プログラミング言語

Kinx

Kray-G, Mr.Diamond Global Blue Publisher
September 1, 2021

目次

図目次

表目次

1 はじめに

1.1 Kinx とは

世の人気を二分するメジャーなスクリプト言語として Ruby と Python があります。それぞれ多くのユーザーを抱え、現実に存在する世の中の問題を解決するために非常に多くのシーンで使われています。ですが、どちらも C ライクではありません。なぜでしょうか。

また、最近ではより安全と言われる静的型付け言語が人気です。しかし動的型付け言語は、スキルさえ問題なければ[1]より高い生産性を発揮できる非常に使い勝手の良い道具であるはずです。C 系統のシンタックスという意味では JavaScript または TypeScript で node.js、という選択肢もありますが、node.js は全く Lightweight ではありません。また、イベントループによる制御自体がスクリプト言語として使いづらいといった側面もありました。

そこで Ruby、Python と同様の目的を持つ、動的型付けの特徴と C 系統のシンタックスを備えた、使いやすいスクリプト言語が欲しいという願望から本プロジェクトはスタートしました。

つまり、Kinx を一言で言えば、「**手に馴染んでいる伝統的な C 言語系統のシンタックスを受け継いだ、汎用スクリプト言語**」となります。

1.2 Kinx の特徴

1.2.1 コンセプト

Kinx のコンセプトは、**Looks like JavaScript, feels like Ruby, and it is a script language fitting in C programmers.** です。

見た目は JavaScript のようであり、使った感触はまるで Ruby のように柔軟な、それでいて C プログラマー（C 系統のシンタックスに慣れ親しんだ多くのプログラマー）に最も馴染むスクリプト言語を目標としています。あなたが C プログラマーであれば、Ruby、Python よりもずっと快適で、手に馴染むような感覚を得ることができるでしょう。

1.2.2 主な特徴

Kinx は**動的型付け言語**であり、**オブジェクト指向プログラミング言語**です。その主な特徴は以下の通りです。

- 動的型付け言語
- オブジェクト指向プログラミング言語

[1] 特に大規模開発において、この「スキルさえ問題なければ」の部分に疑問符が付けられているのが現状です。それが、静的型付け言語人気の理由です。

- C 系統（ほぼ JavaScript）のシンタックス
- クラスと継承、モジュールとミックスイン、ファイバーのサポート
- 高次関数・レキシカル・スコープ・クロージャのサポート
- ガーベジ・コレクションによるメモリ管理
- JIT コンパイルされる `native` 関数。

1.2.3 本書について

本書は「プログラミング言語 Kinx」をまとめた初めての本です。現在サポートされている機能は一通り網羅する予定ですが、今後変更される可能性も勿論あります。あらかじめ、ご了承ください。なお、本書の付録として各種リソース情報を掲載していますので、最新情報などにつきましては、ぜひリンク先をご覧ください。

2
さあ始めよう

2.1 インストール・アンインストール

2.1.1 インストール

2.1.1.1 Windows

Windows での Kinx のインストールは Scoop を使用します。

```
// scoop bucket add is needed only at the first time.
$ scoop bucket add kinx https://github.com/Kray-G/kinx
$ scoop install kinx
```

Bucket の登録は最初に 1 度だけ必要です。Kinx がインストールできたら、パスが通っているか確認するために一度バージョン表示をしてみましょう。

```
$ kinx.exe -v
kinx.exe version 1.1.0 built on 4b2bd84d86fc4e30a78fffbf84805158dc097193
```

Kinx へのパスが通っていることを確認したら、今後追加するかもしれないパッケージ・コマンドに対しても正しくパスが通るように、以下のコマンドを実行しておきます。

```
$ kinx --install-path
```

これで準備は完了です。

2.1.1.2 Linux(Ubuntu)

Linux(Ubuntu) の場合は、インストーラを使ってインストールします。Relases ページから .deb ファイルをダウンロードします[1]。ダウンロードしたディレクトリに移動し、次のようにインストールします。

```
$ sudo apt install ./kinx_1.1.0-0_amd64.deb
```

[1] ファイル名にバージョン番号が含まれます。必要なバージョンをダウンロードしてください

Kinx がインストールできたら、パスが通っているか確認するために一度バージョン表示をしてみましょう。

```
$ kinx -v
kinx version 1.1.0 built on 4b2bd84d86fc4e30a78fffbf84805158dc097193
```

これで準備は完了です。

2.1.2 アンインストール

2.1.2.1 Windows

Windows でアンインストールする場合は、以下のように実施します。

```
$ scoop uninstall kinx
```

これで、Kinx 本体および Kinx パッケージが全て削除されます。また、Bucket も削除したい場合は、以下のようにします。

```
$ scoop bucket rm kinx
```

これで Bucket も削除されます。もう一度登録する場合は、最後 `scoop bucket add` コマンドを実行して登録してください。

2.1.2.2 Linux

Linux でアンインストールする場合は、以下のように実施します。

```
$ sudo apt remove kinx
```

これで、Kinx 本体および Kinx パッケージが全て削除されます。

2.2 Kinx の実行

2.2.1 実行方法

Kinx の実行方法は以下の通りです。

```
$ kinx [options] <kinx-file>.kx
```

標準的な Kinx の拡張子は `.kx` ですが、特に `.kx` であることは意識していません。任意の拡張子のファイルを Kinx スクリプトとして実行できます。

2.2.2 コマンドライン引数

主なコマンドライン引数として、以下が利用可能です。

表 2.1 コマンドライン引数

オプション	意味
-h	ヘルプを表示する。
-c	シンタックスチェックのみ。実際の実行は行わない。
-d	中間表現形式を出力する。実際の実行は行わない。
-D	抽象構文木(AST)を出力する。実際の実行は行わない。
-u	標準入出力での UTF8 変換を行わない。(Windows のみ)
-q	コンパイル時の各種出力を行わない。
-i	ソースを標準入力から入力する。
-v	簡易バージョンを出力する。
--version	詳細バージョンを出力する。
--debug	デバッガー・モードで実行する。
--dot	中間表現を .dot 形式で出力する。
--with-native	-d と一緒に使用し、native 関数のアセンブリコードを出力する。

デバッガー・モードで起動するとデバッガが起動します。デバッガに関しては「10.3 デバッガ」を参照してください。また、--with-native オプションに関しては「6.1.2 native 関数」で一部説明しています。

2.2.3 hello, world.

