プログラミング言語
Konoha
Designer's Notes

倉光君郎

第１版（ドラフト）

本文書に関して

Konoha をご試用いただきましてありがとうございます。

本書は、Konoha 言語のプログラミングガイド、及びその言語仕様をまとめたデザイナーズノートです。長らく、Konoha にはマニュアルがなく、ご不便をかけてきましたが、本書が Konoha の体験を広げる一助になれば幸いでございます。なお、本文書は、「書きかけでもいい」から少しでも多くの情報を提供してほしいという要望に応え、本当に「書きかけ」だったりしてお見苦しいところがたくさんあります。あらかじめ、お詫びしておきます。

また、Konoha 言語自体、なおも開発が行われています。本書は、Konoha 1.0 仕様を前提に書いており、仕様が先行しているため、実装との間にギャップがあります。そこで、各節には⋆, ⋆⋆のようにマークを付けてあり、それぞれ部分的な開発/テストが残っている機能、計画されているが実装されていない機能を表しています。

Konoha ユーザメーリングリストでは、Konoha に関する最新情報を扱っております。是非、ご登録の上、本書の不備や最新機能に関する情報を得て下さい。また、バグレポートも歓迎しております。

> アドレス: konoha-users@sourceforge.jp
> 登録: http://lists.sourceforge.jp/mailman/listinfo/konoha-users

著作権表記

再配布の留意事項

本書は、現在 PDF 版を Konoha プロジェクトの sourceforge サイトから配布しています。将来、出版してくれる出版社が見つかれば、出版したいと考えていますが、電子版の配布もユーザ数の動向を把握するため、可能な限り継続する予定です。ご知人に紹介するときは、PDF ファイルを直接、再配布する代わりに、一次配布サイトからダウンロードするように薦めて頂けますようにお願いします。

http://sourceforge.jp/projects/konoha/releases/

目次

第 1 章

はじめに

Konoha プロジェクトはどのように始まったのか？その背景にある言語設計のフィロソフィーに関して説明する。

1.1 Konoha の始まり*

第 2 章

Konoha の特徴

Konoha の特徴は、ひと言で言えば、「静的に型付けされたスクリプティング言語」である。しかし、スクリプティング言語は、別名「ダイナミック言語」と呼ばれるとおり、動的に型付けされている。そのため、Konoha は、どこかしら保守的でもあり、逆に新しさもあるスクリプティング言語となっている。本章では、Konoha の特徴を簡単なソースコードとともに紹介したいと思う。

2.1 おなじみの言語文法

Konoha の文法は、C/C++ や Java と高い互換性をもっている。これは、はじめてプログラミングを学ぶ者にとって、C/C++, Java などのメインストリーム言語の良き導入になるだけでなく、経験あるプログラマにとって特別な努力を費やすことなく、Konoha プログラミングがはじめられることを意味する。

次は、おなじみ fibonacci 数列を再帰で求める関数である。C/C++, Java でも、もちろん Konoha でも全く同じように実行することができる。

```
int fibo(int n) {
  if(n < 3) return 1;
  return fibo(n-1)+fibo(n-2);
}
```

一方、これが、Perl や Python, Ruby, Lua など、他のスクリプティング言語であったら、やはり独特な文法のため、いきなり簡単な fibonacci 関数も書くことは難しい。次は、「Java 風の文法をもった」といわれる、JavaScript による fibonacci 関数の例であるが、はやり無視できない大きな違いがいくつもある。

```
function fibo(n) {        // JavaScript の場合
  if(n < 3) return 1;
```

```
  return fibo(n-1)+fibo(n-2);
}
```

Konoha が、従来の Java 風スクリプティング言語と比べて、高い文法互換性をもっている理由は、Java と同じく静的に型付けされている点に大きい。これによって、オブジェクト指向プログラミングにおいても、より Java に近いモデリング手法でクラスを設計することができる。

次は、Counter クラスの定義の例である。クラス宣言に始まり、コンストラクタからフィールドまで、通常のプログラミングにおいて、まず問題にならないレベルの互換性がある。

```
class Counter {
  int cnt;
  Counter(int n) { cnt = n; }
  void count() { cnt++; }
  void reset() { cnt = 0; }
}
```

ただし、Konoha は、独立したプログラミング言語である。文法の互換性は、あくまでも C/C++ や Java プログラマへの混乱を最小限にとどめるために導入されている。もちろん、スクリプティング言語として、簡略化されている部分もあるし、独自に拡張されている文法もある。

2.2 対話的な動作: Looks like Java, Runs like Python

Konoha プロジェクトのキックオフ当時の目標は、”Looks like Java, Runs like Python”、つまり「Java のように書けて Python のように実行できる」であった。現在、Konoha は、Python のみならず、Ruby や JavaScript など、様々な言語の「よい設計」を取り込んでいるが、Konoha のスクリプティング言語としてのお師匠といえば、Python なのである。

Python へのオマージュが最もよくあらわれている部分が、対話モードである。`konoha` コマンドを実行すると、`python` コマンドと同様に対話モードが起動する。

```
$ konoha
Konoha 0.3.10 (Rufy) GPL2 (#544, May  2 2009, 09:37:22)
[GCC 4.0.1 (Apple Inc. build 5490)] on macosx_32 (32, UTF-8)
Options: iconv sqlite3 thread regex used_memory:412 kb
>>>
```

>>> は、コマンドプロンプトである。複数行にわたる場合は、2 行目以降は、... となる。ここにプログラム（式もしくはステートメント）を入力すると、次の行にはその実行結果が得られる。

```
>>> print "hello, world"
hello, world
```

Konoha の対話モードは、Lisp の対話的プログラミングほど、それ自体で成り立つプログラミング環境ではない。しかし、Konoha の機能、動作を試したいとき、十分に役立つ機能である。もちろん、先ほどの fibonacci 関数も、対話モードで定義し、その場で実行することもできる。

```
>>> int fibo(int n) {
...     if(n < 3) return 1;
...     return fibo(n-1)+fibo(n-2);
... }
>>> fibo(10)
55
```

2.3 型推論

Konoha の特徴は、静的な型付けである。これは、ごく一般的に言えば、変数の型宣言をプログラマに強制することを意味する。ダイナミック言語に親しんできたプログラマには、これは何かしら面倒な話に聞こえてしまうだろう。

```
int n = 1;                                  // 変数宣言
```

念のため補足しておけば、静的な型付けは、プログラミングモジュールの仕様を明示的に定義することであり、大規模なソフトウェア開発やグループ開発において、ケアレスミスや誤解によるソフトウェアクラッシュを防ぐ重要な機能である。Konoha がスクリプティング言語であっても、静的に型付けされているのは悪い話ではない。

それでも、慣習とは恐ろしいもので、次のようにいきなり変数を使いたくなることもある。

```
n = 1;                                      // 変数宣言なし
```

Konoha は、非常に限定的であるが、実用的に十分な型推論 (type inferencing) の機構を備え、明示的に変数宣言をしなくても、その変数の型を推論する機能を備えている。これは、変数宣言なしであっても、変数の初期化の値からその変数の型を推論する機能である。

```
>>> a = [1,2,3]                             // 推論による型宣言
>>> n = a[1]
>>> typeof(a)                               // 静的な型付けを調べる
Int[]
>>> typeof(n)
Int
```

Konoha の型推論は、プログラミングのしやすさと型宣言による厳密な仕様定義の間で、バランスをとりながら設計されている。また、型を推論できなかった場合、Any 型による

「動的な型付けのオプション」も用意されている。

```
>>> s = null;                                  // 型推論できない
>>> typeof(s)
Any
```

Any 型やダイナミック言語の特性を活かしたプログラミングは、「第 ?? ダイナミック・スタイル」でより詳しく解説する。

2.4 最速水準の実行性能

スクリプティング言語は、元来、プログラムの実行速度より、プログラムの開発速度へ価値をおいた言語設計であった。そのため、実行性能を単純に比較すると、C/C++ や Java の足下にも及ばない。しかし、今日、より複雑なアプリケーション開発にもスクリプティング言語が利用されるようになるにつれ、スクリプティング言語エンジンの性能向上は大きな関心事になっている。

Konoha は、バイトコード型バーチャルマシンで実装されたスクリプティング言語エンジンをもっている。スクリプトは、バイトコードにコンパイルされ、そしてバーチャルマシン上で実行される。

次は、コンパイル済みのバイトコード (`fibo()` 関数) をフォーマッタ機能でダンプした様子である。

```
>>> %dump(Script.fibo)
int main.Script.fibo(int n)
 [4:0x2038e0] SETESP(30) sfp[2]
 [4:0x2038e8] iLTn(99) sfp[2] sfp[1] 3
 [4:0x2038fc] bJIFF(62) 0x20391c sfp[2]
 [4:0x203910] RETo(35) sfp[-1] 1
 [5:0x20391c] iSUBn(87) sfp[5] sfp[1] 1
 [5:0x203930] FCALL(46) sfp[3] 3 sfp[0] int fibo(int n)
 [5:0x203940] iSUBn(87) sfp[6] sfp[1] 2
 [5:0x203954] FCALL(46) sfp[4] 3 sfp[0] int fibo(int n)
 [5:0x203964] iADD(84) sfp[-1] sfp[3] sfp[4]
 [5:0x203970] RET(32)
```

Konoha の大きな特徴は、静的な型付けの特性を活かして、ダイナミック言語より効率のよい実行コードを生成している点である。10 年以上にわたる高速化チューニングが施された Python や世界最速のスクリプティング言語 Lua と比べも、図 2.1 に示すとおり、数倍以上の実行性能を示している。

図 2.1 fibo(36) による性能比較: fibonacci 数列は、数値演算だけでなく、ローカル変数へのアクセスや関数コールなど、プログラミング言語の基礎的な性能を図ることに広く利用されている。

Konoha が大学の研究室生まれであり、プロフェッショナルな開発者が参加していない

現状を勘案すれば、より大幅なチューニングも可能と予想される。また、スクリプティング言語の高速化に関しては、学術的なアプローチにより、新しい高速化技法の研究も進められている。

2.5　高いポータビリティ*

Konoha プロジェクトは、開発者が、TRON プロジェクト出身であったこともあり、TRON OS を含めた組み込み分野やユビキタス応用を視野にいれた言語エンジンの開発を行ってきた。そのため、Konoha は、ポータビリティの高い C ソースコードで書かれ、UNIX 系の OS だけでなく、Windows や TRON(T-Kernel) などで、様々なプラットホームで動作が確認されてきた。(図 refteaboard)

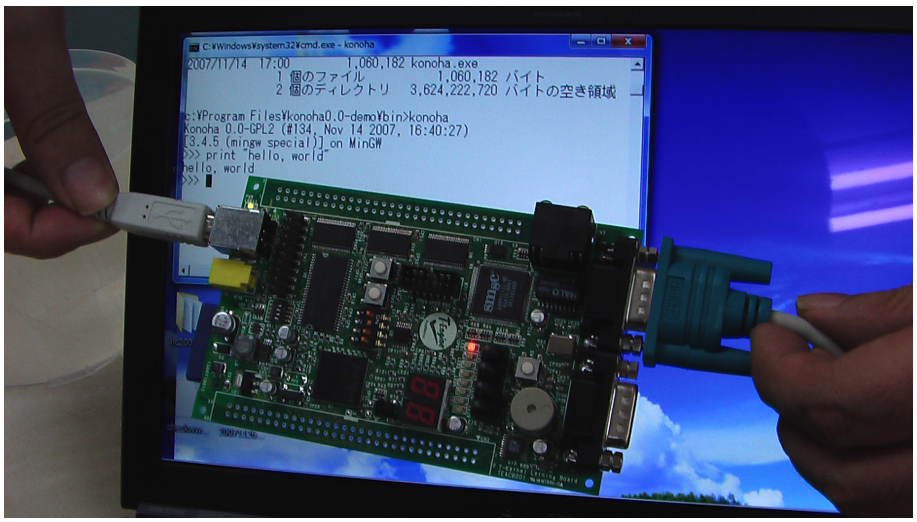

図 2.2　パーソナルメディア製 Teabord (OS:T-Kernel, CPU:ARM, Memory:16Mb) の上で動作する Konoha スクリプティング言語エンジン (2008 年頃撮影)

labelteaboard

2.6　実行前の型検査といつでも実行

型検査とは、型に対する操作、例えば関数コールのとき引数の数や種類が間違っていないか確認することである。ダイナミック言語は、実行時にのみ型検査が行われるため、型エラーが含まれるスクリプトでも実行しないとエラーが発見できない。そのため、全ての実行パスをひとつひとつテスト実行しながら、型エラーを探し、修正する必要がある。

Konoha は、静的な型付け言語であるため、スクリプトの実行前に型検査を機械的に、つまりコンパイラが行うことができる。また、スクリプトを実行することなしに、型エラーや潜在的なミスを全て検証するための専用のオプション (-c) も用意されている。こ

のとき、実際にプログラムが走り始めることはないため、ファイル操作やデータベースが中途半端な状態でプログラムが停止する弊害もない。

```
$ konoha -c sample/err.k
konoha -c err.k
 - [err.k:4]:(errata) added return value
 - [err.k:7]:(error) type error: Int is not string
```

Konoha は、従来の静的な型付け言語と異なる点もある。”Run anytime” コンパイラ技術を採用し、コンパイル中に型エラーが発見されても、型エラーの部分のみ、実行しても安全なコード（ランタイム例外）に書き換えることで、いつでも実行可能なバイトコードを生成することができる。そのため、コンパイルが通らないため実行できないということはなく、スクリプティング言語の開発しやすさが保たれている。

Konoha では、もしエラー箇所を実行したときは、`Source!!`例外が通知されて停止する。プログラマは、エラーを（全部）取り除いてから実行するか、それとも試しに実行させながらバグをとるか、どちらのスタイルの開発でも選ぶことができる。

```
$ konoha sample/err.k
 - [err.k:7]:(error) type error: Int is not string
 ** Source!!: Running errors at [err.k:7]
```

2.7 オブジェクト指向

Konoha は、「全てがオブジェクト」という世界観で統一されたオブジェクト指向プログラミング言語である。整数も null もオブジェクトで表現されている。また、クラスルーム利用において、最先端のオブジェクト指向プログラミングのアイディアを学ぶことができるように設計されている。

- 名前ベースのクラスシステム
- 単一継承、インターフェースによるポリモーフィズム
- ダックタイピングによるポリモーフィズム
- 総称型（Generics)
- メタオブジェクト、アスペクト指向
- データ変換と相互運用性（Mappable Class)

2.7.1 クラスを調べる

オブジェクト指向プログラミング言語は、各クラスにその機能性が集約されている。Konoha では、対話モードから man コマンドで使えば、クラスが提供するオペレーションが表示される。

```
>>> man Int
CLASSNAME
  konoha.Int
  extends konoha.Number
CONST
  Int.MAX: 9223372036854775807
  Int.MIN: -9223372036854775808
OPERATOR
     -x       x != x        x & y        x * y        x + y
  x - y        x / y        x < y       x << y       x <= y
 x == y        x > y       x >= y       x >> y        x ^ y
x mod y        x | y          x++          x--          |x|
      ~x
METHOD
  Int! Int.random(Int n)
FORMATTER
  %bits %d %f %s %u %x
MAPPING
  Int
    ==> Float
    ==> String
```

Konoha は、ソフトウェアモジュールの柔軟性のため、実行時にメソッド等を追加できる。man コマンドは、実行時の状態を表示するため、メソッドの種類は変更になる。

2.8 活版フォーマッタ*

活版印刷 (movable type) は、出版の普及による中世から近代への橋渡しの役割を果たしたため、コンピュータ発明以前の最大の情報技術イノベーションと考えられている。Konoha では、HTML や XML 文書の生成のため広く利用されるテキストフォーマッティングにおいて、活字と同様に再利用性の高い手法を提供することを目指して、組み合わせが自由自在な「活版フォーマッティング (movable formating)」技術を新たに導入している。

基本的なアイディアは、オブジェクト指向モデルに基づく「書式付きフォーマッタ」にある。フォーマッタは、C 言語の printf 風の書式 (`%s`, `%d`) に似ているが、オブジェクトと書式は出力時にダイナミックバインディングされる。つまり、C 言語では型に対して書式が固定的に決まっていたのに対し、Konoha では int 型でも小数形式で出力したいときは `%f` を用いることができる。

```
>>> %s(1)
1
>>> %03d(1)
001
>>> %f(1)
1.00000
>>> %bits(1)
00000000 00000000 00000000 00000001
```

オブジェクト単位のフォーマッタを組み合わせて、自由自在にフォーマッティングを行うことができる。

```
>>> p = 9.80
>>> `<price> $%.2f{p}</price>`
"<price>$9.80</price>"
```

また、フォーマッタはメソッドの一種として、プログラマが自由に定義することもできる。%XML のような複雑なフォーマッタも作成できる。

```
>>> format %XML (Catalog c) """
<product>
  <title> %s{c.name}</title>
  <price> $%.2f{c.price}</price>
</product>
"""


>>> %XML(c)
<product>
 <title>Harry Potter DVD</title>
 <price>$9.80</price>
</product>
```

注意：フォーマッタは、ストリームへの出力という形でフォーマッティングを行うため、大量のデータを低メモリ消費で変換することができる。詳しくは、「第 **??** 章 活版フォーマッタ」で述べる。

2.9　マッピング機能とデータ変換*

2.10　セマンティック・プログラミング*

第 3 章

レキシカル構造

3.1 文字

Konoha は、はじめから多国語環境での利用を前提として設計されている。文字は、Unicode 規格で定められた文字コードが利用可能であり、スクリプトの記述には Web 上で標準である UTF-8 エンコーディングを採用している。また、Konoha の実行環境は、文字コード変換 (iconv もしくは相当) 機構と統合されており、オペレーティングシステムのローカルなエンコーディングも自動的に UTF8 に変換可能して入出力することができる。

さて、Konoha は多国語前提と言っておきながら、識別子（変数名、クラス名、メソッド名）には ASCII 文字の利用しか認めていない。これについてちょっと説明を加えておきたい。そもそも、Java 言語が登場したとき、識別子に Unicode 文字、つまり「日本語名」を認めた。これは、たぶん、彼らなりの非欧文圏への親切だったと思うのあるが、それから 10 年以上を過ぎた現在、日本語で書かれた変数やクラス名をみる機会は全くない。ここから学ぶ教訓は、欧文圏のプログラマにとって「入力すらできない名前」を使わないことも、我々側からの親切であるといえる。

3.2 トークン

トークンとは、プログラミング言語の最小の意味単位である。それぞれ、予約語（Konoha 文法で定義されたキーワード）、識別子、演算子、リテラル（データ値）などの意味を持つ。ソースコードは、Konoha の文法ルールにしたがって、トークンに分割される。

```
return 0;     // 3 つのトークン "return"  "0" ";"
return0;      //   2 つのトークン "return0" ";"
```

空白は、ソースコードからトークンを分割するための区切りとなる記号で、空白はいくつ続いても意味はない。タブ (Tab) や改行も空白と同様に扱われる。そのため、プログラマは自由に空白、タブ、改行を用いて読みやすいソースコードを書くことができる。ただ

し、次節で述べるとおり、改行に関してはひとつだけ留意事項がある。
また、次の文字は前後に空白があるものとみなされる。

```
{ } ( ) [ ] ;  ,
```

また、演算子で用いられる記号 (+ や⋆など) の前後にも空白があるものと解釈される。ただし、読みやすいソースコードを書くため、明示的に空白をいれることが推奨される。

```
a + b                    // 推奨
a+b                      // 非推奨
```

3.3　ステートメント

ステートメントとは、プログラミング言語における文のことである。Konoha のステートメントは、C や C++, Java と同様に、セミコロン (;) で終わる。

```
a = 3;
b = 4;
```

次のように、1 行に複数のステートメントを書くこともできる。

```
a = 3; b = 4;
```

ここまでは、C や C++, Java で採用された正しいスタイルのステートメントである。もし、Konoha を学習用として利用している場合、100% この正しいスタイルを身につけるべきである。

3.3.1　セミコロンの省略

多くのスクリプティング言語は、1 行の終わりをそのままステートメントの終わりと解釈することが多い。そこでセミコロンを強制することは、Konoha は書きにくい言語となってしまう。そこで、Konoha は、ステートメントの途中に改行がくると、セミコロン (;) を忘れたものと拡大解釈している。つまり、次の例でも、同様に、2 つのステートメントになる。

```
a = 3
b = 4
```

こうすると、あちらを立てればこちらが立たずという問題が発生する。

C プログラマは、１行 (80 文字) に収まりきらないステートメントを改行して読みやすくする習慣があり、このケースではステートメントが勝手に分割されてしまうことになりかねない。

```
print a + b
-c;
```

そこで、もしステートメントの途中で改行を入れたときは、２行目以降は字下げしてステートメントが続いていることを明示的に示すことにした。少々面倒な気もするが、多くの C プログラマは読みやすくするため、習慣的に字下げを行っているため、比較的影響が少ないと思われる妥協点である。

```
print a + b
        - c;
```

3.4 コメント

Konoha は、C/C++, Java スタイルのコメントを採用している。// は、行コメントの始まりであり、改行までのテキストがコメントとして無視される。

```
a = b;                                  // 行コメント
```

また、C/C++ や Java と同様に、/* */ で囲むことでコメント化できる。注意すべき点は、Konoha は C 言語と異なり、コメントのネストが可能な点である。

```
/* まとめてコメントアウト
a = 1;
b = 2;  /* コメントの入れ子も OK */
*/
```

3.5 リテラル

リテラルは、プログラム中に直接あらわれたデータ値のことである。型をもった値として扱われる。

次は、基本的なリテラルの例である。

```
12                                      // 整数 12
1.2                                     // 小数 1.2
```

```
"hello world"                    // テキスト
/^world$/                        // 正規表現パターン
true                             // 論理値 true
false                            // 論理値 false
null                             // NULL
```

スクリプティング言語の一般的な特徴は、リテラルが一般的なプログラミング言語に比べ大幅に強化されている点である。Konoha も、データはスクリプトの重要な一部とみなし、配列 (Array), 辞書 (DictMap)、そしてオブジェクトのリテラルをサポートしている。

```
[1,2,3]                          // 配列オブジェクト
{x :1, y: 2}                     // 辞書オブジェクト
C{x: 1, y: 2}                    // クラス C のオブジェクト
```

リテラルは、組み合わせてより複雑なデータを記述することもできる。Konoha のデータリテラルは、JSON などと同様に Web 上でのデータ変換を想定しているため、セキュリティ上の理由から変数や式を含めることはできない。

3.6　識別子

識別子は、名前のことである。Konoha では、識別子は、クラス、変数、メソッド、アノテーションに名づけるために利用される。名前の最初は、アルファベット文字でで始まり、英数字かアンダースコア (_) が続く。次は、全て正しい識別子の例である。

```
i
my_variable_name
v9
Class
MATH_PI
```

Konoha の識別子の特徴は、Java プログラミングの名前付けの慣習 (naming convension) をルール化し、識別子からその種類を判定できるようにした点である。ただし、識別子の一部はコンテキストから判断するしかない場合がある。

