A Scala Tutorial
for Java programmers

Version 1.2
March 15, 2008

Michel Schinz, Philipp Haller, 宮本隆志 (和訳)

Programming Methods Laboratory
EPFL
Switzerland

1 はじめに

この文章は Scala 言語とそのコンパイラについて素早く入門するためのものです。ある程度のプログラミング経験があり、Scala で何が出来るのかの概要が知りたい人を対象にしています。オブジェクト指向プログラミングの基本知識（特に Java での）が前提とされています。

2 最初の例

最初の例として標準的な *Hello world* プログラムを用います。興味深いとはいえませんが、Scala 言語に関する知識をさほど必要とせずに Scala のツールの使い方を簡単に示すことができます。このようになります。

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

Java プログラマにはプログラムの構造はお馴染みのものでしょう。`main` と呼ばれるメソッドが 1 つあり、パラメータとしてコマンドライン引数を文字列配列として受け取ります。メソッドの本体は定義済みメソッドである `println` を友好的な挨拶文を引数として 1 回呼び出しています。`main` メソッドは値を返しません（手続きメソッドです）。それゆえ戻り値型を宣言する必要はありません。

Java プログラマにとって馴染みが薄いのは、オブジェクトの宣言が `main` メソッドを含んでいることでしょう。シングルトン・オブジェクトとして知られるただ一つのインスタンスしか持たないクラスを、このような宣言で導入する事が出来ます。上記の宣言はすなわち、`HelloWorld` と呼ばれるクラスと、同じく `HelloWorld` と呼ばれるそのクラスのインスタンスの両方を宣言します。インスタンスは必要に応じて、つまり最初に使用される時に、生成されます。

賢明なる読者のみなさんは既にお気づきかもしれませんが、ここでは `main` メソッドは `static` として宣言されていません。なぜならば静的メンバ（メソッドやフィールド）は Scala には存在しないからです。Scala のプログラミングでは、静的メンバを定義するのではなくて、シングルトンオブジェクトのメンバを定義します

2.1 例をコンパイルする

この例をコンパイルするには、Scala コンパイラの `scalac` を用います。`scalac` は多くのコンパイラと同じように働きます。引数としてソースファイル 1 つと、もしかしたらオプションを幾つか取り、オブジェクトファイルを 1 つあるいは幾つか生成します。生成されるオブジェクトファイルは標準的な Java のクラスファイルです。

上記のプログラムを `HelloWorld.scala` というファイルに保存したならば、下記のようにコマンドを入力する事でコンパイル出来ます。（不等号 `>` はシェルプロンプトを示すものなので入力しないで下さい。）

```
> scalac HelloWorld.scala
```

これによってクラスファイルが幾つかカレントディレクトリに生成されます。その一つは `HelloWorld.class` と呼ばれ、`scala` コマンドによって直接実行可能なクラスを含んでいます。次節で示します。

2.2 例を走らせる

Scala プログラムをコンパイルしたあとは、scala コマンドにて走らせる事が出来ます。使い方は Java プログラムを走らせるのに使用する java コマンドと非常に良く似ており、同じオプションが使えます。上記の例は下記のコマンドで実行する事が出来、期待通りの出力を行います。

```
> scala -classpath . HelloWorld

Hello, world!
```

3 Java との関わり

Scala の長所の一つは Java のコードと関わるのが非常に簡単だということです。java.lang パッケージの全てのクラスはデフォルトでインポートされますが、他は明示的にインポートする必要があります

それを示す例をお見せしましょう。現在時刻を取得して特定の国、例えばフランス[1]、で使われる表記方法でフォーマットしたいとしましょう。

Java のクラスライブラリには Date と DateFormat のような強力なユーティリティクラスがあります。Scala は Java と継ぎ目無く協調するので、Scala のクラスライブラリに同様なクラスを実装する必要はありません。単に対応する Java パッケージのクラスをインポート可能です。

```
import java.util.{Date, Locale}
import java.text.DateFormat
import java.text.DateFormat._

object FrenchDate {
  def main(args: Array[String]) {
    val now = new Date
    val df = getDateInstance(LONG, Locale.FRANCE)
    println(df format now)
  }
}
```

Scala の import 文は Java のものと大変似ていますが、ずっと強力です。同じパッケージからの複数のクラスを、一行目のように波カッコ（{}）で囲む事によってインポート出来ます。他に違うのは、全てのパッケージあるいはクラスの名前をインポートする時には、アスタリスク (*) の代わりにアンダースコア (_) を使用するという事です。後で示すように、アスタリスクは Scala では有効な識別子 (例えばメソッド名) であるためです。

従って 3 行目の import 文は DateFormat クラスの全てのメンバをインポートします。これによって静的メソッドの getDateInstance と静的フィールドの LONG を直接見える様にします。

main メソッドの最初に、現在時刻をデフォルトで持つ Java の Date クラスのインスタンスを生成します。そして先にインポートした静的な getDateInstance メソッドを用いて日付フォーマットを定義します。最後に各国語化された DateFormat インスタンスによってフォーマットされた現在時刻を表示します。最後の行は Scala の構文の興味深い特徴を示しています。引数を一つ取るメソッドは中置き構文を取る事が出来ます。つまりこの式