では誰もが最初に書くスクリプト、hello, world. を書いてみましょう。以下の内容を記載し、hello.kx というファイル名で保存します。

```
1  System.println("hello, world.");
```

早速実行してみましょう。

```
$ kinx hello.kx
hello, world.
```

このように、Kinx はスクリプト言語ですので複雑で面倒なセットアップ・コードなどは一切不要です。自分のしたいことにフォーカスし、すぐに実行して試すことができます。

2.3 Kinx ツアー

Kinx の全体像を知るために、まずは簡単な Kinx の機能紹介をしていきましょう。各セクションでは、それぞれの機能の解説ページへのリンクを記載しています。機能詳細を確認したい場合は、それぞれのリンク先を確認してみてください。

2.3.1 プログラム構造

プログラムはトップレベルから記述可能です。先の `hello, world.` プログラムのように、プログラムの先頭から直接プログラム・コードを記述できます。

```
1  System.println("hello, world.");
```

2.3.2 コメント

コメントは C/C++ 形式と Perl のような # 形式と両方利用可能です。

```
1  /* Comment */
2  // Comment
```

```
1  # Comment
```

好みに応じて使い分けてください。

2.3.3 変数宣言

変数宣言は `var`、または `const` で宣言します。初期化子を使って初期化できます。初期化子を書かなかった場合、初期値は `null` となります。

```
1  var a = 10;
2  const b = 100;
3  var c;
4  System.println([a, b, c]);
```

```
[10, 100, null]
```

`const` を使用した場合、新たなデータの代入はできなくなります（コンパイル・エラー）。

```
1  const b = 100;
2  b = 10;
```

```
Error: Can not assign a value to the 'const' variable near the <test.kx>:2
```

代入には**分割代入**、**パターンマッチ代入**という方法を使うこともできます。宣言文や、関数の引数でも同じスタイルを使えます。

```
[a, b, , ...c] = [1, 2, 3, 4, 5, 6];
{ x, y } = { x: 20, y: { a: 30, b: 300 } };
{ x: d, y: { a: e, b: 300 } } = { x: 20, y: { a: 30, b: 300 } };
System.println([a, b, c, d, e, { x, y }]);
{ x: d, y: { a: e, b: 300 } } = { x: 20, y: { a: 30, b: 3 } };  // Exception occurs.
```

```
[1, 2, [4, 5, 6], 20, 30, {"x":20,"y":{"a":30,"b":300}}]
Uncaught exception: No one catch the exception.
NoMatchingPatternException: Pattern not matched
Stack Trace Information:
        at <main-block>(test.kx:5)
```

これらについては「5.1.3 分割代入とパターンマッチ」で解説しています。

2.3.4 基本データ型

Kinx は動的型付け言語ですが、内部に型を持っています。内部で保持している型は、以下のプロパティでチェック可能です。

表 2.2 基本データ型

プロパティ	例	意味
.isUndefined	null	初期化されていない値。
.isInteger	100, 0x02	整数。演算では自動的に Big Integer と相互変換される。
.isDouble	1.5	実数。
.isString	"aaa", 'bbb'	文字列。
.isBinary	<1,2,3>	バイトの配列。要素は全て 0x00-0xFF に丸められる。
.isArray	[1,a,["aaa"]]	配列。扱える型は全て保持可能。isObject も true。
.isObject	{ a: 1, b: x }	JSON のようなキーバリュー構造。
.isFunction	function(){}	関数。

Kinx において、null と undefined は同じ意味となります。値が null または undefined の場合、.isUndefined が true となります。一方、null、undefined 以外の値の場合、.isDefined が true になります。

また、真偽値リテラルの true、false は単純に整数の 1、0 のエイリアスとなります。Boolean 型はありませんが Boolean クラスは用意されており、そのインスタンスとして真偽を表す True、False という定数が定義されています。式の中で使用すると、True オブジェクトは真（true）とし

て評価され、False オブジェクトは偽（false）と評価されます。整数値とどうしても区別したい場合はそちらを使用してください。

```
System.println(True ? "true" : "false");
System.println(False ? "true" : "false");
```

```
true
false
```

データ型に関しては、「3 データ型」で詳しく説明しています。

2.3.5 配列とオブジェクト

Kinx では内部的に配列とオブジェクトは同じであり、両方の値を同時に保持できます。配列とオブジェクトに関しては、それぞれ「3.3 配列」「3.5 オブジェクト」を参照してください。

```
var a = { a: 100 };
a.b = 1_000;
a["c"] = 10_000;
a[1] = 10;
System.println(a[1]);
System.println(a.a);
System.println(a.b);
System.println(a.c);
```

```
10
100
1000
10000
```

2.3.6 式・文

2.3.6.1 式（エクスプレッション）

「式（エクスプレッション）」とは、**評価され、値を持つもの**を言います。「式」は組み合わせて使用することができ、四則演算、関数呼び出し、オブジェクト操作等が可能です。「式」の評価が最終的に完了すると、式全体が特定の値を示します。また、その値は変数への代入が可能です。

```
// 式
z = 5 + (a * 2) + some(x)
```

「式」の詳細については「4 式と演算子」で説明しています。

2.3.6.2 文（ステートメント）

一方、「文（ステートメント）」は**手続き**を意味し、値を持ちません。プログラムは「文」の集合であり、「文」の中で「式」が使用されます。

また、「文」には「式文」と「ブロック文」があります。「式文」としては、宣言文、代入文、continue文、break文、return文、throw文、yield文が利用可能です。「式文」の終端はセミコロン（;）であり、このセミコロンは常に必要となります。「ブロック文」としては、ブロック({ ... })、if-else、while、do-while、for、for-in、switch-case/when、try-catch-finally が利用可能です。なお、if 文は C 言語と同様にぶら下がり else です。

```
1  // if 文の例
2  if (expression1) {
3      return a;   // 式文
4  } else if (expression2) {
5      return b;
6  } else {
7      return c;
8  }
```

