Swift 実習マニュアル

第零版 alpha04

大黒学

Swift 実習マニュアル・第零版 alpha04
著者——大黒学

2016 年　2 月 20 日（土）　第零版 alpha04 発行

目次

第 1 章 Swift の基礎 **8**
1.1 プログラム . . . 8
1.1.1 文書と言語 . . . 8
1.1.2 プログラムとプログラミング . . . 8
1.1.3 プログラミング言語 . . . 8
1.1.4 この文章について . . . 8
1.2 言語処理系 . . . 8
1.2.1 言語処理系の基礎 . . . 8
1.2.2 Swift の言語処理系 . . . 9
1.2.3 プログラムの入力 . . . 9
1.2.4 コンパイル . . . 9
1.2.5 機械語のプログラムの実行 . . . 9
1.2.6 機械語のファイルを作らずに実行 . . . 10
1.2.7 エラー . . . 10
1.3 REPL . . . 10
1.3.1 REPL の基礎 . . . 10
1.3.2 Swift の REPL . . . 10
1.3.3 Swift の REPL の終了 . . . 11
1.3.4 式の入力 . . . 11
1.3.5 エラー . . . 11
1.4 文 . . . 12
1.4.1 プログラムの構造 . . . 12
1.4.2 文の列 . . . 12
1.5 空白と注釈 . . . 12
1.5.1 空白 . . . 12
1.5.2 注釈 . . . 13
1.5.3 コメントアウトとアンコメント . . . 14
1.6 型の基礎 . . . 14
1.6.1 型とは何か . . . 14
1.6.2 基本型 . . . 14
1.6.3 真偽値 . . . 14
1.6.4 浮動小数点数 . . . 14
1.6.5 複合的な型 . . . 15

第 2 章 式 **15**
2.1 式の基礎 . . . 15
2.1.1 式と評価と値 . . . 15
2.1.2 式の構造 . . . 15
2.1.3 式の値の型 . . . 15
2.2 リテラル . . . 16
2.2.1 リテラルの基礎 . . . 16
2.2.2 整数リテラル . . . 16
2.2.3 浮動小数点数リテラル . . . 16
2.2.4 マイナスの数値を生成する式 . . . 17
2.2.5 文字列リテラル . . . 17
2.2.6 エスケープシーケンス . . . 17
2.2.7 真偽値リテラル . . . 18
2.3 関数呼び出し . . . 18
2.3.1 関数 . . . 18
2.3.2 ユーザー定義関数と組み込み関数 . . . 18
2.3.3 引数と戻り値 . . . 18
2.3.4 関数呼び出しの書き方 . . . 18

2.3.5 関数呼び出しの評価 . . . 19
2.3.6 絶対値を求める組み込み関数 . . . 19
2.3.7 print . . . 19
2.3.8 外部引数名 . . . 19
2.4 演算子式 . . . 20
2.4.1 演算 . . . 20
2.4.2 演算子式の書き方 . . . 20
2.4.3 算術演算 . . . 20
2.4.4 優先順位 . . . 21
2.4.5 結合規則 . . . 21
2.4.6 丸括弧 . . . 22
2.4.7 符号の反転 . . . 22
2.4.8 文字列の連結 . . . 22
2.4.9 二項演算子を普通の関数名にする方法 . . . 22
2.5 プロパティー . . . 23
2.5.1 プロパティーの基礎 . . . 23
2.5.2 数値から文字列への変換 . . . 23
2.5.3 空文字列かどうかの判定 . . . 23
2.5.4 大文字化と小文字化 . . . 23
2.5.5 メソッド . . . 23

第 3 章 識別子 23
3.1 識別子の基礎 . . . 24
3.1.1 識別子とは何か . . . 24
3.1.2 識別子の作り方 . . . 24
3.2 定数と変数 . . . 24
3.2.1 定数と変数の基礎 . . . 24
3.2.2 代入と初期化 . . . 24
3.2.3 定数宣言 . . . 25
3.2.4 変数宣言 . . . 25
3.2.5 定数名と変数名の値 . . . 26
3.3 代入演算子 . . . 26
3.3.1 代入演算子の基礎 . . . 26
3.3.2 左辺値と右辺値 . . . 26
3.3.3 単純代入演算子 . . . 26
3.3.4 複合代入演算子 . . . 26
3.4 関数宣言 . . . 27
3.4.1 関数宣言の基礎 . . . 27
3.4.2 基本的な関数宣言 . . . 27
3.5 スコープ . . . 28
3.5.1 スコープの基礎 . . . 28
3.5.2 グローバルスコープ . . . 28
3.5.3 ローカルスコープ . . . 29
3.5.4 ローカルスコープのメリット . . . 29
3.6 引数 . . . 29
3.6.1 仮引数 . . . 29
3.6.2 複数の引数を受け取る関数 . . . 30
3.6.3 仮引数名とは異なる外部引数名 . . . 30
3.6.4 デフォルト値 . . . 31
3.7 戻り値 . . . 31
3.7.1 戻り値の型 . . . 31
3.7.2 return 文 . . . 32
3.7.3 return 文を実行しないで終了した関数の戻り値 . . . 32
3.8 タプル . . . 33
3.8.1 タプルの基礎 . . . 33

3.8.2 タプルを生成する式の書き方 . . . 33
3.8.3 タプルの型 . . . 33
3.8.4 番号によるタプルの要素の取得 . . . 33
3.8.5 代入によるタプルの分解 . . . 34
3.8.6 要素名 . . . 34
3.8.7 複数のデータを返す関数 . . . 34

第 4 章 選択 35
4.1 選択の基礎 . . . 35
4.1.1 選択とは何か . . . 35
4.1.2 選択を記述する方法 . . . 35
4.2 比較演算子 . . . 36
4.2.1 比較演算子の基礎 . . . 36
4.2.2 大小関係 . . . 36
4.2.3 等しいかどうか . . . 36
4.3 if 文 . . . 36
4.3.1 if 文の基礎 . . . 36
4.3.2 else 以降を省略した if 文 . . . 37
4.3.3 多肢選択 . . . 38
4.4 switch 文 . . . 40
4.4.1 switch 文の基礎 . . . 40
4.4.2 選択肢の書き方 . . . 40
4.4.3 複数のリテラルを持つ case ラベル . . . 41
4.4.4 範囲による選択肢 . . . 41
4.5 論理演算子 . . . 42
4.5.1 論理演算子の基礎 . . . 42
4.5.2 論理積演算子 . . . 42
4.5.3 論理和演算子 . . . 42
4.5.4 論理否定演算子 . . . 43
4.6 条件演算子 . . . 43
4.6.1 条件演算子の基礎 . . . 43
4.6.2 条件演算子を含む式 . . . 43

第 5 章 繰り返しと再帰 44
5.1 繰り返しの基礎 . . . 44
5.1.1 繰り返しとは何か . . . 44
5.1.2 繰り返しを記述するための文 . . . 44
5.2 for-in 文 . . . 44
5.2.1 for-in 文の基礎 . . . 44
5.2.2 範囲 . . . 45
5.2.3 for-in 文の書き方 . . . 45
5.3 while 文 . . . 46
5.3.1 while 文の基礎 . . . 46
5.3.2 while 文の書き方 . . . 46
5.3.3 無限ループ . . . 46
5.3.4 条件による繰り返しの例 . . . 47
5.4 repeat-while 文 . . . 47
5.4.1 repeat-while 文の基礎 . . . 47
5.4.2 while 文と repeat-while 文の相違点 . . . 48
5.5 break 文と continue 文 . . . 48
5.5.1 break 文 . . . 48
5.5.2 continue 文 . . . 49
5.6 再帰 . . . 49
5.6.1 再帰とは何か . . . 49
5.6.2 基底 . . . 49

5.6.3 関数の再帰的な定義 50
5.6.4 階乗 50
5.6.5 フィボナッチ数列 50
5.6.6 最大公約数 51

第 6 章 オプショナル **51**
6.1 オプショナルの基礎 51
6.1.1 `nil` 51
6.1.2 オプショナル型 51
6.1.3 オプショナルと比較演算子 52
6.2 アンラップ 52
6.2.1 アンラップの基礎 52
6.2.2 感嘆符 53
6.2.3 nil 合体演算子 53
6.3 オプショナルを返す関数 53
6.3.1 この節について 53
6.3.2 文字列から整数への変換 53
6.3.3 文字列から浮動小数点数への変換 53
6.3.4 文字列の読み込み 54
6.3.5 オプショナルを返す関数の宣言 54
6.4 オプショナル束縛構文 55
6.4.1 オプショナル束縛構文の基礎 55
6.4.2 読み込みの繰り返し 55

第 7 章 コレクション **55**
7.1 コレクションの基礎 56
7.1.1 コレクションとは何か 56
7.1.2 コレクションの分類 56
7.1.3 コレクションの変更可能性 56
7.2 配列 56
7.2.1 配列の基礎 56
7.2.2 配列リテラル 56
7.2.3 配列のイニシャライザー 57
7.2.4 添字式 57
7.2.5 配列の要素の変更 58
7.2.6 配列の長さ 58
7.2.7 配列に対する繰り返し 58
7.2.8 配列の末尾への要素の追加 59
7.2.9 配列への要素の挿入 59
7.2.10 配列の要素の削除 59
7.2.11 配列の連結 60
7.3 集合 60
7.3.1 集合の基礎 60
7.3.2 集合のイニシャライザー 60
7.3.3 集合の要素数 60
7.3.4 集合の要素の帰属 61
7.3.5 集合に対する繰り返し 61
7.3.6 集合への要素の追加 61
7.3.7 集合の要素の削除 61
7.3.8 部分集合 62
7.3.9 和集合 62
7.3.10 共通部分 63
7.3.11 差集合 63
7.3.12 対称差集合 63
7.3.13 エラトステネスのふるい 64

7.4 辞書 . . . 65
7.4.1 辞書の基礎 . . . 65
7.4.2 辞書リテラル . . . 65
7.4.3 辞書の添字式 . . . 66
7.4.4 辞書の要素の変更 . . . 66
7.4.5 辞書の要素の削除 . . . 67
7.4.6 辞書の要素数 . . . 68
7.4.7 辞書に対する繰り返し . . . 68
7.4.8 度数分布 . . . 69

第 8 章 クロージャー **69**
8.1 クロージャーの基礎 . . . 70
8.1.1 クロージャーとは何か . . . 70
8.1.2 クロージャーの型 . . . 70
8.1.3 クロージャー式 . . . 70
8.1.4 クロージャー呼び出し . . . 70
8.1.5 クロージャーの戻り値 . . . 71
8.2 引数を受け取るクロージャー . . . 71
8.2.1 簡略仮引数名 . . . 71
8.2.2 引数を渡すクロージャー呼び出し . . . 71
8.2.3 仮引数の宣言 . . . 71
8.3 高階関数 . . . 72
8.3.1 高階関数の基礎 . . . 72
8.3.2 クロージャーを 2 回呼び出す関数 . . . 72
8.3.3 加算をするクロージャーを返す関数 . . . 72
8.3.4 クロージャーを合成する関数 . . . 73
8.4 コレクションの高階メソッド . . . 73
8.4.1 この節について . . . 73
8.4.2 丸括弧の省略 . . . 73
8.4.3 `map` . . . 73
8.4.4 `filter` . . . 74
8.4.5 `reduce` . . . 74
8.4.6 `flatMap` . . . 75
8.4.7 `forEach` . . . 76
8.4.8 `sort` . . . 76

参考文献 **77**

索引 **78**

第 1 章　Swift の基礎

1.1　プログラム

1.1.1　文書と言語

文字を並べることによって何かを記述したものは、「文書」(document) と呼ばれます。

文書を作るためには、記述したいことを意味として持つように、文字を並べていく必要があります。そして、そのためには、文字をどのように並べればどのような意味になるかということを定めた規則が必要になります。そのような規則は、「言語」(language) と呼ばれます。

人間に読んでもらうことを第一の目的とする文書を書く場合は、日本語や中国語やアラビア語のような、「自然言語」(natural language) と呼ばれる言語が使われます。自然言語というのは、人間の社会の中で自然発生的に形成された言語のことです。

言語には、自然言語のほかに、人間が意図的に設計することによって作られた、「人工言語」(artificial language) と呼ばれるものもあります。人間ではなくてコンピュータに読んでもらうことを第一の目的とする文書を書く場合は、通常、自然言語ではなく人工言語が使われます。

1.1.2　プログラムとプログラミング

コンピュータに何らかの動作を実行させるためには、実行してほしいことがどんな動作なのかということを記述した文書をコンピュータに与える必要があります。そのような文書は、「プログラム」(program) と呼ばれます。

プログラムを作成するためには、プログラムを書くという作業だけではなくて、プログラムの構造を設計したり、プログラムの動作をテストしたり、その不具合を修正したりするというような、さまざまな作業が必要になります。そのような、プログラムを作成するために必要となるさまざまな作業の全体は、「プログラミング」(programming) と呼ばれます。

1.1.3　プログラミング言語

プログラムというのも文書の一種ですから、それを書くためには何らかの言語が必要になります。プログラムを書く場合には、プログラムを書くことだけを目的として作られた人工言語を使うのが普通です。そのような、プログラムを書くための専用の言語は、「プログラミング言語」(programming language) と呼ばれます。

プログラミング言語には、たくさんのものがあります。例を挙げると、Fortran、COBOL、Lisp、Pascal、Basic、C、AWK、Smalltalk、ML、Prolog、Perl、PostScript、Tcl、Java、Ruby、Haskell、……というように、枚挙にいとまがないほどです。

1.1.4　この文章について

この文章（「Swift 実習マニュアル」）は、Swift というプログラミング言語を使って、プログラムというものの書き方について説明する、ということを目的とするチュートリアルです。

1.2　言語処理系

1.2.1　言語処理系の基礎

コンピュータというものは異質な二つの要素から構成されていて、それぞれの要素は、「ハードウェア」(hardware) と「ソフトウェア」(software) と呼ばれます。ハードウェアというのは物理的な装置のことで、ソフトウェアというのはプログラムなどのデータのことです。

コンピュータは、さまざまなプログラミング言語で書かれたプログラムを理解して実行することができます。しかし、コンピュータのハードウェアが、ソフトウェアの助力を得ないで単独で理解することのできるプログラミング言語は、ハードウェアの種類によって決まっているひとつの言語だけです。

ハードウェアが理解することのできるプログラミング言語は、そのハードウェアの「機械語」(machine language) と呼ばれます。機械語というのは、人間にとっては書くことも読むことも困難な言語ですので、人間が機械語でプログラムを書くということはめったにありません。

人間にとって書いたり読んだりすることが容易なプログラミング言語で書かれたプログラムをコンピュータに理解させるためには、そのためのプログラムが必要になります。そのような、人

間が書いたプログラムをコンピュータに理解させるためのプログラムのことを、「言語処理系」(language processor) と呼びます（「言語」を省略して、単に「処理系」と呼ぶこともあります)。

言語処理系には、「コンパイラ」(compiler) と「インタプリタ」(interpreter) と呼ばれる二つの種類があります。コンパイラというのは、人間が書いたプログラムを機械語に翻訳するプログラムのことで、インタプリタというのは、人間が書いたプログラムがあらわしている動作をコンピュータに実行させるプログラムのことです。

1.2.2 Swift の言語処理系

Swift の言語処理系としては、現在、OS X 版と Linux 版があります。

OS X 版は、Apple が無料で配布している Xcode というアプリに含まれています。このアプリを Mac にインストールすると、Swift の言語処理系もその一部分としてインストールされます。

Linux 版は、`http://swift.org/download/` からダウンロードすることができます。

1.2.3 プログラムの入力

プログラムをファイルに保存したり、すでにファイルに保存されているプログラムを修正したりしたいときは、「テキストエディター」(text editor) と呼ばれるソフトを使います（テキストエディターは、単に「エディター」(editor) と呼ばれることもあります)。

それでは、Swift の言語処理系を使って、Swift のプログラムを実行してみましょう。

まず、何らかのテキストエディターを使って、Swift のプログラムを入力して、それをファイルに保存します。 ちなみに、Swift のプログラムを保存するファイルに付ける拡張子は、`.swift` です。

それでは、次のプログラムを入力して、`hello.swift` という名前のファイルに保存してください。

```
print("こんにちは、世界。")
```

このプログラムは、「こんにちは、世界。」という文字列を出力して、さらに改行を出力する、という動作をします。

1.2.4 コンパイル

コンパイラを使ってプログラムを機械語に翻訳することを、プログラムを「コンパイルする」(compile) と言います。

Swift のコンパイラは、swiftc という名前です。シェルに対して、

```
swiftc パス名
```

というコマンドを入力すると、パス名で指定されたファイルに保存されている Swift のプログラムがコンパイルされます。ですから、`hello.swift` というファイルに Swift のプログラムが格納されていて、そのファイルがあるディレクトリがカレントディレクトリだとすると、

```
swiftc hello.swift
```

というコマンドをシェルに入力することによって、そのプログラムをコンパイルすることができます。

1.2.5 機械語のプログラムの実行

Swift のコンパイラは、コンパイルの結果としてできた機械語のプログラムを、Swift のプログラムが格納されているファイルのファイル名から `.swift` という拡張子を取り除いた名前のファイルに保存します。ですから、

```
./hello
```

というコマンドをシェルに入力することによって、機械語に翻訳されたプログラムを実行することができます。

実行例

```
$ ./hello
こんにちは、世界。
```

1.2.6 機械語のファイルを作らずに実行

swiftc ではなくて、swift という言語処理系を使うことによって、Swift のプログラムを、それを機械語に翻訳したファイルを作らずに実行することができます。

シェルに対して、

```
swift パス名
```

というコマンドを入力すると、パス名で指定されたファイルに格納されている Swift のプログラムが実行されます。ですから、`hello.swift` というファイルに Swift のプログラムが格納されていて、そのファイルがあるディレクトリがカレントディレクトリだとすると、

```
swift hello.swift
```

というコマンドをシェルに入力することによって、そのプログラムを実行することができます。

1.2.7 エラー

プログラムの中には、何らかの間違いが含まれていることがあります。そのような、プログラムの中に含まれている間違いは、「エラー」(error) と呼ばれます。

言語処理系は、入力されたプログラムがエラーを含んでいる場合、そのエラーについてのメッセージを出力します。そのような、エラーについてのメッセージは、「エラーメッセージ」(error message) と呼ばれます。

それでは、エラーを含んでいる次の Swift のプログラムをファイルに保存してください。

プログラムの例 `error.swift`

```
prink("こんにちは、世界。")
```

先ほどのプログラムとの相違点は、`print` という正しい名前が、`prink` という間違った名前に変わっているというところです。このプログラムを swiftc にコンパイルさせると、swiftc は、次のようなエラーメッセージを出力します。

```
error.swift:1:1: error: use of unresolved identifier 'prink'
prink("こんにちは、世界。")
```

このエラーメッセージは、`prink` という意味不明な名前が使われている、ということを述べています。

1.3 REPL

1.3.1 REPL の基礎

第 1.2 節では、ファイルに保存されているプログラムを Swift の言語処理系に実行させる方法について説明したわけですが、この節では、プログラムをファイルに保存してから実行させるのではなくて、キーボードから直接、プログラムを処理系に入力して実行させる方法について説明したいと思います。

言語処理系は、「REPL」と呼ばれるものと、そうでないものとに分類することができます。REPL というのは、read-eval-print loop の略称で、次の三つの動作を延々と繰り返す、という動作をする処理系のことです。

(1) プログラムまたはその断片を読み込む (read)。
(2) 読み込んだプログラムまたはその断片を実行する (eval)。
(3) 結果を出力する (print)。

ですから、REPL を使うことによって、プログラムまたはその断片をキーボードから直接入力して、それがどのように動作するかということを即座に確かめる、ということができます。

1.3.2 Swift の REPL

swift という言語処理系は、コマンドライン引数を何も渡さないで起動すると、REPL として動作します。

それでは、実際に Swift の REPL を起動してみましょう。`swift` というコマンドをターミナルに入力してみてください。そうすると、

```
Welcome to Apple Swift version ...
```

```
1>
```

というように表示されるはずです（処理系のバージョンが違う場合は、これとは少し違うものが表示されるかもしれません）。この最後に表示されている、

```
1>
```

というのが、Swift の REPL が出力するプロンプトです。プロンプトの中の整数は、何かを REPL に入力するたびに 1 ずつ増えていきます。

Swift の REPL は、コマンドを入力することによって操作することができます。Swift の REPL に対するコマンドは、コロン (:) で始まります。たとえば、 `:help` または `:h` というコマンドを入力すると、コマンドの使い方についてのヘルプが出力されます。

それでは、ヘルプを出力させてみましょう。 `:h` というコマンドを入力して、そののちエンターキーを押してみてください。

```
1> :h

The Swift REPL (Read-Eval-Print-Loop) acts like an interpreter.
（以下略）
```

1.3.3 Swift の REPL の終了

Swift の REPL は、 `:quit` または `:q` というコマンドを入力することによって終了させることができます。

それでは、Swift の REPL を終了させてみましょう。 `:q` というコマンドを入力して、そののちエンターキーを押してみてください。そうすると、REPL が終了して、ターミナルのプロンプトが表示されるはずです。

1.3.4 式の入力

Swift のプログラムの中には、「式」(expression) と呼ばれるものを書くことができます。式については、第 2 章で詳しく説明することになりますが、とりあえずここでは、数学で使われる式のように、何らかの計算を書きあらわしたものだと考えてください。

Swift の REPL には、コマンドだけではなくて、式を入力することもできます。Swift の REPL に式を入力すると、Swift の REPL は、それがあらわしている計算を実行して、その結果を出力します。

それでは、式を Swift の REPL に入力してみましょう。たとえば、 `5+3` という式を入力して、そののちエンターキーを押してみてください。そうすると、次のように、入力した式があらわしている計算が実行されて、 `8` という結果が出力されます。

```
1> 5+3
$R0: Int = 8
Prelude>
```

第 1.2.3 項で入力した、

```
print("こんにちは、世界。")
```

というプログラムも、その全体が 1 個の式になっていますので、REPL に入力することができます。入力すると、

```
こんにちは、世界。
```

と出力されるはずです。ただし、この式の値は、出力されません。

1.3.5 エラー

プログラムの中には、何らかの間違いが含まれていることがあります。そのような、プログラムの中に含まれている間違いは、「エラー」(error) と呼ばれます。

言語処理系は、入力されたプログラムがエラーを含んでいる場合、そのエラーについてのメッセージを出力します。そのような、エラーについてのメッセージは、「エラーメッセージ」(error message) と呼ばれます。

それでは、Swift の REPL に対して、 `hoge` と入力してみてください。そうすると、おそらく、

```
repl.swift:1:1: error: use of unresolved identifier 'hoge'
hoge
^
```

