
func greet(name: String) -> String{
```

```
        var phrase = "Hello, " + name
        phrase = phrase + ".!"
        return phrase
}

// #########ここから追加#########
// 上でdouble(x:), sum(x:,y:), greet(name:)を定義しているので、
// 関数名を入力することで関数を使うことができます。
print(double(x: 10)) // コンソールに 20と表示されます。

var z = sum(x: 100, y: 200)
print(z) // コンソールに 100と 200の和である 300が表示されます。

print( greet(name: "Suzuki")) // <- name:の後ろ"Suzuki"をあなたの名前に置き換えてください。
```

1.5.4 確認問題

以上が関数の宣言方法と実行方法になります。以下の確認問題で関数についての理解を深めましょう。

1. Int 型の値を返す関数 sum を定義してください。関数 sum は以下の仕様を満たします。
   (a) 関数 sum は二つの引数 a と b を持ちます。
   (b) 引数は全て Int 型です。
   (c) 関数 sum の戻り値は a と b の和です。
2. Int 型の値を返す関数 times を定義してください。関数 times は以下の仕様を満たします。
   (a) 関数 times は二つの引数 a と b を持ちます。
   (b) 引数は全て Int 型です。
   (c) 関数 times の戻り値は a と b の積です。
3. Int 型の値を返す関数 diff を定義してください。関数 diff は以下の仕様を満たします。
   (a) 関数 diff は二つの引数 a と b を持ちます。
   (b) 引数は全て Int 型です。
   (c) 関数 diff の戻り値は a と b の差です。
4. Int 型の値を返す関数 div を定義してください。関数 div は以下の仕様を満たします。
   (a) 関数 div は二つの引数 a と b を持ちます。
   (b) 引数は全て Int 型です。
   (c) 関数 div の戻り値は a と b の商です。
5. Int 型の変数 num1 を定義して値 20 を代入してください。
6. Int 型の変数 num2 を定義して値 4 を代入してください。
7. Int 型の変数 result を定義してください。
8. result に関数 sum の実行結果を代入してください。ただし、sum の引数 a は num1、引数 b は num2 とします。
9. result の値を print 関数でコンソールに表示してください。
10. result に関数 times の実行結果を代入してください。ただし、times の引数 a は num1、引数 b は num2 とします。
11. result の値を print 関数でコンソールに表示してください。
12. result に関数 diff の実行結果を代入してください。ただし、diff の引数 a は num1、引数 b は num2 とします。
13. result の値を print 関数でコンソールに表示してください。
14. result に関数 div の実行結果を代入してください。ただし、div の引数 a は num1、引数 b は num2 とします。
15. result の値を print 関数でコンソールに表示してください。

1.5.5 解答例

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

func times(a: Int, b: Int) -> Int{
        return a * b
}

func diff(a: Int, b: Int) -> Int{
        return a - b
}

func div(a: Int, b: Int) -> Int{
        return a / b
}

var num1: Int = 20
var num2: Int = 4
var result: Int

result = sum(a: num1, b: num2)
print(result)

result = times(a: num1, b: num2)
print(result)

result = diff(a: num1, b: num2)
print(result)

result = div(a: num1, b: num2)
print(result)
```

1.6　構造体

いよいよ本章で集大成となる構造体に挑みます。変数や定数および関数をたくさん宣言してくると、煩雑になり、コードがとても読みにくくなります。これらの定数、変数および関数をまとめる機能がプログラミング言語には存在し、構造体やクラスなどとして定義されています。ここでは構造体にのみに絞って解説します。構造体では今まで学習してきた定数、変数から関数までの知識が頻出します。まだマスターできていない方は冒頭に戻って学習し直しましょう。急いでいる方は前の章を参照しながら学習してください。