「文」の詳細については「5 文と制御構造」を参照してください。

2.3.7 関数・クロージャ・ファイバー

関数の利用が可能です。関数には通常の関数の他に、native 関数、クロージャ、ラムダ（匿名関数)、クラス（モジュール）・メソッドなどがあります。

2.3.7.1 関数・クロージャ

関数はレキシカル・スコープを持ち、クロージャの利用も可能です。

```
1  function func(x) {          // 通常の関数
2      return &(y) => x + y;   // 匿名関数、x を束縛したクロージャを返す
3  }
4
5  var f = func(10);           // クロージャを受け取る
6  System.println(f(20));      // => 30
```

```
30
```

通常の関数、および native 関数に関しては「6.1 関数」を、クロージャに関しては「6.2 クロージャ」を参照してください。また、クラス・メソッドに関しては「7.2 クラス」、モジュール・メソッドに関しては「7.3 モジュール」をそれぞれ参照してください。

2.3.7.2 ファイバー

クラス Fiber のコンストラクタに関数を渡すことでファイバーも利用可能です。ファイバーでは yield 文を継続のために使用しますが、この yield 文は通常の関数内で使用することはできませ

ん。ファイバーで指定する関数内でだけ yield が使用できます。ファイバー以外で yield を使用した場合は、実行時エラー（例外）が送出されます。

```
var fiber = new Fiber {
    System.println("fiber 1");
    yield;
    System.println("fiber 2");
};

System.println("main 1");
fiber.resume();
System.println("main 2");
fiber.resume();
System.println("main 3");
```

```
main 1
fiber 1
main 2
fiber 2
main 3
```

ファイバーに関しては、「2.3.7.2 ファイバー」で説明しています。

2.3.8 名前空間・クラス・モジュール

名前空間の利用も可能です。また、Kinx はオブジェクト指向言語ですので、クラスの定義が可能です。一方、モジュールはクラス内に mixin することで、複数のクラスに後から共通の機能を追加できる仕組みです。

```
namespace NS {
    module X {
        public sayHello() {
            System.println("hello");
        }
    }
    class A {
        mixin X;
        private trySayHello() {
            @sayHello();
        }
        public doSayHello() {
            trySayHello();
        }
    }
}
var x = new NS.A();
x.sayHello();   // => hello
```

```
hello
```

名前空間に関しては「7.1 名前空間」を参照してください。また、クラスに関しては「7.2 クラス」を、モジュールに関しては「7.3 モジュール」をそれぞれ参照してください。

2.3.9 ライブラリ

Kinx には便利なライブラリがオールインワンで提供されています。これは、良く使う機能は最初から使えるようにしておくことがスクリプト言語の使い勝手を良くすると考えられるためです。Pyhton における「Batteries Included」哲学と似ています。

オールインワンで提供されるライブラリは以下の通りです。

- **Zip** ... ZIPパスワードだけでなく、AESパスワードを使って ZIP/Unzip が可能。
- **Xml** ... XML DOM をサポート。
- **libCurl** ... HTTP のみ実装済です。(もっと色々できるはずですが)
- **SSH** ... SSH ログインしてコマンドを実行することができます。
- **Socket** ... シンプルな TCP/UDP ソケット。
- **Iconv** ... テキスト・エンコーディング変換。
- **Database** ... SQLite3 を利用する便利な Database クラス。
- **Parser Combinator** ... `Parsec` のような `Parsek` という名前のパーサコンビネータ。
- **PDF** ... HaruPDF ベースの PDF コアライブラリ。
- **JIT** ... 抽象化アセンブラ・ライブラリによる、様々なプラットフォームで使用可能な JIT ライブラリ。

標準ライブラリは「8 標準ライブラリ」で解説しています。かなりボリュームがありますが、参考になるでしょう。

2.3.10 Kinx パッケージ・システム

Kinx にはパッケージ・システムが内蔵されています。パッケージ・システムを使うことで標準にはないライブラリを後から追加したり、必要なライブラリのみをインストールしたりすることができます。

パッケージシステムに関しては「9 パッケージ管理」で詳しく解説しています。

3 データ型

3.1 数値

3.1.1 整数

Kinx は整数を内部的に 2 種類の形で扱っています。1つは 64 bit 整数であり、1つは多倍長整数です。基本的に内部構造として 64 bit 整数、多倍長整数の区別はありますが、これらは自動的に相互変換されて扱われるためユーザー（プログラマー）が特に意識する必要はありません。

3.1.1.1 64 bit 整数

Kinx では基本的に整数は 64 bit 整数です。整数リテラルを書く際は、8進数、10 進数、16 進数で記述できます。表現形式は C 言語と同様で「表 3.1 整数表記ルール」のルールに従います。

表 3.1 整数表記ルール

整数	表記ルール
8進数	`0` で始まる数値。
10進数	`1`〜 `9` で始まる数値。
16進数	`0x` で始まる数値。

値が整数の場合、`.isInteger` が true になります。

```
1  var a = 0x10;
2  System.println([10, a, 010]);
3  System.println(a.isInteger ? "true" : "false");
```

```
[10, 16, 8]
true
```

3.1.1.2 多倍長整数

値が 64 bit の範囲を超えると、自動的に多倍長整数で扱うようになります。多倍長整数は基本

的にどんな数でも扱え、非常に大きい値を扱うことができます。ただし、多倍長整数をリテラルで記述する際は 10 進数のみが利用可能です。

```
var n = 9223372036854775808;    // 多倍長整数
System.println("%d.isBigInteger = %s" % n % (n.isBigInteger ? "true" : "false"));
System.println("%d x 2 = %d" % n % (n * 2));
```

```
9223372036854775808.isBigInteger = true
9223372036854775808 x 2 = 18446744073709551616
```

値が多倍長整数の場合、.isBigInteger が true になります。この時 .isInteger も true となりますので、多倍長整数では .isInteger と .isBigInteger の両方が true となります。

3.1.1.3 64 bit 整数と多倍長整数の自動変換

64 bit 整数として扱われている整数の値の範囲が 64 bit の範囲を超えると、自動的に多倍長整数に拡張されます。逆に、値が 64 bit の範囲に戻ってきた際には自動的に 64 bit 整数として扱われます。この変換は自動的に行われるため、プログラマーが意識する必要はありません。