というようなメッセージが表示されるはずです。これは、hoge というのが何の名前なのか分からない、ということを意味しているエラーメッセージです。

1.4　文

1.4.1　プログラムの構造

プログラムという文書は、構文的な部品から構成されます。そして、プログラムを構文的に構成している部品には、いくつかの階層があります。もっとも下にある階層は文字で、もっとも上にある階層はプログラム全体です。

Swift というプログラミング言語の場合、プログラムを構成している部品の中間的な階層として、「文」(statement) と呼ばれるものや、「式」(expression) と呼ばれるものがあります。

第 1.2 節で、

```
print("こんにちは、世界。")
```

というプログラムを紹介しましたが、このプログラムの全体は、1 個の式になっています。また、この式の中にある、

```
"こんにちは、世界。"
```

という部分も、1 個の式です。このように、式という部品は、その中にさらに式が含まれていることもあります。つまり、式という部品は入れ子にすることができる、ということです。式だけではなくて、文も、入れ子にすることができます。

Swift では、すべての式は、そのままの形で文にすることもできます。式がそのまま文になったものは、「単純文」(simple statement) と呼ばれます。

1.4.2　文の列

Swift のプログラムの中には、文を、いくつでも並べて書くことができます。プログラムの中に文がいくつか並んでいる場合、コンピュータはそれらの文を、原則として、先頭から末尾に向かって 1 回ずつ実行していきます。

文を並べることによって文の列を作る場合、それぞれの文は、改行またはミコロン (;) で区切る必要があります。改行を使って文を区切ると、文は縦に並ぶことになります。文を縦ではなく横に並べたいときは、

```
文; 文; 文; ··· 文
```

というように、それらの文をセミコロンで区切ります。

それでは、実際に、2 個以上の文から構成される文の列を書いて、それがどのように実行されるかを試してみましょう。

プログラムの例　sequence.swift

```
print("菜の花や")
print("月は東に")
print("日は西に")
```

実行例

```
菜の花や
月は東に
日は西に
```

1.5　空白と注釈

1.5.1　空白

空白という文字（スペースキーを押したときに入力される文字）は、Swift のプログラムの意味に影響を与えません。たとえば、

```
print("こんにちは、世界。")
```

というプログラムは、

```
print   (   "こんにちは、世界。"   )
```

と書いたとしても同じ意味になります。

ただし、文字列リテラルの中に空白を挿入した場合は、その空白を含んだ文字列のデータが生成されます。たとえば、

```
print("こ　ん　に　ち　は　、　世　界　。")
```

というプログラムは、

```
こ　ん　に　ち　は　、　世　界　。
```

という文字列を出力します。

名前の途中には、空白を挿入することができません。ですから、`print` という名前を、

```
p r i n t
```

と書くことはできません。

1.5.2 注釈

プログラムを書いているとき、それを読む人間（プログラムを書いた人自身もその中に含まれます）に伝えたいことを、そのプログラムの一部分として書いておきたい、ということがしばしばあります。プログラムの中に書かれたそのような文字列は、「注釈」(comment) と呼ばれます。

注釈は、プログラムを処理するプログラムが、「ここからここまでは注釈だ」ということを認識することができるように、注釈を書くための文法にしたがって書く必要があります。

Swift には、「この部分は注釈である」ということを言語処理系に認識してもらう方法が、二つあります。

そのうちのひとつは、スラッシュスラッシュ(`//`) を使う方法です。プログラムの中に `//` を書くと、その直後から最初の改行までが注釈だと認識されます。たとえば、Swift の言語処理系は、

```
// 私は注釈です。
```

という記述を、注釈だと認識します。

それでは、REPL を使って、スラッシュスラッシュから改行までの部分が本当に注釈だと認識されるかどうかを確かめてみましょう。次のプログラムを REPL に入力してみてください。

```
print("こんにちは、世界。") // 私は注釈です。
```

この場合、入力したプログラムの中にある「私は注釈です。」という部分は、スラッシュスラッシュと改行のあいだに書かれていますので、Swift の言語処理系はその部分を注釈だと認識します。

スラッシュスラッシュを書かなかった場合は、どうなるでしょうか。次のプログラムを REPL に入力してみてください。

```
print("こんにちは、世界。") 私は注釈です。
```

スラッシュスラッシュを書かなかった場合、Swift の言語処理系は、書かれたものをすべて解釈しようとしますので、エラーメッセージが表示されることになります。

プログラムの一部分を注釈として認識してもらう方法の二つ目は、スラッシュアスタリスク(`/*`) とアスタリスクスラッシュ(`*/`) でそれを囲むという方法です。最初の改行までが注釈だと認識されます。たとえば、Swift の言語処理系は、

```
/* 私は注釈です。 */
```

という記述を、注釈だと認識します。

それでは、REPL を使って、スラッシュアスタリスクからアスタリスクスラッシュまでの部分が本当に注釈だと認識されるかどうかを確かめてみましょう。次のプログラムを REPL に入力してみてください。

```
print( /*  私は注釈です。*/ "こんにちは、世界。")
```

この場合、入力したプログラムの中にある「私は注釈です。」という部分は、スラッシュアスタリスクとアスタリスクスラッシュのあいだに書かれていますので、Swift の言語処理系はその部分を注釈だと認識します。

スラッシュアスタリスクとアスタリスクスラッシュを書かなかった場合は、どうなるでしょうか。次のプログラムを REPL に入力してみてください。

```
print( 私は注釈です。 "こんにちは、世界。")
```

スラッシュアスタリスクとアスタリスクスラッシュを書かなかった場合、Swift の言語処理系は、書かれたものをすべて解釈しようとしますので、エラーメッセージが表示されることになります。

改行を含んでいる注釈、つまり 2 行以上の注釈も、その全体を `/*` と `*/` で囲むことによって、注釈だと認識してもらうことができます。たとえば、Swift の言語処理系は、

```
/* 私は、改行を
含んでいる注釈です。 */
```

という記述を、注釈だと認識します。

1.5.3　コメントアウトとアンコメント

プログラムを作成したり修正したりしているとき、その一部分を一時的に無効にしたい、ということがしばしばあります。そのような場合、無効にしたい部分を削除してしまうと、それを復活させるのに手間がかかりますので、削除するのではなくて、注釈にすることによって無効にするという手段が、しばしば使われます。記述の一部分を注釈にすることによって、それを無効にすることを、その部分を「コメントアウトする」(comment out) と言います。逆に、コメントアウトされている部分を復活させることを、その部分を「アンコメントする」(uncomment) と言います。

1.6　型の基礎

1.6.1　型とは何か

同じ性質を持つデータの集合は、「型」(type) と呼ばれます。

Swift のプログラムにおいては、データはかならず、何らかの型に所属しているものとして扱われます。

データ D が型 T に所属しているとき、「D は T を持つ」という言い方をすることもあります。

1.6.2　基本型

Swift のプログラムが扱うデータの型には、さまざまなものがあります。それらのうちでもっとも基本的な一群の型は、「基本型」(base type) と呼ばれます。すべての基本型は、「型名」(type name) と呼ばれる、英字の大文字で始まる名前が与えられています。

基本型には、次のようなものがあります。

`Bool`	真偽値（第 1.6.3 項参照）の型。
`Int`	整数の型。
`Double`	浮動小数点数（第 1.6.4 項参照）の型。
`String`	文字列の型。

1.6.3　真偽値

成り立っているか、それとも成り立っていないか、という判断の対象は、「条件」(condition) と呼ばれます。

条件が成り立っていると判断されるとき、その条件は「真」(true) であると言われます。逆に、条件が成り立っていないと判断されるとき、その条件は「偽」(false) であると言われます。

真を意味するデータと、偽を意味するデータは、総称して「真偽値」(Boolean) と呼ばれます。

1.6.4　浮動小数点数

通常、コンピュータの内部では、小数点以下の桁を持つ数値は、浮動小数点数によって表現されます。「浮動小数点数」(floating point number) というのは、小数点が仮に置かれた数字列と、その数字列の小数点を移動させる整数、という二つのものの組によってあらわされる数値のことです。

小数点が仮に置かれた数字列は「仮数部」(mantissa part) と呼ばれ、仮数部の小数点を移動させる整数は「指数部」(exponent part) と呼ばれます。プラスの指数部は小数点を左へ移動させ、マイナスの指数部は小数点を右へ移動させます。

仮数部として使うことのできる数字列の最大の桁数は、「精度」(precision) と呼ばれます。

1.6.5　複合的な型

型としては、基本型だけではなくて、基本型を組み合わせることによってできる複合的なものもあります。

複合的な型を持つデータとしては、次のようなものがあります。

- 関数 (function)
- タプル (tuple)
- 配列 (array)
- 辞書 (dictionary)

関数については第 2.3 節や第 3.4 節などで、タプルについては第 3.8 節で、配列と辞書については第 7 章で説明することにしたいと思います。

第 2 章　式

2.1　式の基礎

2.1.1　式と評価と値

Swift のプログラムの中には、「式」(expression) と呼ばれるものを書くことができます。

式というのは、コンピュータによる何らかの動作をあらわしています。ですから、コンピュータは、式があらわしている動作を実行することができます。ただし、式の場合は、「実行する」(execute) とは言わずに、「評価する」(evaluate) と言うのが普通です。

式を評価すると、その結果として一つのデータが得られます。式を評価することによって得られるデータは、その式の「値」(value) と呼ばれます。

「評価する」という言葉は、「値を求める」というニュアンスを含んでいます。式を実行することを、「実行する」と言わずに「評価する」と言うのは、「式の実行というのは、単なる動作の実行ではなくて、値を求めるという志向性を持った動作の実行である」という意識が強く働いているからです。

2.1.2　式の構造

式というのは、プログラムというものを組み立てるための部品のようなものだと考えることができます。

多くの場合、式という部品は、より単純な式を組み合わせることによって作られています。たとえば、`5+3` というのはひとつの式ですが、この式は、より単純な式を組み合わせることによって作られています。

`5+3` という式の中にある `5` という部分は、5 という整数を求めるという動作をあらわしている式です。したがって、REPL に対して `5` と入力すると、その式が評価されて、その値が出力されます。

```
   1> 5
$R0: Int = 5
```

同じように、`3` という部分も、3 という整数を求めるという動作をあらわしている式です。つまり、`5+3` という式は、`5` という式と `3` という式を、`+` というものをあいだにはさんで結びつけることによってできているわけです。

どんな式でも、それ自体を、さらに複雑な式の部品にすることができます。たとえば、`5+3` という式を部品にして、`5+3-7` という式を作ることができます。式というのは、組み合わせることによっていくらでも複雑なものを作ることができるという、そんな性質を持っている部品なのです。

2.1.3　式の値の型

REPL は、式が入力されると、その式の値だけではなくて、その式の値の型も出力します。式の値の左側の、コロン (`:`) とイコール (`=`) のあいだです。たとえば、`5` と入力した場合は、

```
$R0: Int = 5
```

と出力されますので、入力した式の値の型は `Int` だということが分かります。

2.2 リテラル

2.2.1 リテラルの基礎

特定のデータを生成するという動作を記述した式は、「リテラル」(literal) と呼ばれます。たとえば、481 のような、何個かの数字を並べることによってできる列は、リテラルの一種です。

リテラルを評価すると、それによって生成されたデータが、その値として得られます。たとえば、481 というリテラルを評価すると、それによって生成された 481 という整数のデータが、その値として得られます。

```
  1> 481
$R0: Int = 481
```

なお、この文章のこれから先の部分では、誤解のおそれがない場合、「○○のデータ」のことを単に「○○」と呼ぶことがあります。たとえば、整数そのものではなくて整数のデータのことを指しているということが文脈から明らかに分かる場合には、整数のデータのことを単に「整数」と呼ぶことがあります。

2.2.2 整数リテラル

整数のデータを生成するリテラルは、「整数リテラル」(integer literal) と呼ばれます。

整数リテラルは、次の 4 種類に分類することができます。

- 10 進数リテラル (decimal integer literal)
- 16 進数リテラル (hexadecimal integer literal)
- 8 進数リテラル (octal integer literal)
- 2 進数リテラル (binary integer literal)

これらの整数リテラルの相違点は、その名前が示しているとおり、整数を表現するための基数です。つまり、それぞれの整数リテラルは、10、16、8、2 のそれぞれを基数として整数を表現します。

10 進数リテラルは、481 とか 3007 というような、数字だけから構成される列です。たとえば、481 という 10 進数リテラルを評価すると、481 というプラスの整数が値として得られます。

16 進数リテラルと 8 進数リテラルと 2 進数リテラルは、基数を示す接頭辞を先頭に書くことによって作られます。基数を示す接頭辞は、16 進数は 0x、8 進数は 0o、2 進数は 0b です。たとえば、0xff と 0o377 と 0b11111111 は、いずれも、255 という整数を生成します。

```
  1> 0xff
$R0: Int = 255
  2> 0o377
$R1: Int = 255
  3> 0b11111111
$R2: Int = 255
```

2.2.3 浮動小数点数リテラル

ひとつの数値を、数字の列と小数点の位置という二つの要素で表現しているデータは、「浮動小数点数」(floating point number) と呼ばれます。

浮動小数点数のデータを生成するリテラルは、「浮動小数点数リテラル」(floating point literal) と呼ばれます。

0.003、41.56、723.0 というような、ドット (.) という文字の左右に数字の列を書いたものは、浮動小数点数のデータを生成するリテラルになります。この場合、ドットは小数点の位置を示します。

```
  1> 3.14
$R0: Double = 3.1400000000000001
```

浮動小数点数を生成するリテラルとしては、

a e b

という形のものを書くことも可能です（e は大文字でもかまいません)。a のところには、ドットを 1 個だけ含むかまたは含まない数字の列を書くことができて、b のところには、数字の列また

は左側にマイナスのある数字の列を書くことができます。この形のリテラルを評価すると、

$a \times 10^b$

という浮動小数点数が生成されます。

リテラル	意味
`3e8`	3×10^8
`6.022e23`	6.022×10^{23}
`6.626e-34`	6.626×10^{-34}

```
  1> 3e8
$R0: Double = 300000000
```

2.2.4 マイナスの数値を生成する式

マイナス (-) という文字を書いて、その右側に整数または浮動小数点数のリテラルを書くと、その全体は、マイナスの数値を生成する式になります。たとえば、`-56` という式はマイナスの 56 という整数を生成して、`-0xff` はマイナスの 255 という整数を生成して、`-8.317` はマイナスの 8.317 という浮動小数点数を生成します。

```
  1> -56
$R0: Int = -56
```

マイナスの数値を生成するこのような式の先頭に書かれるマイナスという文字は、リテラルの一部分ではなくて、第 2.4 節で説明することになる「演算子」(operator) と呼ばれるものです。

2.2.5 文字列リテラル

文字列のデータを生成するリテラルは、「文字列リテラル」(string literal) と呼ばれます。

文字列リテラルは、二重引用符 (`"`) で文字列を囲むことによって作られます。たとえば、

```
"namako"
```

という記述は、文字列リテラルです。

文字列リテラルは、二重引用符で囲まれた中にある文字列を生成します。たとえば、

```
"namako"
```

という文字列リテラルを評価すると、`namako` という文字列が生成されて、その文字列が値として得られます。

```
  1> "namako"
$R0: String = "namako"
```

0 個の文字列から構成される文字列は、「空文字列」(empty string) と呼ばれます。空文字列は、2 個の連続した二重引用符を書くことによって生成することができます。

```
  1> ""
$R0: String = ""
```

2.2.6 エスケープシーケンス

文字の中には、たとえばビープ音や改ページのように、そのままでは文字リテラルや文字列リテラルの中に書くことができない特殊なものがあります。

そのような特殊な文字を生成する文字リテラルや、それを含んだ文字列を生成する文字列リテラルを書きたいときには、「エスケープシーケンス」(escape sequence) と呼ばれる文字列が使われます。エスケープシーケンスは、必ず、バックスラッシュ(\) という文字で始まります[1]。

エスケープシーケンスには、次のようなものがあります。

`\0`	ヌル文字	`\t`	水平タブ
`\'`	一重引用符	`\"`	二重引用符
`\n`	改行	`\r`	キャリッジリターン
`\\`	バックスラッシュ	`\u`n	16 進数 n に対応する `Unicode` 文字

[1] バックスラッシュは、日本語の環境では円マーク (`¥`) で表示されることがあります。

文字列リテラルの中にエスケープシーケンスが含まれていた場合、その文字列リテラルによって生成される文字列は、そのエスケープシーケンスが意味している文字を含むものになります。

```
  1> print("namako\numiushi\nkurage")
namako
umiushi
kurage
```

2.2.7　真偽値リテラル

真偽値のデータを生成するリテラルは、「真偽値リテラル」(Boolean literal) と呼ばれます。
真偽値リテラルとしては、次の二つのものがあります。

`true`　真。
`false`　偽。

```
  1> true
$R0: Bool = true
  2> false
$R1: Bool = false
```

2.3　関数呼び出し

2.3.1　関数

Swift では、何らかの動作を意味しているデータのことを「関数」(function) と呼びます。
コンピュータは、関数というデータがあらわしている動作を実行することができます。
関数があらわしている動作をコンピュータに実行させることを、関数を「呼び出す」(call) と言います。
関数というのはあくまでデータであって、それがあらわしている動作を実行するのはあくまでコンピュータです。しかし、プログラムを書く人間としては、「関数自体が、自分があらわしている動作を実行する」というイメージで考えるほうが思考が単純になります。ですから、このチュートリアルでも、これからは、関数自体が動作をするというイメージで説明をしていきたいと思います。

2.3.2　ユーザー定義関数と組み込み関数

関数を生成して、その関数に名前を与えることを、関数を「宣言する」(declare) と言います。
関数に与えられた名前は、「関数名」(function name) と呼ばれます。
Swift では、「関数宣言」(function declaration) と呼ばれる記述をプログラムの中に書くことによって、関数を自由に宣言することができます（関数宣言の書き方については、第 3 章で説明することにしたいと思います)。プログラムの中に関数宣言を書くことによって宣言された関数は、「ユーザー定義関数」(user-defined function) と呼ばれます。
「ユーザー定義関数」の対義語は、「組み込み関数」です。「組み込み関数」(built-in function) というのは、Swift の言語処理系に組み込まれている関数のことです。組み込み関数は、関数宣言を書かなくても利用することができます。

2.3.3　引数と戻り値

関数は、何らかのデータを受け取って動作します。関数が受け取るデータは、「引数」(argument) と呼ばれます（「引数」は「ひきすう」と読みます)。
関数は、自分の動作が終了したのちに、自分を呼び出した者にデータを返すことができます。関数が返すデータは、「戻り値」(return value) と呼ばれます。
関数 f を動作させて、それに引数としてデータ d を渡すことを、「f を d に適用する」(apply f to d) と言うこともあります。

2.3.4　関数呼び出しの書き方

関数を呼び出したいときは、関数を呼び出すという動作をあらわす式を書きます。そのような式は、「関数呼び出し」(function call) と呼ばれます。
関数呼び出しは、基本的には、

関数名(式)

と書きます。「関数名」のところには、呼び出したい関数の名前を書きます。そして、「式」のところには、関数に渡す引数を求める式を書きます。

たとえば、hoge という名前の関数があって、この関数は引数として 1 個の整数を受け取るとしましょう。このとき、

```
hoge(78)
```

という関数呼び出しを書くことによって、hoge という関数を呼び出して、引数として 78 という整数をそれに渡すことができます。

2.3.5 関数呼び出しの評価

関数呼び出しは式ですから、評価することができます。関数呼び出しを評価すると、関数を呼び出してその関数に引数を渡すという、その関数呼び出しがあらわしている動作が実行されます。そして、呼び出された関数が返した戻り値が、関数呼び出しの値になります。たとえば、

```
hoge(78)
```

という関数呼び出しを評価すると、hoge という関数が呼び出されて、引数として 78 が渡されるわけですが、この関数呼び出しの値は、そのときに hoge が返した戻り値です。

2.3.6 絶対値を求める組み込み関数

すべての組み込み関数には名前が与えられています。ですから、関数呼び出しの中に、組み込み関数の名前と、その関数に渡す引数を求める式を書くことによって、その関数を呼び出して、それに引数を渡すことができます。

Swift の言語処理系には、abs という組み込み関数があります。この関数は、引数として数値を受け取って、その数値の絶対値を戻り値として返します。

a が数値だとするとき、a の絶対値 (absolute value) は、もしも a が 0 またはプラスならば a 自身で、マイナスならば $-a$ です。たとえば、5 の絶対値は 5 で、-5 の絶対値は 5 です。

それでは、abs を呼び出す関数呼び出しを REPL に入力することによって、絶対値を求めてみましょう。

```
  1> abs(5)
$R0: Int = 5
  2> abs(-5)
$R1: Int = 5
```

2.3.7 print

第 1.2.3 項で紹介した、

```
print("こんにちは、世界。")
```

というプログラムは、その全体が、print という名前の組み込み関数を呼び出す関数呼び出しになっています。この関数呼び出しは、print という組み込み関数を呼び出して、

```
"こんにちは、世界。"
```

という文字列を引数としてそれに渡します。

print は、どんなデータを引数として受け取った場合も、戻り値として、「空タプル」(empty tuple) と呼ばれるデータを返します。ですから、print を呼び出す関数呼び出しの値は、常に空タプルです。

Swift の REPL は、入力された式の値が空タプルだった場合、その値を出力しません。ですから、print を呼び出す関数呼び出しを REPL に入力した場合、その値は出力されません。

2.3.8 外部引数名

関数の中には、複数の引数を受け取ることができるものもあります。複数の引数を受け取ることのできる関数を呼び出して、複数の引数を渡すためには、「外部引数名」(external parameter name) と呼ばれるものを使う必要があります。

外部引数名というのは、受け取る引数が何であるかということを意味する名前のことです。2 個目の引数を渡したいときは、1 個目の引数を求める式のうしろにコンマを書いて、そのさらに

うしろに、

外部引数名 : 式

という形のものを書きます。3 個目以降の引数を渡す場合も、同じように、この形の記述をコンマで区切って並べます。

たとえば、print という組み込み関数は、terminator という外部引数名を使って引数を渡すことのできる仮引数を持っています。print は、デフォルトでは、式の値を出力したのちに 1 個の改行を出力するのですが、この外部引数名を使って文字列を渡すと、改行の代わりにその文字列が出力されます。

REPL を使って試してみましょう。

```
  1> print("namako", terminator: "/"); print("umiushi")
namako/umiushi
```

式の値を出力したのち、文字列を何も出力してほしくない、という場合は、terminator を使って空文字列を渡します。

```
  1> print("kitsune", terminator: ""); print("udon")
kitsuneudon
```

2.4　演算子式

2.4.1　演算

Swift の組み込み関数の中には、関数呼び出しではなくて、「演算子式」(operator expression) と呼ばれる式によって呼び出されるものもあります。