構造体のイメージ

構造体のイメージを図にすると図 1.2 のようになります。構造体は先述した通り、定数や変数などのデータや関数などの処理をまとめる機能です。

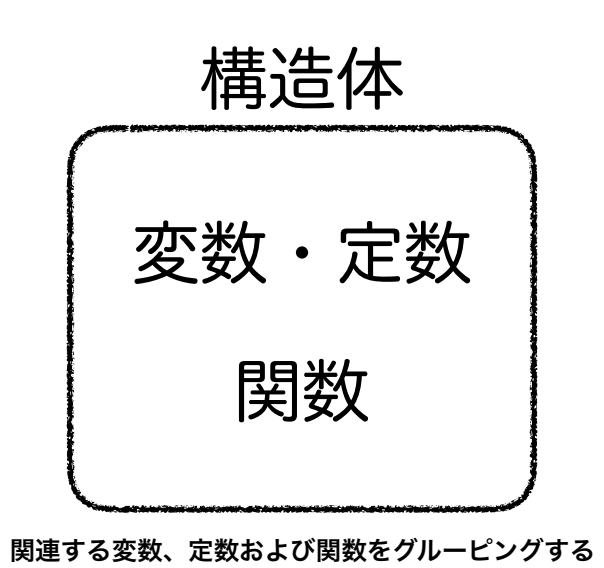

図 1.2　構造体のイメージ

もう少し具体的な例として、例えば果物のりんごをプログラミングを使ってうまく扱いたい時を考えてみます。普段買い物でりんごを買う時などに参照する情報を挙げてみましょう。生産地や生産者、品種、サイズや見た目といったところでしょうか。これらの情報は定数や変数となります。そして、りんご自身の情報から規格を決めたり、商品情報を出力したりできたら便利でしょう。これらの機能はりんご自身の情報を使って処理をするので関数となります。今までの部分をりんご構造体としてイメージを描くと図 1.3 のようになります。りんごに関連した有用な情報と処理をグルーピングすることが構造体の基本です。以上でみてきたように、構造体を用いると変数や定数、関数をグルーピングでき、すっきりとまとめることができるようになります。

構造体の構成要素

構造体の主な構成要素は以下の３つです。

構造体の構成要素

プロパティ　構造体の中で宣言された変数や定数のことです。
イニシャライザ　構造体のプロパティを初期化する特別なメソッドです。
メソッド　構造体の中で宣言された関数のことです。

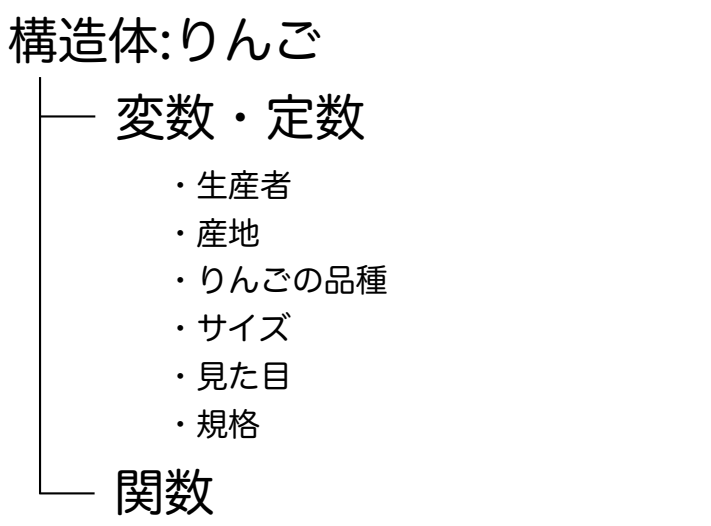

図 1.3　りんご構造体のイメージ

1.6.1　構造体の宣言方法

構造体は struct キーワードにより宣言することができます。構造体名の後ろの { } の中にプロパティ、イニシャライザおよびメソッドを宣言します。プロパティは通常の定数や変数と同様の定義方法です。また、メソッドも通常の関数と同様の定義方法です。イニシャライザはプロパティを初期化するためのメソッドであり、戻り値を持たない関数です。

ソースコード 1.16　構造体の宣言方法

```
1 // 構造体の宣言方法
2 struct 構造体名{
3     プロパティ宣言
4     イニシャライザ宣言
5     メソッド宣言
6 }
```

1.6.2　プロパティ

実際にプロパティを構造体中に宣言してみましょう。以下のソースコードでは Human という構造体を作成し、名前 (name)、年齢 (age)、体重 (weight) データを構造体でグルーピングします。

ソースコード 1.17　Human 構造体の定義例

```
1 // プロパティの宣言方法
2 struct Human{
3     // プロパティの宣言
4     var name = "Tom"
5     var age = 19
6     var weight = 50
7 }
```