多倍長整数と 64 bit 整数の境目を以下のプログラムで確認してみましょう。まず、64 bit 整数の値の範囲は -9223372036854775808 から 9223372036854775807 までとなります。そこで、9223372036854775806 あたりからプラスの方向に増やし、その後、順に元に戻してみます。

```
function disp(n) {
    System.println("%d = %10s %13s"
        % n
        % (n.isInteger ? ".isInteger" : "")
        % (n.isBigInteger ? ".isBigInteger" : "")
    );
}
var n = 9223372036854775806;
for (var i = 0; i < 4; ++i, ++n) {
    disp(n);
}
for (var i = 0; i < 4; ++i, --n) {
    disp(n);
}
```

```
9223372036854775806 = .isInteger
9223372036854775807 = .isInteger
9223372036854775808 = .isInteger .isBigInteger
9223372036854775809 = .isInteger .isBigInteger
9223372036854775810 = .isInteger .isBigInteger
9223372036854775809 = .isInteger .isBigInteger
9223372036854775808 = .isInteger .isBigInteger
9223372036854775807 = .isInteger
```

64 bit 整数と多倍長整数の相互変換が自動的にされているのが分かります。

3.1.1.4 多倍長整数の例

最後に多倍長整数の例として、階乗の計算をしてみましょう。

```
1  function fact(n) {
2      if (n < 1) return 1;
3      return n * fact(n-1);
4  }
5  System.println(fact(500));
```

```
1220136825991110068701238785423046926253574342803192842192413588385845373153881
9760549644750220328186301361647714820358416337872207817720048078520515932928547
7907571939330603772960859086270429174547882424912726344305670173270769461062802
3104526442188787894657547771498634943677810376442740338273653974713864778784954
3848959553753799042324106127132698432774571554630997720278101456108118837370953
1016356324432987029563896628911658974769572087926928871281780070265174507768410
7196243903943225364226052349458501299185715012487069615681416253590566934238130
0885624924689156412677565448188650659384795177536089400574523894033579847636394
4905313062323749066445048824665075946735862074637925184200459369692981022263971
9525971909452178233317569345815085523328207628200234026269078983424517120062077
1464097945611612762914595123722991334016955236385094288559201872743379517301458
6357570828355780158735432768888680120399882384702151467605445407663535984174430
4801289383138968816394874696588175045069263653381750554781286400000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000
```

多倍長整数の演算であっても、この程度の演算であれば体感的に一瞬で完了します[1]。

3.1.2 実数

実数は C 言語でいう double を示します。実数リテラルの書き方としては通常の実数表記のみサポートしており、指数表記には現在対応していません[2]。

値が実数の場合は、`.isDouble` が true になります。また、整数同士の演算であっても、演算結果が実数となる場合は自動的に実数に変換されます。

```
1  var d = 0.5;    // 実数リテラル
2  function eval(f) {
3      r = f(3, 2);
4      System.println("r = %3s, r.isDouble = %s" % r % (r.isDouble ? "true" : "false"));
5  }
6  eval({ => _1 + _2 });
7  eval({ => _1 - _2 });
8  eval({ => _1 * _2 });
9  eval({ => _1 / _2 });
```

```
r =   5, r.isDouble = false
r =   1, r.isDouble = false
r =   6, r.isDouble = false
r = 1.5, r.isDouble = true
```

[1] 5000! の計算でも一瞬でしたが、結果が長すぎて紙面に入りきらなかったので 500! のサンプルにしました。
[2] このくらい対応したほうが良いか。。。

3.2 文字列

文字列が扱いやすくなっているのもスクリプト言語の特徴の一つです。C 言語ではメモリリークやバッファー・オーバーランといったバグを埋め込みやすく、文字列は、良く利用する割に非常に使いづらいものです。そこで、Perl 時代より文字列の扱いが容易だということが、スクリプト言語が人気となった大きな理由でもありました。

Kinx でも同様に文字列を容易に扱えます。

3.2.1 文字列リテラル

文字列のリテラルは、ダブルクォート、またはシングルクォートで囲みます。意味は同じですが、ダブルクォート内ではダブルクォートを、シングルクォート内ではシングルクォートをエスケープする必要があります。

```
1  var a = "\"aaa\", 'bbb'";
2  var b = '"aaa", \'bbb\'';
3  System.println(a == b ? "same" : "different");
```

```
same
```

3.2.2 文字列の演算

文字列の四則演算は、文字列に合わせた動作をします。まず、加算と乗算は以下のように動作します。加算は単純な文字列の連結、乗算は文字列の繰り返し（の連結）です。

```
1  var a = "123456789,";
2  System.println(a + a + a);
3  System.println(a * 4);
```

```
123456789,123456789,123456789,
123456789,123456789,123456789,123456789,
```

文字列に除算を適用すると、文字列を「/」で区切ったパスとして連結します。この時、重複する「/」は一つにまとめられます。

```
1  var a = "path/to";
2  System.println(a / "file.txt");
3  System.println(a / "/file.txt");
```

```
path/to/file.txt
path/to/file.txt
```

また、剰余演算を適用した場合、フォーマッタ・オブジェクトを返します。

```
System.println("0x%02x" % 10);
```

```
0x0a
```

フォーマッタ・オブジェクトは、C 言語の printf のようなフォーマットを行うためのオブジェクトです。詳しくは「3.2.9 フォーマッティング」を参照ください。

3.2.3 添え字アクセス

文字列を添え字でアクセスをした場合、その位置にある文字コードを整数値で返します。したがって、「5 文字目が 'a' である」という判断をする場合は以下のように書きます。

```
if (str[5] == 'a'[0]) {
    /* ... */
}
```

右辺でも添え字アクセスをしていることに注意してください。文字 'a' は実際には文字ではなく**文字列リテラル**のため、最初の文字を示す [0] が必要となります。

また、str[-1] と負の数を指定することによって文字列の末尾からアクセスすることもできます。この場合の str[-1] は最後の文字の文字コードを返します。

3.2.4 =~ 演算子

文字列に対して =~ を適用した場合、右辺値の正規表現にマッチするかどうかを確認します。右辺値が正規表現のオブジェクトではなかった場合、例外が送出されます。復帰値はマッチしたグループの集合となり、マッチしなかった場合は False オブジェクトが返ります。

```
if (g = ("abc" =~ /(.)(bc)/)) {
    System.println(g.toJsonString(true));
}
```

```
[{
    "begin": 0,
    "end": 3,
    "string": "abc"
}, {
    "begin": 0,
    "end": 1,
    "string": "a"
}, {
    "begin": 1,
    "end": 3,
    "string": "bc"
}]
```