3.6.1　変数

変数は、常に英小文字で始まり、英数字、アンダースコアからなる名前をもつ。次は、すべて正しい変数名の例である。

```
n  n2  name  firstName  first_name
```

変数名の先頭にアンダースコアをつけると、それぞれフィールド変数やスクリプト変数をダイレクトに参照する意味となる。

```
_name          // フィールド変数
__name         // スクリプト変数
```

3.6.2 定数

定数は、最初に与えられた値が変更されない特別な変数である。英大文字で始まり、英大文字と数字、アンダースコアからなる。次は、正しい定数名の例である。

```
N  N2  NAME  FIRST_NAME
```

クラス定数は、クラス名とあわせて名付けられる定数である。次は、クラス定数の例である。

```
Int.MAX  Math.PI
```

3.6.3 クラス名

クラス名は、常に英大文字で始まり、英数字からなる。アンダースコア (_) を含めることはできない。次は、全て正しいクラス名である。

```
Int  String  InputStream  C  URN
```

クラス名は、ローカル定数 (アンダースコアなしの定数) と、名前だけから区別することはできない。Konoha コンパイラは、コンテキストによって正しく判断することができるが、ソースコードの読みやすさを考慮に入れたとき、英大文字のみのクラス名 (C や URN) の利用は避けた方がよい。

3.6.4 例外名

例外名は、!!で終わるクラス名である。次は、全て正しい例外名である。

```
Null!!  Security!!  OutOfIndex!!  IO!!
```

Konoha は、Java プログラマへの互換性を配慮して、クラス名が Exception で終わっていた場合、パーサーが自動的に!! に置き換える。つまり、SecurityException は、

Security!!となる。逆に、Exception で終わる名前は、クラス名として用いることはできない。

3.6.5　メソッド名 (関数名)

メソッド名[*1]は、変数と同様に、英小文字で始まる英数字からなる名前を用いる。

Konoha は、Java スタイルの (英語の) 動詞 + 名詞でメソッド名を名付けるスタイルを採用し、名詞の始まりで英大文字を用いることを推奨している。ただし、こればかりはコンパイラでチェックできないので、プログラマの実践に任せるのみである。

次は、推奨スタイルに従ったメソッド名である。

```
get getName readLine query
```

メソッド名の途中でアンダースコアを入れることも認められている。ただし、挿入されたアンダースコアは、取り除かれて、続く 1 文字を英大文字化することで正規化される。

```
get_name                          // getName に正規化
get_host_by_name                  // getHostByName に正規化
read_line                         // readLine に正規化
```

Konoha は、C/C++ など他言語で開発されたライブラリをクラスにバインドして利用することが少なくない。この場合、必ずしも Java 風の名前慣習に従わないライブラリが多く、既存のライブラリ関数をそのまま利用できた方が便利でもある。そこで、Konoha では、メソッドや関数をコールするときのみ、英大文字と英小文字の区別をしない。

次のどの関数呼び出しも、結局、getPid() を呼び出すことになる。

```
getpid()
GetPid()
GET_PID()
```

3.6.6　フォーマッタ名

フォーマッタ名は、% 記号で始まる特別なメソッド名である。命名則は、メソッド名にしたがい、利用するときは英大文字/英小文字の区別はない。次は、フォーマッタ名の例である。

```
%s    %4.2f    %dump    %HTML
```

[*1] 関数は、Script クラスのメソッドであるため、メソッド名のルールに従う。

フォーマッタでは、利用のときに限り、% と名前の間に数字とドット記号 (.) から構成される書式オプションを追加することができる。これは、C 言語の printf 書式の慣例にしたがったものである。

3.6.7 プロパティ変数

プロパティ変数は、実行環境の設定値や環境変数をもった特別な変数名である。$記号で始まり、英数字及びドット記号 (.) から構成される。次は、正しいプロパティ変数名である。

```
$konoha.version
$env.PATH
```

3.6.8 アノテーション

アノテーションは、ステートメントに対して注釈やメタデータを与えるときに用いる。アノテーション名は、Java 言語と同様に、@で始まり、英数字からなる名前である。次は、アノテーション名の例である。

```
@Override   @Final   @Doc
```

3.6.9 ラベル名

英字で始まりコロン (:) で終わるトークンは、予約語を含め、クラス名、変数名、定数名とは関係なく、ラベルとして扱われる。次は、ラベルの例である。

```
default:    ClassName:   variable:   CONST:
```

コロン (:) の前に空白を入れると、そのトークンはラベルと解釈されない。また、より重要な注意は、ラベルの次は、必ず空白を入れることである。コロン (:) の次に空白がない場合は、連続したトークンとみなされる。

```
name : value       // name はラベルでない
name: value        // name: ラベル
name:value         // ひとつの識別子
```

3.7　予約語

Konoha は、「識別子としてプログラム中で利用できない」特別なキーワードをいくつか予約している。次のリストは、Konoha にとって文法の一部、つまり特別な意味をもったキーワードである。

as assert break case catch class continue default defined do double else extends false finally for foreach from goto if import include interface is isa lock namespace new null pragma print return switch this true try typeof using var void where while

いくつかのキーワードは、C や Java, JavaScript などの既存プログラミング言語の記法を解釈するためのエイリアス（別名）として予約されている。

```
abstract                    // @Abstract の別名
boolean                         // Boolean
byte[]                      // Bytes
double                      // Float
final                       // @Final
float                       // Float
int                         // Int
private                         // @Private
protected                   // @Protected
public                      // @Public
```

第 4 章

オペレータ

オペレータとは、1 つ以上の式を結びつけて評価する機構である。+ や=などの演算子と呼び直した方がわかりやすいだろう。近代的なオブジェクト指向プログラミング言語は、ほとんどのオペレータはメソッドの一部として実現し、それをシンタックスシュガー（糖衣構文）として使いやすく提供している。本章では、Konoha が提供するオペレータの全体像を知るため、一般的な紹介を行う。オペレータは、ある特定のクラスや機能と密接に関係したものも多いが、それらは後に続く章において個々に詳説される。

4.1 代入演算子

代入演算子は、変数 x に 右辺式 y を評価した結果を代入 (格納) する演算子である。

```
x = y                              // y の評価値を x に代入する
```

左辺値 x となりうる値は、変数、プロパティ変数、フィールド、配列や辞書に限られる。定数は、未定義の場合に限り、定数値を定義するために代入することができる。次は、代入可能な右辺値の例である。

```
name = "naruto"
$name = "naruto"
p.name = "naruto"
p[0] = "naruto"
p["name"] = "naruto"
P = "naruto"                       // 定数は、1 回のみ代入可能
```

4.1.1　代入式の代入*

Konoha では、代入演算自体も式として扱うことができる。つまり、代入 x = y の評価値は、(代入後の) x となる。したがって、次のように、同じ値を複数の変数へ代入することも可能となる。

```
>>> y = 2
>>> x = y = 1                    // x には、(y = 1) の値が代入
>>> x
1
>>> y
1
```

4.1.2　多値セレクタ**

データ構造から部分的な値を切り出すことは多い。これらのプログラムを簡単に書くため、多値代入を値のセレクタとして利用する拡張計画がある。

```
>>> s = ["naruto", "sakura"]
>>> a, b = s;
>>> a                            // s[0] と同じ (s.opFirst())
"naruto"
>>> b                            // s[1] と同じ (s.opSecond())
"sakura"
```

4.2　コール演算子

コール演算子 () は、メソッド/関数 f、コンストラクタ new C、もしくはフォーマッタ %f に続いて用いられ、メソッド/関数を呼び出すときに用いる。括弧 () の中には、0 個以上のパラメータ/引数を , で区切って与えることができる。コール演算子の評価値は、メソッド/関数の戻り値となる。(ただし、メソッド/関数の戻り値 void 型の場合は、評価値はない。)

```
f()                               // 関数 f() のコール
o.f()                             // o のメソッド f() をコール
new C()                           // クラス C のコンストラクタ
%f(o)                             // o のフォーマッティング
```

コール演算子で与えられるパラメータは、メソッド/関数ごとに個別に定義されている。定義と異なるパラメータでコールしようとしたときは、型エラーとなる。

```
>>> fibo(10)
55
>>> Math.abs(-1)
1.000000
```

4.2.1 コール演算子の省略

コール演算子は、パラメータの数が 1 個であり、かつそれが文字リテラルである場合のみ、括弧 () 自体を省略することもできる。次は、c.query("""...""") と同じであるが、省略した方がすっきりとする。

```
>>> c.query """
select name, salary from PERSON_TBL
  where age > 45 and age < 65;
"""
```

4.3 アドレス演算子

アドレス演算子は、変数 x の値が格納されているアドレスを返す。ただし、Konoha はポインタ演算をサポートしていないため、この機能は完全にクラスルーム目的で導入されている。

```
&x                              // x のアドレス
```

Konoha は、ポインタ型をサポートしていないため、アドレス演算子の演算結果は、int 型である。%p フォーマッタを使えば、アドレスとして表示される。

```
>>> s = "hello, world"
>>> %p(&s)
00000000
```

4.4 比較演算子

比較演算子は、二つの値 x, y を比較するとき用いられ、評価結果は、論理値の true か false である。

```
x == y                          // 等しいか？
x != y                          // 等しくないか？
x < y                           // より小さいか？
x <= y                          // 以下か？
```

```
x > y                          // より大きいか？
x >= y                         // 以上か？
```

Konoha では、全ての型において、x, y の比較が可能である。これは常に意味のある比較となることを保証しない。特に、x と y の型が異なる場合、かつ概念的にも比較が意味をなさない場合は、オブジェクト ID（アドレス値）の比較が返される。どちらにしても、比較演算の評価値は、一意となる。

```
>>> 1 < 2.0                    // 整数と浮動小数点は比較可能
true
>>> 1 == "1"                   // 比較不可能な型
false
```

4.4.1　マッチング演算子

マッチング演算子は、Perl に由来し、主に正規表現と文字列のマッチングに用いられる。

```
x =~ y                         // x は、パターン y と等しいか？
```

マッチング演算子は、==と異なり、対称ではない。パターンは、必ず右辺に与える必要がある。

```
>>> s = "hello, world"
>>> s =~ /w.*d/
true
```

4.4.2　`instanceof` 演算子*

`instanceof` 演算子は、Java に由来し、x がクラス C のインスタンスであるか判定する。Konoha は、純オブジェクト指向言語であり、全ての値は何らかのクラスのインスタンスである。そのため、任意の x に対して、`instanceof` を用いることができる。

```
x instaceof T                  // x は、T のインスタンスか？
```

整数 `1` は、Float のインスタンスではないが、Number のインスタンスである。`instanceof` 演算子は、次のようにテストできる。

```
>>> 1 instanceof Float
false
>>> 1 instanceof Number
true
```

また、null はどのクラスのインスタンスでないため、常に instanceof 演算子の結果は、false となる。ただし、Any 型のみ true となる。

```
>>> null instanceof Object
false
>>> null instanceof Any
true
```

4.5 論理演算子

論理演算子は、論理式（評価結果が論理値となる式）を論理的に連結し、より複雑な論理式を作る演算子である。

```
x && y                    // x かつ y が true のとき
x || y                    // x または y が true のとき
!(y)                      // x が false のとき

x and y                   // x && y と同じ
x or y                    // x || y と同じ
not y                     // !(x) と同じ
```

Konoha の論理演算子は、C/C++, Java 互換の記法と、Python 由来の記法の両方をサポートしている。どちらを使っても構わないが、C/C++, Java との互換性を重んじるなら、&&, ||, ! を用いる方が無難である。

```
>>> a = 1
>>> b = 2
>>> a == 1 && b == 2
true
>>> a == b || a < b
true
>>> !(a == b)
true
```

論理演算を混在したとき、!, &&, || の順に演算される。ただし、() で囲まれた式は優先的に評価される。論理式を混在させるときは、読みやすさのため、() で明示的に優先する論理式を与えるのがよい。

```
>>> a = 1
>>> b = 3
>>> !(a == 1) && b == 3 || a < b
true
>>> !(a == 1) || b == 3 && a < b
false
>>> (!(a == 1) || b == 3) && a < b
true
```

4.5.1　条件演算子

条件演算子は、条件によって評価する式を切り替える演算子である。Konoha は、次の 2 種類の条件演算子をサポートしている。

```
x ? y : z                    // x が true のとき y、そうでなければ z
x ?? y                       // x が null のとき y
```

演算子 `x ? y : z` は、3 項演算子とも呼ばれ、条件式 `x` の評価結果によって、式 `y` か 式 `z` が評価される。3 項演算子は、読みにくくなるので、各式の周りは `()` で囲んだ方がよい。

```
>>> a = 1; b = -1;
>>> a = (a < b) ? (a) : (b);
>>> a
-1
```

演算子 `x ?? y` は、C#で導入された null 検査用の条件演算子である。意味は、`(x != null) ? x : y` と同じである。ただし、null 検査を連続するときは、こちらの方がはるかに読みやすい。

```
>>> name = student["name"] ?? ninja["name"] ?? "unknown";
```

4.6　算術演算子

算術演算子は、2 つの数値 x, y に対し、四則演算を行うときに用いられる。また、文字列など一部のクラスには算術演算子は、連結や分離などの意味で定義されていることもある。

```
x + y                        // 加算
x - y                        // 減算
x * y                        // 乗算
x / y                        // 除算
x % y                        // 余り（モジュロ）
x mod y                      // x % y と同じ
```

モジュロ演算子 `%` は、C/C++, Java で広く採用されているが、Konoha ではフォーマッタ名と区別しにくいので、`mod` を使うか、演算子の前後に明示的に空白を入れることが推奨されている。

算術演算子は、四則演算の優先順位にしたがって評価される。つまり、乗算 (*) と除算 (/) の優先度は、乗算 (+) と除算 (−) より高く、同じ優先度の場合は左から順番に評価される。評価順序を変更する場合は、先に評価したい式を括弧 () で囲めばよい。

```
>>> 1 + 2 * 3
7
>>> 1 + 2 - 3 * 4 / 2
-3
>>> (1 + 2) * 3
9
```

4.6.1 算術代入

算術代入は、代入演算子と算術演算子を組み合わせたシンタックスシュガーである。

```
x += y                    // x = x + y
x -= y                    // x = x - y
x *= y                    // x = x * y
x /= y                    // x = x / y
```

算術代入は、代入可能な右辺値であれば、全て適用することができる。

```
>>> a = [0, 1]
>>> a[0] += 1             // a[0] = a[0] + 1 と同じ
>>> a
[1, 1]
```

ただし、`a[n++] += 1` のような副作用が生じる場合は、期待通りの算術代入にはならない。

4.6.2 インクリメントとデクリメント

Konoha は、”Everything is an Object” の世界で、Int オブジェクトは Immutable なので、本来の意味でインクリメントとデクリメントをサポートできない。しかし、あまりに利用者からの要望が多いため、シンタックスシュガーで対応している。現在のところ、前置、後置演算の区別はない。

```
x++                       // x = x.opNext()
++x                       // x = x.opNext()
x--                       // x = x.opPrev()
--x                       // x = x.opPrev()
```

Konoha では、インクリメント/デクリメント演算子は、ステートメントとして利用することを推奨している。式として利用した場合、その評価順序は、将来にわたって保証されない。

```
>>> for(i = 0; i < 3; i++) print i;
i=0
i=1
i=2
```

4.7　ビット演算子

Konoha は、C スタイルのビット演算子を Int 型に対してサポートしている。スクリプティング言語において、これらの演算子を多用することは稀であり、これらは教育用を主目的[*1]として用意されている。

```
~x                  // ビット反転
x & y               // 論理積
x | y               // 論理和
x ^ y               // 排他論理和 (XOR)
x << y              // 左シフト
x >> y              // 右シフト
```

フォーマッタ `%bits` は、ビット演算子の結果を表示するとき利用できる。

```
>>> %bits(~0)
11111111 11111111 11111111 11111111
>>> %bits(1 << 2)
00000000 00000000 00000000 00000100
```

4.8　シーケンス演算子

Konoha は、シーケンス（順序を保って並んだ集合）に対して、その操作に関する一連の演算子を提供し、それらを総称してシーケンス演算子と呼ぶ。代表的なシーケンスの例は、Array, `byte[]`, String などがある。

```
|s|                 // シーケンス s のサイズ
s[n]                // n 番目の値
s[n] = x            // n 番目に値 x をセットする
s[] = x             // 全部に値 x をセットする
```

[*1] バーチャルマシンにおいて専用命令としてチューニングされないと意味である。

```
x in? s                          // s に x が含まれているか？
s << x                           // s への x の追加
```

シーケンス演算子では、シーケンスの要素の種類によらず、シーケンスのインデックス n に対して、常に $0 \leq n < |s|$ が保証される。(ただし、DictMap や HashMap のように、インデックスの代わりにキーを用いる場合はこの限りでない。)

```
>>> a = [0, 1, 2, 3]
>>> |a|
4
>>> a[1]
1
```

4.8.1 インデックスの範囲

インデックス n が最大値|s|を超えた場合は、OutOfIndex!!例外となる。古い Konoha では、Python と同様に、最後尾から数えてインデックスし直していたが、配列の境界チェックの最適化を考慮に入れて、負のインデックスは見送られた。

```
>>> a[-1]
 ** OutOfIndex!!:
```

4.8.2 シーケンスへの追加

演算子 <<は、特別なシーケンス演算子であり、複数の要素を連続して追加することもできる。

```
>>> a = [1]
>>> a << 2 << 3 << 5 << 7;
>>> a
[1, 2, 3, 5, 7]
```

4.9 スライシング演算子

スライシング演算子は、シーケンスから部分シーケンスを取り出す演算子である。Python のスライシング演算子とよく似ているが、Konoha では部分シーケンスの定義の仕方が 3 種類用意されている。

```
s[x..y]                          // x 番目から y 番目（含む）まで
s[x..<y]                         // x 番目から y 番目まで
                                 //   s[x..(y-1)] と同じ
```

```
s[x..+y]                        // x 番目から y 個の部分
                                //  s[x..(x+y)] と同じ
```

文字列は、シーケンスの一種であり、スライシングで部分文字列を得ることができる。

```
>>> s = "konoha"
>>> s[2..3]
"no"
>>> s[2..<3]
"n"
>>> s[2..+3]
"noh"
```

よく似た演算子が 3 種類存在する理由は、これらはよく混同されて間違えるためである。逆に、3 種類、範囲除外 (..<) やオフセット (..+) をイメージしやすい演算子を導入することで、どれかを正しく使いやすくなる。また、スライシング s[x..y] の範囲 x y は、それぞれ省略可能である。省略したときは、それぞれ x = 0, y = |s| - 1 が初期値として採用される。

```
>>> s = "konoha"
>>> s[2..]
"noha"
>>> s[..4]
"kono"
>>> s[..]                       // シーケンスのコピー
"konoha"
```

4.10　キャスト演算子

Konoha は、オブジェクト間のデータ変換や意味変換を強力にサポートするマッピング機能を独自に導入している。これらは、C/C++, Java 言語でおなじみのキャスト演算子に統合され、利用可能である。

```
(T)x                            // x の型 T への変換（キャスト）
x to? T                         // x は型 T へ変換可能か？
```

マップキャスト演算子 (T)x は、ソース型 typeof(x) とターゲット型 T の関係で動作が異なる。サブタイプの関係が成り立つ場合は、通常どおり、アップキャスト、もしくはダウンキャストされる。それ以外の場合は、マッパーが検索され、それが評価される。

```
>>> Object o = "123";
>>> s = (String)o;                  // ダウンキャスト
>>> n = (int)o;                     // マップキャスト
>>> n
123
```

また、マッピングが定義されていない場合、つまりキャストができない場合は、ClassCast!!例外となる。

```
>>> (int)true
 ** ClassCast!!: Boolean ==> Int
```

これらの演算子は、マッピング機能を応用して実現されている。詳しくは、「第??章 マッピング機能とデータ変換」で述べる。

4.11 イテレータ演算子*

イテレータ演算子は、オブジェクトから (標準) イテレータを得るときに用いる。foreach 文において、イテレータを生成するときに用いる（内部）演算子であるが、プログラマが用いても構わない。

```
x..                                  // x の（標準）イテレータ
```

次の例では、整数配列 Int[] からイテレータ (Int..) を得ている。

```
>>> a = [0, 1, 2]
>>> typeof(a..)
>>> a..
1
2
3
```

イテレータ型や標準イテレータの定義は、「第??章 イテレーション」で詳しく述べる。

4.12 セマンティック演算子**

Konoha は、オブジェクト指向モデルの軽量オントロジ拡張を行い、クラスやオブジェクト間の意味的な関係を比較することができる。

```
x isa? y                             // x は、y か？ IS-A 関係
x === y                              // x と y は意味的に等しいか？
```

Konoha は、文字列や数値に意味タグを付加することができる。次は、華氏 (32.0F) と摂氏 (0.0C) の比較である。

```
>>> Float:F temp = 32.0F;
>>> temp == 0.0C;
false
>>> temp === 0.0C;
true
```

これらの演算子は、マッピング機能を応用して実現されている。詳しくは、「第**??**章 セマンティックプログラミング」で述べる。

第 5 章

ステートメント

プログラムは、ステートメントの集まりである。本章では、Konoha の基本的なステートメントとその文法を説明する。あらかじめ断っておきたいが、Konoha は C/C++, Java とステートメントレベルでの互換性を目標として設計されている。そのため、本章で紹介するステートメントは、（頑張ってかいた割には悲しい事実であるが、）経験豊富なプログラマには読み飛ばしても差し支えない内容である。なお、ステートメントの一般的な訳語は、「文」である。本来、ステートメントに統一すべきかも知れないが、if ステートメントと呼ぶのは冗長な気がするので、広く慣習的に用いられている「if 文」的な呼称を用いている。

5.1 式によるステートメント

もっとも簡単なステートメントは、プログラムの状態を変更する式 (expression) である。代入式は、もっともよく使われるステートメントである。

```
s = "Hello " + name;
i *= 3;
```

インクリメント演算式 (++)、デクリメント演算式 (–) は、代入式の一種であるが、Konoha ではステートメントとして利用することができる。

```
counter++;
```

関数/メソッドのコールも、もうひとつの代表的な式によるステートメントである。

```
System.out.println("hello,world");
Math.sin(0.5);
```

上記以外の任意の式もステートメントとして用いることができる。しかし、プログラムの状態を変更しない式は、コンパイラによって無視され、実際には評価されない。

```
if(a == 1) print a;
a + 1;                          // 無視される
return a;
```

5.2　ブロック

(ステートメント) ブロックは、0 個以上のステートメントまとめてひとつのステートメントとして扱えるようにする。これは、任意のプログラムを中括弧 { }で囲むことで作ることができる。ブロックの終端には、C 時代からの慣習的にセミコロン (;) は必要ない。また、ブロック内では、読みやすさのためインデントするのがよい。

```
{
   x = Math.PI;
   a = Math.cos(x);
   print a;
}
```

注意：Konoha では、メソッド (関数) 内同一スコープであるため、ブロックの中で宣言した変数はブロック外でも有効になる。詳しくは、「第 **??** 章 関数」で述べられる。