この構造体の使い方ですが、構造体を利用する時には (インスタンス化) という操作を行います。インスタンス化とは、構造体を実際のデータとして取り扱えるようにする、構造体の**実体化**を行うことです。すでに定義した Human 構造体のインスタンス化を行います。今定義している Human 構造体は Human() でインスタンス化できます。この Human() を Human 構造体の**インスタンス**といいます。普通、このインスタンスはプログラムの中で再利用するため、定数や変数を

新たに定義して代入します。構造体は複数のデータをまとめるものですから、構造体に定義したプロパティにアクセスできなければ意味がありません。インスタンス化した構造体のプロパティには以下のようにアクセスします。

ソースコード 1.18　プロパティへのアクセス方法

```
1 インスタンス.プロパティ名
```

それでは、構造体のインスタンス化とプロパティへのアクセスを先ほど定義した Human 構造体を使ってみていきます。以下のソースコードを作成してください。

ソースコード 1.19　Human 構造体定義とプロパティへのアクセス

```
1  // プロパティの宣言方法
2  struct Human{
3      // プロパティの宣言
4      var name = "Tom"
5      var age = 19
6      var weight = 50
7  }
8
9  // #####以下を追加#####
10
11 var h = Human() // Human 構造体のインスタンス化 インスタンスは変数h へ代入される。
12
13 var n = h.name // name プロパティへのアクセス 変数n に代入する。
14 var a = h.age // age プロパティへのアクセス 変数a に代入する。
15 var w = h.weight // weight プロパティへのアクセス 変数w に代入する。
16
17 //コンソールに表示する。
18 print(n, a, w)
19
20 //プロパティは他の変数に代入しなくても使える
21 print(h.name, h.age, h.weight)
```

初期値のない構造体

先ほど例にみた Human 構造体のプロパティは全て初期値を付与していました。しかし、最初から値を決められない場合やデフォルト値を持ちたくない場合は、プロパティが初期値を持たない構造体を定義することができます。プロパティが初期値を持たない構造体のプロパティは型推論がはたらかないので型名を明示した定数や変数の定義方法を用います。例として、縦横高さを管理する構造体 Box を定義します。以下のソースコードを作成してください。

ソースコード 1.20　初期値がない構造体の定義

```
1  // 構造体Box の定義
2  struct Box{
3          var length: Float
4          var width: Float
5          var height: Float
6  }
7
8  // 構造体Box のインスタンス化
9  var b = Box(length: 12.0, width: 8.0, height: 5.0)
10
11 // プロパティへのアクセス print 関数でコンソールに表示
12 print(b.length)
```

```
print(b.width)
print(b.height)
```

構造体はインスタンス化が完了するまでに全てのプロパティの値が確定していなければなりません。初期値が無い構造体ではそのインスタンス化時にプロパティの初期値を決めなければなりません。なので、ソースコード 1.20 にある通り、Box 構造体のインスタンス化では Human 構造体のように Box() ではなく、() の中に「**プロパティ名:値**」の組を連ねてプロパティの初期値を指定してインスタンス化を行います。

構造体を定義する時に、初期化するプロパティしないプロパティが混ざっていても構いません。このような場合には定義時に初期化していないプロパティのみを初期化するようにインスタンス化をすれば問題ありません。実は、プロパティを全て初期化している構造体でも上記の Box 構造体のようなインスタンス化が可能です。その場合はインスタンス化時に指定したパラメータでプロパティの値が上書きされます。

1.6.3 イニシャライザ

イニシャライザは構造体のプロパティを初期化するための特別な関数（後に解説するメソッド）です。イニシャライザを定義していない構造体のインスタンス化でもイニシャライザは必ず実行されます。つまり、どんな構造体でもイニシャライザは実行されます。

デフォルトイニシャライザとメンバーワイズイニシャライザ

実は、先ほど構造体をインスタンス化した時の Human() でもイニシャライザが実行されています。全てのプロパティを初期化している構造体のインスタンス化『構造体名 ()』において実行されるイニシャライザは**デフォルトイニシャライザ**といいます。このデフォルトイニシャライザは単に、プロパティすべてにデフォルト値が設定された新しいインスタンスを生成します。