なお、左辺値が正規表現オブジェクトで右辺値が文字列の場合でも同様の動作を行います。

3.2.5 !~ 演算子

文字列に対して !~ を適用した場合は、右辺値の正規表現にマッチ**しないこと**を確認します。右辺値が正規表現のオブジェクトではなかった場合、例外が送出されます。復帰値は true または false になります。true の場合が、マッチしなかった場合です。

```
1  if (g = ("axc" !~ /(.)(bc)/)) {
2      System.println(g);  // => 1
3  }
```

こちらのケースでも同様に、左辺値と右辺値が逆でも同様の動作を行います。

3.2.6 単項 * 演算子

単項 * 演算子を文字列に適用した場合、文字列を配列に変換します。また、逆に配列に単項 * 演算子を適用すると、文字列に戻ってきます。

```
1  var a = *"abc";         // => [97, 98, 99]
2  var b = *[97, 98, 99];  // => "abc"
```

このように、いくつかの型同士での相互変換が可能です。詳しくは「3.8 データ型の相互変換」をご参照ください。

3.2.7 Raw 文字列

%{...} という形式で文字列を記載することで Raw 文字列を作成することが可能です[3]。%-{...} を使うと、先頭と末尾の改行文字をトリミングします。また、%<...>、%(...)、%[...] を使うこともできます。

この表記方法を使うと、複数行にまたがる文字列を直接扱うことができます。以下の例のように記載できます。

```
1  var a = 100;
2  var b = 10;
3  var str = %{
4  This is a string without escaping control characters.
5  New line is available in this area.
6  { and } can be nested here.
7  };
8  System.println(str);
9  var str = %-{
10 This is a string without escaping control characters.
11 New line is available in this area.
12 But newlines at the beginning and the end are removed when starting with '%-'.
13 };
14 System.println(str);
```

[3] ヒアドキュメントのように扱えるため、Kinx ではヒアドキュメントをサポートしていません。

後述する内部式を使う場合の「%」、およびネストした形にならない「{」や「}」に対して「\」でエスケープする必要があります。

また、閉じカッコは対応する開きカッコに対応するものを使用しますが、以下の文字を使うことも可能です。その場合は、開始文字と終了文字は同じ文字となります。例えば、%|...| のような形で使用します。

- “|”、“!”、“^”、“~”、“_”、“.”、“,”、“+”、“*”、“@”、“&”、“$”、“:”、“;”、“?”、“'”、“"”

3.2.8 内部式

文字列の内部で %{...} 形式を使うと、文字列内部に直接式を記述することができます。

```
1  for (var i = 0; i < 5; ++i) {
2      System.println("i = %{i}, i * 2 = %{i * 2}");
3  }
4  // i = 0, i * 2 = 0
5  // i = 1, i * 2 = 2
6  // i = 2, i * 2 = 4
7  // i = 3, i * 2 = 6
8  // i = 4, i * 2 = 8
```

3.2.9 フォーマッティング

文字列に対する % 演算子は、フォーマッタ・オブジェクトを作成します。

```
1  var fmt = "This is %1%, I can do %2%.";
2  System.println(fmt % "Tom" % "cooking");
```

%1% の 1 はプレースホルダ番号を示し、% 演算子で適用した順に合わせて整形します。適用場所が順序通りであれば、C の printf と同様の指定の仕方も可能です。また、さらに C の printf と同じ指定子を使いながら同時にプレースホルダも指定したい場合、$ の前に位置指定子を書き、$ で区切って記述する事もできます。

例えば、16進数で表示したい場合は以下のようにします。

```
1  var fmt = "This is %2%, I am 0x%1$02x years old in hex.";
2  System.println(fmt % 27 % "John");
3  // This is John, I am 0x1b years old in hex.
```

フォーマッタ・オブジェクトに後から値を適用していく場合は、%= 演算子を使って適用していきます。

```
1  var fmt = "This is %1%, I can do %2%.";
2  fmt %= "Tom";
3  fmt %= "cooking";
4  System.println(fmt);
```

実際のフォーマット処理は、以下のタイミングで行われます。

- `System.println` 等で表示するとき。
- 文字列との加算が行われるとき。

明示的にフォーマッタ・オブジェクトからフォーマット文字列を作成するには、`format()` メソッドを使います。

```
1  var fmt = "This is %1%, I can do %2%.";
2  fmt %= "Tom";
3  fmt %= "cooking";
4  setString(fmt.format());
```

3.2.10 文字色・文字装飾

コンソール端末に文字列を表示する際、エスケープシーケンスに則った形で色や装飾を付けることが可能です。

Kinx ではこれを実現するために、文字列に対して直接メソッドが定義されています[4]。指定できるのは、前景色、背景色、および文字装飾です。

3.2.10.1 前景色・背景色

以下のような形で指定すると、前景色を赤色に、背景色を黄色に設定します。

```
1  System.println("The text".red(.yellow));
```

背景色は、前景色を指定するメソッドの引数として指定します。`.yellow` といった形で、`.` で始まることに注意してください。

文字色として指定できるものを「表 3.2 文字色」に示します。

表 3.2 文字色

色	前景色	背景色
黒	`.black()`	`.black`
赤	`.red()`	`.red`
緑	`.green()`	`.green`
黄色	`.yellow()`	`.yellow`
青	`.blue()`	`.blue`
マゼンタ	`.magenta()`	`.magenta`
シアン	`.cyan()`	`.cyan`
白	`.white()`	`.white`

[4] これも「特殊オブジェクト」によって実現されています。特殊オブジェクトに関する詳細は、「3.9 特殊オブジェクト」をご参照ください。

3.2.10.2 文字装飾

文字装飾も文字色と同様に指定します。

```
System.println("The text".red(.white).bold().underline());
```