演算子式によって呼び出される関数は、「演算」(operation) と呼ばれます。そして、演算に与えられた名前は、「演算子」(operator) と呼ばれます。

演算には、受け取る引数の個数が 1 個のもの、2 個のもの、そして 3 個のものがあります。

1 個の引数を受け取る演算は「単項演算」(unary operation) と呼ばれ、そのような演算に与えられた名前は「単項演算子」(unary operator) と呼ばれます。

2 個の引数を受け取る演算は「二項演算」(binary operation) と呼ばれ、そのような演算に与えられた名前は「二項演算子」(binary operator) と呼ばれます。

3 個の引数を受け取る演算は「三項演算」(ternary operation) と呼ばれ、そのような演算に与えられた名前は「三項演算子」(ternary operator) と呼ばれます。

Swift にある三項演算子は、ひとつだけです。それについては、第 4.6 節で説明することにしたいと思います。

2.4.2　演算子式の書き方

単項演算を呼び出してそれに引数を渡す演算子式は、

演算子 式

と書きます。この形の演算子式を評価すると、演算子の右側に書かれた式の値が引数として演算に渡されて、その演算の戻り値が演算子式の値になります。

二項演算を呼び出してそれに引数を渡す演算子式は、

式 $_1$ 演算子 式 $_2$

と書きます。この形の演算子式を評価すると、式 $_1$ と式 $_2$ の値が引数として演算に渡されて、その演算の戻り値が演算子式の値になります。

2.4.3　算術演算

数値に対する計算を動作とする演算は「算術演算」(arithmetic operation) と呼ばれ、そのような演算に与えられた名前は「算術演算子」(arithmetic operator) と呼ばれます。

二項演算でかつ算術演算であるような演算としては、次のようなものがあります。

$a + b$　　a と b とを足し算（加算）する。

$a - b$　　a から b を引き算（減算）する。

$a * b$　　a と b とを掛け算（乗算）する。

a / b　　a を b で割り算（除算）して、その商を求める。
a % b　　a を b で割り算（除算）して、そのあまりを求める。

```
  1> 30+8
$R0: Int = 38
  2> 30-8
$R1: Int = 22
  3> 30*8
$R2: Int = 240
  4> 30/8
$R3: Int = 3
  5> 30%8
$R4: Int = 6
```

これらの演算は、二つの引数の両方が整数の場合は戻り値も整数になります。引数のどちらか一方、あるいは両方が浮動小数点数の場合、戻り値は浮動小数点数になります。

```
  1> 30.0+8
$R0: Double = 38
  2> 30.0/8
$R1: Double = 3.75
  3> 30.0%8
$R2: Double = 6
```

2.4.4 優先順位

ひとつの式の中に 2 個以上の演算子が含まれている場合、その式はどのように解釈されるのでしょうか。たとえば、

```
2+3*4
```

という式は、

```
2+3 *4
```

という構造なのでしょうか。それとも、

```
2+ 3*4
```

という構造なのでしょうか。
この問題は、個々の演算子が持っている「優先順位」(precedence) と呼ばれるものによって解決されます。
優先順位というのは、演算子が左右の式と結合する強さのことだと考えることができます。優先順位が高い演算子は、それが低い演算子よりも、より強く左右の式と結合します。
* と / と % は、 + と - よりも高い優先順位を持っています。ですから、

```
2+3*4
```

という式は、

```
2+ 3*4
```

という構造だと解釈されます。

```
  1> 2+3*4
$R0: Int = 14
```

2.4.5 結合規則

ひとつの式の中に同じ優先順位を持っている 2 個以上の演算子が含まれている場合、その式はどのように解釈されるのでしょうか。たとえば、

```
10-5+2
```

という式は、

```
10-5 +2
```

という構造なのでしょうか。それとも、

```
10-5+2
```

という構造なのでしょうか。

この問題は、同一の優先順位を持つ演算子が共有している「結合規則」(associativity) と呼ばれる性質によって解決されます。

結合規則には、「左結合」(left-associativity) と「右結合」(right-associativity) という二つのものがあります。左結合というのは、左右の式と結合する強さが左にあるものほど強くなるという性質で、右結合というのは、それが右にあるものほど強くなるという性質です。

+、-、*、/、%の結合規則は、左結合です。したがって、

```
10-5+2
```

という式は、

```
10-5+2
```

という構造だと解釈されます。

```
  1> 10-5+2
$R0: Int = 7
```

2.4.6 丸括弧

ところで、2 と 3 とを足し算して、その結果と 4 とを掛け算したい、というときは、どのような式を書けばいいのでしょうか。先ほど説明したように、+ と * とでは、* のほうが優先順位が高くなっていますので、

```
2+3*4
```

と書いたのでは、期待した結果は得られません。

演算子の優先順位や結合規則に縛られずに、自分が望んだとおりに式を解釈してほしい場合は、ひとまとまりの式だと解釈してほしい部分を、丸括弧 (()) で囲みます。そうすると、丸括弧で囲まれている部分は、演算子の優先順位や結合規則とは無関係に、ひとまとまりの式だと解釈されます。

```
  1> (2+3)*4
$R0: Int = 20
  2> 10-(5+2)
$R1: Int = 3
```

2.4.7 符号の反転

- という単項演算は、数値の符号（プラスかマイナスか）を反転させる演算です。

```
  1> -(3.0+5)
$R0: Double = -8
  2> -(3.0-5)
$R1: Double = 2
```

2.4.8 文字列の連結

+ という二項演算は、2 個の引数が両方とも文字列の場合は、それらの文字列を連結して、その結果を戻り値として返します。

```
  1> "kitsune" + "udon"
$R0: String = "kitsuneudon"
```

2.4.9 二項演算子を普通の関数名にする方法

二項演算子は、普通の関数名として、普通の関数呼び出しの中に書くことも可能です。

二項演算子を丸括弧で囲んだものは、普通の関数の名前と同じように、普通の関数呼び出しの中に書くことができます。この場合、1 個目の引数が二項演算の左側の引数になって、2 個目の引数が二項演算の右側の引数になります。2 個目の引数は、外部引数名を使わずに渡します。

```
  1> (/)(30, 8)
$R0: Int = 3
```

2.5 プロパティー

2.5.1 プロパティーの基礎

データは、「プロパティー」(property) と呼ばれる、データを入れることのできる箱を持っています。1 個のデータが持っているプロパティーは、1 個だけとは限りません。個々のプロパティーは、「プロパティー名」(property name) と呼ばれる名前によって識別されます。

データから、そのデータが持っているプロパティーに格納されているデータを取得したいときは、

式 . プロパティー名

という形の式を書きます。この形の式を評価することによって得られる値は、ドット (.) の左側に書かれた式の値が持っている、プロパティー名によって識別されるプロパティーの内容です。

2.5.2 数値から文字列への変換

数値は、`description` というプロパティーを持っています。このプロパティーには、数値を文字列に変換した結果が格納されています。たとえば、768 という数値が持っている `description` には、`"768"` という文字列が格納されています。

```
  1> 768.description
$R0: String = "768"
  2> 3.14.description
$R1: String = "3.14"
```

2.5.3 空文字列かどうかの判定

文字列は、`isEmpty` という名前のプロパティーを持っています。このプロパティーに格納されているのは、文字列が空文字列かそうでないかということを示す真偽値です。空文字列が持っている `isEmpty` には `true` が格納されていて、空文字列ではない文字列には `false` が格納されています。

```
  1> "".isEmpty
$R0: Bool = true
  2> "namako".isEmpty
$R1: Bool = false
```

2.5.4 大文字化と小文字化

文字列が持っている `uppercaseString` というプロパティーには、その文字列に含まれているすべての英字の小文字を大文字に変換した結果が格納されています。

同じように、文字列が持っている `lowercaseString` というプロパティーには、その文字列に含まれているすべての英字の大文字を小文字に変換した結果が格納されています。

```
  1> "namako".uppercaseString
$R0: String = "NAMAKO"
  2> "NAMAKO".lowercaseString
$R1: String = "namako"
```

2.5.5 メソッド

関数が格納されているプロパティーは、「メソッド」(method) と呼ばれます。メソッドであるプロパティー名は、「メソッド名」(method name) と呼ばれます。

たとえば、整数は、`successor` というメソッドと `predecessor` というメソッドを持っています。前者には、その整数の次の整数を返す関数が格納されていて、後者には、その整数の前の整数を返す関数が格納されています。

```
  1> 100.successor()
$R0: Int = 101
  2> 100.predecessor()
$R1: Int = 99
```

第 3 章 識別子

3.1　識別子の基礎

3.1.1　識別子とは何か

Swift のプログラムでは、しばしば、何かに名前を与えておいて、その名前によってその何かを識別する、ということをします。何かに名前として与えることのできるものは、「識別子」(identifier) と呼ばれます。

3.1.2　識別子の作り方

識別子は、次のような規則に従って作ることになっています。

- 識別子を作るために使うことのできる文字は、英字、数字、アンダースコア (_) です。
- 識別子の先頭の文字として、数字を使うことはできません。
- キーワード (keyword) と同じものは識別子としては使えません。

「キーワード」(keyword) というのは、用途があらかじめ予約されている単語のことで、たとえば次のようなものがあります（これだけではありません）。

```
as、break、case、catch、class、continue、default、do、else、enum、extension、false、
for、func、guard、init、inout、internal、if、in、is、let、nil、operator、private、
protocol、public、repeat、return、self、static、struct、subscript、switch、super、
throw、throws、true、try、var、where、while
```

識別子として使うことのできるものの例としては、次のようなものがあります。

```
a   A   a8   namako   back_to_the_future
```

英字の大文字と小文字は区別されますので、たとえば、`a` と `A` は、それぞれを異なるものの名前として使うことができます。

最後の例のように、アンダースコアは、複数の単語から構成される識別子を作るときに、空白の代わりとして使うことができます。

識別子として使うことのできないものの例としては、次のようなものがあります。

`nam@ko`	使うことのできない文字 (`@`) を含んでいる。
`8a`	先頭の文字が数字。
`var`	同じキーワードが存在する。

3.2　定数と変数

3.2.1　定数と変数の基礎

Swift では、データを箱に入れておく、ということができます。データを箱に入れておくと、そのデータを、それが入っている箱の名前で扱うということができるようになります。

データを入れることのできる箱には、「定数」(constant) と「変数」(variable) と呼ばれる 2 種類のものがあります。

定数の名前は「定数名」(constant name) と呼ばれ、変数の名前は「変数名」(variable name) と呼ばれます。定数名も変数名も、第 3.1 節で説明した識別子が使われます。

定数や変数を作ることを、それを「宣言する」(declare) と言います。

データが型を持つのと同じように、定数や変数も型を持ちます。定数や変数に代入することができるのは、その定数や変数と同じ型を持つデータだけです。

3.2.2　代入と初期化

定数または変数にデータを入れることを、そこにデータを「代入する」(assign) と言います。

定数も変数も、それを何らかのデータで「初期化する」(initialize) ということができます。定数や変数をデータで初期化するというのは、それが宣言されると同時に、そこにデータを代入するということです。

Swift では、定数と変数はどちらもデータを入れることのできる箱ですが、それらのあいだには大きな相違点があります。定数は、初期化することはできるのですが、それ以降は、そこにデータを代入することができません。それに対して、変数は、初期化だけではなくて、それ以降も、何度でもデータを代入することができます。

1 個の定数または変数が保持することのできるデータは、1 個だけです。変数にデータを代入すると、それまでその変数に入っていたデータは、失われてしまいます。

3.2.3 定数宣言

定数を宣言する記述は、「定数宣言」(constant declaration) と呼ばれます。
定数宣言は、

`let` 識別子 `:` 型 `=` 式

と書きます。そうすると、「識別子」のところに書かれた識別子を名前とする、「型」のところに書かれた型を持つ定数が宣言されて、「式」のところに書かれた式の値でそれが初期化されます。たとえば、

```
let page: Int = 187
```

という定数宣言を書くことによって、`page` という名前と `Int` という型を持つ、187 という整数で初期化された定数を宣言することができます。

Swift の言語処理系は、「型推論」(type inference) という機能を持っていますので、定数や変数が宣言されたとき、それを初期化するデータの型から、その定数や変数の型を推論することができます。コロンと型を省略した、

`let` 識別子 `=` 式

という形の定数宣言が書かれていた場合は、型推論によって定数の型を決定します。たとえば、

```
let page = 187
```

という定数宣言を書いた場合も、先ほどと同じように、`page` の型は `Int` になります。

3.2.4 変数宣言

変数を宣言する記述は、「変数宣言」(variable declaration) と呼ばれます。
変数宣言は、

`var` 識別子 `:` 型 `=` 式

と書きます。そうすると、「識別子」のところに書かれた識別子を名前とする、「型」のところに書かれた型を持つ変数が宣言されて、「式」のところに書かれた式の値でそれが初期化されます。たとえば、

```
var year: Int = 2037
```

という変数宣言を書くことによって、`year` という名前と `Int` という型を持つ、2037 という整数で初期化された変数を宣言することができます。

定数宣言の場合と同じように、変数宣言の場合も、型推論を利用することができますので、先ほどの変数宣言は、

```
var year = 2037
```

と書いたとしても同じ意味になります。

変数は、初期化をしないということも可能です。初期化しないで変数を宣言したいときは、イコールと式を省略した、

`var` 識別子 `:` 型

という形の変数宣言を書きます。たとえば、

```
var price: Int
```

という変数宣言を書くことによって、`price` という名前と `Int` という型を持つ、初期化されていない変数を宣言することができます。この場合は型推論ができませんので、型を省略することはできません。

なお、変数宣言を REPL に入力する場合は、初期化しないとエラーになります。

3.2.5　定数名と変数名の値

定数名や変数名は、式として評価することができます。定数名や変数名を評価すると、その名前を持つ定数や変数の中に入っているデータが、その値として得られます。

REPL を使って試してみましょう。

```
  1> let page = 187
page: Int = 187
  2> page
$R0: Int = 187
  3> var year = 2037
year: Int = 2037
  4> year
$R1: Int = 2037
```

3.3　代入演算子

3.3.1　代入演算子の基礎

第 3.2.2 項で説明したように、定数は、初期化したのちにそこにデータを代入するということができませんが、変数は、そこに何度でもデータを代入することができます。

変数を宣言したのちに、そこにデータを代入したいときは、「代入演算子」(assingment operator) と呼ばれる演算子を使います。代入演算子は、代入という動作をする演算に与えられた名前です。

代入演算子は、そのすべてが二項演算子で、ほとんどすべての演算子よりも低い優先順位を持っています。

3.3.2　左辺値と右辺値

第 2.1.1 項で説明したように、式を評価すると、その結果として、「値」(value) と呼ばれるデータが得られます。

式を評価することによって得られるものは、必ずしもデータだけとは限りません。場合によっては、データだけではなくて、変数などの箱が得られることもあります。

式を評価することによって得られる、データを入れることのできる箱は、その式の「左辺値」(left value) と呼ばれます。それに対して、式を評価することによって得られるデータは、「右辺値」(right value) と呼ばれます。「値」(value) という言葉は、左辺値と右辺値の総称です。

値として左辺値を持つ式は、必ず右辺値も持ちます。左辺値を持つ式の右辺値は、得られた左辺値の中に格納されているデータです。

変数名は、左辺値が得られる式の一例です。変数名を評価すると、その名前を持っている変数が左辺値として得られます。そして、その変数の内容が右辺値として得られます。

3.3.3　単純代入演算子

= という演算子は、「単純代入演算子」(simple assignment operator) と呼ばれます。これは、代入演算子のうちで、もっとも単純な動作をするものです。

= は二項演算子ですので、= を含む式は、

$$\boxed{\text{式}_1} \ = \ \boxed{\text{式}_2}$$

と書きます。この形の式を評価すると、式$_1$ を評価することによって得られた左辺値に、式$_2$ を評価することによって得られたデータが代入されます。= は戻り値として空タプルを返しますので、= を含む式の値は空タプルです。

それでは、単純代入演算子を使って変数にデータを代入して、そののち変数の内容を調べてみましょう。

```
  1> var a = 20
a: Int = 20
  2> a = 20
  3> a
$R0: Int = 20
```

3.3.4　複合代入演算子

`+=`、`-=`、`*=`、`/=`、`%=` などの演算子は、「複合代入演算子」(compound assignment operator) と呼ばれます。

複合代入演算子を含む式は、`=` を含む式と同じように、

式$_1$ 複合代入演算子 式$_2$

と書きます。この形の式を評価すると、式$_1$ を評価することによって得られた右辺値と、式$_2$ を評価することによって得られた右辺値に対して演算が実行されて、その結果が、式$_1$ を評価することによって得られた左辺値に代入されます。単純代入演算子と同様に、複合代入演算子も、戻り値として空タプルを返しますので、複合代入演算子を含む式の値は空タプルです。

ところで、「演算が実行されて」と書きましたが、ここで実行される「演算」というのは、いったい何なのでしょうか。

複合代入演算子は、すべて、イコール (`=`) の左側に何らかの二項演算子を書いた形になっています。左辺値の内容と式の値に対して実行される演算は、イコールの左側に書かれた二項演算子があらわしている演算です。つまり、☆が二項演算子だとすると、

a ☆= b

という式は、

a = a ☆ b

という式と同じ意味になるということです（ただし、前者は a が 1 回しか評価されないのに対して、後者は a が 2 回評価される、という相違があります)。たとえば、

```
a += 8
```

という式は、

```
a = a + 8
```

という式と同じ意味になります。

それでは、複合代入演算子を使って、変数の内容を変化させてみましょう。

```
  1> var a = 7
a: Int = 7
  2> a += 8
  3> a
$R0: Int = 15
```

3.4 関数宣言

3.4.1 関数宣言の基礎

関数を生成して、その関数に名前として識別子を与えることを、関数を「定義する」(define) と言います。

関数は、「関数宣言」(function declaration) と呼ばれる記述を書くことによって定義することができます。関数宣言は、関数を定義するという動作を意味する文です。

3.4.2 基本的な関数宣言

引数を受け取らず、戻り値も返さない関数を定義する関数宣言は、

```
func 識別子() {
```

文

```
}
```

と書きます。プログラムの中に関数宣言を書いておくと、中括弧の中に書かれた文の列を実行する関数が生成されて、「識別子」のところに書かれた識別子が、その関数に名前として与えられます。

たとえば、次のような関数宣言を書くことによって、`namako` という文字列を出力する関数を生成して、`namako` という識別子を名前としてその関数に与えることができます。

```
func namako() {
    print("namako")
}
```

それでは、この関数宣言を REPL に入力して、定義された関数を呼び出してみましょう。

```
  1> func namako() {
  2.     print("namako")
  3. }
  4> namako()
namako
```

このように、REPL は、入力の途中でエンターキーが押された場合は、大なり (>) ではなくドット (.) をプロンプトの末尾に出力します。

次に、関数宣言をファイルに保存しておいて、それを REPL にコピーアンドペーストしてみましょう。まず、次の関数宣言を、 world.swift というファイルに保存してください。

プログラムの例　world.swift

```
func world() {
    print("What a wonderful world!")
}
```

このように、Swift のプログラムをファイルに保存する場合は、ファイル名の拡張子を .swift にします。

それでは、このプログラムを REPL にコピーアンドペーストして、定義された関数を呼び出してみましょう。

実行例

```
  1> func world() {
  2.     print("What a wonderful world!")
  3. }
  4> world()
What a wonderful world!
```

関数宣言を REPL に入力する場合、それを直接 REPL に入力するという方法では、エラーがあった場合の修正が面倒ですし、後日、もう一度同じものを入力するためには、同じ作業を繰り返す必要があります。ファイルに保存してからコピーアンドペーストするという方法を使えば、修正も簡単ですし、同じ作業を繰り返す必要もなくなります。

3.5　スコープ

3.5.1　スコープの基礎

識別子と、その識別子が名前として与えられたものとのあいだの関係が有効である、プログラムの上での範囲は、その識別子の「スコープ」(scope) と呼ばれます。

3.5.2　グローバルスコープ

Swift では、関数宣言の外でオブジェクトに束縛された識別子は、ひとつのファイルの中に保存されたプログラムの全域というスコープを持つことになります。そのようなスコープは、「グローバルスコープ」(global scope) と呼ばれます。

関数宣言の内部もグローバルスコープの一部分ですから、関数宣言の中でグローバルスコープを持つ識別子を評価すると、その識別子が束縛されているオブジェクトが値として得られます。

プログラムの例　global.swift

```
func function() {
    print("function: \(a)")
}

let a = "I am a string."
function()
```

実行例

```
$ swift global.swift
```

```
function: I am a string.
```

3.5.3 ローカルスコープ

Swift では、関数宣言の中で何かに名前として与えられた識別子は、その関数宣言の中だけというスコープを持つことになります。そのような、関数宣言の中だけという限定されたスコープは、「ローカルスコープ」(local scope) と呼ばれます。

グローバルスコープを持つ識別子と同一の識別子を、ローカルスコープを持つ識別子として使っても、問題はありません。

プログラムの例 `local.swift`

```
func function() {
    let a = "My scope is function."
    print("function: \(a)")
    functionfunction()
    print("function: \(a)")
}

func functionfunction() {
    let a = "My scope is functionfunction."
    print("functionfunction: \(a)")
}

let a = "My scope is global."
print(a)
function()
print(a)
```

実行例

```
$ swift local.swift
My scope is global.
function: My scope is function.
functionfunction: My scope is functionfunction.
function: My scope is function.
My scope is global.
```

3.5.4 ローカルスコープのメリット

ところで、「関数宣言の中で宣言された識別子は、その関数宣言の中だけというスコープを持つ」という規則には、いったいどのようなメリットがあるのでしょうか。

もしも、「ひとつの関数宣言の中で宣言された識別子は、それとは別の関数宣言の中でも有効である」という規則が定められていたとするとどうなるか、ということについて考えてみましょう。その場合、識別子を何かに与えるときには、その識別子がすでに別の関数宣言で使われていないか、ということに細心の注意を払う必要があります。うっかりと同一の識別子を複数の関数宣言の中で使うと、思わぬ不具合が発生しかねません。

つまり、「関数宣言の中で宣言された識別子は、その関数宣言の中だけというスコープを持つ」という規則は、「関数宣言を書くときに、その関数宣言の外でどのような識別子が使われているかということを、まったく気にする必要がない」というメリットを、プログラムを書く人に与えてくれているのです。

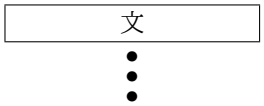
3.6 引数

3.6.1 仮引数

引数を受け取る関数を宣言するためには、その引数を受け取る定数を宣言する必要があります。そのような定数は、「仮引数」(parameter) と呼ばれます。仮引数は、関数が呼び出されたとき、関数が受け取った引数によって初期化されることになります。

引数を受け取る関数を宣言する関数宣言は、

`func` 識別子 `(` 仮引数の宣言 `, ··· ) {`




```
}
```

と書きます。つまり、丸括弧の中に、仮引数の宣言を書くわけです。

仮引数の宣言は、基本的には、

識別子 : 型

と書きます。

次のプログラムの中で宣言されている argument という関数は、引数として受け取った整数を出力します。

プログラムの例　argument.swift

```
func argument(a: Int) {
    print("引数は\(a) です。")
}
```

