Konoha Tutorial

1 Introduction

このチュートリアルではスクリプティング言語、Konoha の使い方の説明を行います。このチュートリアルでは実際にプログラミングの経験がある人、そして Konoha の機能の概要が知りたい人を対象にしています。静的型付けで評価を行う Konoha は、Java ライクの文法で Python のように動きます。このチュートリアルは Konoha のインストールからオブジェクト志向プログラミングまでを説明します。

2 はじめての Konoha

本章では，Konoha のインストール方法の解説と，インタラクティブモードでの ”Hello, world” の実行までをご紹介します.

2.1 Konoha のインストール

Konoha が動作する環境や配布している形式は様々ですが，ここでは Windows のバイナリパッケージからのインストール方法を紹介します．他の OS や，ソースからのビルドを行いたい方は，以下の公式 HP の Konoha インストールガイドをご覧ください.

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

本稿執筆時点で 2010/10/20 が最新更新日です.

ダウンロードした実行ファイルを起動します．インストーラが立ち上がりますので，インストーラの指示する手順に従ってインストールを始めましょう.

2.2 Konoha の起動

この章では Konoha の起動方法として Konoha 対話モードとスクリプトモードの説明をします。

2.2.1 対話モードでの起動

Konoha インストールが完了したら，Konoha が正しく動作するかを確認してみましょう.

Konoha はコマンドプロンプトから利用します.

コマンドで ”konoha” と入力して実行してみてください．正しくインストールできていれば、下のような Konoha の対話型シェルが起動します.

```
$ konoha[Enter]
Konoha 0.7(aomori) winmsi (rev:1927, Oct 20 2010 18:20:20)
[Microsoft Visual C++ 9.0 (1500)] on windows_32 (32, CP932)
Options: rcgc used_memory:191 kb

>>>
```

対話モードでは，入力した式が Konoha によって評価され，その結果を表示することができます．例えば以下のように入力すると，画面に ”Hello, world” と表示されます.

```
>>> OUT.println("Hello, world"); // like Java
Hello, world
>>> OUT << ``Hello, world'' << EOL // like C++
Hello, world
```

また，対話モードでは，変数宣言や，関数・クラス定義や実行など，Konoha スクリプトで使える機能を一通り試すことができます．(詳しくは後述します)

```
>>> void hello() {
... OUT << "Hi!" << EOL;
... }
>>> hello();
Hi!
```

終了したい場合には，”exit”，”quit” と入力して評価するか，キーボードで Ctrl-D を押してください.

```
>>> exit
```

自作したスクリプトを試したいときや Konoha の機能をテストしてみたいときには，この対話モードを活用しましょう.

2.2.2 スクリプトによる起動

このチュートリアルでは対話モードを中心に説明をしますが、Konoha はもちろん記述したスクリプトを読み込むことによる実行も可能です。この場合、“hallo.k” の様に.k 拡張子を用いて保存し、実行の際に引数としてファイル指定を行いますが、文法は対話モードのものと全く同じであるため今回は割愛させていただきます。

3 Konoha のオペランド

この章では Konoha の代表的なオペランドに関する基本的な機能を説明します。対話モードは，式を入力すれば答えが得られるので，Konoha を電卓のように使うことができます．電卓から文字列処理まで行える Konoha の便利な対話モードで Konoha に慣れましょう。

3.1 数値

演算子 +，−，∗，/ は，他のプログラミング言語と同じように動作します．また括弧を使い，グルーピングすることができます．変数への代入は，= 演算子を用います．

```
>>> 2 + 2
4
>>> // This is a comment
>>> 2 + 2 // On the same line as code
4
>>> (50 - 5 * 6) / 4
5
>>> 7 / 3 // Returns the floor
2
>>> int width = 20
20
>>> int height = 5 * 9
45
>>> width * height
900
```