5.2.1　空ステートメント

空ステートメントは、何もしないステートメントである。ブロックを用いて空ステートメントを表すことができる。

```
{}
```

5.3　変数の型宣言

Konoha の変数は、静的に型付けされている。プログラマは、新しい変数を使うとき、型宣言が必要となる。C/C++ や Java と同様、次の構文で変数 $name$ の型 T を宣言することができる。

T　$name$;

また、変数宣言では、必要に応じて、代入演算子によって変数の初期値を与えることもできる。また同じ型であれば、複数の変数をひとつのステートメントで宣言できる。

```
String s;                          // 初期値なし
String t = "hello,world";          // 初期値あり
String u, v = "ABC";               // 同じ型の複数の変数宣言
```

5.3.1 クラスと型

Konoha は、オブジェクト指向プログラミング言語であるため、全ての宣言されたクラス C を「C」型として利用することができる。また、クラス名 C に修飾子を付けることで、その型の null 値の扱いを変更したり、配列型やイテレータ型の短縮名も利用することができる。

```
C                                  C 型
C?                                 @Nullable C
C!                                 @NonNull C
C[]                                C の配列型
C..                                C のイテレータ型
```

Konoha の大きな特徴は、型に@Nullable または@NonNull を修飾できることである。これらは、名前が示すとおり、変数の値として null 値を認めるかどうかであるか、を制約を付加したものである。Java との大きな違いは、何も修飾しなければ、原則として NonNull 型となる点である。この場合、変数宣言の初期値は、null ではなく、そのクラスのデフォルト値となる。

```
>>> String s;                      // 初期値のデフォルト値
>>> s
""
>>> s = null                       // Null 例外が発生
 ** Null!!
```

注意：各クラス C のデフォルト値は、組み込み関数 default(C) を用いて得ることができる。また、デフォルト値の中には Null オブジェクトパターンにしたがって、null 値のように振る舞う値もある。

逆に、C#風に?を型名に付けるか、@Nullable 修飾子を用いれば、null を保持可能な変数となる。この場合の初期値は、null である。

```
>>> String? t;                     // 初期値は null
>>> t
null
```

5.3.2　var 文*

var 文は、JavaScript など一部のスクリプティング言語で採用されている変数宣言のための特別なステートメントである。Konoha では、静的に型付けして型宣言を行うため、var 文では初期値から型推論を行い、静的な型付けを行っている。

```
var i;                                  // Any 型（型推論なし）
var s = "hello,world";                  // s は、String 型
var x = Math.cos(0.75);                 // x は、Float 型
```

注意：var 文は省略可能である。始めての使用する変数への代入は、型推論による変数宣言とみなされる。詳細は、「第?? 章 ダイナミック・スタイル」で解説する。

5.4　if/else 文

if 文は、もっとも基本的な制御ステートメントである。与えられた条件式に応じて、続いて実行するステートメントの切り替えるときに使用する。if 文には、2 種類の形式がある。次は、簡単な形式である。

`if` (*expr*) *stmt*

この形式では、条件式 *expr* がまず評価される。その結果が、true であれば、それに続くステートメント *stmt* が実行される。もし条件式の結果が false であれば、ステートメント *stmt* は実行されない。

```
>>> n = 4
>>> if (n mod 2 == 0) print "even";
even
```

if 文に続くステートメントとして、ブロックを用いることもできる。ブロックを用いれば、0 個以上のステートメントを順次実行することもできる。

```
>>> if (n mod 2 == 0) {
...     print "even";
... }
even
```

5.4.1　else 節

if 文に続く else 節は、別のステートメントを与えるときに利用する。

`if` (*expr*) *stmt*1 `else` *stmt*2

この形式の if 文では、まず条件式 $expr$ が評価され、もしそれが true であればステートメント $stmt1$ が実行され、そうでなければ $stmt2$ が実行される。

```
>>> n = 3
>>> if (n mod 2 == 0) {
...     print "even";
... } else {
...     print "odd";
... }
odd
```

5.4.2 else if

if/else 文は、2 つのステートメントの実行を分岐させることができた。更に、if/else 文を組み合わせ、あたかも else if 文のように用いると、3 つ以上の条件分岐をすっきりと書くことができる。

```
if(n == 1) {
  // n が 1 のとき
}
else if(n == 2) {
  // n が 2 のとき
}
else if(n == 3) {
  // n が 3 のとき
}
else {
   // それ以外のとき
}
```

else if 文は、特別なステートメントではなく、ふつうの if/else 文の組み合わせたに過ぎない。条件式は、上から、つまり `(n == 1)` から、順番に評価されていくため、後に続く条件判定のステートメントほど処理時間が遅くなる。

5.5 switch 文**

switch 文は、多重ディスパッチを行うための専用のステートメントである。文法は、C/C++ や Java と同様に次のとおりである。式 $expr$ の評価結果にマッチする case 節が選択され、そのステートメントが実行される。

```
switch (expr) {
   case c1 : stmt1;
   case c2 : stmt2;
```

```
    ...
    case cn : stmtn;
    default : stmt;
}
```

switch 文を用いると、一般に　if/else 文よりもすっきりと複雑な条件分岐をかくことができる。次は、else if の節で用いた多重分岐の switch バージョンである。注意すべき点は、break 文である。これは、各 case 節は、条件分岐の始まりを表すに過ぎず、明示的に分岐の終端を表している。

```
switch(n) {
  case 1 :
    // n が 1 のとき
  break;
  case 2 :
    // n が 2 のとき
  break;
  case 3 :
    // n が 3 のとき
  break;
  default:
   // それ以外のとき
}
```

Konoha と C/Java との違いは、case 節のマッチングにある。C や Java は、case 節では整数リテラルのみマッチングする条件として与えることができたが、Konoha では任意のリテラルを与えることができる。（ただし、式は与えることができない。）

```
switch(lang) {
  case "perl" :
  case "python" :
    //
  break;
  case "cpp" :
    //
  break;
  default:
   // それ以外のとき
}
```

注意：この辺りが悩みどころで、コードディスパッチの最適化ができなくなるので、switch 文の実装は済んでいない。

5.6 while 文

while 文は、あるステートメントを繰り返し実行させる基本的なステートメントである。ループの制御構造とも呼ばれる。その文法は次のとおりである。条件式 $expr$ の評価結果が true の間、続くステートメント $stmt$ を繰り返し実行する。

```
while (expr) stmt
```

次は、変数 a が 0 より大きい間、繰り返し実行される。

```
a = 1;
while(a > 0) {
   a = Int.random(10);            // 乱数生成
   print a;
}
```

5.6.1 無限ループ

while 文の条件式に定数として true を与えれば、常に真であるため無限にループを繰り返すことになる。

```
while(ture) {
   // 無限ループ
}
```

注意：無限ループは、break 文を内部で用いるか、throw 文による大域ジャンプによって抜け出すことができる。そのため、多くのプログラマは何らかの脱出方法を用意して、確信犯的に無限ループを使用している。Konoha コンパイラは、本当に無限ループかどうか判定をするのは難しいため、エラーも警告も出力しない。

5.6.2 do/while 文

do/while 文は、条件判定の順序が異なる while ループのバリエーションである。

```
do stmt while (expr);
```

do/while ループは、先にステートメント $stmt$ を一度実行したあと、条件式 $expr$ によって繰り返しの判定を行う。while 文と do/while 文の違いは、while 文が条件に次第では一度もステートメントを実行しないこともありえるのに対し、do/while 文の方は必ず１回はループが実行される点にある。

```
do {
   a = Int.random(10);
   print a;
}while(a > 0);
```

5.7　for 文

for 文は、n 回の繰り返しなどを書くときに便利な while 文の置き換えである。最初のステートメント $stmt_1$ でループに入る前の状態の初期化を行い、ステートメント $stmt_3$ でループを実行したのちの状態の変化、条件式 $expr_2$ でループの終了判定を行う。

for （ $stmt_1$; $expr_2$; $stmt_3$ ） $stmt$;

for 文は、ある回数を繰り返し実行するときに多く利用される。実際、次の for 文と while 文は全く同じであるが、多くの慣れたプログラマにとって、for 文の方が while 文よりも読みやすく処理の内容も把握しやすい。

```
for(i = 0; i < 10; i++) {    // for 版
   print i;
}

i = 0;                       // while 版
while(i < 10) {
   print i;
   i++;
}
```

for 文は、C/C++ や Java で広く使われているため、Konoha でも採用となった。ただし、モダンなプログラミングスタイルでは、for 文よりも、イテレーションパターンを扱う foreach 文（後述）の方が好まれる。

5.7.1　for 文とイテレーション

モダンなオブジェクト指向プログラミングでは、繰り返しのループ処理はイテレーションパターンを用いることが多くなっている。Java は、Java5 からイテレーションを扱うため、for 文を独自に拡張している。

```
for(Object e : list.iterator()) {
   print e;
}
```

Konoha では、イテレーションを扱う専用のステートメントとして、後述するとおり、foreach 文を導入している。ただし、Java 風の拡張 for 文も、Konoha スタイルの foreach 文に翻訳されて実行される。

```
foreach(Object e in list.iterator()) {
   print e;
}
```

5.8 break 文と continue 文

Konoha は、break 文と continue 文という 2 種類のループ制御のステートメントをもっている。

break 文を使うと、現在繰り返しているループ構造のブロックから抜け出すことができる。

```
while(true) {
   dice  = Int.random(6) + 1;
   dice2 = Int.random(6) + 1;
   print dice, dice2;
   if(dice == dice2) break;     // ループから抜ける
}
```

continue 文を使うと、現在繰り返しているループ構造のブロックの残りのパートをスキップする。

```
for(i = 0; i < 10; i++) {
   if(i % 2 == 0) continue;     // ループの先頭へ
   print i;
}
```

5.8.1 多重ループとラベル

ループ構造は、while 文や for/foreach 文を組み合わせて、多重ループを作ることができる。多重ループ中では、break 文や continue 文を用いても、どのループを対象としているか曖昧になる。伝統的な C/C++, Java の定義では、最も内側のループのみ対象としてきたが、外側のループを対象としたい場合も多くある。そこで、ループにラベルをつけることで、どのループからの break/continue なのか明示的に示すことができる。また、ラベルを省略すれば、今まで通り暗示的に最も内側のループが対象となる。

```
OUTER:
for(y = 0; y < 8; y++) {
   INNER:
```

```
    for(x = 0; x < 8; x++) {
        print x, y;
        if (x == y) continue OUTER; /* 外側 */
        if (x < y) break; /* 内側 */
    }
}
```

5.9　foreach 文

foreach は、イテレータパターンを扱う専用のループ構造である。foreach 文は、次の２種類のバリエーションが存在するが、どちらもイテレータ評価式 *itr* によるイテレータ内の各オブジェクトをイテレーション変数 *var* に代入し、イテレータ内のオブジェクトがなくなるまで、ステートメント *stmt* の実行を繰り返す。

```
foreach( var  in  itr )  stmt;
foreach( var  from  itr )  stmt;
```

foreach 文は、多くの場合、for 文を用いるよりも可読性が高くなり、また（専用命令のおかげで）高速な処理が期待される。

```
>>> a = [0, 1, 2]
>>> for(i = 0; i < |a|; i++) {    // for 版（古風）
...   n = a[i];
...   print n;
... }
n=0
n=1
n=2
>>> foreach(n from a) {           // foreach 版
...   print n;
... }
n=0
n=1
n=2
```

5.9.1　イテレータの評価式

foreach 文の in/from 節は、任意の式からイテレータをえるため、イテレータ評価式と呼ばれる。Konoha では、イテレータをえる方法は、マッピングによる方法とイテレータ演算子を用いる方法の 2 種類が存在し、foreach 文ではイテレーション要素の型が与えられているかどうかで、自動的に使い分けられる。

型が与えられている場合

イテレーション要素の型が決まっている場合、つまりイテレーション変数の型が明示的に与えられている場合、foreach 文は、マッピングによってイテレータを評価する。つまり、次の場合、イテレーション変数 `line` は String 型であるため、イテレーション評価式は、`(String..)(file)` で評価される。

```
file = new InputStream("file.txt");
foreach(String line from file) {
   print line;
}
```

型が与えられている場合

イテレーション要素の型が決まっていない場合、foreach 文は、イテレーション演算子でイテレータを取り出す。次の場合、`(file)..` で評価される。同時に、このとき得られたイテレーションの型 `(int..)` からイテレーション変数 tt の型は、`int` と型推論される。

```
file = new InputStream("file.txt");
foreach(ch from file) {
   print ch;
}
```

5.9.2 フィルタと変換

foreach 文の in/from は、ほぼ同じ動作をする。違いは、イテレータから得られるオブジェクトとイテレーション変数の型が異なった場合、 フィルタ処理にするか、変換を試みるかの違いである。

in 節：型検査によるフィルタ

foreach 文と in は、多くのプログラミング言語で組み合わせられているペアであり、イテレータ内部のものを加工せずに取り出す意味論に近いため、型検査によるフィルタ処理を行っている。

```
>>> a = ["naruto", 9, "gahra", 1];
>>> foreach(String s in a) {
...   print a;
... }
a="naruto"
a="gahra"
```

from 節：マッピングによる変換

from 節は、本来、次節で述べる通り、 クエリー言語に由来しており、特別に変換の意味と結びつけられたものではないが、Konoha の foreach 文では変換しながらのイテレーション処理と定義されている。変換できなかったオブジェクトは、無視される。

```
>>> a = ["naruto", 9, "gahra", 1];
>>> foreach(String s in a) {
...   print a;
... }
a="naruto"
a="9"                              // String へ変換
a="gahra"
a="1"                              // String へ変換
```

5.9.3　where 節*

Konoha における foreach 文は、SQL や XMLQuery など、クエリー言語をモデルにしている。そのため、クエリー言語の where 節に相当する検索条件を書くことができる。

```
foreach( var  from  itr  where  expr )  stmt;
```

現在は、where 節による条件検索は、Konoha 言語処理系の上で実行されているため、それほど高速ではない。将来は、foreach 文の処理を一部ダイレクトにクエリー展開し、データソースアクセスとスクリプティングをシームレスに統合する計画がある。

```
foreach(Person p
          from 'db:PERSON_TBL'
          where p.age > 20 and p.age <= 60) {
   print p.name, p.salary;
}
```

5.10　ステートメント・アノテーション*

Konoha は、全てのステートメントに対し、アノテーションを付けることで、ステートメントの意味や振る舞いを拡張することができる。アノテーションは、@マークで始まる識別子であり、ひとつのステートメントに対し、複数のアノテーションを同時に与えることもできる。ただし、アノテーションとステートメントの間にセミコロン (;) を入れてはならない。

```
@Date print n;
@Const @Nullable String s;
```

各ステートメントによって、対応しているアノテーションの種類は異なる。未定義のアノテーションを利用しても、現在のところ、エラーも警告も発生しない。将来は、ユーザが自由に拡張したアノテーションを使うフレームワークを用意する予定である。

第 6 章

数値

コンピューティングプロセッサ (CPU) は、コンピュータシステムの中心部であり、様々な形式の整数や浮動点小数の演算をサポートしている。プログラミング言語は、これらを, 例えば C 言語の場合、char, short, int, long long long int, float, double, long double のように、異なる型で区別して利用できるようになっている。一方、Konoha は、プロセッサの性能やメモリ効率を最大利用するより、プログラミングのしやすさに力点を置いている。そのため、最も使いやすい整数と浮動小数点の形式を選んで、Int と Float クラスとして箱詰め (boxing) している。本章では、これらのクラスを用いた数値の扱いを説明する。

6.1 数値と型

Konoha は、2 種類の数値表現をサポートしている。これらは、C 言語における 64 ビット符号付き整数 (long long int) と倍精度浮動小数点数 (double) に相当し、日常的な利用においては自然数や実数を表現するのに十分な値域と精度を持っている。

Konoha では、“Everything is an Object” の設計であり、数値を含め、すべての値はオブジェクトとなっている。Int クラスと Float クラスは、それぞれ整数値と浮動小数点数値を箱詰め (boxing) してインスタンス化するためのクラスである。

```
>>> (1).class
Int
```

注意：「箱詰め (boxing)」の話だけをすると、プログラミング言語実装に詳しい方は不安になるだろう。Konoha の内部をより正確にいえば、オブジェクトとしてアクセスするときのみ、boxing されており、それ以外は、もちろん unboxing された状態で数値処理されている。不必要なメモリ消費は最低限に抑えられている。

6.1.1　int, float 型

Konoha では、クラス名は英大文字で始まり、そのまま型名として用いられる。このルールにしたがえば、整数値、浮動少数点値を用いるための型は、Int, Float 型となる。しかし、C/C++, Java など、ほとんどすべてのプログラミング言語において、これらの型は int, float のように英小文字で始まっている。

ここで、言語設計の統一的美しさにこだわれば、Konoha のルールにしたがってもらえばよいものであるが、「言語」というものは実際に使っている人たちが主人公である。プログラミング言語も、そろそろ半世紀の歴史をもつわけであるから、言語設計者が好き勝手に文法を決められる時代はもう終わったと考えてよいだろう。統一的な美しさはないが、Konoha では、int や float も別名定義し、プログラム中で利用することができる。

最後に、Konoha 風の Int と伝統的な int のどちらを使った方がよいかという問題が残る。Konoha のローカルルールに固執して Int/Float を使うより、C/C++ や Java などとソースコードの交換性を保つため、int/float を使う方がよいと考える。また、同じ理由で long や double 型も定義されており、Int/Float の別名として定義されている。

6.1.2　Decimal**

将来、小数点以下の精度を保持した無限長の数値をサポートすることを計画している。

6.2　数値リテラル

数値は、ふつうに表記すれば、アルファベットで始まる予約語や識別子と容易に区別がつけられる。そのため、数値リテラルと言っても、極めて自然な表記であるが、整数と浮動小数点を区別するため、表記上のルールがある。

6.2.1　整数リテラル

スクリプティング言語では、short や long などサイズの異なる整数を使い分ける用途はあまり想定しにくく、単純に自然数を表現するために整数を用いることが多い。Int クラスでは、数値の上限下限を最も気にしなくて済む 64 ビット整数[*1]を採用している。

Konoha では、10 進数の整数は、数字の列として書ける。整数の値の範囲は、-9,223,372,036,854,775,807 から 9,223,372,036,854,775,807 となる。つまり、日常の利用ではまず足りなくなる心配のない値域をカバーしている。

```
0
3
```

[*1] KNH_USING_INT32 オプションを用いれば、32 ビット整数に変更することも可能である。

```
100000000
```

10 進数表記に加えて、Konoha は、16 進表記と 2 進表記をサポートしている。16 進表記の整数リテラルは、`0x` もしくは `0X` で始まり、続いて数字 (0-9) もしくは 10 から 15 までの数字を表す英字 (A-F) が続く形式である。

```
0xff                                    // 15 * 16 + 15 = 255
0xCAFE911
```

クラスルームでは、結構、2 進数の説明も頻出する。そのため、Konoha は独自の 2 進数リテラルとして、`0b` もしくは、`0B` から始まり、0 もしくは 1 が続く 2 進数で整数を与えることができる。

```
0b1111                                  // 0xff = 255
```

6.2.2 Float リテラル

浮動小数点数 (floating-point numbers) は、全て Float クラスの値として表現される。その名前から想像されるものとは異なり、IEEE 745 標準で定義された 64 ビット倍精度形式を用いている。これらは、C/C++, Java 言語では、通常 double 型で表現される形式である。

Float リテラルは、小数点を必ず含む。通常、整数パートの数列に始まり、小数点、続いて小数点以下の数列が続く形式である。また、e もしくは E によって 10^n 乗形式の位を追加することも可能である。

```
3.14
2345.789
.33333333333
6.02e23
1.473E-32
```

6.2.3 桁取り表記

我々は、日常、大きな数字を間違えることなく扱うため、1000 桁ごとにカンマ (,) を入れて表記することが多い。プログラミング言語では、カンマは特殊な意味をもった演算子の一種として扱われるが、Konoha では、アンダースコア (_) を数値リテラルの任意の箇所に入れることが認められ、カンマの代わりに桁取り表記も可能となる。OCaml 風の拡張であるが、ちょっとだけ読みやすくなる。

```
1_0000
9_223_372_036_854_775_807
```

6.2.4　意味タグ拡張と単位*

Konoha は、セマンティックプログラミングの機能として、Int や Float クラスに対して意味タグを付加することができる。意味タグは、単位など直感的なタグを用いることが可能であり、数値リテラルの後置辞として利用することができる。詳しくは、「第 **??**章 セマンティックプログラミング」で解説する。

```
32.195km                                    // Float:km
80[km/h]                                    // Int[80/km]
```

注意：意味タグが定義されていないときは、意味タグは無視されて、通常の整数、浮動小数点数のリテラルとして扱われる。型チェックはおこなわれないが、読みやすさのために意味タグを使っても構わない。

6.3　数値演算

Konoha では、数値はクラスによって箱詰めされているが、その値はそのままネイティブの数値表現、C 言語で言うところの long long int と double である。したがって、その数値演算の振る舞いは、C 言語の演算子と同じである。特に、浮動小数点演算において計算結果が微妙に異なることがバグレポートとして報告されるが、そもそも浮動小数点演算というのは桁落ち、丸め誤差などが含まれていることを思い出して欲しい。

6.3.1　四則演算

Int, Float は、それぞれの型の範囲において四則演算 (+,-,*./) を定義している。つまり、Int 型間の演算では Int 型の結果がえられ、Float 型間の演算では、結果は Float 型になる。

```
>>> 7 / 2                                  // Int 型の除算
3
>>> 7.0 / 2.0                              // Float 型の除算
3.5000000
```

一方、Int 型と Float 型を混在して演算したときは、Int 型の値は全て自動的に Float 型に変換されたのち、演算される。したがって、結果は Float 型となる。

```
>>> 7 / 2.0                              // 7 は 7.0 へ
3.5000000
>>> 7.0 / 2                              // 2 は、2.0 へ
3.5000000
```

ゼロ除算の場合は、Int, Float ともに、`Arithmetic!!`例外が通知される。

```
>>> 1 / 0                    // ゼロ除算
Arithmetic!!: Zero Divide
```

優先順位

Konoha の四則演算は、日常の数学の演算と同じ順序で計算される。つまり、乗算と除算は、加算と減算に優先して実行される。演算の優先順位を変更するときは、優先対したい演算を () で囲む。

```
>>> 1 + 2 * 3
7
>>> (1 + 2) * 3
9
```

Konoha では、四則演算以外にも数多くの演算子をサポートしている。これらを混在して利用するときは、優先する演算子を () で囲む習慣をつけると、思わぬ誤動作を防ぎ、また読みやすさも向上する。

6.3.2 ビット演算

Konoha は、Int 型の数値のみ、ビット演算をサポートしている。

```
>>> %bits(1)
00000000 00000000 00000000 00000001
>>> %bits(~1)
11111111 11111111 11111111 11111110
>>> %bits(2)
00000000 00000000 00000000 00000010
>>> %bits(1&2)
00000000 00000000 00000000 00000000
>>> %bits(1|2)
00000000 00000000 00000000 00000011
>>> %bits(1<<4)
00000000 00000000 00000000 00010000
>>>
```