実行例

```
  4> argument(68)
引数は 68 です。
```

仮引数は、ローカルスコープを持つことになります。つまり、仮引数のスコープは関数宣言の中だけです。

3.6.2　複数の引数を受け取る関数

複数の引数を受け取る関数を宣言したいときは、受け取る引数の個数と同じ個数の仮引数の宣言を、コンマで区切って並べます。たとえば、

```
func namako(a: Int, b: Int, c: Int) {
    ⋮
}
```

という関数宣言で宣言された namako という関数は、3 個の引数を受け取ります。

複数の引数を受け取る関数を呼び出す場合、2 個目以降の引数を渡すためには、外部引数名を使う必要があります。2 個目以降の引数の外部引数名は、原則的には仮引数名と同じです。したがって、先ほどの namako という関数を呼び出して、 a に 24、 b に 33、 c に 81 を渡したいときは、

```
namako(24, b: 33, c: 81)
```

という関数呼び出しを書けばいい、ということになります。

次のプログラムの中で宣言されている three という関数は、引数として 3 個の整数を受け取って、それらの整数を出力します。

プログラムの例　three.swift

```
func three(a: Int, b: Int, c: Int) {
    print("引数は\(a) と\(b) と\(c) です。")
}
```

実行例

```
  4> three(24, b: 33, c: 81)
引数は 24 と 33 と 81 です。
```

3.6.3　仮引数名とは異なる外部引数名

仮引数名とは異なる識別子を外部引数名にしたいときや、1 個目の引数を外部引数名を使って渡すことができるようにしたいときは、

識別子$_1$ 識別子$_2$: 型

という形の仮引数宣言を書きます。そうすると、識別子$_1$ が外部引数名、識別子$_2$ が仮引数名になります。

プログラムの例 three2.swift

```
func three2(first a: Int, second b: Int, third c: Int) {
    print("引数は\(a) と\(b) と\(c) です。")
}
```

実行例

```
  4> three2(first: 83, second: 46, third: 27)
引数は 83 と 46 と 27 です。
```

3.6.4 デフォルト値

仮引数は、関数を宣言する段階で、あらかじめ特定のデータで初期化しておくことができます。仮引数に初期値として設定されているデータは、「デフォルト値」(default value) と呼ばれます。仮引数をあらかじめデフォルト値で初期化しておくと、その仮引数に引数が渡されなかった場合、その仮引数はデフォルト値で初期化されたままになりますので、引数の代わりとしてデフォルト値が使われることになります。

仮引数をデフォルト値で初期化したいときは、仮引数の宣言として、

識別子 : 型 = 式

という形のものを書きます。そうすると、イコールの右側に書かれた式の値が、仮引数のデフォルト値になります。

プログラムの例 three3.swift

```
func three3(a: Int = 100, b: Int = 200, c: Int = 300) {
    print("引数は\(a) と\(b) と\(c) です。")
}
```

three3 という関数をこのように宣言したとすると、引数を 2 個しか渡さなかった場合には c がデフォルト値のままになり、引数を 1 個しか渡さなかった場合には b と c がデフォルト値のままになり、引数をまったく渡さなかった場合には a と b と c がデフォルト値のままになります。

実行例

```
  4> three3(83, b: 46, c: 27)
引数は 83 と 46 と 27 です。
  5> three3(83, b: 46)
引数は 83 と 46 と 300 です。
  6> three3(83, c: 27)
引数は 83 と 200 と 27 です。
  7> three3(83)
引数は 83 と 200 と 300 です。
  8> three3(b: 46, c: 27)
引数は 100 と 46 と 27 です。
  9> three3()
引数は 100 と 200 と 300 です。
```

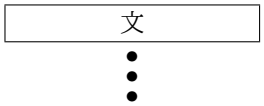
3.7 戻り値

3.7.1 戻り値の型

戻り値を返す関数を宣言するためには、その関数が返す戻り値の型を関数宣言の中に書く必要があります。

戻り値を返す関数を宣言する関数宣言は、仮引数の宣言を囲む丸括弧の右側に、-> という 2 文字を書いて、その右側に戻り値の型を書きます。つまり、

func 識別子 (仮引数の宣言 , ···) -> 型 {



}

と書くわけです。

3.7.2　return 文

戻り値を返す関数を宣言するためには、戻り値の型だけではなくて、もうひとつ、書かないといけないものがあります。

それは、「return 文」(return statement) と呼ばれる文です。

return 文は、

return 式

と書きます。この中の「式」というところには、戻り値として返したいデータを求める式を書きます。return 文は、式を評価して、その値を戻り値にして、関数の動作を終了させる、という動作をあらわしています。

a が 0 でない数値だとするとき、$1/a$ という数値は、a の「逆数」(reciprocal, multiplicative inverse) と呼ばれます。次のプログラムの中で宣言されている recipro という関数は、1 個の数値を引数として受け取って、その逆数を戻り値として返します。

プログラムの例　recipro.swift

```
func recipro(a: Double) -> Double {
    return 1/a
}
```

実行例

```
  4> recipro(4)
$R0: Double = 0.25
```

n が自然数だとするとき、1 から n までの自然数の総和は、

$$\frac{n(n+1)}{2}$$

という計算をすることによって求めることができます。次のプログラムの中で宣言されている sum という関数は、1 個の自然数を引数として受け取って、1 からその自然数までの総和を戻り値として返します。

プログラムの例　sum.swift

```
func sum(n: Int) -> Int {
    return n*(n+1)/2
}
```

実行例

```
  4> sum(10)
$R0: Int = 55
```

3.7.3　return 文を実行しないで終了した関数の戻り値

Swift では、あらゆる関数が戻り値を返します。戻り値の型が明記されておらず、return 文を実行しないで動作を終了する関数も、戻り値を返しています。その場合に関数が返すのは、「空タプル」(empty tuple) と呼ばれる、() というデータです。

実行例

```
  1> func noreturn() {}
  2> print(noreturn())
()
```

3.8 タプル

3.8.1 タプルの基礎

Swift では、「タプル」(tuple) と呼ばれるデータを扱うことができます。タプルは、0 個または 2 個以上の任意のデータを並べることによって作られる列です。

タプルを構成しているそれぞれのデータは、そのタプルの「要素」(element) と呼ばれます。

3.8.2 タプルを生成する式の書き方

タプルは、2 個以上の式をコンマで区切って並べて、その全体を丸括弧で囲むことによって作られた、

(式 , 式 , 式 , ･･･)

という形の式を評価することによって生成することができます。この形の式を評価すると、その中の式が評価されて、それらの式の値から構成されるタプルが生成されて、そのタプルが式全体の値になります。

```
  1> (583, 6.72, "namako", true)
$R0: (Int, Double, String, Bool) = {
  0 = 583
  1 = 6.7199999999999998
  2 = "namako"
  3 = true
}
```

0 個の要素から構成されるタプルは、「空タプル」(empty tuple) と呼ばれます。

空タプルは、0 個の式を丸括弧で囲んだ式を書くことによって生成することができます。

```
  1> print(())
()
```

Swift の REPL は、入力された式の値が空タプルだった場合、その値を出力しません。

```
  1> ()
  2>
```

3.8.3 タプルの型

タプルは、その要素の型から構成される複合的な型を持っています。タプルの型は、0 個または 2 個以上の型をコンマで区切って並べて、その全体を丸括弧で囲むことによって作られた、

(型 , 型 , 型 , ･･･)

という形のものによって記述されます。たとえば、

```
(583, 6.72, "namako", true)
```

というタプルの型は、

```
(Int, Double, String, Bool)
```

と記述されます。同じように、空タプルの型は、`()` と記述されます。

3.8.4 番号によるタプルの要素の取得

タプルを構成しているそれぞれの要素には、先頭から順番に、0 から始まる番号が与えられています。タプルの要素は、その番号を指定することによって、タプルから取り出すことができます。

番号を指定することによってタプルから要素を取り出したいときは、

式 . 番号

という形のものを書きます。ドット (.) の左側にはタプルを求める式、右側には取り出したい要素の番号を 10 進数で書きます。

```
  1> (61, 83, 74, 29, 35).3
$R0: Int = 29
```

3.8.5　代入によるタプルの分解

タプルは、定数や変数に要素を代入する宣言や式を書くことによって、要素に分解することができます。

定数宣言のイコールの左側、変数宣言のイコールの左側、そして単純代入演算子の左側には、識別子をコンマで区切って並べて、その全体を丸括弧で囲んだものを書くことができます。イコールの左側にそのようなものを書いて、右側にはタプルを求める式を書いた場合、そのタプルは要素に分解されて、それぞれの要素が、識別子で指定される定数や変数に代入されます。

```
  1> let (a, b, c) = (88, 47, 63)
a: Int = 88
b: Int = 47
c: Int = 63
```

タプルを構成している要素のすべてではなくて、その一部分だけを定数や変数に代入したい、というときは、識別子の代わりにアンダースコア (_) を書きます。そうすると、アンダースコアの位置の要素は、定数や変数に代入されません。

```
  1> let (a, _, _, b, _, c) = (53, 27, 44, 68, 32, 91)
a: Int = 53
b: Int = 68
c: Int = 91
```

3.8.6　要素名

タプルを構成している個々の要素には、名前として識別子を与えることができます。タプルの要素に名前として与えられた識別子は、「要素名」(element name) と呼ばれます。

タプルの要素に要素名を与えたいときは、タプルを生成する式の中に、

識別子 : 式

という形のものを書きます。そうすると、「式」のところに書いた式の値が要素になって、「識別子」のところに書いた識別子が、その要素の要素名になります。たとえば、

```
(name: "Hiroko", age: 23)
```

という式を評価すると、Hiroko という文字列と 23 という整数から構成されるタプルが生成されて、name という識別子が Hiroko の要素名になって、age という識別子が 23 の要素名になります。

タプルの要素に要素名が与えられている場合、その要素は、番号の代わりに要素名を使ってタプルから取り出すことができます。つまり、

式 . 要素名

と書けばいいわけです。ドット (.) の左側にはタプルを求める式、右側には取り出したい要素の要素名を書きます。

実行例

```
  1> let a = (name: "Hiroko", age: 23)
a: (name: String, age: Int) = {
  name = "Hiroko"
  age = 23
}
  2> a.name
$R0: String = "Hiroko"
  3> a.age
$R1: Int = 23
```

3.8.7　複数のデータを返す関数

Swift では、関数が戻り値として返すことのできるデータの個数は、1 個だけです。しかし、複数のデータから構成される 1 個のデータを戻り値として返すことは可能です。たとえば、複数のデータから構成される 1 個のタプルを生成して、そのタプルを戻り値として返すことによって、複数のデータを返す関数を宣言することができます。

次のプログラムの中で定義されている mtohm という関数は、時間の長さを分で受け取って、それを何時間と何分に分解して、それぞれの部分を要素とするタプルを返します。

プログラムの例　mtohm.swift

```
func mtohm(m: Int) -> (Int, Int) {
    return (m/60, m%60)
}
```

実行例

```
  5> mtohm(150)
$R0: (Int, Int) = {
  0 = 2
  1 = 30
}
```

戻り値としてタプルを返す関数を宣言するとき、タプルの要素の型の左側に識別子とコロンを書いておくと、その識別子が戻り値の要素の要素名になります。

プログラムの例　mtohm2.swift

```
func mtohm(m: Int) -> (hour: Int, minute: Int) {
    return (m/60, m%60)
}
```

実行例

```
  4> mtohm(150)
$R0: (hour: Int, minute: Int) = {
  hour = 2
  minute = 30
}
  5> mtohm(150).hour
$R1: Int = 2
  6> mtohm(150).minute
$R2: Int = 30
```

第 4 章　選択

4.1　選択の基礎

4.1.1　選択とは何か

第 1.4.2 項で説明したように、Swift のプログラムの中には、文を、いくつでも並べて書くことができて、コンピュータはそれらの文を、原則として、先頭から末尾に向かって 1 回ずつ実行していきます。

ですから、いくつかの文を書くことによって、まずこの動作を実行して、次にこの動作を実行して、次にこの動作を実行して……というように、複数の動作を直線的に実行していくという動作を記述することができるわけですが、コンピュータに実行させたい動作は、必ずしも一直線に進んでいくものばかりとは限りません。しばしば、そのときの状況に応じて、いくつかの動作の候補の中からひとつの動作を選んで実行するということも、必要になります。

「いくつかの動作の候補の中からひとつの動作を選んで実行する」という動作は、「選択」(selection) と呼ばれます。

4.1.2　選択を記述する方法

Swift で選択を記述する方法としては、そのための文を書くという方法と、そのための式を書くという方法があります。

選択を記述するための文としては、次の二つのものがあります。

- `if` 文 (`if` statement)
- `swicth` 文 (`switch` statement)

`if` 文については第 4.3 節で、`switch` 文については第 4.4 節で説明することにしたいと思います。

選択を記述するための式は、「条件演算子」(condition operator) と呼ばれる演算子を使って書きます。条件演算子については、第 4.6 節で説明することにしたいと思います。

4.2　比較演算子

4.2.1　比較演算子の基礎

二つのデータのあいだに何らかの関係があるという条件が成り立っているかどうかを調べる二項演算子は、「比較演算子」(comparison operator) と呼ばれます。

比較演算子の優先順位は、加算や乗算などの演算子よりも低くなっています。

比較演算子を含む式を評価すると、演算子の左右にある式が評価されて、それらの式の値のあいだに関係が成り立っているかどうかという判断が実行されます。そして、関係が成り立っているならば真、成り立っていないならば偽が、式全体の値になります。

4.2.2　大小関係

次の比較演算子を使うことによって、オブジェクトの大小関係について調べることができます。

a `>` b　　a は b よりも大きい。

a `<` b　　a は b よりも小さい。

a `>=` b　　a は b よりも大きいか、または a と b とは等しい。

a `<=` b　　a は b よりも小さいか、または a と b とは等しい。

```
  1> 8 > 5
$R0: Bool = true
  2> 5 > 8
$R1: Bool = false
  3> 5 > 5
$R2: Bool = false
  4> 5 >= 5
$R3: Bool = true
```

大小関係があるのは、数値と数値とのあいだだけではありません。文字列と文字列とのあいだにも大小関係があります。

辞書の見出しは、「辞書式順序」(lexicographical order) と呼ばれる順序で並べられています。文字列と文字列とのあいだの大小関係は、辞書式順序で文字列を並べたときに、後ろにあるものは前にあるものよりも大きい、という関係です。

```
  1> "stay" > "star"
$R0: Bool = true
  2> "star" > "stay"
$R1: Bool = false
```

4.2.3　等しいかどうか

二つのオブジェクトが等しいかどうかということは、次の比較演算子を使うことによって調べることができます。

a `==` b　　a と b とは等しい。

a `!=` b　　a と b とは等しくない。

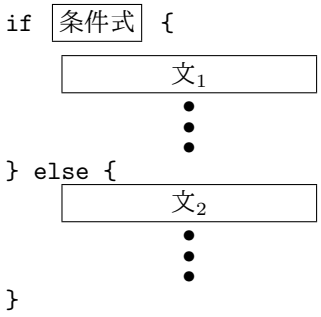
```
  1> 5 == 5
$R0: Bool = true
  2> 5 == 8
$R1: Bool = false
  3> "star" == "star"
$R2: Bool = true
  4> "star" == "stay"
$R3: Bool = false
```

4.3　`if` 文

4.3.1　`if` 文の基礎

何らかの条件が成り立っているかどうかを調べて、その結果にもとづいて二つの動作のうちのどちらかを実行したい、というときは、「`if` 文」(`if` statement) と呼ばれる文を書きます。

`if` 文は、



と書きます。この中の「条件式」のところには、条件をあらわす式、つまり、評価すると値として真偽値が得られる式を書きます。

if 文を実行すると、まず最初に、条件式が評価されます。そして、条件式の値が真だった場合は、$文_1$ とそれに続く文が実行されます（その場合、$文_2$ とそれに続く文は実行されません）。条件式の値が偽だった場合は、$文_2$ とそれに続く文が実行されます（その場合、$文_1$ とそれに続く文は実行されません）。

つまり、if 文は、「もしも条件式が真ならば else よりも上の文を実行して、そうでなければ else よりも下の文を実行する」という動作を意味しているわけです。

それでは、if 文を REPL に入力してみましょう。

```
1> if 8 > 5 {
2.     print("namako")
3. } else {
4.     print("hitode")
5. }
namako
```

この場合、条件式の値は真ですので、namako が出力されて、hitode は出力されません。

```
1> if 5 > 8 {
2.     print("namako")
3. } else {
4.     print("hitode")
5. }
hitode
```

この場合、条件式の値は偽ですので、hitode が出力されて、namako は出力されません。

次のプログラムの中で宣言されている evenodd という関数は、整数を受け取って、それが偶数ならば「偶数」という文字列を返して、そうでなければ「奇数」という文字列を返します。

プログラムの例 evenodd.swift

```
func evenodd(n: Int) -> String {
    if n%2 == 0 {
        return "偶数"
    } else {
        return "奇数"
    }
}
```

実行例

```
  8> evenodd(6)
$R0: String = "偶数"
  9> evenodd(7)
$R1: String = "奇数"
```

4.3.2 else 以降を省略した if 文

二つの動作のうちのどちらかを選択するのではなくて、ひとつの動作を実行するかしないかを選択したい、ということもしばしばあります。そのような場合は、else 以降を省略した if 文を書きます。つまり、

```
if 条件式 {
        文
        ●
        ●
        ●
}
```

という形の if 文を書くわけです。この形の if 文を実行すると、条件式の値が真の場合は文の列が実行されますが、条件式の値が偽の場合は何も実行されません。

REPL を使って試してみましょう。

```
  1> if 8 > 5 {
  2.     print("namako")
  3. }
namako
  4> if 5 > 8 {
  5.     print("namako")
  6. }
  7>
```

次のプログラムの中で宣言されている mtohm という関数は、分を単位とする時間の長さを受け取って、それを「何時間何分」という形式の文字列に変換した結果を返します。ただし、「何分」という端数が出ない場合は、「何時間」という形式の文字列を返します。

プログラムの例　mtohm3.swift

```
func mtohm(m: Int) -> String {
    var hm = (m/60).description + "時間"
    if m%60 != 0 {
        hm += (m%60).description + "分"
    }
    return hm
}
```

実行例

```
  8> mtohm(160)
$R0: String = "2 時間 40 分"
  9> mtohm(180)
$R1: String = "3 時間"
```

「何時間」の部分の右側に「何分」の部分を連結するという動作は、実行するかしないかを選択することになりますので、このように、else 以降を省略した if 文を使って書くことができます。

4.3.3　多肢選択

選択の対象となる動作が 3 個以上あるような選択は、「多肢選択」(multibranch selection) と呼ばれます。多肢選択には、次の二つのタイプがあります。

- ひとつの式の値が何なのかということによって動作を選択するタイプ。
- いくつかの条件のうちのどれが真になるかということによって動作を選択するタイプ。

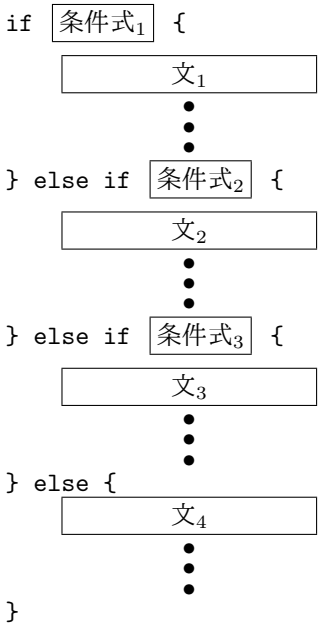
ひとつの式の値による多肢選択は、「switch 文」(switch statement) と呼ばれる文を使うことによって記述することができます（ switch 文については次の節で説明します)。

いくつかの条件による多肢選択は、 if 文の中に、

```
else if 条件式 {
        文
        ●
        ●
        ●
}
```

という形のものをいくつか書くことによって記述することができます。

たとえば、3 個の条件による多肢選択は、



という形の if 文を書くことによって記述することができます。この形の if 文を実行すると、値が真になる条件式が見つかるまで、条件式 $_1$、条件式 $_2$、条件式 $_3$ という順番で条件式が評価されていきます。値が真になる条件式が見つかった場合は、その条件式と同じ番号の文とそれに続く文が実行されます（その場合、それ以降の条件式は評価されません）。すべての条件式の値が偽だった場合は、文$_4$ とそれに続く文が実行されます。

REPL を使って試してみましょう。

```
1> if 5 > 8 {
2.     print("namako")
3. } else if 8 > 5 {
4.     print("hitode")
5. } else {
6.     print("umiushi")
7. }
hitode
```

次のプログラムの中で宣言されている sign という関数は、整数を受け取って、それがプラスならば「プラス」という文字列を返して、そうでなくてマイナスならば「マイナス」という文字列を返して、どちらでもないならば「ゼロ」という文字列を返します。

プログラムの例 sign.swift

```
func sign(a: Int) -> String {
    if a > 0 {
        return "プラス"
    } else if a < 0 {
        return "マイナス"
    } else {
        return "ゼロ"
    }
}
```

実行例

```
 10> sign(5)
$R0: String = "プラス"
 11> sign(-5)
$R1: String = "マイナス"
 12> sign(0)
$R2: String = "ゼロ"
```

4.4　switch 文

4.4.1　switch 文の基礎

前の節で説明したように、ひとつの式の値が何なのかということによって動作を選択するタイプの多肢選択は、「switch 文」(switch statement) と呼ばれる文を使うことによって記述することができます。

switch 文は、

```
switch 式 {
    選択肢
    •
    •
    •
}
```

と書きます。switch 文を実行すると、switch の右に書かれた式が評価されて、その値が何なのかということによって、選択肢の中からひとつが選択されて、それが実行されます。

4.4.2　選択肢の書き方

switch 文の中には、式の値によって選択されるいくつかの選択肢を書く必要があります。選択肢として書くことができるものとしては、次の 2 種類のものがあります。

- case 節 (case clause)
- default 節 (default clause)

case 節は、

```
case リテラル:
    文
    •
    •
    •
```

と書きます。 case 節の先頭に書かれる、

```
case リテラル:
```

という部分は、「case ラベル」(case label) と呼ばれます。

case 節で書かれた選択肢は、 switch の右に書かれた式の値と case ラベルの中に書かれたリテラルの値とが一致した場合に選択されて、その中の文の列が実行されます。

default 節は、

```
default:
    文
    •
    •
    •
```

と書きます。これは、 switch 文の最後の選択肢として書いておく必要があります[1]。 default 節は、 case 節で書かれたどの選択肢も選択されなかった場合（つまり、 case ラベルの中に書かれたどのリテラルの値も、 switch の右に書かれた式の値と一致しなかった場合）に選択されて、その中の文の列が実行されます。