このように、文字色と文字装飾は同時に指定できます。

文字装飾として指定できるのは、「表 3.3 文字装飾」に示す 5 種類です。ただし、端末やフォントによって表現されないケースがあることにご注意ください[5]。

表 3.3 文字装飾

装飾	メソッド	
ボールド	.bold()	太字にする。
下線	.underline()	下線を付ける。
イタリック	.italic()	斜体にする。
リバース	.reverse()	色を反転させる。
ブリンク	.blink()	点滅させる。

3.3 配列

配列は、複数のデータを格納するための入れ物であり、リストです。配列を使うことで、複数のデータを一括して扱うことができるようになります。配列の要素として配列を格納することも可能です。これによって多次元配列を扱うこともできます。

3.3.1 配列リテラル

配列のリテラルは、[...] の形で記述し、任意のデータを混在させて格納することができます。また、最後の要素の後にカンマ（,）を書くこともできます。これは C 言語同様に要素を縦に並べた際に書きやすくするためのものです。最後の要素の後にカンマが置けると、ソースコード上での並べ替えや要素の追加が容易になります。

```
var a = [1, 2, 3, 4, 5];
var b = [
    "item1",
    "item2",
    "item3",
    "item4",
];
var c = [
    1, 2, 3,
    "string",
    [ a, b, ],
];
```

[5] Windows では italic と blink は機能しないようです。

3.3.2 添え字アクセス

配列は添え字でアクセスできます。

```
var a = [1, 2, 3];
var b = [a, 1, 2];
System.println(b[0][2]);    // 3
System.println(a[-1]);      // 3
```

上記例のように、配列の要素として配列を格納した場合、最初の添え字で中身となる配列要素を取り出し、次の添え字でその配列の要素を取り出すことができます。つまり、単純に添え字を重ねることで配列内配列の要素に簡単にアクセスすることができます。

また、添え字に負の数を与えると末尾からアクセスするように動作します。a[-1] は配列 a の最後の要素にアクセスします。

3.3.3 配列の代入

配列構造は左辺値で使用すると右辺値の配列を個々の変数に取り込むことが可能です。これを使用して値をスワップすることも可能です。

```
[a, b] = [b, a];    // Swap
```

スプレッド・レスト演算子を使った分割代入も可能です。

```
[a, ...b] = [1, 2, 3, 4, 5];
// a = 1
// b = [2, 3, 4, 5]
```

これは宣言でも利用でき、宣言と同時に以下の書き方をすることもできます。

```
var a = 3, b = [4], x = 3, y = [4];
{
    var [a, ...b] = [1, 2, 3, 4, 5];
    // a = 1
    // b = [2, 3, 4, 5]
    [x, ...y] = [1, 2, 3, 4, 5];
    // x = 1
    // y = [2, 3, 4, 5]
    [z] = [1, 2, 3, 4, 5];
    // okay z = 1, but scoped out...
}
System.println("a = ", a);      // 3
System.println("b = ", b[0]);   // 4
System.println("x = ", x);      // 1
System.println("y = ", y[0]);   // 2
```

分割代入の詳細は「4.1.3 分割代入」を参照してください。

3.4 バイナリ

バイナリはバイト配列です。全ての要素は 0x00〜 0xFF の範囲にアジャストされ、配列のようにアクセス可能です。

3.4.1 バイナリ・リテラル

バイナリ・リテラルは <...> の形式で記述します。また、バイナリと配列は相互にスプレッド演算子で分割、結合することが可能です。

```
1  var bin = <0x01, 0x02, 0x03, 0x04>;
2  var ary = [...bin];
3      // ary := [1, 2, 3, 4]
4
5  var ary = [10, 11, 12, 257];
6  var bin = <...ary>;
7      // bin := <0x0a, 0x0b, 0x0c, 0x01>
```

ただし、バイナリになった瞬間に 0x00-0xFF に丸められるのでご注意ください。

3.4.2 添え字アクセス

バイナリも配列同様に添え字でアクセスできます。また、同様に添え字に負の数を指定すると末尾からアクセスするように動作します。

```
1  var a = <1, 2, 3>;
2  System.println(a[1]);       // 2
3  System.println(a[-1]);      // 3
```

3.5 オブジェクト

3.5.1 オブジェクト・リテラル

3.6 正規表現

3.6.1 正規表現リテラル

正規表現リテラルは /.../ の形式で扱います。また、リテラル内の「/」は「\」でエスケープする必要があります。以下に例を示します。

```
1  var a = "111/aaa/bbb/ccc/ddd";
2  while (group = (a =~ /\w+\//)) {
3      for (var i = 0, len = group.length(); i < len; ++i) {
4          System.println("found[%2d,%2d) = %s"
5              % group[i].begin
6              % group[i].end
7              % group[i].string);
8      }
9  }
```

/ を多用するような正規表現の場合、%m プレフィックスを付け、別のクォート文字を使うことで回避することもできます。例えば %m(...) といった記述が可能です。これを使って上記を書き直すと、以下のようになります。

```
1  var a = "111/aaa/bbb/ccc/ddd";
2  while (group = (a =~ %m(\w+/))) {
3      for (var i = 0, len = group.length(); i < len; ++i) {
4          System.println("found[%2d,%2d) = %s"
5              % group[i].begin
6              % group[i].end
7              % group[i].string);
8      }
9  }
```

なお、正規表現リテラルを while 等の条件式に入れることができますが、1 点だけ注意点があります。

例えば以下のように記述した場合、str の文字列に対してマッチしなくなるまでループを回すことができます（この時、group にはキャプチャ一覧が入ります）。その際、最後のマッチまで実行せずに途中で break 等でループを抜けると正規表現リテラルの対象文字列が次回のループで正しくリセットされない、という状況が発生します。

```
1  while (group = (str =~ /ab+/)) {
2      /* block */
3  }
```

正規表現リテラルがリセットされるタイミングは以下の 2 パターンです。

- 初回（前回のマッチが失敗して再度式が評価された場合を含む）。
- str の内容が変化した場合。

現時点で本動作は仕様です。扱う際にはご注意願います。

3.6.2 基本的な使い方

3.7 範囲表現

3.7.1 Range クラスと範囲リテラル

3.7.2 Enumerable クラス

3.7.3 基本的な使い方

3.8 データ型の相互変換

3.9 特殊オブジェクト

特殊メソッドを持つオブジェクトを特殊オブジェクトと呼び、String、Integer、Double、Binary、Array があります。特殊メソッドは、特殊オブジェクトが対象としているオブジェクト（String なら文字列）に対して直接作用させることができます。ここではそれらの特殊オブジェクトに関して説明します。