スクリプティング言語では、低レベルな処理を書く機会が少ないため、ビット演算子を活用する機会は少ない。さらに、Konoha では、ビット演算子の記法は他のよく使われるオペレータ (`|a|y` や `OUT << "hi"`など) に転用されているため、演算子の優先度が C/C++, Java と異なることがある。この違いを覚えるよりは、ビット演算子と四則演算子を組み合わせるときは、`()` で囲んで優先する演算子を明示した方がよい。

```
>>> %bits(1+1<<2)                    // 優先度が？
00000000 00000000 00000000 00001000
>>> %bits(1+(1<<2))
00000000 00000000 00000000 00000101
>>> %bits((1+1)<<2)
```

```
00000000 00000000 00000000 00001000
```

Float の数値を IEEE 754 浮動小数点のビットレイアウトにしたがってビット演算をしたいときは、まず `Float.floatToIntBits()` を用いて Int 型へ変換したのち、`Float.intBitsToFloat()` で Float 型に変換することで可能になる。ちなみに、Konoha は 32 ビットシステムであっても、Int と Float はともに 64 ビットで表現される。

```
>>> n = (float)1.0;                    // 数値変換
>>> %bits(n)
00000000 00000000 00000000 00000001
>>> f = Float.floatToIntBits(1.0)      // ビットレイアウト維持
>>> %bits(f)
00111111 11110000 00000000 00000000
```

6.3.3　比較演算

Int 型, Float 型は、ともに比較演算子 (==, !=, <, <=, >=, >) によって数値を比較できる。また、Int 型、Float 型の比較を混在して用いたとき、Konoha は通常、異なる型の比較はオブジェクト ID による比較になるが、特別に自動的に Int クラスを Float クラスに変換して比較を行う。

```
>>> 1 < 2.0
true
>>> 3.2 != 3
false
```

Int 型の比較演算子は、実用上問題がおこらないが、Float 型の場合は、IEEE754 規格に基づいた表現のため、実用上は同値であっても同値とみなされない。そこで、(正規表現のための) マッチング演算子=~を用いると、簡単にほぼ等しいを演算することができる。有効桁数は、小数点以下 5 桁である。

```
>>> 1.00003 == 1.0
false
>>> 1.00003 =~ 1.0
true
```

6.3.4　数値変換

Konoha では、Int 型と Float 型の型変換は、必要に応じて自動的に行われる。これらを明示的におこないたい場合は、キャスト演算を用いることができる。ただし、Konoha の数値変換の振る舞いは、C 言語の数値変換の振る舞いに基づいている。特に、Float から Int への変換は、切り捨てになるため注意が必要である。

```
>>> (int)1.8
1
>>> (int)-1.8
-1
```

, 小数点以下 n 桁で切り上げ、四捨五入、切り下げを行いときは、math パッケージのクラス関数 `Math.ceil(),Math.round(),Math.floor()` を用いる。

```
>>> using math.*;
>>> Math.ceil(1.8)
2.0000000
>>> Math.round(1.8)
2.0000000
>>> Math.floor(1.8)
1.0000000
```

6.3.5 文字列との変換

Konoha は、全てのクラス間の変換がキャスト演算子に統合されている。文字列への変換も文字列からの変換もキャストで行うことができる。ただし、数値の書式を指定したい場合は、次節のフォーマッタを用いる。

```
>>> (String)1.8
"1.8000000"
>>> (int)"123"
123
```

文字列から数値への変換は、数値リテラルの文法も解釈される。

```
>>> (int)"1"
1
>>> (int)"0xff"
255
>>> (float)"6.0221415e23"
602214149999999896780800.000000
```

数値以外の文字列は、それぞれ Int 型、Float 型のデフォルト値へ変換される。デフォルト値の代わりに（変換に失敗したことを知るために）null を得たいときは、Nullable キャストを用いることができる。

```
>>> (int)"hello,world"          // 数値でない
0
>>> (int?)"hello,world"         // 数値でない
null
```

6.4　数値フォーマッタ

数値フォーマッタは、C 言語 printf 書式に由来しているものが多いが、独自に定義されたものは %bits などいくつか存在する。ここでは、代表的な数値フォーマッタについて紹介する。

6.4.1　10 進書式 %d

10 進数 (digit) 書式では、文字列の幅 m を %md のように指定できる。また、%0md ように書式を与えると、空白の代わりに 0 で埋められる。(これは、数値としての順序が文字列化したときも保持されるため、よく用いられる。)

```
>>> %d(123)
"123"
>>> %4d(123)                    // 幅 (4) を指定
" 123"
>>> %-4d(123)                   // 左寄せ
"123 "
>>> %04d(123)                   // 0 パディング
"0123"
```

6.4.2　16 進書式 %x

10 進数 (digit) 書式では、10 進書式同様に、文字列の幅 m を %mx のように指定できる。

```
>>> %x(123)
"7b"
>>> %8x(123)                    // 文字列の幅 8
"      7b"
>>> %08x(123)                   // 0 パディング
"0000007b"
```

6.4.3　小数書式 %f

小数書式は、Float クラスだけでなく、Int クラスもサポートしている。したがって、整数も型変換することないしに、小数書式でフォーマッティング可能である。

```
>>> %f(1)
1.000000
>>> %f(1.0)
1.000000
```

小数書式では、全体の幅 m と小数点以下の桁数 n を %m.nf のように指定できる。

```
>>> %f(3.14159)
"3.141590"
>>> %6.2f(3.14159)                    // 小数点以下 2 桁
"  3.14"
>>> %6.3f(3.14159)                    // 小数点以下 3 桁
" 3.142"
```

6.4.4 ビット書式 %bits

ビット書式は、printf 書式に存在しない独自の書式である。整数や浮動小数点数のビットレイアウトを確認するために導入されており、クラスルーム用途の書式である。

```
>>> %bits(-1)                         // 2 の補数表現 :)
11111111 11111111 11111111 11111111
```

6.4.5 UCS4 文字書式 %c

文字書式は、printf 書式でもおなじみの書式であり、整数から文字コードを用いて文字に変換してくれる。ただし、Konoha 言語では、Unicode 文字コード (UCS4) を用いているため、ASCII 文字コード以外の文字も変換することができる。

```
>>> %c(30)                            // ASCII (<128)
"A"
>>> %c(0x3042)                        // UCS4  (>128)
"あ"
```

6.5 乱数生成

乱数生成は、Int と Float クラスの値をランダムに生成する一種のコンストラクタである。Konoha では、標準乱数生成器として、統計処理分野で愛用される高品質な疑似乱数アルゴリズム Mersenne Twister [1] を採用している。

乱数生成は、クラス関数として提供されている。`Int.random(n)` は、0 から n(含まれない) までの整数乱数を生成し、`Float Float.random()` は、浮動小数点乱数 x ($0.0 < x < 1.0$) を生成するクラス関数である。

例えば、さいころ (1〜6) を作りたいときは、次のように用いることができる。

```
>>> Int.random(6) + 1                 // さいころ
3
>>> Int.random(6) + 1                 // 違う目が出た
6
```

6.5.1　乱数生成器の初期化

Konoha は、起動時の時刻とラインタイム情報から起動する度に乱数生成器を初期化している。そのため、通常、乱数生成器を初期化する必要はない。ただし、明示的に乱数生成器を初期化したいときは、`System.setRandomSeed(seed)` を用いることで、再初期化可能である。(パラメータ `seed` に対し、0 を与えたときのみ、その時刻とランタイム情報からシードが生成される。)

```
>>> System.setRandomSeed(1)     // 初期化
>>> Int.random()                // 初期化後は同じ
1234794094773155764
```

注意：同じ seed 値で初期化すると、乱数生成系列が同じになる。つまり、毎回、同じ順番で同じ乱数生成が繰り返される。ただし、Konoha の乱数生成器は、複数の言語エンジン/スレッドから共有されているため、マルチ言語インスタンスやスレッド実行下での乱数生成は、逆に同一性は保証されなくなる。

第 7 章

文字列

文字は、人類史の始まり以来、最も重要な情報表現の手段である。情報社会といわれる現在においても、社会や文化を支える主要な情報は、文字によって記録され、伝えられている。そのため、プログラミングにおける文字の重要性も明らかといえる。文字やテキストをうまく扱うことは、情報を扱う処理の中心的な役割を占めているからである。

プログラミング言語では、文字を 1 文字、1 文字扱うのは不便なので、複数個以上の文字の並びを「文字列 (string)」として扱う。Konoha は、文字列を表現するため、String クラスを提供し、独自のフォーマティング機能とあわせて、自由自在な文字列操作を実現している。

7.1 文字列と型

ほぼ全てのプログラミング言語は、海外で生まれたという経緯もあって、長らく「文字＝ ASCII コード＝ 1 バイト」という前提で開発されていた。これが不幸の始まりで、低レベルな処理をあまり意識しないで済むはずのスクリプティング言語であっても、日本語を処理する場合はどうしても 1 文字が何バイトに相当するか注意しながら文字列処理を強要されてきた。

Konoha は、21 世紀になってから設計されたプログラミング言語であり、Unicode 文字を採用している。また、1 文字が何バイトでエンコーディングされていても 1 文字と数えている。つまり、文字列処理において、文字コードやバイト数を意識しなくて済むようになっている。

文字列は、0 個以上の Unicode 文字からなる。Konoha では、文字単体を扱う特別な型は存在せず、文字は長さ 1 の文字列として表現される。

```
"a"                          // ASCII 文字
"あ"                         // Unicode 文字
```

どうしても文字コードを扱う必要があるときは、UCS4(31 ビット) コードとして、Int

型を用いて扱うことができる。

```
>>> "a".getChar()          // ASCII コード
97
>>> "あ".getChar()         // UCS4 コード
12354
```

Konoha は、String の内部エンコーディングとして、UTF8 を利用している。本来、文字列は実装依存しない抽象化されたデータ構造であるべきであるが、UTF8 以外はバイト列 (Bytes/byte[]) して扱うことになる。String から、各種エンコーディングにしたがってバイト列に変換する方法も用意されている。

```
s.getBytes("shift_jis")  // Shift_JIS へ変換
new String(byte, "shift_jis");
```

注意：入出力ストリームにはエンコーディングが設定できるようになっているため、エンコーディングを処理するケースはあまりない。

7.2　文字列リテラル

プログラミング言語では、「プログラム要素の識別子」と「データとしての文字列」を区別するため、文字リテラルのルールにしたがって、文字列をデータ値として記述する。Konoha は、C/C++、Python、C#に由来する 3 種類の文字列リテラル、さらに独自のリテラルもサポートしている。ここでは、これらを順番に説明する。

7.2.1　ダブルクオート

最も基本的な文字列リテラルは、文字列をダブルクオート (") で囲む形式である。このリテラルは、C 言語を含め、ほぼ全てのプログラミング言語で広く採用されている。

```
""                          // 空文字列
"hello, world"
"3.14"
"Say \"hi.\""               // エスケープ記号
"いろは"                    // Unicode 文字
```

文字列リテラルとして、そのまま表現できない文字、例えば改行や"は、エスケープ記号 (\) を用いて記述する。Konoha は、次の 6 種類のエスケープシーケンスのみサポートする。

```
sequence        Character represented
\n              改行 (LF)
```

\t	**タブ**
\\	**\記号**
\'	**' シングルクオート** （""内）
\"	**"ダブルクオート** （'' 内）
\:	**:コロン** （'' 内）

7.2.2 シングルクオート

Konoha では、シングルクオート (') も、コロン (:) の扱いに注意すれば、文字列リテラルの始まりと終わりとして利用することができる。シングルクオートを用いると、ダブルクオート (") を含む文字をすっきりと表すことができる。

```
'abc'                    // "abc"と同じ
'Say "hi"'               // "Say \"hi.\"" と同じ
'http\://'               // : はエスケープが必要
```

シングルクオートは、正式には「タグ付き文字列」のリテラルとして定義されている。タグ付き文字列とは、文字列の先頭に'tag:...' のようなタグを付けて、文字列の意味を識別し、その意味にしたがって適切なオブジェクトを生成するリテラルである。

```
're:s$'                  // 正規表現 (Regex)
'file:/tmp/konoha.k'
'isbn:978-4-87311-329-6'
'IO!!:File Not Found'    // 例外 (Exception)
```

注意：「タグ付き文字列」のタグが未定義な場合は、通常の文字列リテラルとして解釈される。ただし、明示的にタグでないことを示すためには、エスケープ (\:) する必要がある。詳しくは、「第**??**章 セマンティックプログラミング」で述べる。

7.2.3 バッククオート

バッククオート (`) も、% 記号の扱いに注意すれば、文字列リテラルの始まりと終わりの記号として利用できる。こちらを用いれば、シングルクオートとダブルクオートが混在する文字列をすっきりと書くことができる。

```
`Naruto said, "I'm a ninja."`
```

バッククオートは、正式には「インライン・フォーマッティング」、つまり文字リテラル中の書式が評価される特別な文字列リテラルとして用いられる。次は、%d{}内の式が評価され、%d フォーマッタで整形された文字列が埋め込まれている。

```
>>> core =2
>>> `make -j%d{core+1}`
"make -j3"
```

もしバッククオートを文字リテラルとして利用するときは、% 文字は、\%、もしくは %% のようにエスケープ処理が必要になる。詳しくは、「第 7.5.4 節 インラインフォーマッティング」で述べる。

7.2.4　トリプルクオート

トリプルクオートは、Python に由来するテキスト用の文字リテラルである。”’ もしくは、”””” で囲まれたテキストは、改行やタブ文字もそのまま文字列として扱われる。ただし、エスケープシーケンスも有効になる。

```
'''
This is a rather long string containing
several lines of text just as you would do in C.
    Note that whitespace at the beginning of the
    line is significant.
'''
```

また、トリプルバッククオート (“‘) を用いれば、ご想像のとおり、インラインフォーマッティングも有効になる。

```
```
configure
make -j%d{core+1}
```
```

Konoha では、読みやすさを向上するため、ちょっとした改良をのトリプルクオートに加えてある。トリプルクオートの先頭が改行の場合は、その改行は無視される。したがって、上記のトリプルクオート文字列は、”c”から始まる。

7.2.5　RAW 文字列

RAW 文字列とは、エスケープシーケンスを無視した文字列リテラルである。Konoha は、C# に由来する記法を採用し、文字リテラルの前に@を付加すると RAW 文字列として用いられる。

```
@"C:\My Document\Programming\Konoha"
```

7.2.6 行リテラル

行リテラルは、ドキュメントや他言語のスクリプトを読みやすく書くための、 Konoha 独自の拡張リテラルである。 # で始まるトークンは、その行の終わりまでをそのまま文字列として解釈される。また、途中で、#が入っていてもそれは文字列の一部として解釈される。ただし、#直後の空白 (スペース) だけは、読みやすさのため挿入と解釈され、それは無視される。行リテラルは、バッククオートと同じく、「インライン・フォーマッティング」を解釈する。

```
# make -j3                  // "make -j3"と同じ
# make -j%d{core+1}
```

行リテラルは、文字列リテラルの連結のところで述べるとおり、ドキュメントや他言語のスクリプトを（読みやすく）書くための記法である。多くのスクリプティング言語では、#はコメントと解釈されるので、多用すると読みにくくなる。次のソースは、Konoha の文法的に正しい用例であるが、読みにくくなる。

```
>>> s = # make -j3
>>> s
"make -j3"
```

7.2.7 文字列リテラルの連結

文字列リテラルは、複数並べると、自動的に連結されてひとつの文字列となる。異なるタイプのクオートを混在させたときは、先頭のクオートタイプで連結される。

```
"hello" "," "world"       // "hello,world"と同じ
```

文字列リテラルの間に改行があると、連結するとき自動的に改行を含めて、連結される。これは、行リテラルの連結のときに、特に威力を発揮する。ちなみに改行を挿入されたくないときは、単に + 演算子で文字列を連結すればよい。

Konoha では、Groovy と同じく、メソッドや関数の第一パラメータに文字列リテラルを書くとき、関数コール演算子 () を省略することができる。行リテラルの連結と組み合わせて利用すると、すっきりとスクリプトが書くことが可能になる。

```
db.query
    # create table book (
    #     name char[20];
    #     title char[80];
    # );
```

下記のソースと完全に同じである。(ただし、どちらが読みやすいか言うまでもない。)

```
db.query('''
create table book (
name char[20];
title char[80];
);''')
```

7.3　文字列と演算子

文字列は、「文字」配列である。ただし、Konoha の String クラスは、不変 (Immutable) オブジェクトとして導入されているため、読み込み専用の「文字」配列となる。

7.3.1　長さとシーケンス

文字列は、0 個以上の文字が並んだシーケンスである。Konoha では、シーケンス s は、|s| 演算子を用いることで、その長さを得ることができる。文字列の場合、そこに含まれる文字の数である。

```
>>> s = "";
>>> t = "ABC";
>>> u = "ABC いろは";
>>> |s|
0
>>> |t|
3
>>> |u|
6
```

Konoha の文字列の長さは、多国語化された設計となっているため、文字のバイト数に関わらず、文字数が文字列の長さとなる。ASCII(1 バイト) 文字列を前提としたバイト数でないことに注意したい。もし文字列のバイト数をしりたい場合は、一旦、バイト配列 (byte[]) に変換すればよい。ただし、このとき UTF8 エンコーディングが用いられ、それにしたがい日本語は 3 バイト以上になっている。

```
>>> s = "ABC いろは";
>>> b = (byte[])s;          // バイト配列へ変換
>>> |b|
12
```

文字列は、文字の配列として扱うことができる。配列のインデックスは、0 から始まり、|s| − 1 である。

```
>>> s = "ABC いろは";
>>> s[0]
```

```
"A"
>>> s[|s|-1]                    // 最後の一文字
"は"
```

注意：Konoha の文字列は、変更不能 (immutable) である。通常の配列のように値を変更することができない。

文字列から部分文字列 (substring) を取り出すときは、シーケンススライス演算子を用いることができる。

```
>>> s = "ABC いろは";
>>> s[..2]
"ABC"
>>> s[3..]
"いろは"
>>> s[2..4]
"C いろ"
```

7.3.2　連結

文字列の連結は、Java と同様に、加算 (+) 演算子によって書くことができる。この加算は、もちろん結合則や交換則に基づくものではないが、操作が直感的であり、広く使われている。

```
>>> "Uzumaki" + " " + "Naruto"
"Uzumaki Naruto"
```

Konoha は、文字列の加算を最も優先されるオペレータとして定義している。つまり、文字列と文字列以外のオブジェクトを混在して換算した場合、全て文字列に変換されたのち連結される。

```
>>> "Naruto " + 8 + 1                // %s(8) + %s(1) と同じ
"Naruto 81"
```

もし数値演算を優先したい場合は、括弧 () で囲んで先に計算する。

```
>>> "Naruto " + (8 + 1)              // %s(8 + 1) と同じ
"Naruto 9"
```

7.3.3　連結、除去、分割

Konoha では、加算に加えて、減算、乗算、除算の演算子にもそれぞれ各演算子の意味に近い文字列操作として定義されている。これらも数学的な性質は継承しないが、文字列処理としてときに便利である。

文字列の減算 `s - t` は、文字列 s から指定された文字列 t を取り除く操作である。これは、`s.replace(t, "")` に相当する。

```
>>> "Uzumaki Naruto" - "a" - "u";
"Uzmki, Nrto"
```

文字列の乗算 `s * n` は、文字列の四則演算の中で唯一整数値を引数にとる。しかし、そこから得られる結果は極めて直感的に、文字列 s を n 個連結したものである。

```
>>> "Naruto" * 3;
"NarutoNarutoNaruto"
>>> "hello" * 0
""
```

文字列の除算 `s / t` は、文字列 s を文字列 t で分割したときの先頭の文字列である。文字列の余算 `s % t` は、分割された残りの文字列となる。これは、`s.split(t)` の簡易版として利用される。

```
>>> "Uzumaki Naruto" / " "
"Uzumaki"
>>> "Uzumaki Naruto" % " "              // mod のこと
"Naruto"
```

7.3.4　文字列の比較

文字列は、関係演算子によって比較することができる。そのとき、単純にアルファベットオーダー (UTF8 のコード順) で比較され、英大文字/英小文字は無視されない。

```
>>> "Uzumaki" == "Naruto"
false
>>> "Uzumaki" > "Naruto"
ture
```

7.4　文字列メソッド

Konoha は、Java に由来する String クラスのメソッドを（一部の例外はあるが）そのままサポートしている。また、より多くの文字列操作メソッドも拡張している。本節でとりあげるメソッドは、その一部である。しかし、本節のメソッドだけで、Konoha プロジェクトで Konoha 言語のエンジンのソースコードを生成しているドメイン言語 (DSL) を十分に作成することができる。

7.4.1 equals()

Java は、少々、プログラマには不便な特徴であったが、==演算子をオブジェクトの参照（リファレンス）の比較演算子として定義していた。そのため、文字列 s,t の値を比較するとき、s.equals(t) のような特別なメソッドを利用していた。

Konoha は、同じ文字列同士であれば、==, !=, <, <=, >, >=演算子を用いて比較することができる。これはより直感的なプログラミングを可能にしているが、Java とちょっとした互換性を高めるため、同様に s.equals(t) メソッドも導入している。

```
>>> s = "naruto";
>>> s.equals("naruto");      //  == 演算子と同じ
true
>>> s == "naruto"
ture
```

equals:IgnoreCase()

equals() は、英大文字/小文字を区別する。:IgnoreCase バリエーションを利用すれば、英大文字/英小文字を無視してマッチングを行うことができる。

```
>>> s = "Uzumaki"
>>> s.equals("uzumaki")
false
>>> s.equals:IgnoreCase("uzumaki")
true
```

これは、Java の equalsIgnoreCase() と同じである。Konoha では、独自のメソッド・バージョン管理を利用して、equals() のバリエーションを管理している。:IgnoreCase バリエーションは、equals() 以外にも、startsWtih(), endsWith(), indexOf(), lastIndexOf() などに用意されている。

7.4.2 startsWith(), endsWith()

Konoha は、正規表現によるパターンマッチングが利用できるが、多くの場合、文字列の先頭、もしくは末尾のみマッチングするだけで事足りることが多い。メソッド s.startsWith(t) は、文字列 s の先頭が文字列 t と一致するか判定するときに利用できる。逆に、メソッド s.endsWith(t) は、文字列 s の末尾が文字列 t と一致するか判定できる。

```
>>> s = "Uzumaki Naruto"
>>> s.startsWith("Uzumaki")
true
>>> s.startsWith("Naruto")
```

```
false
>>> s.endsWith("Uzumaki")
false
>>> s.endsWith("Naruto")
true
```

startsWith() と endsWith() は、英大文字/小文字を区別する。equals() と同様に:IgnoreCase バージョンを利用すれば、英大文字/英小文字を無視してマッチングを行うことができる。

```
>>> s = "Uzumaki Naruto"
>>> s.startsWith("uzumaki")
false
>>> s.startsWith:IgnoreCase("uzumaki")
true
```

7.4.3　検索: indexOf(), lastIndexOf()