次のプログラムの中で宣言されている ntoweek という関数は、曜日の番号（1 が月曜日、2 が火曜日、……）を引数として受け取って、その番号があらわしている曜日を出力します。

プログラムの例　ntoweek.swift

```
func ntoweek(n: Int) {
    switch n {
        case 1:
            print("月曜日")
        case 2:
```

[1] case 節による選択肢がすべての可能性を網羅している場合、 default 節は書かなくてもかまいません。

```
            print("火曜日")
        case 3:
            print("水曜日")
        case 4:
            print("木曜日")
        case 5:
            print("金曜日")
        case 6:
            print("土曜日")
        case 7:
            print("日曜日")
        default:
            print("未定義番号")
    }
}
```

実行例

```
 21> ntoweek(3)
水曜日
 22> ntoweek(8)
未定義番号
```

4.4.3 複数のリテラルを持つ case ラベル

switch 文の選択肢としては、いくつかのデータのどれかひとつと式の値とが一致した場合に選択されるものを作ることもできます。

そのような選択肢を作りたい場合は、case ラベルの中に、コンマで区切って、複数のリテラルを書きます。そうすると、その case ラベルを持つ選択肢は、case ラベルの中に書かれたリテラルの値のうちのどれかひとつが、switch の右に書かれた式の値と一致した場合に選択されます。

次のプログラムの中で宣言されている month という関数は、月の番号を引数として受け取って、その番号があらわしている月が大の月ならば「大の月」、小の月ならば「小の月」、存在しない月ならば「存在しない月」という文字列を出力します。

プログラムの例 month.swift

```
func month(m: Int) {
    switch m {
        case 1, 3, 5, 7, 8, 10, 12:
            print("大の月")
        case 2, 4, 6, 9, 11:
            print("小の月")
        default:
            print("存在しない月")
    }
}
```

実行例

```
 11> month(8)
大の月
 12> month(9)
小の月
```

4.4.4 範囲による選択肢

switch 文の選択肢としては、式の値と一致した場合に選択されるものだけではなく、式の値がデータの範囲の中にある場合に選択されるものを作ることもできます。

そのような選択肢を作りたい場合は、case ラベルの中に、

$\boxed{\text{リテラル}_1}$ ... $\boxed{\text{リテラル}_2}$

という形のものを書きます。そうすると、リテラル_1 の値を下限、リテラル_2 の値を上限とする範囲の中に、switch の右に書かれた式の値が入っている場合に、その選択肢が選択されます。たとえば、

```
    case 30...70:
```

という case ラベルを書くことによって、switch の右に書かれた式の値が 30 から 70 までの範囲に入っている場合に選択される選択肢を作ることができます。

次のプログラムの中で宣言されている agegroup という関数は、引数として年齢を受け取って、その年齢の年齢層を出力します。

プログラムの例　agegroup.swift

```
func agegroup(age: Int) {
    switch age {
        case 0...4:
            print("幼年期")
        case 5...14:
            print("少年期")
        case 15...24:
            print("青年期")
        case 25...44:
            print("壮年期")
        case 45...64:
            print("中年期")
        default:
            print("高年期")
    }
}
```

実行例

```
 17> agegroup(21)
青年期
 18> agegroup(82)
高年期
```

4.5　論理演算子

4.5.1　論理演算子の基礎

引数が真偽値で戻り値も真偽値であるような演算は「論理演算」(logical operation) と呼ばれ、そのような演算に与えられた名前は「論理演算子」(logical operator) と呼ばれます。

Swift には、次の三つの論理演算子があります。

a && b　　a かつ b である。

a || b　　a または b である。

! a　　a ではない。

&& と || は、関係演算子よりも低くて代入演算子よりも高い優先順位を持っています。そして、&& は、|| よりも高い優先順位を持っています。

4.5.2　論理積演算子

&& は、「論理積演算子」(logical AND operator) と呼ばれます。これは、二つの条件が両方とも成り立っているかどうかを判断したいとき、つまり、A かつ B という条件が成り立っているかどうかを判断したいときに使われる論理演算子で、次のような動作をします。

```
true   &&  true     ⟶  true
true   &&  false    ⟶  false
false  &&  true     ⟶  false
false  &&  false    ⟶  false
```

4.5.3　論理和演算子

|| は、「論理和演算子」(logical OR operator) と呼ばれます。これは、二つの条件のうちの少なくともひとつが成り立っているかどうかを判断したいとき、つまり、A または B という条件が成り立っているかどうかを判断したいときに使われる論理演算子で、次のような動作をします。

```
true   ||  true     ⟶  true
true   ||  false    ⟶  true
false  ||  true     ⟶  true
false  ||  false    ⟶  false
```

次のプログラムの中で宣言されている leapyear という関数は、引数として西暦の年を受け取って、その年がうるう年ならば「うるう年」という文字列を、うるう年ではないならば「平年」という文字列を戻り値として返します。

プログラムの例 leapyear.swift

```
func leapyear(y: Int) -> String {
    if y%4 == 0 && y%100 != 0 || y%400 == 0 {
        return "うるう年"
    } else {
        return "平年"
    }
}
```

実行例

```
  8> leapyear(2080)
$R0: String = "うるう年"
  9> leapyear(2100)
$R1: String = "平年"
 10> leapyear(2400)
$R2: String = "うるう年"
```

4.5.4 論理否定演算子

! は、「論理否定演算子」(logical negation operator) と呼ばれます。これは、真偽値を反転させたいとき、つまり、A ではないという条件が成り立っているかどうかを判断したいときに使われる論理演算子で、次のような動作をします。

```
! true     ⟶  false
! false    ⟶  true
```

4.6 条件演算子

4.6.1 条件演算子の基礎

これまで説明してきたように、Swift には、 if 文と switch 文という、選択を記述するための文があります。しかし、選択を記述する方法は、 if 文または switch 文を書くというものだけではありません。1 個の式を書くことによって選択を記述する、ということも可能です。

選択をあらわす式は、「条件演算子」(conditional operator) と呼ばれる、 ?: という三項演算子を使うことによって記述されます。

4.6.2 条件演算子を含む式

条件演算子を含む式は、

条件式 ? 式$_1$: 式$_2$

と書きます。「条件式」のところには、評価すると値として真偽値が得られる式を書きます。

条件演算子を含む式を評価すると、まず最初に、条件式が評価されます。そして、条件式の値が真だった場合は、式$_1$ が評価されます（その場合、式$_2$ は評価されません）。条件式の値が偽だった場合は、式$_2$ が評価されます（その場合、式$_1$ は評価されません）。そして、評価された式$_1$ または式$_2$ の値が、条件演算子を含む式の全体の値になります。

REPL で試してみましょう。

```
  1> true ? "namako" : "umiushi"
$R0: String = "namako"
  2> false ? "namako" : "umiushi"
$R1: String = "umiushi"
```

プログラムの例　hundred.swift

```
func hundred(a: Int) -> String {
    return a.description + "は 100" +
        ((a >= 100) ? "以上" : "未満") + "です。"
}
```

実行例

```
  5> hundred(120)
$R0: String = "120 は 100 以上です。"
  6> hundred(80)
$R1: String = "80 は 100 未満です。"
```

第 5 章　繰り返しと再帰

5.1　繰り返しの基礎

5.1.1　繰り返しとは何か

コンピュータに実行させたい動作は、必ずしも、一連の動作をそれぞれ一回ずつ実行していけばそれで達成される、というものばかりとは限りません。しばしば、ほとんど同じ動作を何回も何十回も何百回も実行しなければ意図していることを達成できない、ということがあります。

「同じ動作を何回も実行する」という動作は、「繰り返し」(iteration) と呼ばれます。

この章では、繰り返しというのはどのように記述すればいいのか、ということについて説明します。

5.1.2　繰り返しを記述するための文

繰り返しは、繰り返したい回数と同じ個数の文を書くことによって記述することも可能ですが、そのような書き方だと、繰り返しの回数に比例してプログラムが長くなってしまいます。ですから、多くのプログラミング言語は、繰り返しを簡潔に記述することができるようにする機能を持っています。

Swift では、次の 3 種類の文のうちのどれかを使うことによって、繰り返しを簡潔に記述することができます。

- `for-in` 文 (`for-in` statement)
- `while` 文 (`while` statement)
- `repeat-while` 文 (`repeat-while` statement)

`for-in` 文については第 5.2 節で、`while` 文については第 5.3 節で、`repeat-while` 文については第 5.4 節で説明することにしたいと思います。

5.2　`for-in` 文

5.2.1　`for-in` 文の基礎

`for-in` 文 (`for-in` statement) は、データが集まってできているデータに対して、そこからひとつのデータを取り出して、そのデータに対して何かを実行する、ということを繰り返したいときに使われる文です。

ただし、データが集まってできているデータならば、どんなものでも `for-in` 文を使って繰り返しを記述することができる、というわけではありません。たとえば、タプルというのは、データが集まってできているデータですが、`for-in` 文を使って、タプルから要素を取り出して何かを実行するということを繰り返すことはできません。

`for-in` 文を使って繰り返しを記述することができるデータのことを、「繰り返し可能データ」(iterable data) と呼ぶことにしたいと思います。

5.2.2 範囲

繰り返し可能データに分類されるデータとしては、「範囲」(range) と呼ばれるものがあります。範囲は、このデータからこのデータまで、というように、下限と上限を指定することによって生成されるデータの列です。

範囲の型は、

```
Range<型>
```

と記述されます。この中の「型」のところには、範囲を構成しているデータの型が入ります。たとえば、整数から構成される範囲の型は、

```
Range<Int>
```

と記述されます。

範囲は、「範囲演算」(range operation) と呼ばれる演算を使うことによって生成することができます。範囲演算子に与えられた名前は、「範囲演算子」(range operator) と呼ばれます。

範囲演算子には、次の二つのものがあります。

a`...`*b*　*a* から *b* までで、*a* も *b* も含む。たとえば、`3...6` は、3、4、5、6 という列。「閉範囲演算子」(closed range operator) と呼ばれる。

a`..<`*b*　*a* から *b* までで、*a* は含むが、*b* は含まない。たとえば、`3..<6` は、3、4、5 という列。「半開範囲演算子」(half-open range operator) と呼ばれる。

5.2.3 for-in 文の書き方

for-in 文は、

```
for 識別子 in 式 {
    文
    ●
    ●
    ●
}
```

と書きます。この中の「式」のところには、値として繰り返し可能データが得られる式を書きます。

for-in 文を実行すると、まず最初に、in の右側の式が 1 回だけ評価されます。そして、その式の値として得られた繰り返し可能データを構成している 1 個 1 個のデータに対して、for-in 文の中に書かれた文の列が実行されます。文の列が実行される直前には、毎回、定数が宣言されて、繰り返し可能データから取り出されたデータで、その定数が初期化されます。for の右側の識別子は、その定数の名前になります。たとえば、

```
for i in 3...6 {
    print(i)
}
```

という for-in 文を実行すると、まず最初に、`3...6` という式が評価されて、3 から 6 までの範囲が値として得られます。そして、その範囲を構成している 1 個 1 個の整数に対して、

```
print(i)
```

という文が実行されます。この文が実行される直前には、毎回、定数が宣言されて、3、4、5、6 という整数のそれぞれで、その定数が初期化されます。i という識別子は、その定数の名前になります。

REPL を使って試してみましょう。

```
1> for i in 3...6 {
2.     print(i)
3. }
3
4
5
6
```

次のプログラムの中で宣言されている divisor という関数は、引数としてプラスの整数を受け取って、そのすべての約数を出力します。

プログラムの例　divisor.swift

```
func divisor(n: Int) {
    for i in 1...n {
        if n%i == 0 {
            print(i, terminator: " ")
        }
    }
    print("")
}
```

実行例

```
  9> divisor(96)
1 2 3 4 6 8 12 16 24 32 48 96
```

5.3　while 文

5.3.1　while 文の基礎

繰り返しを記述したいときは、基本的には for-in 文を使えばいいわけですが、 for-in 文では記述することが困難な繰り返しも存在します。たとえば、条件による繰り返しは、 for-in 文では記述することが困難です。

条件による繰り返しというのは、繰り返しの対象となる動作を実行するたびに、繰り返しを続行するか終了するかということを、何らかの条件が成り立っているかどうかを判断することによって決定する、というタイプの繰り返しのことです。このようなタイプの繰り返しは、 for-in 文では記述することが困難です。

そこで登場するのが、条件による繰り返しを表現するために存在する、「 while 文」(while statement) と呼ばれる文と、「 repeat-while 文」(repeat-while statement) と呼ばれる文です。

ただし、この節で説明するのは while 文だけです。 repeat-while 文については、次の節で説明することにしたいと思います。

5.3.2　while 文の書き方

while 文は、

```
while 条件式 {
        文
        •
        •
        •
}
```

と書きます。この中の「条件式」のところには、条件をあらわす式、つまり、評価すると値として真偽値が得られる式を書きます。

while 文は、次のような動作をあらわしています。

(1) 条件式を評価する。その値が偽だった場合、 while 文の動作は終了する。

(2) 条件式の値が真だった場合は、文の列を実行する。

(3) (1) に戻って、ふたたび同じ動作を実行する。

5.3.3　無限ループ

while 文を使うと、永遠に終わらない繰り返しというものを記述することも可能になります。永遠に終わらない繰り返しは、「無限ループ」(infinite loop) と呼ばれます。

true という真偽値リテラルを評価すると、真を意味するデータが値として得られますので、 while 文の条件式としてそれを書くと、その while 文は無限ループになります。

たとえば、次の while 文は、 kurage という文字列の出力を永遠に繰り返します。

```
while true {
```

```
        print("kurage")
    }
```

この while 文の実行を終了させたいときは、Ctrl-C、つまりコントロールキーを押しながら C のキーを押す、という操作をします。

5.3.4 条件による繰り返しの例

条件による繰り返しの例として、二つの整数の最大公約数を求めるという処理について考えてみることにしましょう。

n がプラスの整数で、m が 0 またはプラスの整数だとするとき、n と m の両方に共通する約数のうちで最大のもののことを、n と m の「最大公約数」(greatest common measure, GCM) と呼びます（m が 0 の場合は、n と m の最大公約数は n だと定義します)。たとえば、54 と 36 の最大公約数は 18 です。

二つの整数の最大公約数は、「ユークリッドの互除法」(Euclidean algorithm) と呼ばれる方法を使えば、きわめて簡単に求めることができます。ユークリッドの互除法というのは、

ステップ 1　与えられた二つの整数のそれぞれを、n と m という変数に代入する。

ステップ 2　m が 0 ならば計算を終了する。

ステップ 3　n を m で除算して、そのあまりを r という変数に代入する。

ステップ 4　m を n に代入する。

ステップ 5　r を m に代入する。

ステップ 6　ステップ 2 に戻る。

という計算を実行していけば、計算が終了したときの n が、最初に与えられた二つの整数の最大公約数になっている、というものです。ステップ 2 からステップ 6 までは、

　m が 0 ではないあいだ、ステップ 3 からステップ 5 までを繰り返す。

ということだと考えることができますので、その部分は、while 文を書くことによって記述することができます。

次のプログラムの中で宣言されている gcm という関数は、二つの整数を受け取って、それらの最大公約数を返します。

プログラムの例　gcm.swift

```
func gcm(a: Int, b: Int) -> Int {
    var n: Int = a
    var m: Int = b
    var r: Int
    while m != 0 {
        r = n%m
        n = m
        m = r
    }
    return n
}
```

実行例

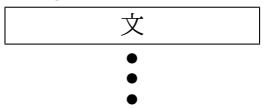
```
 12> gcm(54, b: 36)
$R0: Int = 18
```

5.4 repeat-while 文

5.4.1 repeat-while 文の基礎

repeat-while 文は、

```
repeat {
```



```
} while 条件式
```

と書きます。「条件式」のところには、評価すると値として真偽値が得られる式を書きます。

repeat-while 文は、次のような動作をあらわしています。

(1) 文の列を実行する。
(2) 条件式を評価する。その値が偽だった場合、repeat-while 文の動作は終了する。
(3) 条件式の値が真だった場合は、(1) に戻って、ふたたび同じ動作を実行する。

5.4.2 while 文と repeat-while 文の相違点

while 文と repeat-while 文は、どちらも、条件式の値が真であるあいだだけ文の列の実行を繰り返す、という動作をあらわしています。では、この 2 種類の文は、どこに相違点があるのでしょうか。

while 文と repeat-while 文の相違点は、「最初に何をするか」というところにあります。while 文の場合、最初の動作は「条件式の評価」です。それに対して、repeat-while 文の場合、最初の動作は「文の列の実行」です。

その結果として、while 文と repeat-while 文とでは、条件式の値が最初から偽だった場合に異なる動作をすることになります。

REPL を使って、動作を調べてみましょう。まず、次のプログラムを REPL に入力してみてください。

```
var i = 10
while i <= 5 {
    print(i)
    i += 1
}
```

このプログラムは、何も出力しないで終了します。その理由は、

```
i <= 5
```

という条件式の値が最初から偽だからです。

それでは、次のプログラムを REPL に入力すると、どうなるでしょうか。

```
var i = 10
repeat {
    print(i)
    i += 1
} while i <= 5
```

このプログラムは、条件式の値が最初から偽であるにもかかわらず、10 を出力してから終了します。その理由は、条件式の評価よりも先に文の列が実行されるからです。

5.5 break 文と continue 文

5.5.1 break 文

for-in 文や while 文や repeat-while 文を使って繰り返しを記述するとき、場合によっては、何らかの条件が成り立ったときに繰り返しを途中で終了するようにしたい、ということがあります。そのようなときに使われるのが、「break 文」(break statement) と呼ばれる文です。

break 文は、

```
break;
```

と書きます。

for-in 文、while 文、repeat-while 文の中に break 文を書いておくと、それが実行されたとき、for-in 文、while 文、repeat-while 文の実行は終了します。

REPL を使って試してみましょう。

```
1> for i in 1...5 {
2.     if i == 4 {
3.         break;
4.     }
```

```
    5.      print(i)
    6. }
1
2
3
```

5.5.2 continue 文

break 文は、for-in 文や while 文や repeat-while 文による繰り返しを途中で終了させたいときに使われるわけですが、繰り返しを終了させるのではなくて、繰り返しの対象となっている動作の実行をスキップしたい、ということもあります。そのようなときに使われるのが、「continue 文」(continue statement) と呼ばれる文です。

continue 文は、

```
continue;
```

と書きます。

for-in 文、while 文、repeat-while 文の中に continue 文を書いておくと、それが実行された場合だけ、繰り返しの対象となっている動作のうちで、それ以降の部分の実行がスキップされます。

REPL を使って試してみましょう。

```
1> for i in 1...5 {
2.      if i == 3 {
3.          continue;
4.      }
5.      print(i)
6. }
1
2
4
5
```

5.6 再帰

5.6.1 再帰とは何か

この節では、「再帰」(recursion) と呼ばれるものについて説明したいと思います。

再帰というのは、全体と同じものが一部分として含まれているという性質のことです。再帰という性質を持っているものは、「再帰的な」(recursive) と形容されます。

ここに、1 台のカメラと 1 台のモニターがあるとします。まず、それらを接続して、カメラで撮影した映像がモニターに映し出されるようにします。そして次に、カメラをモニターの画面に向けます。すると、モニターの画面には、そのモニター自身が映し出されることになります。そして、映し出されたモニターの画面の中には、さらにモニター自身が映し出されています。このときにモニターの画面に映し出されるのは、再帰という性質を持っている映像、つまり再帰的な映像です。

また、先祖と子孫の関係も再帰的です。なぜなら、先祖と子孫との中間にいる人々も、やはり先祖と子孫の関係で結ばれているからです。

5.6.2 基底

再帰という性質を持っているものは、全体と同じものが一部分として含まれているわけですが、その構造は、内部に向かってどこまでも続いている場合もあれば、どこかで終わっている場合もあります。

再帰的な構造がどこかで終わっている場合、その中心には、その内部に再帰的な構造を持っていない何かがあります。そのような、再帰的な構造の中心にあって、その内部に再帰的な構造を持っていないものは、その再帰的な構造の「基底」(basis) と呼ばれます。

先祖と子孫の関係では、親子関係というのが、その再帰的な構造の基底になります。

5.6.3　関数の再帰的な定義

関数は、再帰的に定義することが可能です。関数を再帰的に定義するというのは、定義される当の関数を使って関数を定義するということです。再帰的な構造を持っている概念を取り扱う関数は、再帰的に定義するほうが、再帰的ではない方法で定義するよりもすっきりした記述になります。

関数を再帰的に定義する場合は、それが循環に陥ることを防ぐために、基底について記述した選択肢を作っておくことが必要になります。

5.6.4　階乗

n が 0 またはプラスの整数だとするとき、n から 1 までの整数をすべて乗算した結果、つまり、

$$n \times (n-1) \times (n-2) \times \cdots \times 1$$

という計算の結果は、n の「階乗」(factorial) と呼ばれて、$n!$ と書きあらわされます。ただし、$0!$ は 1 だと定義します。

たとえば、$5!$ は、

$$5 \times 4 \times 3 \times 2 \times 1$$

という計算をすればいいわけですから、120 ということになります。

階乗という概念は、再帰的な構造を持っています。なぜなら、階乗は、

$$\begin{cases} & 0! = 1 \\ n \geq 1 \text{ ならば} & n! = n \times (n-1)! \end{cases}$$

というように再帰的に定義することができるからです。

階乗を求める関数も、再帰的に定義することができます。次のプログラムは、階乗を求める `factorial` という関数を再帰的に定義しています。

プログラムの例　`factorial.swift`

```
func factorial(n: Int) -> Int {
    if n == 0 {
        return 1
    } else if n >= 1 {
        return n * factorial(n-1)
    } else {
        return 0
    }
}
```

実行例

```
  8> factorial(5)
$R0: Int = 120
```

5.6.5　フィボナッチ数列

第 0 項と第 1 項が 1 で、第 2 項以降はその直前の 2 項を足し算した結果である、という数列は、「フィボナッチ数列」(Fibonacci sequence) と呼ばれます。フィボナッチ数列の第 0 項から第 12 項までを表にすると、次のようになります。

n	0	1	2	3	4	5	6	7	8	9	10	11	12
第 n 項	1	1	2	3	5	8	13	21	34	55	89	144	233

フィボナッチ数列というのは再帰的な構造を持っている概念ですので、その第 n 項 (F_n) は、

$$\begin{cases} & F_0 = 1 \\ & F_1 = 1 \\ n \geq 2 \text{ ならば} & F_n = F_{n-2} + F_{n-1} \end{cases}$$

というように再帰的に定義することができます。

フィボナッチ数列の第 n 項を求める関数も、再帰的に定義することができます。次のプログラムは、フィボナッチ数列の第 n 項を求める `fibonacci` という関数を再帰的に定義しています。