一方で、デフォルト値が設定されていない構造体では、() の中に「**プロパティ名:値**」の組を連ねてプロパティの初期値を指定してインスタンス化を行いましたが、このインスタンス化では**メンバーワイズイニシャライザ**が実行されています。メンバーワイズイニシャライザは、構造体インスタンスのプロパティを初期化するための簡略方法で、プロパティの初期値を、メンバーワイズイニシャライザに名前で指定して渡すことができます。

ここまでの内容を実例で確認しましょう。

ソースコード 1.21　デフォルトイニシャライザとメンバーワイズイニシャライザ

```
// 構造体Human の定義
struct Human{
    var name = "Tom"
    var age = 19
    var weight = 50
}

// Human をインスタンス化
var h = Human() // デフォルトイニシャライザが実行される。

// 構造体Box の定義
struct Box{
        var length: Float
        var width: Float
        var height: Float
}

// Box をインスタンス化
var b = Box(length: 12.0, width: 8.0, height: 5.0) // メンバーワイズイニシャライザが実行される。
```

ソースコード 1.21 では Human と Box を定義しています。Human では全てのプロパティにデフォルト値を設定しているため、そのままインスタンス化が可能です。なので、Human() でインスタンス化をすることができ、その際デフォルトイニシャライザが実行されます。Box では全てのプロパティにデフォルト値が設定されていないので、インスタンス化時にプロパティに値をセットする必要があります。したがって、インスタンス化は Box(length: 12.0, width: 8.0, height: 5.0) とする必要があり、メンバーワイズイニシャライザが実行されます。

イニシャライザ

ここから一般的なイニシャライザの説明をします。先に説明したようにイニシャライザは構造体のプロパティを初期化するための特別な関数（後に解説するメソッド）です。なので、書式は関数とほぼ同様となります。関数の書式との違いは func キーワードを使わないこと、関数名が init のみであること、戻り値がないことの 3 点です。具体的な書式をソースコード 1.22 に記しました。

ソースコード 1.22　イニシャライザの書式

```
1 // イニシャライザの書式
2 init( 引数名: 型名, 引数名: 型名,... ){
3     処理
4 }
```

イニシャライザはプロパティを初期化するための特別な関数なので、イニシャライザ中で全てのプロパティの値を決めている状態にしなければなりません。逆に、プロパティを全て決定できれば、イニシャライザの引数はどんなものでも構いません。また、イニシャライザは複数定義することができます。ただし、引数の数や引数の名前で、それぞれのイニシャライザを区別できなくてはなりません。以上がイニシャライザのルールです。

それでは具体例をみていきましょう。先ほど定義した Box を少し変えていきます。

ソースコード 1.23　Box によるイニシャライザの例

```
1  // 構造体Box の定義
2  struct Box{
3          var length: Float // たて
4          var width: Float // よこ
5          var height: Float // 高さ
6          var volume: Float // 体積
7
8          // イニシャライザその１
9          // たて x よこ x 高さ から体積を計算して代入するパタン
10         init(tate: Float, yoko: Float, takasa: Float){
11                 self.length = tate // self キーワードは Box 構造体自身を指す
12                 self.width = yoko
13                 self.height = takasa
14                 self.volume = tate * yoko * takasa
15         }
16
17         // イニシャライザその２
18         // たて、よこ、体積から高さを計算して代入するパタン
19         init(tate:Float, yoko:Float, taiseki: Float){
20                 self.volume = taiseki
21                 self.length = tate
22                 self.width = yoko
23                 self.height = taiseki / ( tate * yoko )
24         }
25
26 }
```

```
27
28 // Box をインスタンス化
29 var b = Box(tate: 1.0, yoko: 2.0, takasa: 3.0) // イニシャライザその１
30 var c = Box(tate: 1.0, yoko: 2.0, taiseki: 8.0) // イニシャライザその２
31
32 // 変数b と c のプロパティの確認をします。
33 print("b のプロパティ確認")
34 print(b.length)
35 print(b.width)
36 print(b.height)
37 print(b.volume)
38 print("c のプロパティ確認")
39 print(c.length)
40 print(c.width)
41 print(c.height)
42 print(c.volume)
```