関数 index() 系の文字列操作は、C 言語の標準ライブラリから存在する古典的な文字列ライブラリである。Konoha では、Java 同様に、文字でなく、部分文字列の検索ができる。

メソッド s.indexOf(t) は、文字列 s の先頭から部分文字列 t を探し、その位置を返すメソッドである。逆に、メソッド s.lastIndexOf(t) は、文字列 s の末尾から部分文字列 t を探し、その位置を返すメソッドである。位置は、文字列のインデックスと同様に 0 から始まる。ともに、部分文字列 t を発見できなかった場合は、-1 を返す。

```
>>> s = "Uzumaki Naruto"
>>> s.indexOf("Naruto")
>>> s.indexOf("Sakura")
-1
>>> s.indexOf("a")
4
>>> s.lastIndexOf("a")
9
```

最もよく使う利用法は、シーケンススライスと組み合わせて、文字列を分割するときである。

```
>>> s = "Name: Uzumaki Naruto"
>>> s[.. s.indexOf(": ")]
"Name"
>>> s[s.indexOf(": ")+1 ..].trim()
"Uzumaki Naruto"
```

7.4.4 split(), tokenize()

メソッド s.split(t) は、文字列 s を分割文字列 t によって単純に分割するメソッドである。分割された文字列は、String[] に格納される。(分割文字列 t) は、含まれない。)

```
>>> s = "Naruto, Sakura, Sasuke"
>>> s.split(", ")
["Naruto", "Sakura", "Sasuke"]
>>> s.split(",")
["Naruto", " Sakura", " Sasuke"]
```

特別な用例としては、分割文字列を省略すると、文字列は 1 文字ずつ split() される。

```
>>> s = "naruto"
>>> s.split()
["n", "a", "r", "u", "t", "o"]
```

もう少し意味のある単位でトークンを分割したい場合は、メソッド s.tokenize() も利用できる。これは、文字列 s を Konoha のレキシカル構造にしたがってトークンに分割するメソッドである。

```
>>> s = '''
print (1+1), 'hello,world';
'''
>>> s.tokenize()
["print", "(", "1", "+", "1", ")",
   ",", "'hello,world'", ";"]
```

7.4.5 置換: replace()

メソッド s.replace(t, u) は、文字列 s 中に出現する部分文字列 t を与えられた文字列 u によって置き換えるメソッドである。Konoha では、文字列は不変オブジェクトであり、破壊的操作はできないため、新しい String オブジェクトが生成され、メソッドの戻り値として返される。

```
>>> s = "Haruno Naruto"
>>> t = s.replace("Haruno", "Uzumaki")
>>> t
"Uzumaki Naruto"
>>> s                                  // 元の文字列は変わらない
"Haruno Naruto"
```

注意：replace() メソッドは、正規表現パターンによる置き換えも可能である。詳しくは、「第 7.6 節 正規表現」で述べる。

7.4.6　toUpper(), toLower()

メソッド s.toUpper() は、文字列 s の英文字を英大文字に変換し、メソッド s.toUpper() は、文字列 s の英文字を英小文字に変換するメソッドである。

```
>>> s = "Uzumaki Naruto"
>>> s.toUpper()
"UZUMAKI NARUTO"
>>> s.toLower()
"uzumaki naruto"
```

注意：Konoha は、メソッド名も含めて、できるだけ Java クラスライブラリとの互換性を意識しているが、本メソッドのみはどうしても toUpperCase(),toLowerCase() という名前にできなかった。String クラスの数少ない例外である。

7.4.7　メソッドのパッケージ拡張*

Konoha は、実行中であってもクラスに新しいメソッドを追加したり、既存の振る舞いを変更することができる。String クラスも例外ではなく、パッケージのインポートによって、メソッドが拡張されることがある。

次は、japanese パッケージをインポートしたことで、新しいメソッドが追加されたり、既存のメソッドが日本語処理に対応している例である。

```
>>> using japanese;                    // 日本語パッケージ
>>> ("なると").toKatakana()            // カタカナへの変換
"ナルト"
>>> ("ナルト　").trim()                // 全角空白も除去される
"ナルト"
```

7.5　フォーマッティング

フォーマッタは、オブジェクトを様々な書式で文字列化する Konoha 独自の機能である。理想的には、全てのオブジェクトはクラス階層にしたがってフォーマッタが定義され、全てのフォーマッティングはオブジェクト指向モデルによってよく再利用されるべきである。しかし、現実問題としては、再利用性を考慮に入れるよりも、とりあえずアドホックでもフォーマッティングができることが望ましいことが多い。

Konoha では、「テンプレート・フォーマッティング」と「インライン・フォーマッティング」の 2 種類のフォーマッティングを導入している。どちらの場合も、定義済みのフォー

マッタは、テンプレート書式の一部として用いることができる。

7.5.1 テンプレート・フォーマッティング

テンプレート・フォーマッティングは、文字列としてテンプレートを作成し、そのテンプレートをビルトイン関数 format() を用いることで、パラメータと組み合わせてフォーマットされた文字列を生成する仕組みである。

```
>>> format("We are at [%s{0}:%d{1}]:", __file__, __line__)
"We are at [(shell):1]"
```

テンプレートは、テンプレート書式にしたがえば、あとは通常の文字列として扱うことができる。次は、変数に格納されたテンプレートをフォーマット処理したときである。

```
>>> fmt = "We are at [%s{0}:%d{1}]:";
>>> s = format(fmt, __file__, __line__)
>>> s
"We are at [(shell):4]"
```

7.5.2 テンプレート書式

テンプレート書式は、"%x{n}"スタイルで書く。これは、「フォーマッタ %x に n 番目のパラメータの値を適用する」と意味になる。ここでインデックスは、0 から始まる。わざわざ、パラメータのインデックスを指定できる理由は、英語と日本語のような語順が違うテンプレートでも、パラメータの順番を気にしなくて適用するためである。

```
>>> fmt = "%s{1}内で %s{0}が見つかりません."
>>> fmt = "%{0}: Not found in %s{1}"
```

7.5.3 フォーマッティング関数

テンプレートは、ビルトイン関数 format() を用いて、フォーマッティングされる。フォーマットされた文字列は、関数の戻り値として返される。

```
>>> format(fmt, "Class", "file.k")
"Class: Not found in file.k"
```

%(...) は、format(...) のショートカット名である。下の例は、上のフォーマッティングと全く同じである。

```
>>> %(fmt, "Class", "file.k")
"Class: Not found in file.k"
```

フォーマッティング時のパラメータは、Konoha において数少ないコンパイル時に型検査されないパートである。フォーマッタは、与えられたパラメータのクラスに対して、動的にバインドされる。もし指定されたフォーマッタが定義されてなければ、自動的に `%empty` が適用され、そこには文字列が表示されない。

もし厳密な型検査を求める場合は、関数等でラッピングして用いるとよい。

7.5.4　インライン・フォーマッティング

Konoha では、バッククオートによる文字リテラルと、#で始まるライン文字列リテラルに対してのみ、インデックスの代わりに変数やその他の式を直接埋め込むことができるインライン・フォーマッティングを認めている。

```
c.query
   # select name, salary from PERSON_TBL
   # where age > %d{age} and age < %d{age2}
```

7.6　正規表現

正規表現 (regular expression) は、文字列のパターンを記述する小さな言語である。与えられた正規表現パターンは、有限オートマトンにコンパイルされ、入力文字列に対して高速にマッチングされる。

スクリプティング言語では、Perl を代表例として文字列処理の中心的機能として正規表現を言語に統合している。Konoha では、独自の「正規表現」という概念は存在しないが、GNU regex, Oniguruma, PCRE(Perl Compatible Regular Expression) など、既存の正規表現ライブラリを選択して利用することができる。

7.6.1　正規表現リテラル

正規表現パターンは、Regex クラスのインスタンスとして表現される。Konoha では、Perl, Ruby, JavaScript などと同様に、専用の正規表現リテラルが用意されている。正規表現リテラルは、/記号で囲まれた文字列であり、そこで表現された文字列は正規表現パターンと解釈される。

```
pattern = /s$/
```

正規表現リテラルは、Regex コンストラクタで生成されるオブジェクトと全く同じである。また、Regex コンストラクタは、@Const 属性であるため、コンパイル時にオブジェクトが生成されるため、性能の面でも同じである。しかし、正規表現リテラルの方を使う

ことが圧倒的に多い。何よりもシンプルに記述できるからである。特別なオプションを指定するときだけ、現在のところ、コンストラクタを使う必要がある。

```
pattern = new Regex("s$", "i");          // 英大文字/小文字を無視..
```

7.6.2 正規表現エンジン

Konoha は、標準的な正規表現エンジンとして、POSIX regex ライブラリを利用することが多い。現在の正規表現エンジンは、`$konoha.regex` で確認することができる。

```
>>> $konoha.regex
"GNU regex library"
```

正規表現エンジンは、正規表現パッケージをインポートすることで切り替えることができる。主な正規表現パッケージは、以下のとおりである。

- oniguruma — kosako 氏が開発し、Ruby で正式採用されている高性能な正規表現ライブラリ
- pcre — Perl 互換の正規表現ライブラリ

切り換えは、パッケージのインポートと同期している。つまり、正規表現パッケージをインポートすれば、正規表現ドライバが入れ替えられ、それ以降の正規表現は新しいエンジンに切り替わる。

```
>>> using oniguruma;
>>> $konoha.regex
"oniguruma"
>>> pattern = /^*.:*.$/            // 以後は、鬼車を利用
```

注意：正規表現ライブラリをインポートする以前の正規表現オブジェクトは更新が反映されない。

また、正規表現ライブラリごとの異なる機能を明示的に使い分けたいときは、正規表現リテラルの代わりに、`'regex:'`、`'pcre:'`、`'onig:'` のように、それぞれのライブラリを識別する意味タグを用いることで、正規表現を用いることができる。

```
using oniguruma;
s =~ 'onig:^*.:*.$';          // 鬼車でマッチング
s =~ 'regex:^*.:*.$';           // regex でマッチング
```

7.6.3　正規表現と演算子

Konoha は、Perl 由来の正規表現マッチング演算子をサポートしている。しかし、JavaScript もサポートしていないようだから、この演算子は中止しようかと悩んでいる。

```
>>> s = "name: naruto";
>>> s =~ /^*.: *.$/
true
```

7.6.4　正規表現とメソッド

String クラスは、いくつかのメソッドを正規表現パターンに対応させている。

`s.serach(p)` は、文字列 `s` の先頭から、正規表現パターン `p` でマッチング検索を行い、最初にマッチした文字列インデックスを返す。`indexOf()` の正規表現版である。

```
>>> text.search(/ha/);
```

`s.match(p)` は、文字列 `s` に対して、正規表現パターン `p` でマッチングを行い、そのマッチング結果の文字列を返す。複数個マッチした場合は、全て返す。

```
>>> s = "1 + 2 = 3";
>>> s.match(/\d/);
["1", "2", "3"]
```

また、正規表現パターンにおいてグルーピングを用いれば、マッチングした文字列に続いて、グルーピングされた部分文字列も得ることできる。

```
>>> url = "http://www.website.org/konoha";
>>> url.match(/(\w+):\/\/([\w.]+)\/(\S*)/);
["http://www.website.org/konoha",
   "http", "www.website.org", "konoha"]
```

`s.replace(p, fmt)` は、文字列 `s` に対して、正規表現パターン `p` でマッチングを行い、その結果をフォーマッティング `fmt` によって置き換える演算子である。

```
>>> s = "1 + 2 = 3";
>>> s.replace(/(\d)/, "%s{1}.0");
"1.0 + 2.0 = 3.0"
```

7.6.5 正規表現とマッピング**

最近の Perl 互換の正規表現では、名前付きのグルーピングを提供している。文字列を正規表現からパースして、直接オブジェクトへマッピングする機能を計画している。

```
>>> Class Url {
...   String site;
...   String path;
... }
>>> url = "http://www.website.org/konoha";
>>> pattern = /(\w+):\/\/(?site:[\w.]+)\/(?path:\S*)/;
>>> u = (Url with pattern)url;
>>> u
{site: "www.website.org", path: "/kohoha"}
```

第 8 章

配列とリスト

配列 (array) は、複数個のオブジェクトをコレクションとして扱う最も基本的なデータ構造である。Konoha では、配列は Array クラスとして実現されている。クラス名は、単に Array であるが、Java の ArrayList と同様に、可変長配列（Growing Array）であり、リストやスタックなどのデータ構造を扱う場合にも活用できる。

8.1 配列と型

Konoha の配列型は、2 種類のスタイルがある。ひとつは、Array 型で、任意の型の要素をもつ配列である。もうひとつは、C や Java でおなじみの T[] スタイルの型で、T 型の要素をもつ配列である。両者は、一見全くことなるようにみえるが、ともに Array のいわゆる総称型の別名で、Array<Any>と Array<T>である。基本的な性質は大きく変わらない。

本書では、Konoha スタイルで 2 種類の配列型を併有して書くが、実際のソースコードでは古典的な配列スタイル、総称型スタイルのどれを利用しても構わない。

Konoha	**配列**	**総称型**
Array	Any[]	Array<Any>
int[]	int[]	Array<int>
String[]	String[]	Array<String>
T[]	T[]	Array<T>

Unbox された配列: int[] float[]

プログラマの立場からみれば、int[] と Array<int>は区別なく利用できる。ただし、これはあくまでも見かけ上の話である。Konoha 内部では、より正確に言えば、整数 (int と浮動小数点数 (float) は unbox された形で整数要素を格納する専用のクラス実装に切り替えている。ちなみに boolean[] は、現在のところ unbox されていない。

8.1.1　Array と Object[] の違い

Object は、全てのオブジェクトの上位クラスである。そのため、Object[] は、Array と同様に任意の型のオブジェクトを要素に持つことができる。ただし、「第 ?? 章　ダイナミック言語スタイル」で述べるとおり、Object と Any は、キャストの取扱いが異なるため、両者には注意すべき違いが生じる。

まず、Object[] の方であるが、基本的に Java の場合と同じ振る舞いをする。ただし、配列の要素型は、コンパイルした時点では Object 型なので、次の例のとおり、実行時に扱う要素が明らかに String 型であっても、明示的なダウンキャストが必要となる。

```
>>> String[] o = ["naruto", "sakura"];
>>> typeof(o[0])                    // 要素型は Object
Object
>>> String s = (String)o[0];        // ダウンキャストが必要
>>> s
"naruto"
```

これに対し、Any 型は、実行時に動的な型検証が入るため、静的なキャストは不要になる。こちらの方が便利な反面、気づかないうちにダウンキャストや型変換を行っているため、実行効率が落ちることがある。

```
>>> Any[] a = ["naruto", "sakura"];
>>> typeof(a[0])                    // 要素型は動的
Any
>>> String s = a[0];                // キャストは不要
>>> s
"naruto"
```

8.1.2　null 要素*

Konoha の配列は、null 値を配列要素に持つことができない。ただし、Any 型はいかなる場合でも null 値を求めるため、Array 型のみ null を要素にもつことができる。

```
>>> String[] s = ["naruto"];
>>> s[0] = null;
 ** Null!!

>>> Array a = ["naruto"];
>>> a[0] = null;
>>> a[0]
null
```

8.1.3 バイト配列 byte[]

Konoha の整数型は、64 ビット長の Int 型に単一化されているため、バイト (8 ビット整数) を表す型はない。多少、メモリが無駄かも知れないが、int でバイトを表すことになる。しかし、バイト列（バイナリデータ）となると、int[] を使うのは、多少の無駄では済まない。そこで、バイト列を扱う専用のクラスとして、Bytes クラスが用意されている。byte[] は、ごたぶんに漏れず、Bytes の別名である。

```
byte[] buf = new byte[4096];
```

バイト配列 int[] は、配列とほぼ同じ性質を持つ。本章で取り上げるメソッドや演算子をほぼそのまま利用することができる。バイト列を扱うときの特有な性質のみ、「第 8.6 節 バイト配列」で取り上げる。

8.2 配列の生成

配列は、配列リテラルでデータとして要素を列挙する方法と、プログラム中から new 演算子を用いて生成する方法の 2 種類で生成することができる。

8.2.1 配列リテラル

配列リテラルは、Python や JavaScript などで採用されているデータ値を列挙し、四角括弧 [, ,] で囲む記法を採用している。C/C++ や Java の配列リテラルとは異なるが、こちらの方が JSON (JavaScript Object Notation) として Web 上のデータ交換にも利用されている。

```
[1, 2, 3]                       // 1, 2, 3 の配列
```

配列リテラルの要素は、Konoha がサポートするリテラルであれば含めることができる。ただし、変数や式を含めることは、セキュリティ上の観点から制限されている。

```
[
        "naruto",
        [1, 2, 3, 4],
]
```

8.2.2　配列リテラルと型推論

Konoha における重要な特徴は、リストの要素から配列リテラルの型が推論される点である。ただし、配列リテラルの型推論リールは、いたって単純である。まず、最初の要素の型 T を配列型 $T[]$ の候補として、残りの要素がすべて T か、そのサブタイプであるとき、`T[]` と推論する。そうでないときは、Array クラスとなる。

```
[]                              // 空の Array
[1,2,3]                         // Int[] の配列
["Naruto", "Sakura"]            // String[] の配列
["Naruto", 9]                   // Array
```

8.2.3　Array コンストラクタ

Array クラスは、空の Array オブジェクトを生成するコンストラクタをサポートしている。空の配列とは、大きさが 0 の配列である。空の配列に意味があるのかと思われるかもしれないが、Konoha の Array は、可変長配列であり、後述する `add()` メソッドによって要素を追加することができる。

```
>>> a = new Array();            // 空の Array を生成
>>> |a|                         // サイズは？
0
>>> a.add(1)                    // 追加
>>> a
[1]
```

可変長配列は、許容量を超えると、再度メモリーアロケーションを行いながら、配列の大きさを成長させている。ある程度、配列が成長する大きさがわかっている許容量の初期値を設定することもできる。

```
>>> a = new Array(100);         // 初期許容量
>>> a
[]
```

8.2.4　T[] コンストラクタ

T[] スタイルの配列は、Java と同様に、配列の大きさをあらかじめ与えて、配列を生成することができる。

```
>>> ss = new String[4];         // 要素数 4 個の配列 String[]
>>> ss
```

```
["", "", "", ""]
>>> |ss|
4
```

このとき、生成された配列の要素は、要素型のデフォルト値 (`default(C)`) で満たされる。(Java と異なり、null ではない。) デフォルト値が不要な場合は、要素数を 0 にして空の配列を生成することもできる。

```
>>> ss = new String[0];          // 空の String[]
>>> ss
[]
```

8.2.5 配列リテラルとコンストラクタ

配列は、整数や文字列と異なり、変更可能なオブジェクトである。そのため、配列リテラルをコンパイル時に定数プールで共有すると、変更されたときリテラルと要素の内容の間で一貫性が保証できなくなる。Konoha では、配列リテラルをコンパイル時にオブジェクト化せず、new 演算子によるコンストラクタ式に変換してコンパイルしている。

```
>>> int[] genArray123() {
...     return [1, 2, 3];
... }
>>> a = getArray123()            // 配列を生成
>>> a.reset()                    // 変更しても
>>> getArray123()                // もうひとつ生成
[1, 2, 3]
```

もし配列リテラルで定義した配列を共有したい場合は、先に定数（もしくはスクリプト変数）に格納しておく方法がある。（こちらの方が毎回、配列を生成しない分、高速になる。）ただし、共有されている配列オブジェクトを変更した場合、その変更も共有されるため、注意が必要となる。

```
>>> LIST123 = [1, 2, 3];         // 定数（ここで生成）
>>> int[] genArray123() {
...     return LIST123;
... }
>>> a = getArray123()
>>> a.reset()                    // 変更すると
>>> getArray123()                // 変更が共有される
[]
```

注意： Array クラスは、`toImmutable()` メソッドによって、変更不可能な属性を与えることができる。ただし、この機能は、配列の型システムの見直しによって変更される可能性がある。

8.3　配列と演算子

Konoha は、C/C++ 由来のインデックスによる配列へのアクセスから Python 風のシーケンス演算子、スライシング演算子をサポートしている。

8.3.1　シーケンス演算子

配列は、0 個以上の要素が並んだ典型的なシーケンスである。Konoha では、シーケンス s は、`|s|` 演算子を用いることで、その長さを得ることができる。配列の場合は、要素の数である。

```
>>> a = [0, 1, 2, 3]
>>> |a|
4
```

配列 `a` のインデックスは、C/C++, Java 言語と同様に 0 から始まり、`|a| - 1` である。その範囲を超えると、`OutOfIndex!!`例外が通知される。

```
>>> a = [0, 1, 2, 3]
>>> a[0]
0
>>> a[|a|-1]
3
>>> a[4]
 ** OutOfIndex!!
```

配列は、基本的に変更可能である。正しいインデックスを指定すれば配列の値を変更できる。

```
>>> a = [0, 1, 2, 3]
>>> a[1] = 10                    // a[1] を変更
>>> a
[0, 10, 2, 3]
```

注意：初期の Konoha では、python 風に最後尾からインデックスを数える $-n$ 表記が利用できた。このインデックスは便利であったが、C/C++, Java などの既存の配列処理との互換性や最適化処理の困難さを理由に廃止となった。

8.3.2　スライシング

配列 `a` から部分配列を取り出すときは、スライス演算子 `a[m..n]` を用いることができる。m と n は、省略可能で、省略した場合は、それぞれ 0 と最後の添字 (`|a|-1`) と解釈される。

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> a[2..]                          // a[2] から
[2, 3, 4, 5, 6, 7, 8]
>>> a[2..6]                         // a[2] から a[6] まで
[2, 3, 4, 5, 6]
>>> a[..<6]                         // a[2] から a[5] まで
[2, 3, 4, 5]
>>> a[2..+6]                        // a[2] から 6 個
[2, 3, 4, 5, 6, 7]
>>> a[..6]                          // a[6] まで
[0, 1, 2, 3, 4, 5, 6]
>>> a[..<6]                         // a[6] まで（含まない）
[0, 1, 2, 3, 4, 5]
>>> a[..+6]                         // 最初の 6 個
[0, 1, 2, 3, 4, 5, 6]
```

8.3.3 要素の判定 in?

配列では、in?演算子を用いることで、配列中に与えられた要素が含まれているかどうか判定することができる。

```
>>> a = [1, 2, 4]
>>> 2 in? a                         // 含まれるか？
true
>>> 0 in? a
false
```

注意：Array クラスの in?は、単純な線形探索であるため、配列が大きくなるにつれ効率は悪くなる。実行性能が気になる場所では、HashSet を代わりに用いた方がよい。

8.3.4 集合演算**（調査中）

配列は、シーケンスなので文字列と同様に、+ 演算子によって連結したり、−演算子によって要素の差分をとることができる。

```
>>> a = [1, 2, 3, 5]
>>> b = [2, 4]
>>> a + b                           // 連結
[1, 3, 5, 2, 4]
>>> a - b                           // 差分
[1, 3, 5]
```

Konoha では、配列に対し、集合として積 (intersection) や和 (union) の演算も定義している。ただし、これらの演算子では、配列を集合として扱うため、順序を無視し、また重複する要素も自動的に除去される。