プログラムの例　fibonacci.swift

```
func fibonacci(n: Int) -> Int {
    if n == 0 || n == 1 {
        return 1
    } else if n >= 2 {
        return fibonacci(n-2) + fibonacci(n-1)
    } else {
        return 0
    }
}
```

実行例

```
  8> fibonacci(7)
$R0: Int = 21
```

5.6.6　最大公約数

第 5.3.4 項で、条件による繰り返しの例として、ユークリッドの互除法を使って二つの整数の最大公約数を求めるという処理について説明しましたが、ユークリッドの互除法は、二つの整数を n と m とするとき、次のように再帰的に記述することも可能です。

- m が 0 ならば、n が、n と m の最大公約数である。
- m が 1 以上ならば、n を m で除算したときのあまりを求めて、その結果を r とする。そして、m と r の最大公約数を求めれば、その結果が n と m の最大公約数である。

次のプログラムは、2 個のプラスの整数の最大公約数を求める gcm という関数を、ユークリッドの互除法を使って再帰的に定義しています。

プログラムの例　gcm2.swift

```
func gcm(n: Int, m: Int) -> Int {
    if m == 0 {
        return n
    } else if m >= 1 {
        return gcm(m, m: n%m)
    } else {
        return 0
    }
}
```

実行例

```
  8> gcm(54, m: 36)
$R0: Int = 18
```

第 6 章　オプショナル

6.1　オプショナルの基礎

6.1.1　nil

Swift では、「nil」と呼ばれるデータを扱うことができます。これは、「データが存在しない」ということを意味するデータです。

nil というキーワードは、式として評価することができて、評価すると、その値として nil が得られます。ただし、REPL にいきなり nil と入力しても、エラーメッセージが出力されるだけです。

そのときに出力されるエラーメッセージが示しているように、nil というキーワードを式として評価することができるのは、それの評価が可能な文脈の中にそのキーワードがあるときだけです。

6.1.2　オプショナル型

nil というキーワードを式として評価することのできる文脈というのは、「オプショナル」(optional) と呼ばれるデータが必要とされる文脈のことです。

オプショナルというのは、「オプショナル型」(optional type) と呼ばれる型を持つデータのことです。

オプショナル型は、既存の何らかの型に nil を加えた型です。T という型があるとするとき、それに nil を加えることによってできるオプショナル型を、「オプショナルな T 型」と呼んで、

```
Optonal<T> または T?
```

と書きます。

オプショナルな T 型の変数には、T のデータだけではなくて、nil を代入することもできます。

```
  1> var a: Int? = 37
a: Int? = 37
  2> print(a)
Optional(37)
  3> a = nil
  4> print(a)
nil
```

このように、print は、受け取った引数がオプショナルだった場合、

```
Optional(...)
```

という形でそれを出力します。

オプショナルというのは、箱のようなものだと考えることができます。この箱には、1 個のデータを入れることができます。また、データを入れないで空のままにしておくことも可能です。nil というのは、オプショナルという箱が空になっているということを示すデータのことです。

6.1.3　オプショナルと比較演算子

オプショナルの中のデータは、== または != を使うことによって、何らかのデータと等しいかどうかを調べることができます。また、数値や文字列のような大小関係の比較ができるデータのオプショナルについては、その中のデータと何らかのデータとのあいだの大小関係を比較することも可能です。

```
  1> var a: Int? = 37
a: Int? = 37
  2> a == 37
$R0: Bool = true
  3> a > 50
$R1: Bool = false
```

オプショナルは、nil と比較することも可能です。大小関係を調べる演算子でオプショナルと nil を比較した場合、nil は、どのデータよりも小さいデータとして扱われます。

```
  1> var a: Int? = 37
a: Int? = 37
  2> a == nil
$R0: Bool = false
  3> a > nil
$R1: Bool = true
  4> a = nil
  5> a == nil
$R2: Bool = true
```

6.2　アンラップ

6.2.1　アンラップの基礎

オプショナルは、そのままでは、本来の型のデータとして処理することができません。たとえば、Int? という型のオプショナルは、たとえそれが nil ではなかったとしても、それに対して加算や乗算をすることはできません。

オプショナルを、本来の型のデータとして処理するためには、オプショナルという箱から本来の型のデータを取り出す必要があります。オプショナルという箱から本来の型のデータを取り出すことを、オプショナルを「アンラップする」(unwrap) と言います。

オプショナルという箱が空の場合、つまりそのオプショナルが nil の場合、それをアンラップすることはできません。

6.2.2 感嘆符

値としてオプショナルが得られる式の右側に感嘆符 (!) を加えた式を評価すると、その値として、オプショナルをアンラップしたデータが得られます。

REPL を使って試してみましょう。

```
  1> var a: Int? = 37
a: Int? = 37
  2> a!
$R0: Int = 37
```

アンラップのための感嘆符を書く場合、その左側の式との間に、`a !`というように空白を入れることはできません。かならず、`a!`というように詰めて書く必要があります。

6.2.3 nil 合体演算子

オプショナルをアンラップする方法は、感嘆符を使うというものだけではありません。「nil 合体演算子」(nil coalescing operator) と呼ばれる 2 項演算子を使うことによっても、オプショナルをアンラップすることができます。

オプショナルが nil の場合、感嘆符によるアンラップはエラーになります。それに対して、nil 合体演算子によるアンラップでは、オプショナルが nil の場合でもエラーにはなりません。

nil 合体演算子は 2 項演算子ですので、2 個の引数を受け取ります。左側の引数はオプショナルで、右側の引数は、左側の引数が nil だった場合に返すデータです。左側の引数の型が`T?`だとすると、右側の引数の型は`T`または`T?`である必要があります。

```
  1> var a: Int? = 37
a: Int? = 37
  2> a ?? -1
$R0: Int = 37
  3> a = nil
  4> a ?? -1
$R1: Int = -1
```

6.3 オプショナルを返す関数

6.3.1 この節について

この節では、まず、組み込み関数の中にある、オプショナルを返す関数を紹介します。そしてそののち、オプショナルを返す関数の宣言の書き方について説明したいと思います。

6.3.2 文字列から整数への変換

データを生成する関数は、「イニシャライザー」(initializer) と呼ばれます。

Swift の処理系には、型名と同じ名前の関数が組み込まれています。そのような関数は、その名前の型のデータを生成するイニシャライザーです。

`Int`のデータを生成する`Int`というイニシャライザーは、文字列を`Int`のデータに変換するために使うことができます。`Int`を呼び出して、引数として文字列をそれに渡すと、`Int`は、その文字列を整数に変換した結果を返します。ただし、その場合に`Int`が返すデータの型は、`Int`ではなくて`Int?`です。もしも、整数に変換することができない文字列を引数として受け取った場合、`Int`は、戻り値として nil を返します。

```
  1> Int("37")
$R0: Int? = 37
  2> Int("namako")
$R1: Int? = nil
```

6.3.3 文字列から浮動小数点数への変換

`Int`の場合と同じように、`Double`というイニシャライザーを使うことによって、文字列を浮動小数点数に変換することもできます。`Double`の戻り値の型は`Double?`です。

```
  1> Double("3.14")
```

```
    $R0: Double? = 3.1400000000000001
      2> Double("namako")
    $R1: Double? = nil
```

6.3.4　文字列の読み込み

readLine という組み込み関数は、標準入力から 1 行の文字列を読み込んで、それを戻り値として返します。

readLine の戻り値の型は String? です。読み込みがファイルの終わり (end of file, EOF) に到達した場合は、戻り値として nil を返します。

次のプログラムは、キーボードから 1 行の文字列を読み込んで、その文字列を出力します。

プログラムの例　readline.swift

```
print("文字列を入力してください。: ", terminator: "")
var s = readLine()
print("あなたが入力した文字列は「\(s!)」です。")
```

実行例

```
$ swift readline.swift
文字列を入力してください。: umiushi
あなたが入力した文字列は「umiushi」です。
```

readLine は、デフォルトでは、読み込んだ行から、その末尾にある改行を取り除いた結果を戻り値として返します。改行コードを取り除く前の文字列を戻り値として返してほしい場合は、stripNewLine という外部引数名を使って、引数として真偽値の偽を渡します。

6.3.5　オプショナルを返す関数の宣言

戻り値を返す関数を宣言するとき、どのような引数を受け取った場合でも返すべき戻り値が存在する、という場合は問題はないのですが、引数によっては返すべき戻り値が存在しない、という場合は、どうすればよいのでしょうか。

たとえば、マイナスの整数の階乗というのは定義されていません。ですから、階乗を求める関数は、引数がマイナスの整数だった場合は、返すべき戻り値が存在しません。

5.6.4 で紹介した、階乗を求める関数は、引数がマイナスだった場合、戻り値として 0 を返しています。しかし、マイナスの整数の階乗は存在しないわけですから、0 を返すというのは好ましい動作とは言えません。

このような、場合によっては返すべき戻り値が存在しない関数を宣言する場合、Swift では、戻り値をオプショナルにするという方法を使うことができます。つまり、返すべき戻り値が存在しない場合は nil を返すように宣言するわけです。

階乗を求める関数は、オプショナルを使うことによって、次のように宣言することができます。

プログラムの例　factorial2.swift

```
func factorial(n: Int) -> Int? {
    if n == 0 {
        return 1
    } else if n >= 1 {
        return n * factorial(n-1)!
    } else {
        return nil
    }
}
```

実行例

```
 10> factorial(5)
$R0: Int? = 120
 11> factorial(-5)
$R1: Int? = nil
```

6.4 オプショナル束縛構文

6.4.1 オプショナル束縛構文の基礎

if 文または while 文の中の、条件式を書くところには、

let 識別子 = 式

という形のものを書くことができます。この中の「式」というところには、オプショナルを値とする式を書きます。

if 文または while 文の条件として、この形のものを書いた場合、条件は、式の値として得られたオプショナルが nil ではない場合は真と判定されて、nil の場合は偽と判定されます。

さらに、式の値として得られたオプショナルが nil ではない場合、「識別子」のところに書かれた識別子を名前として持つ定数が宣言されて、オプショナルをアンラップした結果がその定数に代入されます。let ではなくて var と書くことによって、定数ではなくて変数を宣言することもできます。

REPL を使って試してみましょう。

```
   1> var a: Int? = 37
a: Int? = 37
   2> if let b = a {
   3.     print(b)
   4. } else {
   5.     print(a)
   6. }
37
   7> a = nil
   8> if let b = a {
   9.     print(b)
  10. } else {
  11.     print(a)
  12. }
nil
```

このような、オプショナルが nil 以外か nil かということで動作を選択して、オプショナルが nil ではない場合にだけ、それが代入された定数または変数を宣言する形の if 文または while 文は、「オプショナル束縛構文 」(optional binding syntax) と呼ばれます。

6.4.2 読み込みの繰り返し

ファイルからデータを読み込んでそれを処理する、ということをファイルの終わりに到達するまで繰り返す、という処理は、オプショナル束縛構文の while 文を使うことによって記述することができます。

次のプログラムは、標準入力からテキストデータを読み込んで、それぞれの行の先頭に行番号を付けたものを標準出力に出力します。

プログラムの例 linenumber.swift

```
var n = 0
while let line = readLine() {
    n += 1
    print("\(n): \(line)")
}
```

実行例

```
$ swift linenumber.swift < linenumber.swift
1: var n = 0
2: while let line = readLine() {
3:     n += 1
4:     print("\(n): \(line)")
5: }
```

第7章 コレクション

7.1　コレクションの基礎

7.1.1　コレクションとは何か

Swift では、複数のデータから構成されていて、その中のデータを変更したり、そこにデータを追加したり削除したりすることができるデータのことを、「コレクション」(collection) と呼びます。

ちなみに、タプルは、複数のデータから構成されていますが、その中のデータを変更したり、そこにデータを追加したり削除したりする、ということができませんので、それはコレクションではありません。

7.1.2　コレクションの分類

コレクションは、次の三つのものに分類することができます。

- 配列 (array)
- 集合 (set)
- 辞書 (dictionary)

配列については 7.2 で、集合については 7.3 で、辞書については 7.4 で説明することにしたいと思います。

7.1.3　コレクションの変更可能性

コレクションに対しては、その中のデータを変更したり、そこにデータを追加したり削除したりすることができるわけですが、そのような変更ができるのは、そのコレクションが変数に代入されている場合だけです。定数に代入されているコレクションや、変数にも定数にも代入されていないコレクションに対しては、いかなる変更もできません。

7.2　配列

7.2.1　配列の基礎

同じ型のデータを並べることによって作られる、順序を持つコレクションは、「配列」(array) と呼ばれます。

配列を構成しているそれぞれのデータは、その配列の「要素」(element) と呼ばれます。

配列を構成している要素の個数は、その配列の「長さ」(length) または「大きさ」(size) と呼ばれます。

配列の型は、

`[`型`]` または `Array<`型`>`

と記述されます。この中の「型」のところには、配列の要素の型を書きます。たとえば、`Int` の要素から構成される配列の型は、

`[Int]` または `Array<Int>`

と記述されます。

配列は、次の方法のうちのいずれかを使うことによって生成することができます。

- 配列リテラルを書くという方法。
- イニシャライザーを呼び出すという方法。

7.2.2　配列リテラル

式をコンマで区切って並べて、その全体を角括弧で囲むことによって作られる式、つまり、

`[式，式，式，･･･]`

という形の式は、「配列リテラル」(array literal) と呼ばれます。

配列リテラルを評価すると、その中に書かれた式が評価されて、それらの式の値から構成される配列が生成されて、その配列が式全体の値になります。

```
  1> [58, 74, 23, 90, 61]
$R0: [Int] = 5 values {
```

```
    [0] = 58
    [1] = 74
    [2] = 23
    [3] = 90
    [4] = 61
}
```

7.2.3 配列のイニシャライザー

配列の型の記述は、配列のイニシャライザーを呼び出す記述として使うことができます。

配列のイニシャライザーに、引数として範囲を渡すと、その範囲を構成しているそれぞれのものを要素とする配列が生成されます。

```
  1> [Int](4...7)
$R0: [Int] = 4 values {
  [0] = 4
  [1] = 5
  [2] = 6
  [3] = 7
}
```

配列のイニシャライザーを使うことによって、要素の初期値と長さが指定された配列を生成する、ということもできます。要素の初期値と長さは、次の外部引数名を使って渡します。

`repeatedValue`	要素の初期値。
`count`	長さ。

```
  1> [Int](count: 4, repeatedValue: 888)
$R0: [Int] = 4 values {
  [0] = 888
  [1] = 888
  [2] = 888
  [3] = 888
}
  2> [String](count: 3, repeatedValue: "namako")
$R1: [String] = 3 values {
  [0] = "namako"
  [1] = "namako"
  [2] = "namako"
}
```

0 個の要素から構成される配列は、「空配列」(empty array) と呼ばれます。

空配列は、何も引数を渡さないで配列のイニシャライザーを呼び出すことによって生成することができます。

```
  1> [Int]()
$R0: [Int] = 0 values
```

7.2.4 添字式

配列を構成しているそれぞれの要素には、それが並んでいる順番のとおりに、0 番から始まる番号が与えられています。ですから、配列を構成しているそれぞれの要素は、その番号によって識別することができます。

要素の番号によって要素を指定するための式は、「添字式」(subscript expression) と呼ばれます。添字式は、

式$_1$ [式$_2$]

と書きます。この中の 式$_1$ のところには配列を求める式を書いて、式$_2$ のところには要素の番号を求める式を書きます。

添字式を評価すると、その値として、番号によって指定された要素が得られます。

```
  1> let a = ["namako", "umiushi", "hitode", "isoginchaku"]
a: [String] = 4 values {
  [0] = "namako"
  [1] = "umiushi"
  [2] = "hitode"
```

```
    [3] = "isoginchaku"
}
  2> a[2]
$R0: String = "hitode"
```

7.2.5 配列の要素の変更

定数ではなく変数に配列を代入しておくと、その配列に対して、さまざまな変更を加えることが可能になります。

配列が代入されている変数は、その配列を構成しているそれぞれの要素が格納された箱が、その内部にあると考えることができます。

変数に代入された配列の要素を指定する添字式を評価すると、指定された要素が右辺値として得られるだけではなくて、指定された要素が格納されている箱がその左辺値として得られます。ですから、代入演算子の左側に添字式を書くことによって、配列の要素を変更することができます。

```
  1> var a = [28, 33, 47, 59]
a: [Int] = 4 values {
  [0] = 28
  [1] = 33
  [2] = 47
  [3] = 59
}
  2> a[2] = 111
  3> a
$R0: [Int] = 4 values {
  [0] = 28
  [1] = 33
  [2] = 111
  [3] = 59
}
```

7.2.6 配列の長さ

配列が持っている count というプロパティーには、その配列の長さが格納されています。

また、配列が持っている isEmpty というプロパティーには、その配列が空配列ならば真、そうでなければ偽が格納されています。

```
  1> [3, 8, 2, 7, 4, 5, 0, 1].count
$R0: Int = 8
  2> [5, 6, 3, 2].isEmpty
$R1: Bool = false
  3> [Int]().isEmpty
$R2: Bool = true
```

7.2.7 配列に対する繰り返し

配列は繰り返し可能データですので、for-in 文を使うことによって、配列を構成している要素をひとつずつ処理する繰り返しを記述することができます。

```
  1> let a = [75, 43, 89, 21]
a: [Int] = 4 values {
  [0] = 75
  [1] = 43
  [2] = 89
  [3] = 21
}
  2> for i in a {
  3.     print(i)
  4. }
75
43
89
21
```

7.2.8 配列の末尾への要素の追加

配列が持っている append というメソッドを呼び出すことによって、配列の末尾に要素を追加することができます。 append には、引数として、追加したい要素を渡します。

```
  1> var a = [81, 22, 64]
a: [Int] = 3 values {
  [0] = 81
  [1] = 22
  [2] = 64
}
  2> a.append(333)
  3> a
$R0: [Int] = 4 values {
  [0] = 81
  [1] = 22
  [2] = 64
  [3] = 333
}
```

7.2.9 配列への要素の挿入

配列が持っている insert というメソッドを呼び出すことによって、配列の任意の位置に要素を挿入することができます。 insert に渡す 1 個目の引数は、挿入したい要素です。要素を挿入する位置は、 atIndex という外部引数名で渡します。この引数は、挿入したい位置の右側にある要素の番号です。末尾の要素に 1 を加えた番号を渡すことによって、配列の末尾に要素を追加することもできます。

```
  1> var a = [64, 38, 29]
a: [Int] = 3 values {
  [0] = 64
  [1] = 38
  [2] = 29
}
  2> a.insert(555, atIndex: 2)
  3> a.insert(777, atIndex: 4)
  4> a
$R0: [Int] = 5 values {
  [0] = 64
  [1] = 38
  [2] = 555
  [3] = 29
  [4] = 777
}
```

7.2.10 配列の要素の削除

配列が持っている removeAtIndex というメソッドを呼び出すことによって、配列の任意の要素を削除することができます。 removeAtIndex は、引数として受け取った番号の要素を削除して、その要素を戻り値として返します。

```
  1> var a = [81, 23, 54, 67]
a: [Int] = 4 values {
  [0] = 81
  [1] = 23
  [2] = 54
  [3] = 67
}
  2> a.removeAtIndex(2)
$R0: Int = 54
  3> a
$R1: [Int] = 3 values {
  [0] = 81
  [1] = 23
  [2] = 67
}
```

7.2.11　配列の連結

+ という二項演算子は、引数が二つとも配列の場合は、左の引数の右側に右の引数を連結した結果を戻り値として返します。

```
  1> [26, 47] + [81, 34]
$R0: [Int] = 4 values {
  [0] = 26
  [1] = 47
  [2] = 81
  [3] = 34
}
```

7.3　集合

7.3.1　集合の基礎

集合 (set) は、オブジェクトを単純に集めることによって作られるコレクションです。

集合と配列との相違点は、集合の要素には順序がないということ、そして集合は要素の重複を許さないということです。要素の重複を許さないというのは、1 個の集合の中に同一の要素が 2 個以上存在することはできない、ということです。

集合の型は、

Set<型>

と記述されます。この中の「型」のところには、集合の要素の型を書きます。たとえば、Int の要素から構成される集合の型は、Set<Int> と記述されます。

集合は、イニシャライザーを呼び出すことによって生成することができます。

7.3.2　集合のイニシャライザー

集合の型の記述は、集合のイニシャライザーを呼び出す記述として使うことができます。

集合のイニシャライザーに、引数として範囲を渡すと、その範囲を構成しているそれぞれのものを要素とする集合が生成されます。

```
  1> Set<Int>(4...7)
$R0: Set<Int> = {
  [0] = 5
  [1] = 6
  [2] = 7
  [3] = 4
}
```

集合のイニシャライザーに、引数として配列を渡すと、その配列の要素から構成される集合が生成されます。引数として渡した配列の中に、同一の要素が 2 個以上存在していた場合、それらの要素のうちで集合の要素になるのは 1 個だけです。

```
  1> Set<Int>([5, 7, 3, 7, 8, 7])
$R0: Set<Int> = {
  [0] = 5
  [1] = 7
  [2] = 3
  [3] = 8
}
```

0 個の要素から構成される集合は、「空集合」(empty set) と呼ばれます。

空集合は、引数を何も渡さないで集合のイニシャライザーを呼び出すことによって生成することができます。

```
  1> Set<Int>()
$R0: Set<Int> = {}
```

7.3.3　集合の要素数

集合が持っている count というプロパティーには、その集合の要素数が格納されています。

また、集合が持っている isEmpty というプロパティーには、その集合が空集合ならば真、そうでなければ偽が格納されています。

```
  1> Set<Int>([3, 8, 2, 7, 4, 5, 0, 1]).count
$R0: Int = 8
  2> Set<Int>([5, 6, 3, 2]).isEmpty
$R1: Bool = false
  3> Set<Int>().isEmpty
$R2: Bool = true
```

7.3.4 集合の要素の帰属

集合が持っている contains というメソッドは、引数として受け取ったデータが要素として自身に含まれている（帰属している）ならば真を返して、含まれていなければ偽を返します。

```
  1> let a = Set<Int>([83, 24, 51])
a: Set<Int> = {
  [0] = 51
  [1] = 83
  [2] = 24
}
  2> a.contains(24)
$R0: Bool = true
  3> a.contains(67)
$R1: Bool = false
```

7.3.5 集合に対する繰り返し

集合は繰り返し可能データですので、for-in 文を使うことによって、集合を構成している要素をひとつずつ処理する繰り返しを記述することができます。ただし、集合の要素には順序がありませんので、処理される順番は定まっていません。

```
  1> let a = Set<Int>([75, 43, 89, 21])
a: Set<Int> = {
  [0] = 89
  [1] = 75
  [2] = 43
  [3] = 21
}
  2> for i in a {
  3.     print(i)
  4. }
89
75
43
21
```