ソースコード 1.23 は構造体 Box の定義と、インスタンス化をおこなったものです。Box のプロパティは縦 (length) 横 (width) 高さ (height) とその体積 (volume) を意味しています。この Box にはイニシャライザを２つ定義しています。１つは縦横高さを入力するもので、縦横高さから体積を計算できるので、イニシャライザの引数から体積を計算しています。デフォルトイニシャライザやメンバーワイズイニシャライザでは全てのプロパティの値を直接決めておく必要がありましたが、イニシャライザを使うと、あるロジックからプロパティの値を決めるということが可能になります。この点が、デフォルトイニシャライザやメンバーワイズイニシャライザとの決定的な違いです。

self キーワード

ソースコード 1.23 のイニシャライザの中に **self.** という記述があります。この self キーワードは構造体自身を指しています。つまり、今の場合構造体 Box 自身を指します。self は構造体自身を指すので、**self. プロパティ名**で Box に定義されたプロパティへアクセスすることができます。構造体やクラス、列挙型を定義する際には self キーワードは頻繁に使用しますので、今のうちに使えるようになりましょう。

ソースコード 1.24　構造体自身のプロパティにアクセスする方法

```
1 // プロパティへのアクセス方法
2 self.プロパティ名
```

1.6.4　メソッド

メソッドは構造体に定義された関数です。構造体に定義されたプロパティの変換をしたり、計算をする時に必要となる機能です。また、それに限らず、外部から入力されたデータについて処理をすることも可能です。メソッドの書式は関数と同様です (ソースコード 1.25)。

ソースコード 1.25　メソッドの書式

```
1 // メソッドの宣言方法
2 func メソッド名（ 引数名: 型名, 引数名: 型名,... ） -> 戻り値の型名 {
3     処理
4     return 戻り値
5 }
```

もし、構造体内で定義されたプロパティを変更したい場合は func キーワードの前に mutating キーワードを付与します (ソースコード 1.26)。

ソースコード 1.26　構造体自身のプロパティを変更する場合のメソッドの書式

```
// メソッドの宣言方法（構造体自身のプロパティの値を変更する場合）
mutating func メソッド名（ 引数名: 型名, 引数名: 型名,... ） -> 戻り値の型名 {
    処理
    return 戻り値
}
```

それでは Human 構造体にメソッドを追加していきます。ソースコード 1.27 を作成してください。

ソースコード 1.27　メソッドの例

```
// 構造体Human の定義
struct Human{
        // プロパティの定義
    var name: String
    var age: Int
    var weight: Float

    // イニシャライザ
        init(name:String,age:Int,weight:Float){
                self.name = name
                self.age = age
                self.weight = weight
        }

        // 挨拶文を作るメソッド
        func greet() -> String{
                return "Hello, my name is " + self.name + ". Nice to meet you."
        }

        // 年齢を修正するメソッド
        mutating modAge(age: Int) -> Int{ // 年齢プロパティを変更するのでmutating キーワードを書く
        self.age = age
                return age
        }

}

// Human をインスタンス化
var kumi = Human(name: "Kumi", age: 29, weight: 49.5)

// 年齢が本当は 19才だったので、年齢の修正
kumi.modAge(age: 19)
print(kumi.age) // 年齢を表示
print(h.greet()) // 挨拶文を表示
```

1.6.5　構造体は型である

ここまで学習してきた構造体ですが、実は構造体は型として機能します[*5]。今、構造体の中で別の構造体を利用する場面を考えてみましょう。学校のクラス（教室）の情報を管理するための構造体 ClassRoom を定義します。クラス（教室）はクラスの名前、生徒の人数、先生の情報を管理することにしましょう。ここで、先生の情報を構造体 Human として定

[*5] 型として機能するものとしては構造体の他にクラスと列挙型があります。

義してみましょう。今までの Human 構造体を以下のコード 1.28 にあるように変更します。構造体 ClassRoom はソースコード 1.28 のようになります。ここで問題です。プロパティ teacher には構造体 Human を使いたいですが、この時の型はどのようになるでしょうか。