```
>>> a = [1, 2, 3, 5]
>>> b = [2, 4]
```

```
>>> a & b                          // 集合積
[2]
>>> a | b                          // 集合和
[1, 2, 3, 4, 5]
```

8.4　多次元配列

Konoha の配列は、基本的に 1 次元配列である。n 次元配列を使うというのは、単純に、C 言語同様に、適切に次元を計算してからインデックス操作を行えばよい。例えば、それぞれの次元の大きさが X, Y, Z の 3 次元配列を使いたい場合は、次のようにインデックス (index) を計算できる。

```
int[] a = new int[X * Y * Z];
index = x + X * y + (X * Y) * z;
a[index]
```

当初、多次元配列に関しては、これ以上の関心がなかった。それは、Konoha 言語の開発を含め、過去のプログラミング経験から多次元配列がなくて困ったことはほとんどなかったからである。しかし、これも「クラスルーム」から来た要求であるが、プログラミング練習課題には、オセロやら五目並べやらと、2 次元配列を使うものが多い。学生たちから「2 次元配列は使えないのですか？」と質問がくる度に、シンタックスシュガーでさっさと対応すべきではないか？、ということになった。

そういうわけで、Konoha の多次元配列は、型としては 1 次元配列のときと同じ Array クラスである。ただし、コンストラクタで 2 次元もしくは 3 次元の大きさを設定すると、その次元情報はメタデータとして記録される。このメタ情報を用いてインデックスをを再計算している。確かに、上記のプログラムよりは、すっきりする。

```
int[] a = new int[X, Y, Z];
a[x, y, z]
```

次は、2 次元配列の利用例である。クラスルームで使う程度であれば、ふつうの 2 次元配列として利用することができる。

```
>>> a = new Int[8, 8];            // 2 次元配列
>>> a[1, 7] = 1
>>> a[1, 7]
1
>>> a[8, 1]                       // 次元単位でチェック
 ** OutOfIndex!!
```

多次元配列は、1 次元配列なので、1 次元配列として操作することもできる。多次元配列としての型安全性は備えていない。

```
>>> a = new Int[8, 8];          // 2 次元配列
>>> typeof(a)
Int[]
>>> |a|                         // 大きさは、1 次元
64
>>> a.add(100)                  // 追加もできる
>>> |a|
65
>>> a[64]                       // 1 次元配列の操作も可
100
```

8.5 配列とメソッド

Konoha は、Java の Collection フレームワークの標準的なメソッドをベースにして Array 関係のメソッドを定義している。

8.5.1 リスト: add(), insert(), remove()

Konoha の配列は、全て可変長であり、リストのように要素数のわからないデータを実行時に必要に応じて伸縮させながら利用できる。

メソッド `add(v)` は、最もよく使われるメソッドである。配列の最後尾に新しい要素 `v` を追加する。演算子 <<と同じ働きをする。

```
>>> a = [0, 1, 2]
>>> a.add(100)                  // 追加
>>> a
[0, 1, 2, 100]
>>> a.add(30000)                   // 追加
>>> a
[0, 1, 2, 100, 300000]
```

メソッド `insert(n, v)` は、`n` 番目に要素 `v` を挿入するときに利用し、メソッド `remove(n)` は、`n` 番目からを取り除くとき利用する。どちらの場合は、配列の大きさは、それぞれ増減する。

```
>>> a = [0, 1, 2]
>>> a.insert(1, 100)            // 挿入
>>> a
[0, 100, 1, 2]
>>> a.remove(2)                    // 削除
>>> a
[0, 100, 2]
```

Konoha の配列は、リストのように利用できるが、双方向リンクリスト構造ではないため、`insert()`, `remove()` ともに配列要素をコピーするだけのコストがかかる。`add()` は、十分に高速に要素を追加することができる。

8.5.2　キューとスタック: first(), pop()

Konoha の配列は、キュー (FIFO) やスタック (FILO) として利用することができる。

次は、配列のキューとしての操作例ある。キューに追加するときは、配列と同じく、`add()` を用いる。`first()` は最初の要素を取り出すメソッドである。キューが空であれば、ブロックされるか、もしくは Thread!!例外が投げられる。(先に、`|queue|` でキューの大きさを検査する必要がある。)

```
>>> queue = []
>>> queue.add(1)                // 最後尾に追加
>>> queue << 2 << 3             // 更に追加
>>> queue
[1, 2, 3]
>>> queue.first()               // キューから取り出す
1
>>> queue
[2, 3]
```

次は、スタックとしての操作例である。スタックに要素を追加するのは同じく `add()` を用いる。`pop()` は最後に追加した要素を取り出すメソッドである。スタックが空の場合は、`pop()` は、StackUnderFlow!!例外を投げる。(注意：add() と pop() が非対称なコードはバグなので、スタックサイズを検査する必要はない。)

```
>>> stack = []
>>> stack.add(1)                // 最後尾に追加
>>> stack << 2 << 3             // 更に追加
>>> stack
[1, 2, 3]
>>> stack.pop()                 // スタックから取り出す
3
>>> stack
[1, 2]
```

注意：配列は、スタックとして利用するとき、配列の大きさに関わらず、十分に高速である。しかし、キュー操作の場合は、`first()` のとき配列要素のコピーが発生するため、配列の大きさに応じてコストがかかる。

8.5.3　整列：sort()

Konoha は、高速なソートアルゴリズム (qsort 相当) によって配列の要素の並びかえることができる。大小の比較は、オブジェクト間の標準的な比較方法、つまり < 演算子によって行われる。

```
>>> a = [3, 2, 1, 5, 4]
>>> a.sort()                    // ソート
```

```
>>> a
[1, 2, 3, 4, 5]
```

sort() は、いわゆる昇順ソート (assending sort) である。降順ソート (decending sort) のためのメソッドは用意されていない。ただし、sort() と reverse() を組み合わせると、降順になる。

```
>>> a
[1, 2, 3, 4, 5]
>>> a.reverse()
[5, 4, 3, 2, 1]
```

比較方法の入れ替え**

sort() メソッドは、標準 C ライブラリのソートアルゴリズムと同様に、配列の要素の比較方法を切り替えることができる。Konoha では、関数ポインタの代わりに、クロージャを用いる。

次は、文字列の比較をケースインセンシブなバージョンに変更した例である。スクリプト関数 cmpr は、qsort ライブラリの比較関数と同様に、比較した結果が小さければ負の数 (-1), 等しければ 0、大きければ正の数 (1) を返す。それを delegate() でクロージャ化し、sort() メソッドに渡している。

```
>>> int cmpr(String a, String b) {
...     return a.compareTo:IgnoreCase(b);
... }
>>> s = ["SASUKE", "sakura", "naruto"]
>>> s.sort(delegate(cmpr));    // クロージャ化
>>> s
["naruto", "sakura", "SASUKE"]
```

sort() メソッドが受け付けるクロージャの型は、Array<T>に対して、int(T, T) である。クロージャに関しては、「第 15 章 クロージャ」を参考にして欲しい。

8.5.4 ランダム化：shuffle()

ゲーム開発、負荷分散、セキュリティ対策など、様々な場面で配列の要素をランダム化する機会は多い。shuffle() は、配列の要素をランダムに並べるメソッドである。(乱数は、Int.random() 同様に Mersenne Twister を利用している。)

```
>>> a = [1, 2, 3, 4, 5]
>>> a.shuffle()
[3, 2, 1, 5, 4]
```

8.5.5　サーチ: indexOf(), lastIndexOf(), binarySearch()

サーチは、配列中から指定された要素を探すことである。配列は、indexOf() は、先頭から探して最初に見つかったインデックスを返す。lastIndexOf() は、逆に最後尾から検索して最初に見つかった箇所のインデックスを返す。ともに発見できなかった場合は、-1 を返す。

```
>>> a = [1, 2, 3, 2, 1]
>>> a.indexOf(2)                    // 先頭から検索
1
>>> a.lastIndexOf(2)                // 後尾から検索
3
>>> a.indexOf(0)                    // 見つからない
-1
```

indexOf() と lastIndexOf() は、どのような配列に対しても検索が可能である。しかし、単純に順番に検索するため、検索時間は配列の大きさに比例し、効率はあまりよいといえない。もし配列が sort() で整列された状態であるときは、binarySearch() を用いると、バイナリーサーチアルゴリズムにより効率よく検索することができる。

```
>>> a = [1, 3, 2, 4, 0]
>>> a.sort()
>>> a
[0, 1, 2, 3, 4]
>>> a.binarySearch(3)               // バイナリサーチ
3
>>> a.binarySearch(5)               // 見つからない
-1
```

8.5.6　文字列への連結: format()*

format(fmt, delim) メソッドは、配列の各要素に対し、与えられたフォーマッタ fmt を適用し、更に delim で連結してフォーマットする機能である。

```
>>> a = ["naruto", 9]
>>> a.format("%s", ",")
"naruto,9"

>>> a.format("%k", EOL)
"naruto"
9
>>>
```

8.6 バイト配列: byte [ ]

バイト配列は、バイナリデータを扱うための専用の配列であり、ヒープ上のメモリを確保し、それをダイレクトに扱うことができる。バイト配列の大きさは、バイト数である。

```
>>> buf = new byte[4096];
>>> typeof(buf)
Bytes
>>> |buf|
4096
```

各要素のとりうる値は、0 〜 255 (0xff) であり、この値域に当てはまらない整数は丸め込まれる。(型エラーとならない。)

```
>>> buf[0]
0
>>> buf[0] = -1
255
```

`sort()` や `shuffle()` など、バイナリ操作に関係ないメソッドをサポートしていないが、それ以外は配列と同じように操作することができる。以下は、バイト配列に特徴的な機能のみ紹介する。

8.6.1 バイトコピー: Bytes.memcpy()*

クラス関数 `Bytes.memcpy(dst, s1, src, s2, n)` は、memcpy() のバイト配列版である。バイト配列 dst の先頭から s1 バイト目へ バイト配列 src の s2 バイト目から、n バイトコピーする。名前が示すとおり、C 標準ライブラリが提供する memcpy を利用しているため、スクリプト上で配列操作するより格段に高速である。

```
>>> Bytes.memcpy(dst, 0, src, 0, 256)
```

8.6.2 ストリームへの変換*

バイト配列は、InputStream, OutputStream にへラップして操作することができる。(これらは、Java の ByteArrayInputStream, ByteArrayOutputStream に相当する。)

`ba.wrapInputStream(offset, length)` は、バイト配列 ba の offset バイト目から length 分に相当する部分を読む InputStream をつくる。offset と length はそれぞれ省略すると、最初から最後まで、と解釈される。

```
>>> ba = Bytes.readFile('coredump');
```

```
>>> bin = ba.wrapInputStream();      // (InputStream)ba
>>> data = bin.readData();
```

ba.wrapOutputStream(offset) は、バイト配列 ba の offset バイト目から追加で書き込む OutputStream をつくる。offset は省略すると、「最初から」と解釈される。バイト配列も可変長であるため、メモリが許す限り追加することができる。（上限制限は必要か？）特に、OutputStream ストリームやフォーマッタの一部として利用できるため、便利である。

```
>>> ba = new byte[4096];             // 初期の大きさ
>>> bout = ba.wrapOutputStream(0);   // (OutputStream)ba
>>> bout.println("hello, world");
>>> bout.close()

>>> %dump(ba)                        // ba
```

InputStream や OutputStream の詳しい情報は、「第 **??** 章 ストリーム」で紹介する。こちらもあわせて、参照してほしい。

第 9 章

イテレータ

Konoha の Iterator クラスとそれを用いたプログラミングに関して執筆する予定である。

第 10 章

ディクショナリ

Konoha の DictMap クラスとそれを用いたプログラミングに関して執筆する予定である。

第 11 章

関数

関数 (function) は、プログラムの一部をモジュール化するための代表的な手段である。名前が示すとおり、数学の関数 $y = f(x)$ をモデルとして、引数 x に対して結果 y が得られるという形でプログラムのモジュール化とパラメータ化を実現する。

Konoha における関数は、単に Script クラスのメソッドである。しかし、関数のアイディアはオブジェクト指向プログラミングより古く、クラス概念を導入しなくても利用できる伝統的なモジュールであるため、オブジェクトを意識しないで呼び出せるメソッドを「関数」と呼んでいる。関数をファーストクラスオブジェクトとして扱う手法を「クロージャ」と呼んでいる。

11.1 関数を定義する

プログラマは、既存の関数を利用するだけなく、新しい関数を自由に定義することができる。Konoha では、関数定義のための特別なステートメントは用意されておらず、C/C++, Java と同様に、型宣言にパラメータとモジュールを追加する形式で行う。

$$T\ f\ (T_1\ p_1, T_2\ p_2, ...) \quad stmt$$

T 型の評価値を返す関数は、Konoha の名前規則にしたがい、英小文字で始まる名前 f を持ち、0 個以上のパラメータ (変数) $p_1, p_2, ...$ をとるように定義する。モジュール本体は、続けて $stmt$ の部分に書き、ここでは一般的にステートメントブロックを用いてプログラムを構成する。

次は、$f(x) = x + 1$ という式を関数でモジュール化した例である。関数の評価結果は、return 文で書く。

```
int f(int x) {
   return x + 1;
}
```

一旦、関数を定義すると、関数をコール演算子で呼び出すことで、何回でもモジュール化されたプログラムを再利用することができる。

```
>>> int f(int x) {
...     return x + 1;
... }
>>> y = f(1)
>>> y
2
>>> f(2)
3
```

プログラミング言語における関数は、引数や戻り値に対して「型」が必要な点が特徴的であるといえる。Konoha は、C/C++, Java などの静的な型付け言語と同様に、関数定義のときに、これらの型を明示的に宣言することになっている。原則、パラメータや戻り値の型推論は行わない。

これは、変数宣言における型推論に比べ、極めて保守的なスタンスをとっているように思われるかも知れない。関数（及びメソッド）は、異なる開発主体が開発するソフトウェア境界のインターフェースとなり、今日のソフトウェア開発ではインターフェースを明示的に仕様化することが望ましいとされている。更に、型検査などで機械的に検査されることがもっと望ましいとされている。そういうわけで、Konoha の関数定義では、しかし、プログラミングのしやすさよりも、ソフトウェア工学的な実践面を重視した言語設計を採用している。

11.1.1　function 文

Konoha は、動的な型付けを可能とする Any 型を持っているため、型宣言なしの関数定義もそれらの型を Any 型として扱うことで推論なしに定義可能である。したがって、次のような JavaScript 風の関数定義もそのままかくことができる。

```
function f(x) {
   return x + 1;
}
```

注意： Konoha では、function 文は、無名関数を定義するときに用いることを前提に導入されている。したがって、戻り値の型が決定的であれば、型推論される。詳しくは、「第 15 章 クロージャ」を参考にして欲しい。

11.2　return 文

リターン

11.2.1 戻り値*

モジュールの処理が終れば、その処理内容を評価結果として返すことができる。これを戻り値 (return value)、もしくは返り値と呼ぶ。T は、戻り値の型であるが、void 型を指定すれば特定の値を返さない関数も作成することができる。

11.3 パラメータ

関数の引数は、モジュール化されたプログラムの振る舞いをかえることに役立っている。そのため、これをモジュールのパラメータ化と呼び、関数の引数のことを「パラメータ (parameter)」と呼ぶこともある。

11.3.1 参照渡し

関数の引数は、モジュール間のデータをやり取りするインターフェースになる。プログラミング言語の教科書を読めば、値渡し (call by value)、参照渡し (call by reference)、名前渡し (call by name) の 3 種類存在することになっているが、

Konoha は、結論から言えば、全て参照渡し (call by reference) である。理由は、”Everything is an Object”の世界であり、関数コールの度にオブジェクトをコピーするのはコストが大きすぎるためである。オブジェクト自体の代わりに、オブジェクトの参照、つまりオブジェクトのポインタを渡すことで、効率よい関数コールを実現している。

```
>>> void f(String t) { print %p(t); }
>>> s="a"
>>> print %p(s)
[(shell):3] "0x6cc00"              // s と
>>> f(s)
[(shell):4] "0x6cc00"              // t は同じポインタ
>>>
```

参照渡しで注意すべき点は、関数内部でオブジェクトの値を変更すると、関数を呼び出した外部でもその変更が継続することである。(関数の外と内で、同じオブジェクトを参照しているから当たり前といえば、当たり前である。)

```
>>> void g(Array a) {
...   a << 1;
... }
>>> Array list = [];
>>> list
[]
>>> g(list)
>>> list                           // 関数内のリスト操作が残る
[1]
```

注意：Int, Float, String オブジェクトは、不変オブジェクトなので、関数内で変更しても、そのとき実際はオブジェクトのコピーが行われている。そのため、関数内部の変更によって外部のオブジェクトまで変更されることはない。

11.3.2　可変長パラメータ**

Konoha は、可変長変数を採用することができる。

11.3.3　パラメータ初期値*

11.4　関数内同一スコープ

Konoha は、C/C++ や Java とは大きく異なり、ブロックレベルの変数スコープをサポートしていない。関数内で宣言された全ての変数は、どこで宣言されても関数内で利用できる。ただし、変数宣言より先に変数を使うことはできない。

例えば、次のコードにおいて、変数 k, j は、スコープ内として扱われる。

```
void test(int n) {
   int i = 0;
   if(n > 10) {
      int j = 0;
      for(int k = 0; k < n; k++) {
         print k;
      }
      print k;                          // スコープ外ではない
   }
   print j;                             // スコープ外ではない
}
```

次のような例外処理のコードを書くときも都合がよい。

```
try {
   InputStream in = new InputStream("file.txt");
   for(String line from in) {
      print line;
   }
}
catch(IO!! e) {
   print format("Error at %d", in.line);
}
finally {
   in.close();
}
```

また、Konoha は同一スコープ内であっても同じ型であれば、何回、型宣言を行っても

エラーとならない。そのため、コピー&ペーストする場合も支障とならない。

11.5 関数呼び出し

コール演算子 () は、メソッド/関数 f、コンストラクタ new C、もしくはフォーマッタ %f に続いて用いられ、メソッド/関数を呼び出すときに用いる。括弧 () の中には、0 個以上のパラメータ/引数を , で区切って与えることができる。コール演算子の評価値は、メソッド/関数の戻り値となる。(ただし、メソッド/関数の戻り値 void 型の場合は、評価値はない。)

```
f()                    // 関数 f() のコール
o.f()                  // o のメソッド f() をコール
new C()                // クラス C のコンストラクタ
%f(o)                  // o のフォーマッティング
```

コール演算子で与えられるパラメータは、メソッド/関数ごとに個別に定義されている。定義と異なるパラメータでコールしようとしたときは、型エラーとなる。

```
>>> fibo(10)
55
>>> Math.abs(-1)
1.000000
```

11.5.1 コール演算子の省略

コール演算子は、パラメータの数が 1 個であり、かつそれが文字リテラルである場合のみ、括弧 () 自体を省略することもできる。次は、c.query("""...""") と同じであるが、省略した方がすっきりとする。

```
>>> c.query """
select name, salary from PERSON_TBL
  where age > 45 and age < 65;
"""
```

11.6 ビルトイン関数

関数は、通常、プログラム実行時に評価される。しかし、いくつかのプログラミング言語機能は、コンパイル時に評価しなければならなかったり、もしくは評価した方が効率がよい場合がある。Konoha では、これらのコンパイラと密接に結びついた機能をビルトイン関数として提供している。

11.6.1 スタティックな型付け typeof()

ビルトイン関数 `typeof(expr)` は、プログラム中で、与えられた式 expr のスタティックな型を調べるビルトイン関数である。

```
>>> a = 1
>>> typeof(a)
Int
>>> typeof("hello,world")
String
```

これに対し、`(expr).class` は、実行時の expr がもつクラス（型）を調べるメソッドである。typeof(expr) と instanceof の関係が成り立つが、必ずしも一致するとは限らない。また、型をもたないトークン（例えば予約語など) を評価した場合、void 型が返される。

`typeof()` 関数の評価は、コンパイラの型検査時に行われる。つまり、あまりおすすめではないが、次のように型名の代わりに型宣言にも利用できる。

```
int f(int n) {
   typeof(n) m = n + 1;
   return m;
}
```

11.6.2 識別子の存在 defined()

ビルトイン関数 `defined(name)` は、識別子 name が定義されているかどうか調べる関数である。

```
>>> defined(Class)
true
>>> defined(System.LINUX)
false
>>> defined(a)
false
>>> a = 1
>>> defined(a)
true
```

`defined()` 関数の評価も、コンパイラの型検査時に行われ、必ず論理値 true もしくは false に置き換えられている。

```
if(!defined(System.LINUX)) {
   print os.uname;
}
```

11.6.3 デフォルト値をえる default()

デフォルト関数 default(expr) は、プログラム中で、与えられた式 expr の型からそのデフォルト値をえる関数である。

```
>>> default(String)
""
>>> default(%s(1))
""
>>> default(Script)                    // Script は実行依存
main.Script
```

多くのデフォルト値は、スタティックに解決される。しかし、いくつかのクラスは、実行コンテクストに依存する。そのため、デフォルト値も存在する。そのため、default() は typeof() と同様にスタティックに型まで決定し、一部のデフォルト値は実行時に得ている。

11.6.4 デリゲートの生成 delegate()

ビルトイン関数 delegate() は、デリゲートクロージャを生成する関数である。デリゲートの詳細は、「第??節 デリゲート」で述べる。

デリゲート delegate(o, m) は、閉包するオブジェクト o と呼び出すメソッド名 g をセットにしてクロージャを生成する。

```
>>> String s = "name";
>>> f = delegate(s, split);
>>> f()                                // s.split() が呼ばれる
["n", "a", "m", "e"]
```

もしくは、delegate(f) のように関数名 f のみを引数として与え、クロージャを生成することもできる。

```
>>> g = delegate(fibo);
>>> g(10)                              // Script.fibo(10) が
呼ばれる
55
```

11.6.5 フォーマッティング format()

format() は、「第??節 テンプレート・フォーマッティング」のとおり、テンプレート・フォーマッティングのためのビルトイン関数である。この関数がビルトイン関数である理由は、もし可能であれば、テンプレートのインライン展開を行い、フォーマッティン

グの性能を向上させるためである。

11.6.6 likely()/unlikely()*

likely()/unlikely() は、論理値 (true/false) を受け取り、その値をそのまま返すビルトイン関数である。ほとんどのプログラマには、この関数の存在意義からして疑わしいものかも知れないが、Linux カーネル開発者など一部からは非常に好まれている。少なくとも、ソースコードが読みやすくなるという効果は認められるようである。

```
if(unlikely(a == b)) {
    // あまり起こらないケース
}
```

現在、Konoha では、クラスルーム利用において、コンピュータアーキテクチャと結びつけて、プログラミングの深淵（の一部）を説明するときに役立つように導入している。

将来、優れた Konoha コンパイラが登場したとき、likely()/unlikely() をひょっとしたら正しく解釈して、分岐予測の最適化を行ってくれるかも知れない。現在は、無害で副作用のないビルトイン関数であるが、もし使うのなら正しく使うことをオススメする。

第 12 章

クラスとオブジェクト

Konoha のオブジェクト指向プログラミングは、名前ベースのクラス (nominal class)、単一継承 (single-inheritance) など、主に Java 言語のそれから影響を受けている。

12.1 class 宣言

新しいクラスは、class 文を用いて宣言することで定義できる。class 文の最も簡単なシンタックスは、次のとおりである。クラス継承を含めたシンタックスは、「第 13.1 節 クラス継承」で扱う。