7.3.6 集合への要素の追加

集合が持っている insert というメソッドを呼び出すことによって、集合に要素を追加することができます。 insert には、引数として、追加したい要素を渡します。

```
  1> var a = Set<Int>([81, 22, 64])
a: Set<Int> = {
  [0] = 64
  [1] = 22
  [2] = 81
}
  2> a.insert(333)
  3> a
$R0: Set<Int> = {
  [0] = 333
  [1] = 22
  [2] = 81
  [3] = 64
}
```

7.3.7 集合の要素の削除

集合が持っている remove というメソッドは、削除することができます。 remove は、引数として受け取ったデータが要素として含まれているならば、それを削除します。

removeは、戻り値としてオプショナルを返します。戻り値は、要素を削除した場合はその要素で、含まれていない場合はnilです。

```
  1> var a = Set<Int>([81, 23, 54, 67])
a: Set<Int> = {
  [0] = 23
  [1] = 81
  [2] = 67
  [3] = 54
}
  2> a.remove(54)
$R0: Int? = 54
  3> a
$R1: Set<Int> = {
  [0] = 23
  [1] = 81
  [2] = 67
}
  4> a.remove(94)
$R2: Int? = nil
```

7.3.8 部分集合

集合が持っているisSubsetOfというメソッドは、引数として受け取った集合が自身の部分集合ならば真を返して、そうでなければ偽を返します。

isSubsetOfに渡す引数は、範囲や配列でもかまいません。この点は、これから紹介するunionやintersectなど、引数として集合を受け取る他のメソッドについても同様です。

```
  1> let a = Set<Int>([58, 62, 24, 73])
a: Set<Int> = {
  [0] = 58
  [1] = 73
  [2] = 24
  [3] = 62
}
  2> a.isSubsetOf(10...80)
$R0: Bool = true
  3> a.isSubsetOf(30...80)
$R1: Bool = false
```

集合が持っているisSupersetOfというメソッドは、引数として受け取った集合が自身の上位集合ならば真を返して、そうでなければ偽を返します。

```
  1> let a = Set<Int>([58, 62, 24, 73])
a: Set<Int> = {
  [0] = 58
  [1] = 73
  [2] = 24
  [3] = 62
}
  2> a.isSupersetOf([58, 24])
$R0: Bool = true
  3> a.isSupersetOf([58, 24, 19])
$R1: Bool = false
```

7.3.9 和集合

集合が持っているunionというメソッドは、引数として受け取った集合と自身との和集合を返します。

```
  1> let a = Set<Int>([38, 24, 76])
a: Set<Int> = {
  [0] = 24
  [1] = 76
  [2] = 38
}
  2> a.union([24, 76, 51])
$R0: Set<Int> = {
```

```
    [0] = 38
    [1] = 24
    [2] = 51
    [3] = 76
}
```

集合が持っている unionInPlace というメソッドは、自身を、引数として受け取った集合と自身との和集合に変更します。

```
  1> var a = Set<Int>([38, 24, 76])
a: Set<Int> = {
  [0] = 24
  [1] = 76
  [2] = 38
}
  2> a.unionInPlace([24, 76, 51])
  3> a
$R0: Set<Int> = {
  [0] = 38
  [1] = 24
  [2] = 51
  [3] = 76
}
```

7.3.10 共通部分

集合が持っている intersect というメソッドは、引数として受け取った集合と自身との共通部分を返します。

```
  1> let a = Set<Int>([38, 24, 76])
a: Set<Int> = {
  [0] = 24
  [1] = 76
  [2] = 38
}
  2> a.intersect([24, 76, 51])
$R0: Set<Int> = {
  [0] = 76
  [1] = 24
}
```

集合が持っている intersectInPlace というメソッドは、自身を、引数として受け取った集合と自身との共通部分に変更します。

7.3.11 差集合

集合が持っている subtract というメソッドは、引数として受け取った集合の要素と一致しない自身の要素から構成される集合、すなわち、自身から引数を引いた差集合を返します。

```
  1> let a = Set<Int>([38, 24, 76])
a: Set<Int> = {
  [0] = 24
  [1] = 76
  [2] = 38
}
  2> a.subtract([24, 76, 51])
$R0: Set<Int> = {
  [0] = 38
}
```

集合が持っている subtractInPlace というメソッドは、自身を、自身から引数を引いた差集合に変更します。

7.3.12 対称差集合

集合が持っている exclusiveOr というメソッドは、引数として受け取った集合と自身との対称差集合を返します。

```
  1> let a = Set<Int>([38, 24, 76])
```

```
    a: Set<Int> = {
      [0] = 24
      [1] = 76
      [2] = 38
    }
      2> a.exclusiveOr([24, 76, 51])
    $R0: Set<Int> = {
      [0] = 51
      [1] = 38
    }
```

集合が持っている exclusiveOrInPlace というメソッドは、自身を、引数として受け取った集合と自身との対称差集合に変更します。

7.3.13　エラトステネスのふるい

2 から n までの範囲にあるすべての素数を求めるための手順としては、「エラトステネスのふるい」(sieve of Eratosthenes) と呼ばれるものがよく知られています。

エラトステネスのふるいは、次のような手順です。

(1) 2 から n までのすべての整数から構成される集合を作る。

(2) 2 を i とする。

(3) 次の (4) と (5) を、i^2 が n 以下であるあいだ繰り返す。

(4) i が集合の中にあるならば、その倍数のうちで集合の中にあるものをすべて取り除く。ただし、i 自身は取り除かない。

(5) i に 1 を加算した整数を i とする。

この手順が終了すると、素数だけが集合の中に残ります。

次のプログラムの中で定義されている eratosthenes という関数は、プラスの整数 n を受け取って、エラトステネスのふるいを使って 2 から n までの範囲にあるすべての素数を求めて、それらの素数から構成される集合を返します。

プログラムの例　eratos.swift

```
func eratosthenes(n: Int) -> Set<Int> {
    var sieve = Set<Int>(2...n)
    var i = 2
    while i*i <= n {
        if sieve.contains(i) {
            var j = i+i
            while j <= n {
                sieve.remove(j)
                j += i
            }
        }
        i += 1
    }
    return sieve
}
```

実行例

```
 16> eratosthenes(20)
$R0: Set<Int> = {
  [0] = 17
  [1] = 5
  [2] = 7
  [3] = 3
  [4] = 11
  [5] = 13
  [6] = 19
  [7] = 2
}
```

7.4 辞書

7.4.1 辞書の基礎

「辞書」(dictionary) と呼ばれるデータは、集合と同じように、順序を持たないコレクションです。

集合と辞書との相違点は、要素の構造です。集合の個々の要素は単独のデータですが、辞書の要素は、2 個のデータがペアになったものです。

辞書の要素を構成しているペアのデータのそれぞれは、「キー」(key) と「値」(value) と呼ばれます。

キーは、個々の要素を識別するために使われるデータです。したがって、1 個の辞書の中に、同じキーを持つ要素が 2 個以上存在することはできません。それに対して、値は、要素の識別には使われませんので、1 個の辞書の中に、同じ値を持つ要素が 2 個以上存在することも可能です。

辞書は、現実の世界に存在する辞書に、よく似ています。現実の辞書は、見出し語と、その見出し語についての説明とをペアにした項目から構成されています。キーというのは見出し語に相当して、値というのは見出し語についての説明に相当します。そして、それぞれの項目は、見出し語によって識別されます。

辞書の型は、

`[`型$_1$`:` 型$_2$`]` または `Dictionary<`型$_1$`,` 型$_2$`>`

と記述されます。型$_1$ のところにはキーの型、型$_2$ のところには値の型を書きます。たとえば、キーが `String` で値が `Int` であるような要素から構成される辞書の型は、

```
[String: Int] または Dictionary<String, Int>
```

と記述されます。

7.4.2 辞書リテラル

「辞書リテラル」(dictionary literal) と呼ばれる式を書くことによって、その式の中に記述された要素から構成される辞書を生成することができます。

辞書リテラルは、キーと値のペアをあらわす記述から構成されます。キーと値のペアをあらわす記述は、「キー値ペア」(key/value pair) と呼ばれます。

キー値ペアは、

式$_1$`:` 式$_2$

というように、コロンの左右に式を書いたものです。式$_1$ のところにはキーを求める式を書いて、式$_2$ のところにはには値を求める式を書きます。たとえば、

```
"namako": 707
```

というキー値ペアを書くことによって、`namako` という文字列がキーで、707 という整数が値であるような辞書の要素を記述することができます。

コロンとその右側の式とのあいだには、通常、1 個の空白を書きます。ただしこれは、文法でそのように決められているわけではなくて、あくまでスタイルの問題です。

辞書リテラルは、キー値ペアをコンマで区切って並べて、その全体を角括弧で囲むことによって作られます。つまり、

`[`キー値ペア`,` キー値ペア`,` `···``]`

という形のものが辞書リテラルだということです。

辞書リテラルを評価すると、その中の式が評価されて、それらの式の値から構成される辞書が生成されて、その辞書が辞書リテラル全体の値になります。

```
  1> ["namako": 68, "umiushi": 31, "hitode": 74]
$R0: [String : Int] = 3 key/value pairs {
  [0] = {
    key = "hitode"
    value = 74
  }
  [1] = {
    key = "namako"
```

```
    value = 68
  }
  [2] = {
    key = "umiushi"
    value = 31
  }
}
```

0 個の要素から構成される辞書は、「空辞書」(empty dictionary) と呼ばれます。

空辞書は、引数を何も渡さないで、辞書のイニシャラーザーを呼び出すことによって生成することができます。

```
  1> [String: Int]()
$R0: [String : Int] = 0 key/value pairs
```

7.4.3 辞書の添字式

配列の場合と同じように、辞書についても、「添字式」(subscript expression) と呼ばれる式を書くことによって、それを構成している特定の要素を指定することができます。

辞書の添字式は、配列の場合と同じように、

式$_1$ [式$_2$]

と書きます。この中の 式$_1$ のところには辞書を求める式を書いて、式$_2$ のところには指定したい要素のキーを求める式を書きます。

添字式を評価すると、その値として、キーによって指定された要素が得られます。ただし、得られる要素はオプショナルです。もしも、指定されたキーを持つ要素が存在しない場合、添字式の値は nil になります。

```
  1> let a = ["namako": 68, "umiushi": 31]
a: [String : Int] = 2 key/value pairs {
  [0] = {
    key = "namako"
    value = 68
  }
  [1] = {
    key = "umiushi"
    value = 31
  }
}
  2> a["umiushi"]
$R0: Int? = 31
  3> a["hitode"]
$R1: Int? = nil
```

7.4.4 辞書の要素の変更

定数ではなく変数に辞書を代入しておくと、その辞書に対して、さまざまな変更を加えることが可能になります。

辞書が代入されている変数は、その辞書を構成しているそれぞれの要素が格納された箱が、その内部にあると考えることができます。

変数に代入された辞書の要素を指定する添字式を評価すると、指定された要素が右辺値として得られるだけではなくて、指定された要素が格納されている箱がその左辺値として得られます。ですから、代入演算子の左側に添字式を書くことによって、辞書の要素を変更することができます。

```
  1> var a = ["umiushi": 31]
a: [String : Int] = 1 key/value pair {
  [0] = {
    key = "umiushi"
    value = 31
  }
}
  2> a.updateValue(444, forKey: "umiushi")
$R0: Int? = 31
  3> a.updateValue(87, forKey: "hitode")
```

```
$R1: Int? = nil
  4> a
$R2: [String : Int] = 2 key/value pairs {
  [0] = {
    key = "hitode"
    value = 87
  }
  [1] = {
    key = "umiushi"
    value = 444
  }
}
```

辞書の要素を変更する代入を実行したときに、指定されたキーを持つ要素が存在しなかった場合は、そのキーを持つ要素が辞書に追加されて、代入されたデータがその値になります。

```
  1> var a = [String: Int]()
a: [String : Int] = 0 key/value pairs
  2> a["hitode"] = 87
  3> a["hitode"]
$R0: Int? = 87
```

辞書が持っている updateValue というメソッドを使うことによっても、辞書の要素を変更することができます。このメソッドには、1 個目の引数として要素の新しい値、forKey という外部引数名でキーを渡します。指定されたキーを持つ要素が存在しなかった場合は、そのキーを持つ要素が辞書に追加されて、1 個目の引数がその値になります。

このメソッドの戻り値は、オプショナルです。指定された要素が存在した場合はその要素の変更前の値が戻り値になって、存在しなかった場合は nil が戻り値になります。

```
  1> var a = ["namako": 68, "umiushi": 31]
a: [String : Int] = 2 key/value pairs {
  [0] = {
    key = "namako"
    value = 68
  }
  [1] = {
    key = "umiushi"
    value = 31
  }
}
  2> a.updateValue(444, forKey: "umiushi")
$R0: Int? = 31
  3> a.updateValue(87, forKey: "hitode")
$R1: Int? = nil
```

7.4.5 辞書の要素の削除

辞書が持っている removeValueForKey というメソッドは、引数としてキーを受け取って、そのキーを持つ要素を辞書から削除します。このメソッドは、戻り値として削除した要素を返します。指定されたキーを持つ要素が存在しない場合の戻り値は nil です。

```
  1> var a = ["namako": 68, "umiushi": 31]
a: [String : Int] = 2 key/value pairs {
  [0] = {
    key = "namako"
    value = 68
  }
  [1] = {
    key = "umiushi"
    value = 31
  }
}
  2> a.removeValueForKey("umiushi")
$R0: Int? = 31
  3> a
$R1: [String : Int] = 1 key/value pair {
  [0] = {
```

```
    key = "namako"
    value = 68
  }
}
```

辞書が持っている removeAll というメソッドは、辞書を構成しているすべての要素を削除します。

```
  1> var a = ["namako": 68, "umiushi": 31]
a: [String : Int] = 2 key/value pairs {
  [0] = {
    key = "namako"
    value = 68
  }
  [1] = {
    key = "umiushi"
    value = 31
  }
}
  2> a.removeAll()
  3> a
$R0: [String : Int] = 0 key/value pairs
```

7.4.6 辞書の要素数

辞書が持っている count というプロパティーには、その辞書の要素数が格納されています。

また、辞書が持っている isEmpty というプロパティーには、その辞書が空辞書ならば真、そうでなければ偽が格納されています。

```
  1> [7: 2, 5: 3, 8: 6, 4: 0, 3: 9].count
$R0: Int = 5
  2> [3: 8, 4: 2, 6: 5].isEmpty
$R1: Bool = false
  3> [Int: Int]().isEmpty
$R2: Bool = true
```

7.4.7 辞書に対する繰り返し

辞書は繰り返し可能データですので、for-in 文を使うことによって、辞書を構成している要素をひとつずつ処理する繰り返しを記述することができます。集合の場合と同じように、辞書の要素にも順序はありませんので、処理される順番は定まっていません。

```
  1> let a = ["namako": 68, "umiushi": 31, "hitode": 74]
a: [String : Int] = 3 key/value pairs {
  [0] = {
    key = "hitode"
    value = 74
  }
  [1] = {
    key = "namako"
    value = 68
  }
  [2] = {
    key = "umiushi"
    value = 31
  }
}
  2> for i in a {
  3.     print(i)
  4. }
("hitode", 74)
("namako", 68)
("umiushi", 31)
```

この実行結果から分かるように、for-in 文を使って辞書に対する繰り返しを実行した場合、定数には、キーと値から構成されるタプルが代入されます。ですから、for と in のあいだに、

(識別子 , 識別子)

という形のものを書くことによって、キーと値のそれぞれを別々の定数に代入することができます。

```
  1> let a = ["namako": 68, "umiushi": 31, "hitode": 74]
a: [String : Int] = 3 key/value pairs {
  [0] = {
    key = "hitode"
    value = 74
  }
  [1] = {
    key = "namako"
    value = 68
  }
  [2] = {
    key = "umiushi"
    value = 31
  }
}
  2> for (key, value) in a {
  3.     print("\(key) = \(value)")
  4. }
hitode = 74
namako = 68
umiushi = 31
```

7.4.8 度数分布

何かの度数分布を求めたいときは、辞書を使うと便利です。

次のプログラムの中で定義されている frequency という関数は、整数の配列を受け取って、その配列を構成している整数の度数分布を示す辞書を返します。

プログラムの例 frequency.swift

```
func frequency(a: [Int]) -> [Int: Int] {
    var f = [Int: Int]()
    for i in a {
        if let n = f[i] {
            f[i] = n+1
        } else {
            f[i] = 1
        }
    }
    return f
}
```

実行例

```
 12> frequency([1, 1, 1, 2, 1, 2, 2, 1, 1, 0, 1, 2, 0, 2, 1])
$R0: [Int : Int] = 3 key/value pairs {
  [0] = {
    key = 2
    value = 5
  }
  [1] = {
    key = 0
    value = 2
  }
  [2] = {
    key = 1
    value = 8
  }
}
```

第 8 章　クロージャー

8.1　クロージャーの基礎

8.1.1　クロージャーとは何か

Swift では、動作をあらわしているデータのことを「クロージャー」(closure) と呼びます。

Swift では、関数というのは、言語処理系に組み込まれているクロージャー、または関数宣言によって生成されたクロージャーのことだと考えることができます。

クロージャーがあらわしている動作をコンピュータに実行させることを、クロージャーを「呼び出す」(call) と言います。

クロージャーは、引数を受け取って、戻り値を返すことができます。

8.1.2　クロージャーの型

クロージャーはデータですので、型を持っています。

1 個の引数を受け取るクロージャーの型は、

型$_1$ -> 型$_2$

と記述されます。この中の 型$_1$ のところには引数の型を書いて、型$_2$ のところには戻り値の型を書きます。たとえば、1 個の整数を引数として受け取って、文字列を戻り値として返すクロージャーの型は、

```
Int -> String
```

と記述されます。

2 個以上の引数を受け取るクロージャーの型は、

(型$_1$, 型$_2$, ··· 型$_n$) -> 型$_m$

と記述されます。この中の 型$_1$ から 型$_n$ までのところには、それぞれの引数の型を書きます。

引数を受け取らないクロージャーの型は、

() -> 型

と記述されます。

8.1.3　クロージャー式

クロージャーは、「クロージャー式」(closure expression) と呼ばれる式を評価することによって生成することができます。

引数を受け取らず、戻り値も返さないクロージャーを生成するクロージャー式は、

```
{
    文
    ⋮
}
```

と書きます。ただし、1 個の文から構成されるクロージャー式は、

{ 文 }

というように 1 行で書くのが普通です。たとえば、1 個の文字列を出力するクロージャーは、

```
{ print("namako") }
```

というように書きます。

8.1.4　クロージャー呼び出し

クロージャーを呼び出したい時は、クロージャーを呼び出すという動作をあらわす式を書きます。そのような式は、「クロージャー呼び出し」(closure call) と呼ばれます。

クロージャー呼び出しは、

式$_1$ (式$_2$, ···)

と書きます。この中の 式$_1$ のところにはクロージャーを求める式を書いて、丸括弧の中には、クロージャーに渡す引数を求める式を書きます。クロージャーが返した戻り値は、クロージャー呼び出しの値になります。

```
  1> let a = { print("namako") }
a: () -> () = 0x0000000100570000 （中略） at repl1.swift)
  2> a()
namako
```

クロージャー呼び出しの中にクロージャー式を書くためには、そのクロージャー式を丸括弧で囲む必要があります。

```
  1> ({ print("namako") })()
namako
```

8.1.5 クロージャーの戻り値

クロージャー式の中に `return` 文を書くと、そのクロージャー式によって生成されるクロージャーは、その `return` 文の中に書かれた式の値を戻り値として返すことになります。

```
  1> ({ return 888 })()
$R0: Int = 888
```

クロージャー式の中に、`return` 文だけしか文がない場合は、その `return` 文の中の `return` という部分は、省略することができます。

```
  1> ({ 888 })()
$R0: Int = 888
```

8.2 引数を受け取るクロージャー

8.2.1 簡略仮引数名

クロージャー式の中に、「簡略仮引数名」(shorthand argument name) と呼ばれる名前を書くと、それを名前とする仮引数が暗黙的に宣言されます。

簡略仮引数名は、

```
$0、$1、$2、・・・
```

というように、ドルマーク (`$`) と数字から構成されます。数字は、渡される引数の順番をあらわしています。1 番目に渡された引数は `$0` が受け取り、2 番目に渡された引数は `$1` が受け取ります。3 番目以降も同様です。

8.2.2 引数を渡すクロージャー呼び出し

引数を渡すクロージャー呼び出しは、

式$_1$ (式$_2$, 式$_3$, ・・・)

と書きます。そうすると、式$_2$ の値が 1 個目の仮引数に渡されて、式$_3$ の値が 2 個目の仮引数に渡されて、それ以降の式の値も、その順番のとおりに仮引数に渡されます。

```
  1> ({ $0*100+$1*10+$2 })(8, 7, 6)
$R0: Int = 876
```

8.2.3 仮引数の宣言

クロージャーの仮引数の宣言は、クロージャー式の中に明示的に書くことも可能です。そうすることによって、簡略仮引数名ではなく、自由な識別子を仮引数名として使うことができます。

クロージャー式の中に仮引数の宣言を書きたいときは、左中括弧の直後に、

(仮引数の宣言 , ・・・) -> 型 `in`

という形のものを書きます。矢印 (`->`) の右側には、戻り値の型を書きます。戻り値の型は、型推論が可能な場合は省略することもできるのですが、なるべく書いておくほうがいいと思います。

仮引数の宣言の書き方は、関数宣言の場合と同じように、

識別子 : 型

と書きます。たとえば、クロージャー式の左中括弧の直後に、

```
(n: Int) -> Bool in
```

と書くことによって、nという仮引数に整数を受け取って、戻り値として真偽値を返すクロージャーを生成することができます。

```
  1> ({ (n: Int) -> Bool in n%2 == 0 })(6)
$R0: Bool = true
```

8.3　高階関数

8.3.1　高階関数の基礎

クロージャーというのは、データの一種です。したがって、クロージャーは、引数としてクロージャーを受け取ることができます。また、クロージャーは、戻り値としてクロージャーを返すこともできます。

引数としてクロージャーを受け取るクロージャーや、戻り値としてクロージャーを返すクロージャーは、「高階クロージャー」(higher-order closure) と呼ばれます。

関数というのはクロージャーの一種ですから、引数としてクロージャーを受け取る関数や、戻り値としてクロージャーを返す関数を作ることも可能です。そのような関数は、「高階関数」(higher-order function) と呼ばれます。また、高階関数であるようなメソッドは、「高階メソッド」(higher-order method) と呼ばれます。

8.3.2　クロージャーを 2 回呼び出す関数

引数としてクロージャーを受け取る関数の例として、引数として受け取ったクロージャーを 2 回呼び出す関数を定義してみましょう。

次のプログラムの中で定義されている twice という関数は、引数として、

```
Int -> Int
```

という型のクロージャーと、1 個の整数を受け取って、その整数を引数にしてそのクロージャーを呼び出して、その戻り値を引数にして再びそのクロージャーを呼び出して、その戻り値を返します。たとえば、引数を 2 乗するクロージャーと、3 を引数として twice に渡すと、3 の 2 乗の 2 乗が戻り値として得られます。