ソースコード 1.28　構造体中のプロパティを別の構造体にする例

```
struct Human{
        var name: String // 氏名
        var age: Int // 年齢
        var sex: String // 性別  male(男性) or female(女性)

        init(name:String, age:Int, sex: String){
                self.name = name
                self.age = age
                self.sex = sex
        }
}

struct ClassRoom{ // 教室の情報を管理するClassRoom 構造体
        let name: String
        var num: Int
        var teacher: ???? // <- ????はどうなる？

        init( name: String, num: Int, teacher: ????){ // <- ????はどうなる？
                self.name = name
                self.num = num
                self.teacher = teacher
        }
}
```

実は、上記ソースコードの????は Human になります。

デフォルトの型とカスタムな型

先述した通り、構造体は型として機能します。つまり、構造体[*6]を使うことでよく使用する Int、String、Float、Bool などの型のように使用できる型を作ることができます。つまり、カスタマイズした型を作ることができます。実は、Swift のデフォルト型である Int や String も構造体です。なので、Int や String もそれぞれプロパティやメソッドを持っています。これらの型がプロパティやメソッドを持つことを確認するために、以下のソースコードを作成してください。

ソースコード 1.29　Int 型および String 型のプロパティとメソッド

```
var x: Int = -10
var y: String = "HELLO"

print(x)
print(y)

print(x.magnitude) // Int 型のプロパティ magnitude 値の大きさを表示する 大きさは 10
print(y.count) // String 型のプロパティ count 文字数を表示する 文字数は 5

x.negate() // Int 型のメソッド negate() 正負を入れ替える処理
y.lowercased() // String 型のメソッド lowercased() 全て小文字にする処理

```

[*6] クラスと列挙型も

```
print(x) // 10になっている
print(y) // hello になっている
```

Int 型のプロパティ magnitude はその値の大きさを表し、今 x は-10 なので、x.magnitude は 10 となります。String 型のプロパティ count は文字数を表し、今 y は HELLO の 5 文字なので y.count は 5 となります。Int 型のメソッド negate() は正負を入れ替える処理なので、x.negate() とすると x は-10 から 10 となります。String 型のメソッド lowercased() は文字列中の大文字を全て小文字にするという処理なので、y.lowercased() を実行すると y は”hello” となります。

以上で確認したように、デフォルトの型も構造体であり、同じようにプロパティとメソッドを持ちます。これからプログラミングを学習する中で、**a.b** という形が出てきたら a は構造体、クラスもしくは列挙型で b はプロパティになります。b() という形であればメソッドになります。これは非常に重要なことなので、覚えておきましょう。

それでは話を戻して、前節のソースコード 1.28 の正しいコードを以下に記します。ソースコード 1.30 では Human と ClassRoom の宣言の他、変数として定義し、各種値の確認まで行なっています。

ソースコード 1.30　構造体中のプロパティを別の構造体にする例

```
struct Human{
        var name: String // 氏名
        var age: Int // 年齢
        var sex: String // 性別  male(男性) or female(女性)

        init(name:String, age:Int, sex: String){
                self.name = name
                self.age = age
                self.sex = sex
        }
}

struct ClassRoom{ // 教室の情報を管理するClassRoom 構造体
        let name: String
        var num: Int
        var teacher: Human // <- teacher は Human 型

        init( name: String, num: Int, teacher: Human){ // teacher は Human 型
                self.name = name
                self.num = num
                self.teacher = teacher
        }
}

var teacherAlice = Human(name: "Alice", age: 29, sex: "female")
var classRoom = ClassRoom(name: "1-A", num: 30, teacher: teacherAlice)

print(classRoom.name)
print(classRoom.num)
print(classRoom.teacher.name) // <- teacher は Human 型なので teacher.name として先生の名前にアクセスする
print(classRoom.teacher.age)
print(classRoom.teacher.sex)
```