数値は，整数と浮動小数点数があります．これらは，数値リテラルとして，小数点以下の数値の有無で区別されます．明示的な型変換には，型キャスト演算子を用います．整数と浮動小数点数が混在するときは，整数が自動的に浮動小数点数に変換されます．

```
>>> typeof(3)
konoha.Int
>>> typeof(3.0)
konoha.Float
>>> (float)3
3.000000
>>> (int)3.5
3
>>> typeof(3)
konoha.Int
>>> typeof(3.0)
konoha.Float
>>> (float)3
3.000000
```

3.2 配列，リスト

配列を宣言するには，各要素をカンマで区切った上で，角括弧 ([]) で囲みます．このとき，要素の型によって，配列の型が決定されます．配列は，インデックスを指定して要素を取り出したり，変更することができます．

```
>>> int[] a = [0, 1, 2]
[0, 1, 2]
>>> typeof(a)
konoha.Array<konoha.Int>
>>> a[1] = a[1] + 2
3
>>> a
[0, 3, 2]
```

配列は，可変長配列 (growing array) であるため，リストとしても活用することができます．配列の最後に要素を追加するときは，*add*() メソッドや << 演算子を用います．

```
>>> int[] a = ["a", "b"]
["a", "b"]
>>> a.add("c")
>>> a
["a", "b", "c"]
>>> a << "d" << "e"
>>> a
["a", "b", "c", "d", "e"]
```

また多次元配列も宣言できます。

```
>>> String[][] a;
[]
>>> a << [``a'', ``b''];
>>> a << [``a''];
>>> a << [``a'', ``b'', ``c''];
>>> a
[[``a'', ``b''], [``a''], [``a'', ``b'', ``c'']]
```

3.3 文字列

ダブルクォート (”) やシングルクォート (’) で囲まれた値は文字列になります．シングルクォートで囲まれた $'A'$ のような 1 文字は，その文字コードの整数値として扱われることがあります．これは，コンパイル時の型検査で判断されます．トリプルクォート (”””” や ’’’) は，その範囲内の改行やクォートもそのまま文字コードとして解釈します．

```
>>> "Naruto"
"Naruto"
>>> 'A'
65
>>> '''Naruto said "Hi."
... Sakura said "Hi."'''
"Naruto said \"Hi.\"\nSakura said \"Hi.\""
```

文字列は，演算子を用いることで数値リテラル同様、自由に編集することができます．フォーマッティング機能を使えば，データ変換操作を変換子で統一的に行うことができます.

```
>>> String fname = "Naruto"
"Naruto"
>>> String lname = "Uzumaki"
"Uzumaki"
>>> lname + " " + fname
"Uzumaki Naruto"
>>> "%s"(1.23)
"1.23"
```

文字列は，文字の配列として扱えます．文字列の先頭から n 番目の文字を取り出したいときは，$n-1$ のインデックスを指定します.

```
>>> String name = "Naruto"
"Naruto"
>>> |name|
6
>>> name[2]
"r"
```

スライス演算子 *to* を用いることで，部分文字列を取り出すことができます．最後の一文字を含めたくない場合は，*to* の代わりに *until* を用いることもできます.

```
>>> String s = "0123456789"
"0123456789"
>>> s[1 to 8]
"12345678"
>>> s[1 until 8]
"1234567"
```

4 Konoha でプログラミング

Konohaで実際にプログラムを書いて処理を行いましょう。この章ではKonohaでプログラムを製作するのに必要な型と関数の説明を行います。なお、Konohaは Javaの文法を基本的に蹴踏しているため、このチュートリアルではKonohaの文法を省略してあります。

4.1 型

今までの例でも見てきて分かるように、Konohaは型宣言を行うスクリプティング言語です。“int”, “float”, “Array”, “String” など、Javaで見覚えのある型ばかりですが、それぞれ整数、浮動小数点、配列、文字列の型として宣言されます。

4.1.1 型推論による静的型付け

Konohaは静的型付け言語ですが、オブジェクトを宣言するときはCやJavaのように型を宣言しなければエラーであるということはありません。なぜならKonohaは型推論の機能も備えているからです。宣言時の型推論においてオブジェクトの型を静的に決定することで型安全なプログラミング言語を実現しています。

```
>>> int a = 0;// Type Declaration
0
>>> b = ``This is String'' // Type Inference
``This is String''
```