ただし、以下の観点から乱用はお勧めしません。

- **ライブラリの追加（組み込み特殊メソッドの追加）で使う可能性がある**こと。
- **標準ライブラリの中での挙動が変わり、正しく動作しなくなる可能性がある**こと。

ここでは「仕組みとしての特殊オブジェクト」という観点で、説明します。

3.9.1 特殊オブジェクトの例

例えば、特殊オブジェクト String に対して以下のように関数定義してみましょう。

```
1  String.greeting = function(name) {
2      System.println("Hello, I am %{name}.");
3  };
```

すると、以下のように書くことができるようになります。

```
1  "John".greeting();
```

実行してみると、以下のように出力されます。

```
1  Hello, I am John.
```

対象となる "John" が String.greeting 関数の name に引き渡されたのが分かるでしょう。つまり、対象オブジェクトが特殊オブジェクトの第一引数として渡されるといった動作をします。この仕組みを使って、文字列に対する操作関数を自由に定義することができます。

特殊オブジェクトに定義されたメソッドを特殊メソッドと呼び、「C 特殊メソッド」に良く使う特殊メソッドをまとめましたので、ご参照ください。

4
式と演算子

4.1 演算子

4.1.1 演算子と優先順位

表 4.1 演算子の優先順位

#	要素	演算子例
1	要素	変数, 数値, 文字列, ...
2	後置演算子	++, --, [], .property, ()
3	前置演算子	!, +, -, ++, --
4	パターンマッチ	=~, !~
5	べき乗	**
6	乗除	*, /, %
7	加減	+, -
8	ビットシフト	<<, >>
9	大小比較	<, >, >=, <=
10	等値比較	==, !=
11	ビットAND	&
12	ビットXOR	^
13	ビットOR	|
14	論理AND	&&
15	論理OR	||
16	三項演算子	? : , function(){}
17	代入演算子	=, +=, -=, *=. /=. %=, &=, |=, ^=, &&=, ||=

4.1.2 短絡演算子

4.1.3 3項演算子

4.1.4 case-when

4.1.5 パイプライン演算子

4.1.6 関数合成演算子

A 中間表現

A.1 Kinx 中間表現一覧

B

Kinx リソース

C 特殊メソッド

C.1 Integer 特殊メソッド

表 C.1 特殊メソッド（整数）

メソッド	意味
Integer.times(val, f)	i = 0 〜 (val - 1) の範囲として、f があれば f(i) の結果で、無ければ i で配列を作成して返す。
Integer.upto(val, max, f)	i = val 〜 max の範囲で引数として f(i) を呼ぶ。
Integer.downto(val, min, f)	i = min 〜 val の範囲で引数として f(i) を呼ぶ。
Integer.toString(val, rdx)	val を文字列に変換する。rdx は基数を表す。
Integer.toDouble(val)	val を Double に変換する。

64bit 整数と多倍長整数はシームレスに相互変換されますので、Integer 特殊オブジェクトは多倍長整数にも適用されます。ただし多倍長整数にループ系のメソッドが適用された場合、ループ回数が極端に大きくなりますのでご注意ください。

C.2 Double 特殊メソッド

表 C.2 特殊メソッド（実数）

メソッド	意味
Double.toString(val, fmt)	val を文字列に変換する。fmt は % で始まり、a、A、e、E、f、F、g、G のいずれかで終わる文字列。省略時は %g
Double.toInt(val)	val を Integer に変換する。

実数に関する特殊メソッドは、基本的に型変換しか用意されていません。

C.3 String 特殊メソッド

表 C.3 特殊メソッド（文字列）

メソッド	意味
String.startsWith(str, s)	文字列が s で始まれば true。

メソッド	意味
String.endsWith(str, s)	文字列が s で終れば true。
String.toUpper(str, [s, e])	str の s 文字目（0〜）から e 文字目の前まで大文字化する。引数省略時は全てを大文字化する。
String.toLower(str, [s, e])	str の s 文字目（0〜）から e 文字目の前まで小文字化する。引数省略時は全てを小文字化する。
String.trim(str, [c])	str の先頭と末尾から c で指定された文字を削除する。引数省略時は空白文字（空白、タブ、改行）を削除する。
String.trimLeft(str, [c])	str の先頭から c で指定された文字を削除する。引数省略時は空白文字（空白、タブ、改行）を削除する。
String.trimRight(str, [c])	str の末尾から c で指定された文字を削除する。引数省略時は空白文字（空白、タブ、改行）を削除する。
String.find(str, s)	文字列の中で s が見つかった位置（0〜）を返す。見つからなかった場合は -1 を返す。
String.subString(str, s[, l])	str の s 文字目（0〜）から l 文字分の部分文字列を返す。
String.replace(str, c, r)	str から c にマッチする部分を全て r に置換する。c は文字列または正規表現オブジェクトが指定可能。
String.toInt(str)	str を整数値に変換する。
String.toDouble(str)	str を実数値に変換する。
String.parentPath(str)	str をパスと認識し、親パスとなる部分文字列を返す。例）"ab/cd/ef.x".parentPath() → "ab/cd"
String.filename(str)	str をパスと認識し、親パス部分を削除したファイル名部分文字列を返す。例）"ab/cd/ef.x".filename() → "ef.x"
String.stem(str)	str をパスと認識し、ファイル名の stem 部分文字列を返す。例）"ab/cd/ef.x".stem() → "ef"
String.extnsion(str)	str をパスと認識し、ファイル名の拡張子部分文字列を返す。例）"ab/cd/ef.x".extnsion() → ".x"
String.split(str, sep)	str を sep を区切り文字として分割し、配列として返す。sep は文字列、または正規表現オブジェクトが指定可能。

メソッド	意味
String.each(str, f)	str を 1 文字ずつ分割し、それを引数にして f 関数を呼び出す。f 関数の第 2 引数に index (0〜) も渡される。