```
class C {
  members // クラスのメンバー
}
```

クラスのメンバーには、フィールド変数 (field variable)、メソッド (method)、コンストラクタ (constructor) が含まれる。

- **フィールド変数**は、SmallTalk ではインスタンス変数、C++ ではメンバ変数、Java ではフィールドとそろぞれ異なる呼称をもつが、オブジェクトの内部状態を表す変数である。Java ではフィールド変数と呼ばないのが正しいが、Konoha では変数の種類をその位置によって区別する統一呼称を用いているので、「フィールド変数」と呼んでいる。
- **メソッド**は、SmallTalk ではメッセージ、C++ ではメンバ関数とも呼ばれるが、オブジェクトの内部状態を操作する手続きである。メソッドは、関数のフォームをとり、パラメータ（メソッドにおける引数のこと）を受けて処理を行い、その結果を戻り値として返す。Konoha では、オブジェクトへの操作が明確なときはメソッド/パラメータ、そうでないときは手続き型言語風に関数/引数と呼び分けている。どちらも同じである。

- **コンストラクタ**は、新しくオブジェクトを生成するときに、new 演算子を用いて呼び出される特別なメソッドである。なお、Konoha は、ガベージコレクションの機構を備えているため、明示的なオブジェクト破壊を行う必要なない。

さて、次は Konoha におけるクラス定義の例である。フィールド変数、コンストラクタ、メソッドの定義をもっている。フィールド変数は、class ブロックの一番先頭で宣言することが（よみやすいため）一般的であるが、定義する順番は自由に変えても構わない。

```
class Person {
   String _name;                    // フィールド変数
   int age;
                                    // コンストラクタ
   Person(String name, int age) {
      _name = name;
      _age = age;
   }
   String getName () {              // メソッド
      return _name;
   }
}
```

Konoha は、スクリプティング言語の柔軟さを提供するため、メソッドやコンストラクタを既存のクラスに追加するを認めている。しかし、フィールド変数のみは、class ブロックであらかじめ定義しなければならない。

12.1.1　クラスの種類

Konoha は、いくつかの特性の異なったクラスを分類している。これらは、class **文** の前に、アノテーションを与えることで宣言することができる。

アノテーション	説明
@Private	名前空間からみることはできない。
@Final	これ以上、継承することはできない
@Singleton	Singleton 関数
@Interface	インターフェースとして利用することが可能
@Release	デバッグの完了したクラス

12.1.2　Glue クラス

12.2　フィールド変数

フィールド変数は、アンダースコア (_) で始まる変数名をもつ。これは、Java プログラマがよく使っているローカル変数とフィールドを区別する慣習に由来するが、フィールド変数宣言のとき、アンダースコアを付け忘れなくても、コンパイラは自動的にフィールド変数にアンダースコアを付加する。

```
class Person {
   String _name;
   int age;        // もし _ を付けなくても
   Person(String name, int age) {
      _name = name;
      _age = age;  // _ は、フィールド変数へのアクセスに必要
   }
}
```

重要な留意点は、フィールド変数宣言のとき、アンダースコアを付けないことは、自動的な getter/setter メソッドの生成を意味する。これは、次節で述べるとおり、フィールド変数の public 宣言に等しい。

```
class Person {
   ...
   int age;
   int getAge() { return _age;}            // 自動生成
   void setAge(int age) { _age = age; }    // 自動生成
}
```

12.2.1 フィールドアクセサ

Konoha は、全てのフィールド変数はいわゆる private である。オブジェクト外部から直接参照はできず、全て getter/setter メソッドを用いて行う。

```
>>> p = new Person("naruto", 17);
>>> p.getName();
"naruto"
>>> p.setAge(18);
>>> p.getAge();
18
```

フィールドへのアクセスに getter/setter メソッドを多用するのは、Java が産み出したトレンドである。しかし、これは何かと不便に感じる人も多い。そこで、Konoha は、フィールドアクセサ (. 演算子) を、getter/setter を呼び出すための省略形 (シンタックスシュガー) として再利用している。

```
>>> p = new Person("naruto", 17);
>>> p.name;                    // 実は p.getName()
"naruto"
>>> p.age = 18;                // 実は p.setAge(18)
>>> p.age;
18
```

今までどおり、フィールドアクセスをしているように見えて、実は全て getter/setter メソッドによってふるまいを変更することができる。

12.2.2　バーチャルフィールド

バーチャルフィールドとは、フィールド変数の実体がなくても、getter/setter を通して、あたかもフィールドが存在するようにふるまうことである。

```
class Person {
    ...
    String name;
    String getFirstName() {
        return _name.split(" ")[0];
    }
}

>>> p = new Person("naruto uzumaki", 17);
>>> p.name                          // 実は p.getName()
"naruto uzumaki"
>>> p.firstName                     // 実は p.getFirstName()
"naruto"
```

Konoha では、そのオブジェクトのクラス Class インタフェースを通して、フィールド変数、（バーチャル）フィールドのリストを得ることができる。

```
>>> p = new Person("naruto", 17);
>>> p.class.fiedls("variable")
["_name", "_age"]
>>> p.class.fields("getter")
["name", "age", "firstName", "lastName"]
>>> p.class.fields("setter")
["name", "age"]
```

12.2.3　this 参照

キーワード `this` は、自オブジェクト参照への参照である。`this` を用いたフィールドアクセサを用いると、フィールド変数を直接参照する代わりに、getter/setter を経由してアクセスすることになる。

```
    ...
    int age;
    boolean isDrinkable() {
        return (this.age => 20);
    }
```

フィールドアクセサによる特別な結果を期待する場合でもない限り、`_age` で直接フィールド変数を参照した方が高速である。次のように、getter/setter メソッド内で this 参照を用いると、予期しない依存関係から無限ループに陥る危険性もある。

```
    int _age;                              // やってはいけない
    int getAge() { return this.age;}
    void setAge(int age) { this.age = age; }
```

Konoha コンパイラは、getter/setter 内では、this 参照を使うとエラーになる。

12.3 メソッド

メソッドは、オブジェクトに対してメッセージを送受信する手段である。0 個以上のパラメータを受け取り、0 もしくは 1 つの戻り値を返すことができる。次の `isChild()` は、0 個のパラメータ、Boolean 型の戻り値をもったメソッドの例である。

```
class Person {
   String name;
   int age;
    ...
   boolean isChild () {
       return (_age < 21);
     }
}
```

12.3.1 メソッド追加

Java 言語では、class 文の中でのみメソッドを定義することができました。Konoha では、動的言語の柔軟さを実現するため、既存のクラスに対して、スクリプト中どこでも新たなメソッドを追加定義することができます。次の Perso.isChild() 関数は、class Person 内で定義された isChild() と同じ定義になります。

```
boolean Person.isChild () {
   return (_age < 21);
}
```

メソッド追加は、これからインスタンス化されるオブジェクトのみでなく、（追加した時点で）既にインスタンス化されたオブジェクトにも影響が及びます。

```
>>> Person p = new Person("naruto", 17);
>>> boolean Person.isChild () {
...   return (_age < 21);
... }
>>> p.isChild()
true
```

12.3.2　抽象メソッド

抽象メソッドは、メソッドのインターフェースのみ定義され、実装パートが定義されていないメソッドである。Konoha では、メソッドにつづくブロックが存在しなければ、抽象メソッドと解釈される。また、メソッドのコンパイルに失敗したとき、自動的に抽象メソッドとして扱われる。

```
class Person {
   String name;
   int age;
    ...
   boolean hasFriend();              // 抽象メソッド
}
```

抽象メソッドの本来の用途は、抽象クラスやインターフェースなど、クラス設計においてポリモーフィズムを実現する手段としての利用である。厳密なスタティック言語では、全ての抽象クラスが実装されていない限り、クラスのインスタンス化はできない。しかし、Konoha は、抽象メソッドが含まれているクラスであっても、インスタンス化を認めている。これは、ラピッドプロトタイピングにおいて、単純に抽象メソッドを未実装なメソッドとして扱うためである。

抽象メソッドを実行すると、実行時例外となる。

```
>>> p = new Person("naruto", 17);
>>> p.hasFriend()
AbstractMethod!!: Person.hasFriend()
```

我々は、実行前にスタティックな抽象メソッドを検査する機構を検討している。現在のところ、実行時に `isAbstract()` によって検査する方法しかない。

```
>>> String.isAbstract()
false
>>> Person.isAbstarct()
true
```

12.3.3　パラメータの初期値

メソッドの多重定義 (overloding) は、異なる型のパラメータを同じ名前で定義することである。オブジェクトプログラミング言語の標準的な機能として、C++ や Java 言語で広く採用されているが、Konoha はメソッド再定義をサポートしていない。その代替機能として、パラメータの初期値を設定できる。

```
class {
```

```
    ...
    void Person.say(String msg="hello") {
        OUT << _name << " says " + msg << EOL;
    }
}
```

初期値が設定されたパラメータは、Nullable 型となる。したがって、パラメータを省略すると、自動的に null が与えられたものと解釈され、設定された初期値が渡される。

```
>>> Person.says;
void main.Person.says(String? msg);
>>> p.say("aloha");
naruto says aloha
>>> p.say();                                    // 省略, p.say(null) と同義
naruto says aloha
```

Konoha は、メソッドの多重定義をサポートしていない理由は、パラメータ初期値を活用することで、多くの場合のメソッド多重定義のバリエーションをカバーできるからである。あわせて、マップキャスト機能や Any 型パラメータを活用することで、いくつもメソッドを多重定義することなしに、同等なプログラミングが可能となる。

12.3.4 メソッドのバージョニング

Konoha は、複数のメソッド実装をバージョン管理する機構を備えている。まず、メソッドのバージョニングは、オリジナルなメソッドに対し、パラメータの型と戻り値の型はそのまま同じで、メソッド名にバージョンタグ (:tag) を付けることで作成できる。

次は、:JA タグを付加して定義した says() メソッドの日本語版である。

```
void Person.says:JA(String msg = "こんにちは") {
    OUT << _naame << "は、" << msg << "と言った." << EOL;
}
```

注意：(まだ厳密に決まった仕様ではないが、) 英大文字で始まるタグは、言語ロケール用に予約されている。小文字で始まるタグは、ユーザが自由に利用して構わない。

バージョニングされたメソッドは、呼び出すときに明示的にバージョンタグを付けることで呼び出すことができる。

```
>>> p.says();
naruto says hello
>>> p.says:JA();                                // 日本語版の利用
naruto はこんにちはと言った.
```

将来の拡張計画

将来の Konoha では、型による自動的なバージョン切り換えも導入を予定している。この機構には、セマンティックプログラミングの意味タグ型を流用する予定であるが、セマンティックプログラミングとの一貫性に関して検討を必要としている。

次は、Person クラスに対し、意味タグを追加して、意味タグ型 Person:JA 型の例である。意味タグ型は、同じタグが付けられたメソッドを優先して呼び出す出されることになる。

```
>>> Person:JA p = new Person("なると", 17);
>>> p.says();                              // says:JA() を利用する
naruto はこんにちはと言った.
```

12.4　クラス関数

クラス関数は、クラス名を直接レシーバにして利用できる特別なメソッドである。通常、メソッドは、クラスのオブジェクトを操作するための手段である。したがって、メソッドを利用するとき、クラスがインスタンス化されている必要がある。しかし、特定のインスタンスから独立しており、インスタンスを特定できない、もしくはする必要がない場合がある。

乱数生成メソッド `random()` は、特定の数値（インスタンス）の操作というより、それに独立したユーティリティ機能となる。ただし、Int や Float クラス程度に操作を区別する必要もある。Konoha では、これらは、クラス関数としてそれぞれ定義されて提供されている。

```
>>> Int.random()
143980198322
>>> Float.random()
0.39810
```

クラス関数は、メソッド定義のとき、`@Static` アノテーションを付けることで定義することができる。

```
@Static
int Int.max(int a, int b) {
   return (a > b) ? : a : b;
}
```

`@Static` アノテーションは、Java のスタティックメソッドに由来している。事実、見かけ上、クラス関数はスタティックメソッドと同様に利用することができる。しかし、実際はクラスのデフォルト値をレシーバとしてメソッドを呼んでいる。this キーワードを用

いれば、自オブジェクトとして参照もできる。

```
>>> Int.max(1, 2)
2
>>> (default(Int)).max(1, 2)      // 同じ
2
```

また、シングルトンクラスは、@Static アノテーションがなくても、自動的にクラス関数として解釈される。そのため、クラス関数としてコールしてもよいし、メソッドとしてコールしても構わない。

```
>>> System.exit()                 // System はシングルトン
>>> os = default(System)
>>> os.exit()                     // 同じ
```

Konoha は、シンプルな言語構造を実現するため、Java 風のスタティックフィールドをサポートしていない。クラスで共有したい変数は、単にスクリプト変数を用いることができる。

12.5 コンストラクタ

12.6 オペレータとメソッド

Konoha では、全てのオペレータはメソッドのシンタックスシュガーである。例えば、加算演算 x + y は、次のように単純にメソッド opAdd() に置き換えられて実行されている。

```
x.opAdd(y)                              // x + y
```

そこで、演算子と対応つけられた op で始まるメソッドを定義すれば、オペレータの振る舞いを変更することも可能である。ただし、演算子の再定義は、Konoha ではあまり積極的に推奨されない。

x + y	x.opAdd(y)	x - y	x.opSub(y)
x * y	x.opMul(y)	x / y	x.opDiv(y)
x mod y	x.opMod(y)		
x++	x = x.opNext()	x--	x = x.opPerv()
x == y	x.opEq(y)	x != y	x.opNeq(y)
x < y	x.opLt(y)	x <= y	x.opLte(y)
x > y	x.opEq(y)	x >= y	x.opNeq(y)
x & y	x.opLand(y)	x | y	x.opLor(y)
x ^ y	x.opXor(y)	~x	x.opLnot(y)
x << y	x.opLshift(y)	x >> y	x.opRshift(y)
|x|	x.getSize()	y in? x	x.opHas(y)
x[n]	x.get(n)	x[n] = y	x.set(n)
x[] = y	x.setAll(n)	x[s..e]	x.opSubsete(s,e)
x[s..<e]	x.opSubset(s,e)	x[s..+n]	x.opOffset(s,n)

第 13 章

クラス継承と抽象化

クラス継承 (class inheritance) は、オブジェクト指向プログラミング黎明期の一大関心事であった。しかし、オブジェクト指向プログラミングが広く普及し、多種多様なオブジェクトのデザインパターンが実践されるにつれ、それほど関心を集める話でなくなった。

Konoha は、そんな時代に設計されたので、「クラス継承」に関して、必要最小限の機能をサポートするのみにとどめている。(しかも、現在のところ、ほとんどクラス継承を使う機会はない。)

13.1 クラス継承

Konoha のオブジェクト指向プログラミングは、Java から強い影響を受けている。クラス継承を用いるクラス定義は、Java 同様、 class 文において extends 節によって上位クラス D を指定することができる。クラス継承によって定義されたクラス D は、上位クラス C に対し、「サブクラス (subclass)」と呼ぶ。

```
class D extends C {
   members // クラスのメンバー
}
```

Konoha は、Java のオブジェクト指向プログラミングの特徴を強く受け継いで設計されている。

クラス継承とは、上位クラスの振る舞い、つまりメソッドを引き継ぐことである。サブクラスでは、上位クラスのメソッドを継承するのみならず、新たなメソッドを追加することもできる。次のクラス D は、クラス C のサブクラスである。メソッド `C.sayHi()` を継承し、新たに `D.sayBye()` を追加している。

```
>>> class C {
...     void sayHi() { print "hi"; }
... }
```

```
>>> class D extends C {
...     void sayBye() { print "bye"; }
... }

>>> D d = new D();
>>> d.sayHi()            // c.sayHi() が呼ばれる
hi
>>> d.sayBye()           // 追加されたメソッド
bye
```

13.1.1　final クラス

Konoha では、クラス定義において、Java 言語同様、final 修飾子、もしくは@Final アノテーションを付けることで、クラス継承不可能なクラスとして宣言できる。

```
final class C {
   void sayHi() { print "hi"; }
}

class D extends C {     // C は継承できない
   void sayBye() { print "bye"; }
}
```

13.2　オーバーライド

オーバーライドは、継承されたメソッドをサブクラスにおいて上書き、つまり新たに作り直す機構である。オブジェクトに対しポリモーフィズム (polymorphism) を与える重要な機構であるが、同時に、ダイナミックバインディングによる性能低下が議論となる機構でもある。

Konoha は、C++ や C#の流儀にしたがって、@Virtual アノテーションが付けられたメソッドのみオーバライドすることができる。これは、実際のプログラミングにおいて、メソッドがオーバーライドされることは稀であり、Java のようにデフォルトでオーバーライドを認めることによる性能ロスを無視できないと判断したためである。プログラマがダイナミックバインディングを避けるため final を追加するようなテクニックを駆使することなく、ふつうに書いて最適な速度が得られる方を好んだといえる。

```
class C {
   @Virtual void sayHi() { print "hi"; }
}

class D extends C {
```

```
    // オーバライド
    @Override void sayHi() { print "Hey"; }
    void sayBye() { print "bye"; }
}
```

注意：@Virtaul メソッドは、ダイナミックバインディングされるため、どうしてもスタティックバインディングに比べると遅くなる。Konoha では、インラインキャッシュによる高速化の工夫をしているが、オーバーライドする必要のないメソッドは@Virtual 宣言しないようにすべきである。また、抽象メソッドもオーバーライドされたメソッドも、同様に、ダイナミックバインドされる。

13.2.1 ポリモーフィズム

13.3 super

sf super は、上位クラスのフィールドやメソッドへアクセスするためのリファレンスを表す表現である。

13.4 インターフェース*

Konoha は、Java と同様に、単一継承 (single inheritance) である。複数のクラスの振る舞いを継承するクラスを定義したいときは、インターフェースを実装する方法を用いる。次は、クラス D とクラス E のインターフェースを実装することを宣言したクラスである。

```
class C implements D, E {
   members // クラスのメンバー
}
```

Konoha は、Java と異なり、インターフェースを定義する専用の interface 文は存在しない。その代わり、任意のクラスをインターフェースとして実装することができる。

```
>>> class C {
...     void sayHi() { print "hi"; }
...     void sayBye() { print "bye"; }
... }

>>> class D implements C {
...     void sayHi() { print "Hai"; }
... }
>>> D d = new D();
>>> d.sayHi()                          // 自分で作る
Hai
>>> d.sayBye()                         // 継承されない
```

```
** AbstractMethod!!
```

注意: インターフェースは、あくまでもメソッドの設計（シグネチャ）のみ継承し、実装は継承しない。Konoha では、抽象メソッドとして継承される。そのため、新たに追加しない限り、implements されたインターフェースのメソッドは、`Abstract!!`例外になる。

13.4.1　インターフェースへのメソッド追加**

13.5　Object クラス

Object クラスは、全てのクラスの共通した上位クラスである。全てのオブジェクトは、Object クラスのメソッド（オペレータ、フォーマッタも含む）を継承している。

第 14 章

例外処理

例外は、動的な非局所ジャンプの一種である。

14.1 例外クラス Exception

Exception は、例外の種類、状態を表現するクラスである。

クラス Exception は、Konoha において例外を実現する唯一のクラスである。例外の種類は、例外クラス内部の表現として分類され、どの例外も Exception クラスのインスタンスとなる。

```
>>> e = new Security!!();
>>> e.class
Exception
>>> e = new IO!!();
>>> e.class
Exception
```

14.2 throw 文

Konoha は、プログラム実行中に何らかの異常を検出したとき、例外をスローする。例えば、0 除算が発生した場合、その異常状態は `Arithmetic!!`例外がスローされたことで知ることができる。

```
>>> a = 0;
>>> 1 / a;
** Arithmetic!!: Divide by Zero
```

同様に、プログラマ自身も、専用のステートメント throw を使うことによって、例外をスローすることができる。最も標準的な例外のスローは、新しく例外クラスを作成し、それを throw 文に与える方法である。

```
>>> throw new Security!!("Something wrong");
** Security!!: Something wrong
```

Konoha の throw 文では、文字列（エラーメッセージ）をそのまま投げることができる。文字列の先頭が例外名のタグであれば、例外の種類が適切に判別され、そうでなければ単純に Exception!!になる。

```
>>> throw "Security!!: Something wrong";
** Security!!: Something wrong
```

14.3　try-catch 文

try-cacth 文は、try ブロック内で発生した例外を捕捉（キャッチ）し、プログラムの正常化を行うための制御構造である。

```
try {
    // ブロックでスローされた例外に対する
}
catch(IO!! e) {
    // プログラムの正常化
}
```

catch 節は、複数種類の例外をそれぞれ処理することができる。例外はソースコードの順に先頭からマッチング処理を行われ、最初にマッチングした catch 節が処理される。マッチングは、e instanceof E!!による半順序マッチングであり、複数の catch 節の条件にマッチする可能性があるが、最初のひとつ以外は catch 節は無視される。

```
try {
    // ブロックでスローされた例外に対する
}
catch(Security!! e) {
    // プログラムの正常化
}
catch(SQL!! e) {
    // プログラムの正常化
}
catch(IO!! e) {
    // プログラムの正常化
}
```

また、どの catch 節の例外処理にもマッチしなかった場合は、例外は捕捉されることなく、そのまま継続してスローされ続ける。

14.4 finally 節

第 15 章

クロージャ

Konoha の Closure クラスとそれを用いたプログラミングに関して執筆する予定である。

第 16 章

スクリプトと名前空間

スクリプトは、Konoha のプログラムの単位である。プログラムは、スクリプトファイルに記述され、それがロードされたのち、コンパイルされたスクリプトは Script クラスと名前空間 NameSpace クラスによって管理される。本章では、Script と NameSpace クラスを紹介しながら、Konoha 特有のスクリプティング機能も紹介する。

16.1 名前空間 NameSpace

Konoha は、大規模なソフトウェア開発を想定しているわけでないが、シンプルかつ十分な名前空間の機構を備えている。全てのスクリプトは、最初、デフォルトの名前空間 `main` をもっている。

現在の名前空間は、システム変数`__ns__`で確認することができる。

```
>>> __ns__
"main"
```

システム変数`__ns__`は、NameSpace 型で、現在の名前空間のオブジェクトを返す。通常、名前空間は、using 文などを通して自動的に一貫性をもって管理されている。

16.1.1 クラス識別子とニックネーム*

クラスは、class 文で宣言されたとき、その名前空間を前置名にして宣言される。実は、これが Konoha システム内での正式なクラス識別子となる。

```
>>> class C ;
>>> C                                    // 正式なクラス識別子
main.C
```

Konoha は、クラス識別子を型の名前として仕様できない。あくまでもクラス名を用いる。それぞれの名前空間は、クラス名とクラス識別子を結びつけたニックネーム表をもっ

ている。