プログラムの例　twice.swift

```
func twice(c: Int -> Int, n: Int) -> Int {
    return c(c(n))
}
```

実行例

```
  4> twice({ $0*$0 }, n: 3)
$R0: Int = 81
```

8.3.3　加算をするクロージャーを返す関数

戻り値としてクロージャーを返す関数の例として、加算をするクロージャーを返す関数を定義してみましょう。

次のプログラムの中で定義されている add という関数は、引数として整数 n を受け取って、戻り値としてクロージャーを返します。この関数が返すクロージャーは、引数として整数 m を受け取って、n と m を加算した結果を返します。

プログラムの例　add.swift

```
func add(n: Int) -> Int -> Int {
    return { n + $0 }
}
```

実行例

```
  4> add(5)(3)
$R0: Int = 8
```

8.3.4 クロージャーを合成する関数

クロージャー a と b があるとします。このとき、引数を b に渡して、その戻り値を引数として a に渡す、という動作をするクロージャーを生成することを、a と b を「合成する」(compose) と言います。

次のプログラムの中で定義されている compose という関数は、引数として、

```
Int -> Int
```

という型のクロージャーを二つ受け取って、それらのクロージャーを合成した結果を戻り値として返します。

プログラムの例 compose.swift

```
func compose(a: Int -> Int, b: Int -> Int) -> Int -> Int {
    return { a(b($0)) }
}
```

実行例

```
  4> compose({ $0+77 }, b: { $0*100 })(42)
$R0: Int = 4277
```

8.4 コレクションの高階メソッド

8.4.1 この節について

コレクションは、便利な高階メソッドをいくつも持っています。この節では、コレクションが持っている高階メソッドを紹介したいと思います。

8.4.2 丸括弧の省略

引数として 1 個のクロージャーだけを受け取るクロージャーに、クロージャー式の値を引数として渡す場合、そのクロージャー式を囲む丸括弧は、省略してもかまいません。たとえば、

```
hoge({ $0+$1 })
```

というクロージャー呼び出しは、丸括弧を省略して、

```
hoge { $0+$1 }
```

と書いてもかまいません。

8.4.3 map

コレクションが持っている map というメソッドは、引数としてクロージャーを受け取って、それによってそれぞれの要素を処理して、それらの処理の結果から構成される配列を戻り値として返します。

要素の型が T である配列または集合が持っている map に引数として渡すクロージャーは、引数の型が T である必要があります。もしも、T -> U という型のクロージャーを引数として渡したとすると、map の戻り値は、[U] という型の配列になります。

```
  1> [5, 3, 2, 8].map { $0*$0 }
$R0: [Int] = 4 values {
  [0] = 25
  [1] = 9
  [2] = 4
  [3] = 64
}
```

キーの型が K で、値の型が V である辞書が持っている map に引数として渡すクロージャーは、引数の型が (K, V) である必要があります。もしも、

```
(K, V) -> U
```

という型のクロージャーを引数として渡したとすると、map の戻り値は、[U] という型の配列になります。

```
  1> ["namako": 83, "umiushi": 46].map { ($0.0, $0.1) }
$R0: [(String, Int)] = 2 values {
```

```
    [0] = {
      0 = "namako"
      1 = 83
    }
    [1] = {
      0 = "umiushi"
      1 = 46
    }
  }
```

8.4.4 filter

コレクションが持っているfilterというメソッドは、戻り値として真偽値を返すクロージャーを引数として受け取って、それによってそれぞれの要素をテストして、戻り値として真が得られた要素から構成される配列を戻り値として返します。

要素の型がTである配列または集合が持っているfilterに引数として渡すクロージャーは、

```
T -> Bool
```

という型である必要があります。filterの戻り値は、[T]という型の配列になります。

```
  1> [7, 6, 3, 5, 4, 1, 8, 9].filter { $0%2 == 0 }
$R0: [Int] = 3 values {
  [0] = 6
  [1] = 4
  [2] = 8
}
```

キーの型がKで、値の型がVである辞書が持っているfilterに引数として渡すクロージャーは、

```
(K, V) -> Bool
```

という型である必要があります。filterの戻り値は、[(K, V)]という型の配列になります。

```
  1> ["namako": 83, "umiushi": 46].filter { $0.1 > 60 }
$R0: [(String, Int)] = 1 value {
  [0] = {
    0 = "namako"
    1 = 83
  }
}
```

8.4.5 reduce

コレクションが持っているreduceというメソッドは、引数として1個のデータとクロージャーを受け取って、そのデータを初期値として、そのクロージャーによってそれぞれの要素を連鎖的に処理していって、最後の処理の結果を戻り値として返します。

reduceは、combineという外部引数名でクロージャーを受け取ります。

要素の型がTである配列または集合が持っているreduceに、1個目の引数としてUという型のデータを渡したとすると、reduceに渡すクロージャーは、

```
(U, T) -> U
```

という型である必要があります。reduceの戻り値は、Uという型のデータになります。

```
  1> [3, 5, 2, 7].reduce("", combine: { $0+$1.description })
$R0: String = "3527"
```

二項演算というのは、2個の引数を受け取るクロージャーですから、reduceには、クロージャーとして二項演算を渡すこともできます。たとえば、reduceにクロージャーとして+を渡すことによって、コレクションの合計を求めることができます。

```
  3> [3, 5, 2, 7, 8, 1, 6, 4].reduce(0, combine: +)
$R2: Int = 36
```

キーの型がKで、値の型がVである辞書が持っているreduceに、1個目の引数としてUという型のデータを渡したとすると、reduceに渡すクロージャーは、

```
(U, (K, V)) -> U
```

という型である必要があります。reduceの戻り値は、Uという型のデータになります。

```
  1> ["umiushi": 35, "namako": 27].reduce(
  2.     "", combine: { $0+$1.0+$1.1.description })
$R0: String = "namako27umiushi35"
```

8.4.6 flatMap

コレクションが持っているflatMapというメソッドは、mapと同じように、引数としてクロージャーを受け取って、それによってそれぞれの要素を処理して、それらの処理の結果から構成される配列を戻り値として返します。

flatMapの動作はmapとほとんど同じですが、違っている点が二つあります。

一つは、戻り値として配列を返すクロージャーを受け取った場合にどのような配列を生成するかということです。その場合、mapは、クロージャーの戻り値から構成される配列、つまり配列の配列を生成します。それに対して、flatMapは、クロージャーの戻り値を要素に分解して、それらの要素から構成される配列を生成します。

```
  1> [3, 8].map { [$0, $0] }
$R0: [[Int]] = 2 values {
  [0] = 2 values {
    [0] = 3
    [1] = 3
  }
  [1] = 2 values {
    [0] = 8
    [1] = 8
  }
}
  2> [3, 8].flatMap { [$0, $0] }
$R1: [Int] = 4 values {
  [0] = 3
  [1] = 3
  [2] = 8
  [3] = 8
}
```

もう一つの相違点は、戻り値としてオプショナルを返すクロージャーを受け取った場合に、その戻り値をどうするかということです。その場合、mapは、クロージャーが返したオプショナルをそのまま配列の要素にします。それに対して、flatMapは、オプショナルの戻り値をアンラップした結果を配列の要素にします。もしも戻り値がnilだった場合、そのnilは配列の要素にはなりません。

```
  1> [4, 5, 2, 7, 6].map {
  2.     (n: Int) -> Int? in
  3.     n%2 == 0 ? n : nil
  4. }
$R0: [Int?] = 5 values {
  [0] = 4
  [1] = nil
  [2] = 2
  [3] = nil
  [4] = 6
}
  5> [4, 5, 2, 7, 6].flatMap {
  6.     (n: Int) -> Int? in
  7.     n%2 == 0 ? n : nil
  8. }
$R1: [Int] = 3 values {
  [0] = 4
  [1] = 2
  [2] = 6
}
```

8.4.7 forEach

コレクションが持っている forEach というメソッドは、map と同じように、引数としてクロージャーを受け取って、それによってそれぞれの要素を処理します。

map と forEach との相違点は、前者はクロージャーの戻り値から構成される配列を生成するのに対して、後者はそれをしないというところにあります。ですから、クロージャーの戻り値から構成される配列が必要ではない場合は、map よりも forEach を使う方がいいでしょう。

```
  1> [5, 3, 2, 8].forEach { print($0) }
5
3
2
8
```

8.4.8 sort

配列の要素を、大小関係にもとづいて並べ替えることを、配列を「ソートする」(sort) と言います。

小さなものから大きなものへという順序は「昇順」(ascending order) と呼ばれ、大きなものから小さなものへという順序は「降順」(descending order) と呼ばれます。

配列が持っている sort というメソッドは、配列をソートして、その結果として得られた配列を戻り値として返します。

sort は、引数としてクロージャーを受け取ります。要素の型が T である配列が持っている sort に引数として渡すクロージャーは、

```
(T, T) -> Bool
```

という型である必要があります。sort は、左にある要素を 1 個目の引数、右にある要素を 2 個目の引数としてクロージャーを呼び出したときに、それが常に真を返すような順序になるように配列をソートします。

比較演算は、そのまま引数として sort に渡すことができます。< または <= を渡すと配列は昇順にソートされ、> または >= を渡すと配列は降順にソートされます。

```
  1> [5, 8, 3, 2].sort(<)
$R0: [Int] = 4 values {
  [0] = 2
  [1] = 3
  [2] = 5
  [3] = 8
}
```

次の実行例は、配列の配列を、要素の長さの昇順にソートしています。

```
  1> [[5, 1, 3], [7, 6, 2, 9], [], [8, 0], [4]].sort {
  2.     $0.count < $1.count
  3. }
$R0: [[Int]] = 5 values {
  [0] = 0 values
  [1] = 1 value {
    [0] = 4
  }
  [2] = 2 values {
    [0] = 8
    [1] = 0
  }
  [3] = 3 values {
    [0] = 5
    [1] = 1
    [2] = 3
  }
  [4] = 4 values {
    [0] = 7
    [1] = 6
    [2] = 2
    [3] = 9
  }
}
```

sort は、ソートの結果を戻り値として返すだけで、配列自体は変化させません。

sortInPlace というメソッドは、 sort と同じように配列をソートするのですが、その結果を戻り値として返すのではなくて、配列自体を変化させます。

```
  1> var a = [5, 8, 3, 2]
a: [Int] = 4 values {
  [0] = 5
  [1] = 8
  [2] = 3
  [3] = 2
}
  2> a.sortInPlace(<)
  3> a
$R0: [Int] = 4 values {
  [0] = 2
  [1] = 3
  [2] = 5
  [3] = 8
}
```

参考文献

[Swift,2015] *The Swift Programming Language (Swift 2.2)*, Apple Inc., 2015.

[荻原,2016] 荻原剛志、『詳解 Swift 改訂版』、SB クリエイティブ、2016、ISBN 978-4-7973-8625-7。

索引

! （アンラップ）, 53
! （演算子）, 42, 43
!= （演算子）, 36, 52
", 17
$, 71
%= （演算子）, 27
&& （演算子）, 42
(), 22
* （演算子）, 20
*/, 13
*= （演算子）, 27
+ （演算子）, 20, 22, 60, 74
+= （演算子）, 27
-, 17
- （演算子）, 20, 22
-= （演算子）, 27
.
    タプルの——, 33, 34
    浮動小数点数の——, 16
    プロパティーの——, 23
... （演算子）, 45
..< （演算子）, 45
.swift （拡張子）, 9, 28
/ （演算子）, 21
/*, 13
//, 13
/= （演算子）, 27
:, 11
:h （REPL コマンド）, 11
:help （REPL コマンド）, 11
:q （REPL コマンド）, 11
:quit （REPL コマンド）, 11
;, 12
< （演算子）, 36
<= （演算子）, 36
= （演算子）, 26
== （演算子）, 36, 52
> （演算子）, 36
>= （演算子）, 36
?: （演算子）, 43
\, 17
% （演算子）, 21
_, 34
|| （演算子）, 42
10 進数リテラル, 16
16 進数（エスケープシーケンス）, 17
16 進数リテラル, 16
2 進数リテラル, 16
8 進数リテラル, 16

abs, 19
append, 59
Apple, 9
atIndex, 59
AWK, 8

Basic, 8
Bool, 14
break 文, 48, 49

C, 8
case 節, 40
case ラベル, 40
COBOL, 8
combine, 74
contains, 61
continue 文, 49
count, 57, 58, 60, 68
Ctrl-C, 47

default 節, 40
description, 23
Double, 14, 53

else
    ——以降を省略した if 文, 37
EOF, 54
exclusiveOr, 63
exclusiveOrInPlace, 64

false, 18
filter, 74
flatMap, 75
for-in 文, 44, 48, 49, 58, 61, 68
for-in 文
    ——の書き方, 45
forEach, 76
forKey, 67
Fortran, 8

Haskell, 8

if 文, 35, 36, 55
    else 以降を省略した——, 37
insert, 59, 61
Int, 14, 53
intersect, 63

`intersectInPlace`, 63
`isEmpty`, 23, 58, 60, 68
`isSubsetOf`, 62
`isSupersetOf`, 62

Java, 8

`let`, 25
Linux, 9
Lisp, 8
`lowercaseString`, 23

Mac, 9
`map`, 73, 75, 76
ML, 8

nil, 51
nil 合体演算子, 53

OS X, 9

Pascal, 8
Perl, 8
PostScript, 8
`predecessor`, 23
`print`, 19, 20, 52
Prolog, 8

`readLine`, 54
`reduce`, 74
`remove`, 61
`removeAll`, 68
`removeAtIndex`, 59
`removeValueForKey`, 67
`repeat-while` 文, 44, 46–49
`repeatedValue`, 57
REPL, 10
　　——の終了, 11
　　——のヘルプ, 11
　　Swift の——, 10
`return` 文, 32, 71
Ruby, 8

Smalltalk, 8
`sort`, 76
`sortInPlace`, 77
`String`, 14
`stripNewLine`, 54
`subtract`, 63
`subtractInPlace`, 63
`successor`, 23
Swift, 8
　　——の REPL, 10
　　——の言語処理系, 9
swift, 10
swiftc, 9
`switch` 文, 35, 38, 40

Tcl, 8
`terminator`, 20
`true`, 18

Unicode 文字, 17
`union`, 62
`unionInPlace`, 63
`updateValue`, 67
`uppercaseString`, 23

`var`, 25

`while` 文, 44, 46, 48, 49, 55
　　——の書き方, 46

Xcode, 9

値, 65
　　式の——, 15, 26
　　式の——の型, 15
　　定数名の——, 26
　　変数名の——, 26
あまり, 21
アンコメント, 14
アンダースコア, 24, 34
アンラップ, 52, 75

一重引用符, 17
イニシャライザー, 53, 56, 60
　　集合の——, 60
　　配列の——, 57
入れ子, 12
インタプリタ, 9

右辺値, 26
うるう年, 43

英字, 24
エスケープシーケンス, 17
エディター, 9
エラー, 10, 11
エラーメッセージ, 10, 11
エラトステネス
　　——のふるい, 64
演算, 20
演算子, 17, 20
演算子式, 20
円マーク, 17

大きい, 36
大きいかまたは等しい, 36

大きさ
　　配列の——, 56, 58
大文字化, 23
オプショナル, 52, 75
オプショナル型, 52
オプショナル束縛構文, 55
親子関係, 49

改行, 17
階乗, 50, 54
外部引数名, 19, 30
書き方
　　`for-in` 文の——, 45
　　`while` 文の——, 46
　　関数呼び出しの——, 18
　　選択肢の——, 40
角括弧, 56, 65
加算, 20
仮数部, 14
型, 14, 24
　　クロージャーの——, 70
　　式の値の——, 15
型推論, 25, 71
型名, 14
かつ, 42
カメラ, 49
仮引数, 29, 71
関数, 15, 18, 27
　　——の再帰的な定義, 50
　　——の定義, 27
　　——を宣言する, 18
関数宣言, 18, 27, 70
関数名, 18
関数呼び出し, 18, 20, 22
　　——の書き方, 18
　　——の評価, 19
感嘆符, 53
簡略仮引数名, 71

偽, 14
キー, 65
キー値ペア, 65
キーボード, 54
機械語, 8, 9
基数, 16
奇数, 37
帰属
　　集合の要素の——, 61
基底, 49
基本型, 14
逆数, 32
キャリッジリターン, 17
共通部分, 63

空辞書, 66
空集合, 60
偶数, 37
空タプル, 19, 26, 27, 32, 33
空配列, 57
空白, 12, 24
空文字列, 17, 20, 23
組み込み関数, 18
繰り返し, 44
　　辞書に対する——, 68
　　集合に対する——, 61
　　配列に対する——, 58
　　読み込みの——, 55
繰り返し可能データ, 44, 58, 61, 68
クロージャー, 70
　　——の型, 70
　　——の戻り値, 71
　　引数を受け取る——, 71
クロージャー式, 70
クロージャー呼び出し, 70
グローバルスコープ, 28

結合規則, 22
言語, 8
言語処理系, 9
　　Swift の——, 9
減算, 20

高階クロージャー, 72
高階関数, 72
高階メソッド, 72
降順, 76
合成, 73
構造
　　式の——, 15
コメントアウト, 14
小文字化, 23
コレクション, 56, 73
コロン, 11
コンパイラ, 9
コンパイル, 9
コンマ, 33, 56, 65

再帰, 49
再帰的, 49
　　関数の——な定義, 50
最大公約数, 47
削除
　　辞書の要素の——, 67
　　集合の要素の——, 61
　　配列の要素の——, 59
左辺値, 26
三項演算, 20
三項演算子, 20, 43
算術演算, 20
算術演算子, 20

時間, 34, 38
式, 11, 12, 15
　　――の値の型, 15
　　――の構造, 15
　　選択をあらわす――, 43
　　タプルを生成する――, 33
識別子, 24
辞書, 15, 56, 65
　　――に対する繰り返し, 68
　　――の要素数, 68
　　――の要素の削除, 67
　　――の要素の変更, 66
辞書式順序, 36
辞書リテラル, 65
指数部, 14
自然言語, 8
子孫, 49
集合, 56, 60
　　――に対する繰り返し, 61
　　――のイニシャライザー, 60
　　――の要素数, 60
　　――の要素の帰属, 61
　　――の要素の削除, 61
　　――への要素の追加, 61
終了
　　REPL の――, 11
商, 21
上位集合, 62
条件, 14
条件演算子, 35, 43
乗算, 20
昇順, 76
小の月, 41
省略
　　`else` 以降を――した `if` 文, 37
　　丸括弧の――, 73
初期化, 24
除算, 21
処理系, 9
真, 14
真偽値, 14, 18
真偽値リテラル, 18
人工言語, 8

水平タブ, 17
数字, 24
数値
　　――から文字列への変換, 23
スコープ, 28
スペースキー, 12

整数, 14
整数リテラル, 16
精度, 14
セミコロン, 12
宣言, 24
宣言する
　　関数を――, 18
先祖, 49
選択, 35
　　――をあらわす式, 43
選択肢
　　――の書き方, 40
　　範囲による――, 41

挿入
　　配列への要素の――, 59
総和, 32
添字式, 57, 58, 66
ソート, 76
ソフトウェア, 8

対称差集合, 63, 64
代入, 24
代入演算子, 26
大の月, 41
多肢選択, 38, 40
タプル, 15, 33, 44, 56, 68
　　――を生成する式, 33
単項演算, 20
単項演算子, 20
単純代入演算子, 26
単純文, 12

小さい, 36
小さいかまたは等しい, 36
注釈, 13

追加
　　集合への要素の――, 61
　　配列への要素の――, 59

定義
　　関数の――, 27
　　関数の再帰的な――, 50
定数, 24, 29, 56
定数宣言, 25
定数名, 24
　　――の値, 26
テキストエディター, 9
ではない, 42, 43
デフォルト値, 31

度数分布, 69
ドット
　　タプルの――, 33, 34
　　浮動小数点数の――, 16
　　プロパティーの――, 23
ドルマーク, 71

長さ

配列の——, 56, 58

二項演算, 20, 74
二項演算子, 20
二重引用符, 17

ヌル文字, 17

ハードウェア, 8
配列, 15, 56
——に対する繰り返し, 58
——のイニシャライザー, 57
——の大きさ, 56, 58
——の長さ, 56, 58
——の要素の削除, 59
——の要素の変更, 58
——の連結, 60
——への要素の挿入, 59
——への要素の追加, 59
配列リテラル, 56
バックスラッシュ, 17
範囲, 45, 57, 60
——による選択肢, 41
範囲演算, 45
範囲演算子, 45
半開範囲演算子, 45
反転
符号の——, 22

比較演算子, 36, 76
引数, 18, 19, 29, 70
——を受け取るクロージャー, 71
左結合, 22
等しい, 36
等しくない, 36
評価
関数呼び出しの——, 19
評価する, 15
標準入力, 54

ファイルの終わり, 54, 55
フィボナッチ数列, 50
複合代入演算子, 27
符号
——の反転, 22
浮動小数点数, 14, 16
浮動小数点数リテラル, 16
部分集合, 62
ふるい
エラトステネスの——, 64
プログラミング, 8
プログラミング言語, 8
プログラム, 8
プロパティー, 23
プロパティー名, 23
プロンプト, 11

分, 34, 38
文, 12
——の列, 12
文書, 8

平年, 43
閉範囲演算子, 45
変換
数値から文字列への——, 23
変更
辞書の要素の——, 66
配列の要素の——, 58
変数, 24, 56
変数宣言, 25
変数名, 24
——の値, 26

または, 42
丸括弧, 22, 33, 34
——の省略, 73

右結合, 22
見出し語, 65

無限ループ, 46

メソッド, 23
メソッド名, 23
メリット
ローカルスコープの——, 29

文字列, 14
——の連結, 22
数値から——への変換, 23
文字列リテラル, 17
戻り値, 18, 31, 32, 70
クロージャーの——, 71
モニター, 49

約数, 46

ユークリッドの互除法, 47, 51
ユーザー定義関数, 18
優先順位, 21

要素, 33, 56
辞書の——の削除, 67
辞書の——の変更, 66
集合の——の帰属, 61
集合の——の削除, 61
集合への——の追加, 61
配列の——の削除, 59
配列の——の変更, 58
配列への——の挿入, 59
配列への——の追加, 59
要素数
辞書の——, 68

集合の——, 60
要素名, 34
曜日, 40
呼び出す, 18, 70
読み込み
——の繰り返し, 55
キーワード, 24

リテラル, 16

列
文の——, 12
連結
配列の——, 60
文字列の——, 22

ローカルスコープ, 29
——のメリット, 29
論理演算, 42
論理演算子, 42
論理積演算子, 42
論理否定演算子, 43
論理和演算子, 42

差集合, 63
和集合, 62, 63