構造体が型として機能していることを理解できたと思います。色々な構造体を定義して、使ってみましょう。

1.6.6　確認問題

以下の確認問題で構造体についての理解を深めましょう。

1. 構造体 Algebra を宣言してください。Algebra は以下の仕様を満たします。
   (a) プロパティとして Float 型の変数 a,b を二つもつ
   (b) 引数を a,b とするイニシャライザをもち、引数 a はプロパティ a に、引数 b はプロパティ b に代入される
   (c) a と b の和を返すメソッド sum をもつ
   (d) a と b の差を返すメソッド diff をもつ
   (e) a と b の積を返すメソッド times をもつ
   (f) a と b の商を返すメソッド div をもつ
2. 定数 al を定義し、構造体 Algebra をインスタンスを代入してください。イニシャライザの引数 a,b の値は 2.5,2.0 とします。
3. プロパティ a をコンソールに表示してください。
4. プロパティ b をコンソールに表示してください。
5. メソッド sum の実行結果をコンソールに表示してください。
6. メソッド diff の実行結果をコンソールに表示してください。
7. メソッド times の実行結果をコンソールに表示してください。
8. メソッド div の実行結果をコンソールに表示してください。

1.6.7 解答例

```
1  struct Algebra{
2          var a: Float
3          var b: Float
4          init(a: Float, b: Float){
5                  self.a = a
6                  self.b = b
7          }
8
9          func sum() -> Float{
10                 return self.a + self.b
11         }
12
13         func diff() -> Float{
14                 return self.a - self.b
15         }
16
17         func times() -> Float{
18                 return self.a * self.b
19         }
20
21         func div() -> Float{
22                 return self.a / self.b
23         }
24 }
25
26 let al = Algebra(a: 2.5, b: 2.0)
27 print(al.a)
28 print(al.b)
29 print(al.sum())
30 print(al.diff())
31 print(al.times())
```

```
32 print(al.div())
```

1.7　スコープ

プログラミング言語にはスコープと呼ばれる機能があります。スコープとは定数、変数、関数、型の定義の有効範囲のことで、局所的なローカルスコープと大域的なグローバルスコープの２種類あります。全てのデータはこの２種類のスコープのどちらかに属することになります。同じスコープ内に同じ名前を複数定義することはできず、各スコープ内で名前は一意でなければなりません。

1.7.1　ローカルスコープ

関数、制御構文や型の内部で定義された変数や定数その他が有効な範囲をローカルスコープといいます。ローカルスコープはその範囲内で参照可能で、外部からは参照できません。原則 {} の内部がローカルスコープとなります[*7]。ローカルスコープで宣言されたものは限定的な範囲からの参照になるため、意図しない変更が起こりにくいというメリットがあります。以下のソースコードは関数中に定義した変数を関数外で使用した例ですが、ローカルスコープ外でアクセスしようとしているためエラーとなります。

ソースコード 1.31　グローバルスコープの例

```
1 func exampleFunc(){
2         var localVar = "This␣variable␣is␣belong␣to␣local␣scope."
3 }
4
5 print(localVar) // コンパイルエラー
```

関数や制御構文、型の内部で定義したローカル変数などを外部で使用しないように注意しましょう。

1.7.2　グローバルスコープ

グローバルスコープはどの関数にも型にも含まれないスコープで、大域的なスコープです。グローバルスコープで定義されたものはファイル外も含めどこからでも参照可能です。

ソースコード 1.32　グローバルスコープの例

```
1 var globalVar = "This␣variable␣is␣belong␣to␣global␣scope."
2
3 func exampleFunc(){
4         print(globalVar)
5 }
```

グローバルスコープで宣言された変数や定数、その他はどこからでも同じ名前で参照でき、意図しない変更を招きやすいので、使用には注意が必要です。

[*7] guard 文など一部例外があります。

1.8　練習問題

今までに解説してきた、定数や変数、データ型、関数、構造体に関するプログラミングを練習してみましょう。

1.8.1　定数と変数・データ型・演算子編

1. 定数名を name、値を”Bob” とする定数を宣言してください。
2. 定数名を pi、型を Double、値を 3.141592 とする定数を宣言してください。
3. 変数名を radius、型を Double、値を 2.0 とする変数を宣言してください。
4. 変数名を area、型を Double とする変数を宣言してください。値は円の面積とします。円の半径は radius、円周率は pi を用いてください。
5. 半径 radius の値を 5.0 に変更してください。
6. area を再計算してください。

解答例

```
let name = "Bob" // 型推論で$\rm{String}$型に決定されている。
print(name) // 値の確認のため出力
let pi:Double = 3.141592
var radius:Double = 2.0
var area:Double = radius * radius * pi // 円の面積の公式：半径*半径*円周率
print(area) // 値の確認のため出力
radius = 5.0 // 変数に値を代入
area = Double(radius * radius) * pi // 円の面積の公式は半径*半径*円周率
print(area) // 値の確認のため出力
```