```
>>> class C
>>> %dump(__ns__)                    // ニックネーム表の確認
Script     main.Script
C          main.C
```

16.1.2　パッケージとクラス名*

パッケージは、パッケージ名を名前空間としてもったスクリプトである。例えば、math パッケージ内の Math クラスは、クラス識別子は `math.Math` となる。

konoha パッケージ外の名前空間に属すクラス識別子は、自動的にニックネーム表に登録されない。そこで、明示的にニックネーム表に登録する必要がある。

using 文は、もしパッケージが未ロードであればパッケージのロードを行ったのち、指定されたクラスをニックネーム表に登録するステートメントである。

```
>>> using math.Math;
>>> Math
math.Math
>>> %dump(__ns__)
Script     main.Script
C          main.C
Math       math.Math
```

もし指定したパッケージ内にニックネーム表と同じクラス名が存在する場合は、既存の名前を優先する。強制的に新しい名前を用いたい場合は、`@Override` を使うこともできる。ただし、予期せぬ混乱をまねく可能性があるのであまりおすすめできない。

もし、パッケージ内の全てのクラスをロードしたい場合は、ワイルドカードを用いることができる。パッケージ内に多くのクラスが存在するときは、こちらを使うと便利である。

```
>>> using math.*;
```

また、クラス名を省略すると、パッケージのみのロード、もしくは強制的な再ロードを意味する。このときは、ニックネーム表は更新されない。

```
>>> using math;
```

16.2　スクリプトクラス Script

Konoha は、スクリプトファイルを読み始める前に、新しい Script クラスを生成する。このとき、名前空間ごとに異なるクラスを生成するため、名前空間が `main` であ

れば、クラス識別子は main.Script のように生成される。また、新しく生成された main.Script は、Konoha が標準的に提供する基本パッケージ konoha.Script の性質を継承している。

Script クラスとスクリプトファイルは、ちょうど class Script 宣言のブロック中にスクリプトファイルが置かれたものと考えるとわかりやすい。グローバル（に見えた）変数は、実は Script クラスのフィールド変数に相当し、関数も Script のメソッドとなる。Konoha 言語では、それぞれ**スクリプト変数**、**スクリプト関数**と呼ばれる。

class Script

```
{
// スクリプトファイル の始まり


// スクリプトファイル の終わり
}
```

スクリプトファイルが、通常のクラス宣言と異なる点は、直接、ステートメントを記述できる点である。(これは、**スクリプトステートメント**と呼ばれる。) 例えば、次はスクリプトステートメントの例である。

```
for(i = 0; i < 10; i++) print i;
```

スクリプトステートメントは、無名関数によってラップされて実行される。つまり、上のスクリプトステートメントは、次の lambda() 関数定義と呼び出しと同義である。

```
Any lambda() {
   for(i = 0; i < 10; i++) print i;
}
lambda();
```

注意：次節で述べるとおり、変数 i は、lambda() 関数内のローカル変数となるため、注意が必要である。

16.3 スクリプト変数

スクリプト変数は、Script クラスのフィールド変数である。グローバル変数のように振る舞うが、実は名前空間が違えば、その名前空間に固有の Script オブジェクトを通してアクセスしているため、参照されることはない。つまり、スクリプト変数のスコープは名前空間によって分離されている。

```
// スクリプト変数
card = new int[54];                    // スクリプト変数
```

Script クラスは、特別な`@Singleton`クラスとして導入されている。インスタンスの数がひとつに限定されるため、実行中であっても一貫性を乱すことなくフィールド変数を増やすことができる。その上限数は、おおよそ 128 変数くらいである。上限数がはっきりしない理由は、Konoha の最適化オプションの状態で、UNBOXFIELD が採用された場合、int と float はそれぞれ 2 変数分の領域を消費することがあるためである。どちらにしても、通常のプログラミングにおいてスクリプト変数を使い切ることはないし、そのようなスクリプトは（メンテナンス性がないため）書いてはいけない。

ちょっとしたテクニックであるが、スクリプト中で一次的に利用する変数は、ブロックステートメントなどを利用してローカル変数として扱うこともできる。また、スクリプト変数が不足するようなプログラムはメンテナンス不能になるため、書いてはいけない。

```
card = new int[54];                    // スクリプト変数
{
   int i;                          // ローカル変数
   for(i = 0; i < |card|; i++) {
      card[i] = Int.random(i);
   }
}
```

16.3.1　変数スコープ

Konoha は、単純化された 3 つの変数スコープをもつ。変数名は、まずローカル変数を検索し、続いてフィールド変数、最後にスクリプト変数を検索する。明示的に、フィールド変数やスクリプト変数を参照したい場合は、変数名の前に_, __をそれぞれ付ける。もしくは、フィールド変数、スクリプト変数を宣言するときにあらかじめ付けておいても構わない。

```
>>> String s = "script";          // スクリプト変数
>>> class Person {
>>> String s = "field";           // フィールド変数
...     void test() {
...          String s = "local";  // ローカル変数
...          print s, _s, __s;
...       }
...    }
>>> t = new T();
>>> t.test()
s="local", _s="field", __s="script"
```

16.3.2　定数の定義

スクリプトの重要な役割は、定数の定義である。Konoha では、名前ルールによって、グローバル定数、ローカル定数、クラス定数がある。

```
GLOBAL_ =
LOCAL =
Math.PI_2 =
```

第 17 章

デバッグ

スクリプティング言語は、一般に「プログラミングしやすい」言語とみなされている。その理由のひとつは、ラピッドプロトタイプピングによる、よくテストしながらの開発スタイルにあると考えられる。しかし、どのようなソースコードにも必ずバグは存在し、モダンなプログラミング言語は、バグを発見するのを支援する機能を備えている。

Konoha は、いくつかの、例によって、本格的なデバッグ機能ではないが、クラスルーム利用においてデバッグの基本技法を学ぶのに十分なデバッグ機能をサポートしている。

17.1 DEBUG ブロックとデバッグモード

デバッグは、バグの原因を分析するデバッグ情報をえることから始まる。そのためには、ソースコードを眺めているより、実際にプログラムを実行させて、内部状態の出力をえる方がバグを発見しやすい。

次は、簡単なバグの例である。プログラムが停止しなくなる理由を調べるため、引数 n を表示している。

```
int fibo(int n) {
        if(n < 0) print n;                    // デバッグ情報の出
力
        if(n == 0) return 0;
        return fibo(n-1) + fibo(n-2);
}
```

デバッグ情報は、特別な解析ツールを利用しない限り、プログラマ自身がプログラムして出力することになる。つまり、デバッグ情報の出力もソースコードの一部となる。こうなると、正規のソフトウェア機能とデバッグ情報の出力パートが混在し、いろいろ不都合が生じる。

Konoha では、DEBUG ブロックと呼ぶ専用のステートメントブロックを導入し、デバッグのためのソースコードを分離して書くことができる。これは、後から述べるとおり、リリース/デバッグの実行モードを切り替えることで、デバッグ情報の出力を切り替え

ることが可能にする。

```
int fibo(int n) {
        DEBUG {                                    // デバッグ情報の出
力
                if(n < 0) print n;
        }
        if(n == 0) return 0;
        return fibo(n-1) + fibo(n-2);
}
```

17.1.1　デバッグモードとリリースモード

Konoha は、デバッグモードとリリースモードの 2 種類の実行モードを持っている。デバッグモードは、開発者向けのモードであり、デバッグ情報を出力しながら実行する。リリースモードは、一般のユーザの利用を前提としたモードであり、逆にデバッグ情報を抑制して実行するモードである。

Konoha は、通常、リリースモードでスクリプトを実行する。デバッグモードで実行する場合は、起動オプションに -g を用いる。

```
$ konoha -g script.k
```

注意：対話モードは、-g を付けなくてもデバッグモードとして起動される。

17.1.2　実行時のモード切り換え

Konoha 内部では、デバッグ/リリースモードの切り換えは、実行コンテクスト (Context) 単位で行っている。そのため、Context クラス関数を通して、実行中にモードを切り替えることもできる。

```
Context.setDebug(true);              // デバッグモードに変更
Context.setDebug(false);             // リリースモードに変更
```

「実行時のモード切り換え」を正しく使うためには、Konoha のスクリプト実行の内側まで注意する必要がある。スクリプトの実行は、厳密に言えば、コンパイル（コード生成）とコードの実行からなり、リリース/デバッグモードはコンパイル時のコード生成にも適用される。特に、リリースコンパイル時は、実行速度向上と利用メモリ量軽減を意図したコード最適化のため、DEBUG ブロックと（後述する）print 文、assert 文は無視される。したがって、リリースコンパイルされたコードは、デバッグモードで実行してもデバッグ情報は表示されない。

17.1.3 @Release アノテーション

クラス宣言やメソッド定義のとき、`@Release` アノテーションを付けると、その対象となるクラスとメソッドは、デバッグコンパイルであっても常にリリース版のコードが生成される。十分に開発が完了した部分は、`@Release` アノテーションを付けることで、開発チームにもコンパイラにも開発の進捗を明確に伝えることができる。

```
@Release
class Person {
        ...
}
```

注意: クラスに、`@Release` アノテーションを付けると、そのクラスの全てのメソッドがリリースとなる。

17.2 print 文

Konoha における正式なプリントアウト方法は、いわゆる OUT 定数を用いた OutputStream クラスを用いた標準出力である。一方、古式ゆかしい print 文もサポートしている。これは、クラスルームにおいて、「print デバッグ」を教えるとき print 文がないと都合がわるいからである。もちろん、誰にとっても「print デバッグ」は有効であり、Konoha における print 文は、「print デバッグ」専用の機能を備えている。

print 文は正式なプログラムの一部ではなく、デバッグ/リリースモードによって、プリントアウト出力のありなしが切り替わる。リリースコンパイルでは、そもそも print 自体、無視される。

もうひとつの特徴は、print 文は、デバッグ情報として加工された情報が出力されることである。まず、必ず print 文が実行された位置が [ファイル名:行番号] として出力される。変数の値を出力するときは、（ご親切に）変数名も自動的に出力される。

```
>>> a = 0;
>>> print a;
[shell.c:4] a=0
```

単一の print 文において、複数の情報を出力する場合は、カンマで区切る。変数名以外の式を与えると、式の評価結果が表示される。

```
>>> a = 0;
>>> b = 1;
>>> print a, b, (a+b);
[shell.c:4] a=0, b=1, 1
```

17.2.1　@Date アノテーション*

print 文は、@Date アノテーションを付けることができる。@Date アノテーションは、print 文出力に対し、デバッグ情報を出力した時刻を表示するオプションである。

```
>>> @Date print a, b, (a+b);
2009/12/24 00:00:00 [shell.c:4] a=0, b=1, 1
```

17.2.2　ロッギング**

Konoha プロジェクトでは、print 文とロギング (syslog) との統合を検討し、@Log(ALART) のようなわかりやすい拡張オプションを提供する予定である。

```
>>> @Log(PANIC) print name;
```

17.3　assert 文

アサーション (assertion) は、表明と訳される。assert 文は、プログラムが正しく動作するための前提条件を表明し、プログラムが表明された条件通り正しく動作しているか確認するデバッグ機能を提供する。

まず、正しいプログラムという概念を考えるため、簡単な例を考えてみたい。次の fibo() 関数は、引数 n において n > 0 を満たす整数が与えられるという前提でプログラミングされている。しかし、利用者の方は原理的に fibo(-1) とコールすることもできる。(その場合は、StackOverflow!!がスローされる。)

```
>>> int fibo(int n) {
...         if(n == 1 || n == 2) return 1;
...         return fibo(n-1) + fibo(n-2);
... }
>>> fibo(-1)
** StackOverflow!!
```

ここで、「引数が負の数でも動作するように fibo() 関数を作らなかった」方に責任があるのか、それとも「fibo(-1) をコールした」方が悪いのか、という責任問題が発生する。どちらにバグの責任があるかはっきりしない場合は、デバッグすることができない。

今回は、もし前者の立場をとれば、あらゆる引数値の範囲を確認し、その範囲を超えた場合は、何らかの例外をスローする方法も考えられる。(ちなみに、勝手に引数の値を修正してプログラムを継続する修正はあまり望ましいデバッグではない。)

```
int fibo(int n) {
```

```
        if(n > 0) {
                if(n == 1 || n == 2) return 1;
                return fibo(n-1) + fibo(n-2);
        }
        throw new Arithmetic!!("Negative fibo");
}
```

逆に、負のフィボナッチ数列は通常、定義されていないため、そもそも `fibo(-1)` がおかしいと考えてもよい。そのように考えても、プログラミング言語側はそこまで察することはできない。そこで、何らかの方法で `fibo(n)` の正しい動作条件を表明しておく必要がある。

assert 文は、プログラムの正しい動作を表明するステートメントである。assert 文に続く、条件式には、正しい動作のとき true になるように書く。

```
int fibo(int n) {
        assert(n > 0);
        if(n == 1 || n == 2) return 1;
        return fibo(n-1) + fibo(n-2);
}
```

assert 文による表明は、実行時における動的な検査で判定される。もし表明に違反した場合は、`Assertion!!`例外がスローされ、プログラムの実行は正常化されるまで実行が中断される。ただし、`Assertion!!`例外の標準的な対応は、表明に違反したコードを探して、デバッグを行うことである。

```
>>> fibo(-1)
** Assertion!!: n > 0
```

注意すべき点は、assert 文はデバッグ作業を助けるステートメントである点である。一般に、assert 文は、頻繁にコールされる関数に用いると、パフォーマンス低下の原因となる。そのため、リリースコンパイルでは除去してコンパイルされる。リリース前に、徹底的にソフトウェアテストを行い、`Assertion!!`例外が発生しないようにデバッグする必要がある。

17.3.1 リリース・アサーション

アサーションは、デバッグ用の機能である。しかし、リリース版であっても、致命的な実行結果をもたらす前に、アサーション機能によって処理を中断させたいと考える場合もある。Konoha では、`@Release` アノテーションを assert 文の前に追加することで、リリースコンパイル時でも assert 文がコード生成されるようになる。

```
@Release assert(n > 0);
```

注意：頻繁にアクセスされる箇所に、リリース・アサーションを用いると、性能低下の原因となる。リリース・アサーションの導入は慎重に検討すべきである。

17.3.2　assert 拡張文法

C/C++ では、`assert()` は関数 (マクロ) で提供されている。Konoha では、基本的に互換性を保った形で、ただし条件式に続けてステートメントを記述できるように拡張してある。したがって、assert 文の文法は次のとおりになる。

`assert` (*expr*) *stmt*

ここでは、まず条件式 *expr* を評価し、その結果が true なら何もおこらない。もし、false の場合、それに続くステートメント *stmt* が実行され、最後に `Assertion!!` 例外がスローされる。

何のための拡張文法かと言えば、要するに `Assertion!!` が発生したとき、その瞬間のデバッグ情報を表示するためである。(ただし、意外と便利である。)

```
assert(n > MAX) {
        print n, MAX;
}
```

17.4　utest 文**

Konoha プロジェクトでは、ユニットテストをサポートするための専用のステートメントを導入する計画である。

```
@Name("test name")
utest (n > 0) {       // テスト結果の表明
        // テストコード
}
```

17.5　ブレークポイント**

Konoha の対話モードでは、デバッガ (gdb) の代表的なデバッグ機能をサポートする予定である。これらの機能は、バイトコードの実行時書き換えの API が整備したのち、拡張される予定である。

第 18 章

Konoha ライブラリ

Konoha は、C 言語のライブラリ (libkonoha, konoha.dll) として提供され、アプリケーションに組み込んで利用することができる。

18.1 Konoha インスタンス

Konoha インスタンスは、Konoha スクリプティング言語エンジンと実行コンテクストから構成され、`konoha_t` という型で定義されている。Konoha 言語の機能は、全て Konoha インスタンスを通して実行される。

次は、最も簡単な Konoha インスタンスの利用例である。まず、Konoha インスタンスの生成 (`konoha_open()`) し、Konoha ステートメントを実行 (`konoha_eval()`) し、最後に Konoha インスタンスを解放 (`konoha_close()`) している。

```
#include<konoha.h>
int main(void)
{
   konoha_t k = konoha_open(0);
   konoha_eval(k, "print 'hello,world';");
   konoha_close(k);
   return 0;
}
```

Konoha インスタンスは、実行コンテクストを保持しているので、同じ Konoha インスタンス上で定義されたスクリプト変数やスクリプト関数は、その定義を持続して評価が行える。

```
#include<konoha.h>
int main(void)
{
```

```
    konoha_t k = konoha_open(0);
    konoha_eval(k, "a=1;");
    konoha_eval(k, "b=2;");
    konoha_eval(k, "print a + b;");
    konoha_close(k);
    return 0;
}
```

また、スクリプトファイルをロードする API も用意されている。スクリプトのロードのみであるため、main() 関数は実行されない。

```
#include<konoha.h>
int main(void)
{
    konoha_t k = konoha_open(0);
    konoha_include(k, "file.k");
    konoha_close(k);
    return 0;
}
```

18.1.1 マルチ・インスタンス

Konoha ライブラリは、マルチ・インスタンス対応である。つまり、複数のインスタンスを同時につくることが可能である。同時に作成できるインスタンスの数は、システムのリソースが続く限り、無制限である。

```
#include<konoha.h>
int main(void)
{
    konoha_t k1 = konoha_open(0);
    konoha_t k2 = konoha_open(0);
    ...
    konoha_close(k1);
    konoha_close(k2);
    return 0;
}
```

Konoha のインスタンスは、メモリ効率はあまりよくないが、完全に独立したメモリ領域を持つように設計されている。したがって、マルチスレッド環境においては、スレッドごとに Konoha インスタンスを割り当てる限り、ブロッキングもレースコンディションも起こらない。もし異なるスレッドから、同じ Konoha インスタンスを利用したときは、実行コンテクストが混在して破滅的な結果を招くことがある。

18.2 C からのスクリプト関数の利用*

Konoha で開発されたプログラムは、スクリプト関数を通して利用することができる。

```
konoha_fcall(k, "funcname(%s)", "value");
konoha_fcall_int(k, "funcname(%s)", "value");
konoha_fcall_float(k, "funcname(%s)", "value");
konoha_fcall_text(k, "funcname(%s)", "value");
```

第 19 章

C/C++ ライブラリの利用

スクリプティング言語は、別名「グルー (glue) 言語」とも呼ばれる。それは、C/C++ など、他のプログラミング言語で開発されたライブラリを「糊のごとく」はり合わせて利用することが一般的な利用形態であるためである。実際、多くの C/C++ ライブラリが、Konoha にバインドされて、Konoha から利用可能になっている。

19.1 C 言語関数のバインド

最も簡単なバインドは、C 言語の関数を Konoha のクラス関数へのバインドである。まずは、C 言語の標準ライブラリの `math.h` を例にしながら、バインド機構を紹介する。

19.1.1 パッケージの作成

Konoha では、バインドされた C/C++ ライブラリは、全てパッケージ機構で管理される。まず、パッケージ名を決めてパッケージスクリプトを開発する。

ここでは、`math` パッケージとする。パッケージスクリプトのファイル名は、`math.k` となる。新しく作成したパッケージは、次のどこかのディレクトリに保存する。

1. **環境変数**
   `{$KONOHA_PACKAGE}/math/math.k`
2. `Konoha` **インストールディレクトリ**
   `{$konoha.home}/package/math/math.k`
3. **ローカルパッケージ**
   `~/.konoha/math/math.k`
4. **一時的なパッケージ**
   `./.konoha/math/math.k`

注意: パッケージスクリプトは、上の順番で検索され、最初に発見したスクリプトがロードされる。今回の例としている math パッケージは、インストールされているため、

実際に試している読者はパッケージ名を変更するなど工夫が必要である。

さて、肝心のパッケージスクリプト math.k の内容であるが、特別なことはない。単純に、バインドする (予定の) 関数を Konoha からみた名前や型で設計するだけである。メソッドの実体は、バインドするため、抽象メソッドとして宣言するのみである。

```
class Math;
@Static float Math.sin(float x);
@Static float Math.abs(float x);
...
```

注意：Konoha は、オブジェクト指向プログラミング言語であるため、クラスが機能の単位の中心となる。そのため、非オブジェクト指向言語の C 関数であっても、何らかのクラス関数としてバインドする必要がある。（スクリプト関数にバインドすることはできない。）上記の例では、Math クラスをクラス関数のバインド先として定義している。

19.1.2 グルー関数の作成

Konoha の内部では、メソッドは knh_fmethod 型で定義された関数で実装されている。次は、konoha.h ヘッダファイルに定義されている knh_fmethod 型の抜粋である。コンテクストとスタックフレームポインタを引数として受け取り、戻り値は void 型である。（ただし、一読してメソッドとわかるためと fastcall の切り換えのため、METHOD マクロで定義されている。）

```
#define METHOD void KNH_CC_FASTCALL
typedef METHOD (*knh_fmethod)(Ctx *, knh_sfp_t *);
```

グルー関数とは、knh_fmethod 型に準拠したラッパー関数のことである。その機能は、単に、Konoha 側のスタックフレームポインタで渡されたパラメータを C 言語の型に変換し、C 関数を実行し、そして C 関数の実行結果を再び、Konoha 側のスタックに戻している。

```
// Static float Math.abs(float x);
METHOD Math_abs(Ctx *ctx, knh_sfp_t *sfp)
{
   double v = (double)sfp[1].fvalue;    // C 言語の型へ変換
   v = fabs(v);
   KNH_RETURN_Float(ctx, sfp, v);       // Konoha のスタック
へ戻す
}
```

グルー関数の仕様は、次節で詳しく述べるが、sfp[1] には第一引数、sfp[2] には第二引数、sfp[n] には第 n 引数の値が格納されている。例えば、次のような場合、第一引数の値は、-1.0 であり、それは sfp[1].fvalue に格納されている。

```
>>> Math.abs(-1.0)
1.0000
```

グルー関数の特徴は、型チェック済みの引数が必ず渡される点である。これは、パッケージスクリプトで宣言したメソッド定義に基づいている。つまり、グルー関数の開発者は、特別な場合をのぞいて sfp[1] の型をチェックする必要はない。

逆に、戻り値は、グルー関数の開発者が型チェック済みの値を正しく返さなければならない。Math.abs() の戻り値の型は、float であるため、専用のマクロ KNH_RETURN_Float を用いて、Konoha スタックに戻り値を設定し、同時にグルー関数から抜け出している。

注意：グルー関数内で設定する戻り値の型が間違っていた場合は、ほぼ例外なくクラッシュする原因となる。グルー関数の作者は細心の注意を払う必要がある。

19.1.3 共有ライブラリのコンパイル

19.2 グルー関数とメソッド

19.3 構造体とクラス

19.4 関数ポインタとクロージャ

参考文献

[1] http://www.math.sci.hiroshima-u.ac.jp/ m-mat/MT/mt.html

付録 A

入手方法

```
$ tar zxfv konoha-0.x.y.tar.gz
$ cd konoha-0.x.y
$ ./configure
$ make
$ sudo make install
```

索引