4.1.2 型検査

前の章にも出ましたが、Konohaは型安全なプログラミング言語です。静的型付けに起因した機能としてこの型検査があります。その名の通り代入や宣言した型が正しいものかどうか判断してくれます。

```
>>> b = ``This is String not a number.'' // b's type is St
``This is String not a number.''
>>> b = 0; // Type Miss Match!
 - [(eval):1]:(error) not numeric: String
```

4.2 関数

ここでは関数定義について説明します。

```
>>> int square(int n) {
... return n * n; // return n-th power of two
... }
>>> square(3);
9
>>> square(10);
100
```

もちろん，複雑な計算にもKonohaを使うことができます．例えば，Fibonacchi数列を出力するプログラムは，Konohaでは次のように書く事ができます.

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

4.3 Func 型

この章の最後に少し高度な Konoha の使い方を紹介します。コールバック関数と呼ばれるもので関数の引数に渡す関数のことを指します。

```
>>> float aSecond(int n) {
... return (float)n / 2;
... }
>>> float aThird(int n) {
... return (float)n / 3;
... }
>>> float func(int n, Func<Int=>Float> f){
... return f(n);
... }
>>> func(10, aSecond);
5.000000
>>> func(10, aThird){
3.333333
```

Func<Int=>Float> というのは引数が Int 型 (“=>” の左側)、戻り値が Float 型 (“=>” の右側) である関数を表す、Func 型と呼ばれる機能です。同じ func() という関数でもコールバック関数によって動作が異なっているのがわかります。使用例として、Event を受け取ったときに発動する関数として利用されています.

5 Konoha でオブジェクト指向プログラミング

Konoha はオブジェクト指向プログラム言語であり、名前ベースのクラス、単一継承など Java からの影響を強く受けています. その一方で Java とは違い、緩やかな変更で微調整できる設計思想を導入し、スクリプティング言語としての特性を活かせるようになっています。

5.1 クラスの定義

フィールド変数は、アンダースコア (_) の接頭辞がついた名前で宣言します。これは、Java プログラミングでよく用いる慣習に由来し、ローカル変数とフィールドを区別するためです. Konoha ではこの他にキーワード this を使用することも可能です。

```
class Shape {
int _area;
 Shape(int area){
  _area = area;
 }
}
```

Konoha では、コンパイル済みのコードに対して型安全であれば、実行時にメソッドを追加したり、その振る舞いをかえることができます。インタラクティブにクラスメソッドを追加できるのはスクリプティング言語ならではと言えます。

```
>>>Shape s = new Shape(25);
>>>void Shape.getArea() {
  return Shape.area;
 }
>>> s.getArea();
 25
```

変数名の接頭辞にアンダースコアをつけなくても宣言することができます。(こちらの用例の方がより一般的でしょう。) このとき、Konoha コンパイラは自動的に接頭辞にアンダースコアを追加し、あわせて自動的に getter/setter メソッドが生成されます。

フィールド変数をどちらの形式で宣言しても、メソッドからフィールド変数へアクセスするときは、 _name のように アンダースコアを接頭辞につける必要があります。

続いて仮想フィールドの説明になります。実際にフィールド変数として宣言していなくても特定のメソッドを用いるとあたかもフィールド変数であるように振舞うことが出来ます。

```
class Shape {
  String getName(){
    return ``shape'';
  }
}
>>> Shape.name
shape
```

5.2 継承

継承とは親オブジェクトが持つ属性、メソッドを子が受け継ぐものです。これによりコード量を減らすことが出来、ソースコードの冗長化を防げます。Konoha は Java と同じく単一継承しか出来ません。

```
class Shape {
 int width,height;
 Shape(int w, int h){
  _width = w;
  _height = h;
 }
 int getArea(){
  return _width * _height;
 }
}
class Rectangle extends Shape{
 Rectangle(int w, int h){
  super(w, h);
 }
}

>>> Rectangle r = new Rectangle(5, 4);
>>> r.getArea();
```

```
  20
```