C.4 Array 特殊メソッド

表 C.4 特殊メソッド（配列）

メソッド	意味
Array.length(ary)	配列 ary の要素数を返す。
Array.keySet(obj)	オブジェクト obj のキー一覧を配列として返す。
Array.push(ary, e)	配列 ary の最後の要素として e を追加する。
Array.pop(ary)	配列 ary の最後の要素を返し、ary から削除する。
Array.unshift(ary, e)	配列 ary の最初の要素として e を追加する。
Array.shift(ary)	配列 ary の最初の要素を返し、ary から削除する。後ろの要素は順次先頭方向に移動する。
Array.join(ary, sep)	配列 ary の要素を全てつなげた文字列として返す。要素と要素の間は sep でつなぐ。
Array.reverse(ary)	配列 ary の要素を逆順にした新たな配列を返す。
Array.flatten(ary)	配列 ary の要素をフラットにした形で新たな配列として返す。
Array.toString(ary)	配列 ary を文字列化して返す。デフォルトのセパレータは ", "
Array.toJsonString(ary, idt)	配列 ary を JSON 文字列化して返す。idt を true とすると、インデントした形で整形する。
Array.apply(ary, f)	f(ary) を実行する。
Array.each(ary, f)	ary の各要素 e に対して f(e, index) を実行する。
Array.map(ary, f)	ary の各要素 e に対して f(e, index) の結果の集合となる新たな配列を返す。
Array.flatMap(ary, f)	map した新たな配列をフラットにして返す。
Array.filter(ary, f)	ary の各要素 e に対して f(e, index) の結果が真となるもののみの集合として新たな配列を返す。
Array.reject(ary, f)	ary の各要素 e に対して f(e, index) の結果が偽となるもののみの集合として新たな配列を返す。

メソッド	意味
Array.reduce(ary, f, i)	前の要素までの結果を r（初期値は i、デフォルト null）として、ary の各要素 e に対し f(r, e) を行った最終結果を返す。
Array.sort(ary, comp)	任意の ary の要素 e1, e2 に対する comp(e1, e2) の結果を利用して ary 全体をソートした新たな配列を返す。
Array.clone(obj)	obj のディープ・コピーを作成する。
Array.extend(obj, obj2)	obj の内容を obj2 の内容で拡張する。同じキーがあれば上書きする。
Array.println(ary)	ary の各要素 e に対して System.println(e) を適用する。

push、pop、unshift、shift は破壊的な操作を行います。Array の特殊メソッドはオブジェクトにも適用されることに注意してください。使い分ける場合は呼び出された後に区別します。詳しくは「2.3.5 配列とオブジェクト」をご参照ください。

C.5 Binary 特殊メソッド

表 C.5 特殊メソッド（バイナリ）

メソッド	意味
Binary.length(bin)	バイナリ bin の要素数を返す。
Binary.push(bin, e)	バイナリ bin の最後の要素として e を追加する。
Binary.pop(bin)	バイナリ bin の最後の要素を返し、bin から削除する。
Binary.unshift(bin, e)	バイナリ bin の最初の要素として e を追加する。
Binary.shift(bin)	バイナリ bin の最初の要素を返し、bin から削除する。後ろの要素は順次先頭方向に移動する。
Binary.join(bin, sep, fmt)	バイナリ bin の要素を全てつなげた文字列として返す。要素と要素の間は sep でつなぐ。fmt が省略された場合 "0x02x" でフォーマットされる。
Binary.reverse(bin)	バイナリ bin の要素を逆順にした新たなバイナリを返す。
Binary.toString(bin, sep, fmt)	バイナリ bin を文字列化して返す。デフォルトのセパレータは ", "。fmt が省略された場合 "0x02x" でフォーマットされる。
Binary.apply(bin, f)	f(bin) を実行する。
Binary.each(bin, f)	bin の各要素 e に対して f(e, index) を実行する。

メソッド	意味
Binary.map(bin, f)	bin の各要素 e に対して f(e, index) の結果の集合となる新たなバイナリを返す。
Binary.filter(bin, f)	bin の各要素 e に対して f(e, index) の結果が真となるもののみの集合として新たなバイナリを返す。
Binary.reject(bin, f)	bin の各要素 e に対して f(e, index) の結果が偽となるもののみの集合として新たなバイナリを返す。
Binary.reduce(bin, f, itr)	前の要素までの結果を r（初期値は itr、デフォルト null）として、bin の各要素 e に対し f(r, e) を行った最終結果を返す。
Binary.sort(bin, c)	任意の bin の要素 e1, e2 に対する c(e1, e2) の結果を利用して bin 全体をソートした新たなバイナリを返す。
Binary.clone(bin)	bin のコピーを作成する。
Binary.println(bin)	bin の各要素 e に対して System.println(e) を適用する。

基本的にバイナリ列に適用することになるため、あまり複雑なことはできません。Array と同様に push、pop、unshift、shift は破壊的な操作を行います。push、pop、unshift、shift は破壊的な操作を行います。

また、バイナリに対するメソッドは、配列と比較してネイティブ・コードで最適化されているケースがいくつかあります。例えば、Array.sort(ary, compfunc) と Binary.sort(bin, compfunc) はクイックソートですが、Binary のほうは compfunc が省略された場合に stdlib.h の qsort を使うように最適化されています。また、Binary.reverse() も C ルーチンで実装されています。つまり、Binary に対しての降順ソートは bin.sort(&(a, b) => b <=> a) よりも bin.sort().reverse() としたほうが速い結果が出ます。

```
 1  function test(name, f) {
 2      var tmr = new SystemTimer();
 3      var sorted = f();
 4      var disp = [sorted[0], sorted[1], "...", sorted[-1]];
 5      System.println("%{name}%{disp} => ", tmr.elapsed());
 6  }
 7
 8  var a = 10000.times(&() => Integer.parseInt(Math.random() * 100));
 9  var bin = <...a>;
10  test("sort1", &() => bin.sort(&(a, b) => b <=> a));
11  test("sort2", &() => bin.sort().reverse());
```

上記プログラムで実測してみた結果が以下となります。

```
sort1[99, 99, "...", 0] => 0.045096
sort2[99, 99, "...", 0] => 0.000582
```

100倍くらい速くなりました。ただし、ここまで差が出るケースはまれですので、普段はあまり意識する必要はありません。