1.8.2　関数編

1. 次の仕様を満たす関数をプログラムしてください。関数名は tax、引数名は price、引数の型は Double、返り値の型は Double で返り値の値は price を 1.08 倍したもの。
2. 関数 tax を実行してください。入力値は 100.0 とします。実行結果を print() でコンソールに出力してください。
3. 三角形の面積を求める関数をプログラムしてください。関数名は triangleArea、引数名は底辺が width、高さが height で共に Float 型、返り値の型は Float 型とします。
4. 関数 triangleArea を実行して結果を print() でコンソールに出力してください。ただし底辺は 5.0、高さは 10.0 とします。

解答例

```
//1
func tax(price:Double) -> Double{
    return price * 1.08
}
//2
print(tax(price:100.0))
//3
func triangleArea(width: Float, height: Float) -> Float{
    return width * height / 2.0
}
```

```swift
//4
print(triangleArea(width: 5.0, height: 10.0))
```

1.8.3 構造体編

1. 名前を Algebra とする構造体を宣言してください。この構造体は以下の仕様を満たします。
   (a) 変数の Int 型プロパティ a,b が宣言されています。プロパティ a と b の初期値はそれぞれ 2 と 3 です。
   (b) メソッド add が宣言され、返り値は二つのプロパティの和
   (c) メソッド sub が宣言され、返り値は二つのプロパティの差
   (d) メソッド times が宣言され、返り値は二つのプロパティの積
   (e) メソッド div が宣言され、返り値は二つのプロパティの商
2. 変数 al を宣言し、構造体 Algebra のインスタンスを代入してください。
3. プロパティ a を print() でコンソールに出力してください。
4. プロパティ b を print() でコンソールに出力してください。
5. メソッド add の結果を print() でコンソールに出力してください。
6. メソッド sub の結果を print() でコンソールに出力してください。
7. メソッド times の結果を print() でコンソールに出力してください。
8. メソッド div の結果を print() でコンソールに出力してください。Swift では原則同じ型同士での演算となるため、出力も演算子した型と同じになります。したがって、ここでは 2/3 で答えは商 0 あまり 2 となり、商の部分だけが取られます。
9. プロパティ a の値を 6 に変更してください。
10. メソッド add の結果を print() でコンソールに出力してください。
11. メソッド sub の結果を print() でコンソールに出力してください。
12. メソッド times の結果を print() でコンソールに出力してください。
13. メソッド div の結果を print() でコンソールに出力してください。
14. メソッド setting を追加してください。メソッドの引数は x,y とし、それぞれプロパティ a,b を x,y に更新するものとします。
15. メソッド setting によりプロパティ a,b の値をそれぞれ 12,4 に変更してください。
16. メソッド add の結果を print() でコンソールに出力してください。
17. メソッド sub の結果を print() でコンソールに出力してください。
18. メソッド times の結果を print() でコンソールに出力してください。
19. メソッド div の結果を print() でコンソールに出力してください。

解答例

```swift
//1
struct Algebra{
    //(a)
    var a: Int = 2
    var b: Int = 3

    //(b)
    func add() -> Int{
        return self.a + self.b
    }

    //(c)
```

```
    func sub() -> Int{
        return self.a - self.b
    }

    //(d)
    func times() -> Int{
        return self.a * self.b
    }

    //(e)
    func div() -> Int{
        return self.a / self.b
    }

    //14
    mutating func setting(x: Int, y: Int){
        self.a = x
        self.b = y
    }
}
//2
var al = Algebra()
//3
print(al.a)
//4
print(al.b)
//5
print(al.add())
//6
print(al.sub())
//7
print(al.times())
//8
print(al.div())
//9
al.a = 6
//10
print(al.add())
//11
print(al.sub())
//12
print(al.times())
//13
print(al.div())
//15
al.setting(x: 12, y: 4)
//16
print(al.add())
//17
print(al.sub())
//18
print(al.times())
```

```
//19
print(al.div())
```

超速 Swift プログラミングツアーは以上です。お疲れ様でした。次はいよいよ本格的なプログラミング学習の始まりです！