以上のように子である Rectangle クラスでは getArea() メソッドでは定義されていませんが、親である Shape クラスのものを継承していることがわかります。

5.3 ポリモーフィズム

ここでは Konoha におけるポリモーフィズムについて説明します。ポリモーフィズムとは同じ名前のメソッドでもクラスによって違う結果を返すというものです。なお、クラスやメソッド、フィールドに対してコントロールを行う修飾子を Konoha ではアットマーク (@) を付けて行ないます。

```
class Shape {
 int width,height;
 Shape(int w, int h){
  _width = w;
  _height = h;
 }
 @Virtual int getArea(){
  return _width * _height;
 }
}
class Rectangle extends Shape{
 Rectangle(int w, int h){
  super(w, h);
 }
 int getArea(){
  return _width * _height;
 }
}
class Triangle extends Shape{
 Rectangle(int w, int h){
  super(w, h);
 }
 int getArea(){
  return _width * _height / 2;
 }
}

>>> Rectangle r = new Rectangle(5, 4);
>>> r.getArea();
  20
>>> Triangle t = new Triangle(5,4);
>>> t.getArea();
  10
```

同じ getArea() という面積を返すメソッドでも Rectangle と Triangle では面積が異なるため返ってくる値も異なります。

6 その他の機能

ここでは今までの説明から応用してできることや、今までの説明から独立しているが強力であり欠かすことの出来ない機能を紹介します。

6.1 delegate

ますはソースコードとその実行結果を御覧ください。

```
>>> class Num{
...   int _base;
...   Num(int base){
...     _base = base;
...   }
...   int power(int exp){
...     int i, ret = 0;
...     for (i = 0; i < exp; i++)
...       ret *= _base;
...     }
...   }
... }
>>> Num n = new Num(3);
Num{base: 2}
>>> Func<Int=>Int> f = delegate(n, power);
Func<Int=>Int>:0x100214680
>>> f(3);
8
```

4.3 節に出た Func 型を利用していますが、ここで Func 型に渡している delegate という関数は Num クラス内で定義された power メソッドです。つまり、コールバック関数としてクラス内メソッドを利用するものです。ただの関数を渡すのとクラス内でメソッドの一番の違いはやはり、インスタンス別に処理が変わるというところでしょう。上記の例でも基数の値はインスタンスにより変化します。

6.2 パッケージの使用

Konoha は様々なパッケージを用意しています。たとえば、Konoha で高度な数学関数を使用したいと考えます。必要なパッケージはプログラムの先頭にて宣言します。

```
>>> using konoha.math.*; // import Math Package
... Math.sin(Math.PI / 2);
1.000000
```

6.3 include

パッケージの使用と似ていますが、こちらは実際に自分で作った konoha スクリプトを対話モードや他のファイルから参照できるものです。四角形の面積を返すプログラムを予め製作しておきます。

```
class Rectangle{ // This program is named rect.k
  int _w, _h;
  Rectangle(int w, int h){
    _w = w;
    _h = h;
  }
  int getArena(){
```

```
    return _w * _h;
  }
```

```
$ konoha -i rect.k
>>> Rectangle r = new Rectangle(3,4);
Rectangle{w: 3, h: 4}
>>> r.getArea();
12
```

konoha を対話モードで呼び出すときに-i のオプションをつけて、導入したいスクリプトを相対パスで指名します。対話モードではなく、他のスクリプトから呼び出したい場合は

```
include ''rect.k``
```

の様にプログラムの先頭で宣言しておきます。

7 終わりに

この度は、「初めての Konoha」をお読みいただきまして誠にありがとうございます。Konoha のインストールから関数、OOP プログラミングまでをひと通り説明しました。Konoha を楽しんでもらえたでしょうか？