[1] 例えばスイスでフランス語を話す地域などの他の地方でも同じ記法が使用されます。

```
df format now
```

は、次の式の文字数の少ない表記方法だといえます。

```
df.format(now)
```

これは些細な文法規則に見えるかもしれませんが実は重要な事柄です。詳しくは次の節で述べます。

Java との統合に関するこの節を終えるにあたって、Scala では直接に Java のクラスを継承したり Java のインターフェイスを実装したりすることが出来る、という事も述べておくべきでしょう。

4 全てはオブジェクト

Scala は純粋なオブジェクト指向ですがそれは全てが、数や関数も含めて、オブジェクトであるという意味においてです。この点において Java とは異なります。なぜならば Java ではプリミティブ型 (例えば `boolean` や `int`) と参照型とを区別しており、また関数を値として扱えないからです。

4.1 数はオブジェクト

数もオブジェクトなのでメソッドを持ちます。実際、下記のような式

```
1 + 2 * 3 / x
```

はメソッド呼び出しだけから成り立っています。前の節で見たように、下記の式と等価だからです。

```
1.+(2.*(3./(x)))
```

これは、`+`, `*` なども Scala では有効な識別子だということも、意味しています。

4.2 関数はオブジェクト

多分 Java プログラマにとってより驚くことは、関数もまた Scala ではオブジェクトだということでしょう。従って関数を、引数として渡したり、変数に格納したり、他の関数から戻り値にしたり出来ます。関数を値として扱うこの能力は、**関数プログラミング** と呼ばれる大変興味深いプログラミング・パラダイムの基礎の一部です。

なぜ関数を値として用いるのが有用であるかの非常に簡単な例として、1 秒毎に何かアクションを行うタイマー関数について考えてみましょう。行うアクションをどのように渡せば良いでしょうか。関数として、が極めて論理的です。このような関数を渡す様な簡単な例は、多くのプログラマは良くご存知でしょう。ユーザインタフェイスのコードにおいて、何かイベントが起こった時に呼び出されるコールバック関数を登録する際に、良く使われるからです。

下記のプログラムで、タイマー関数は `oncePerSecond` と呼ばれ、コールバック関数を引数として取ります。この関数の型は `() => unit` と書かれ、引数無しで戻り値無しである全ての関数の型です (`unit` 型は C/C++ の `void` に似ています) 。このプログラムの `main` 関数は単にこのタイマー関数を、端末に文章を表示するコールバック関数を付けて呼び出すだけです。別の言い方をするならば、このプログラムは 1 秒毎に “time flies like an arrow” という文章を表示し続けます。

```
object Timer {
  def oncePerSecond(callback: () => unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def timeFlies() {
    println("time flies like an arrow...")
  }
  def main(args: Array[String]) {
    oncePerSecond(timeFlies)
  }
}
```

文字列を表示する為に System.out のメソッドではなく、予め定義された println メソッドを使用しているということに留意して下さい。

4.2.1 無名関数

このプログラムは理解しやすいですが、もう少し洗練することが出来ます。まず最初に、関数 timeFlies は後で onecePerSecond 関数に渡す為だけに定義されていることに留意して下さい。関数に名前を付けるのは、一度きりしか使われないならば不必要であるように思われます。実のところ oncePerSecond に渡すためだけにこの関数を作成出来れば良いでしょう。Scala では無名関数、その名の通り名前の無い関数、を使えば可能です。私たちのタイマープログラムを timeFlies の代わりに無名関数を使って書き直したものはこのようになります。

```
object TimerAnonymous {
  def oncePerSecond(callback: () => unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def main(args: Array[String]) {
    oncePerSecond(() =>
      println("time flies like an arrow..."))
  }
}
```

この例で無名関数を使っている事は、関数の引数リストを本体と分離している右矢印'=>' によって判ります。引数リストが空であることは矢印の左側の空の括弧の組で判ります。関数の本体は上の timeFlies と同じです。

5 クラス

前に見たように Scala はオブジェクト指向言語であり、それゆえにクラスの概念があります。Scala のクラスは Java の構文と似た構文で宣言します。重要な違いは Scala のクラスはパラメタを持つ事が出来るということです。以下の複素数の定義で示します。

```
class Complex(real: double, imaginary: double) {
  def re() = real
  def im() = imaginary
}
```

この複素数クラスは２つの引数、複素数の実部と虚部を取ります。これらの引数は Complex クラスのインスタンスを生成する時に、new Complex(1.5, 2.3) のよう

に渡されます。このクラスは２つのメソッド re と im を持ち、実部と虚部へのアクセスを与えます。

これらの２つのメソッドの戻り値型が明示的に与えられていないことに留意すべきです。コンパイラが自動的に推論し、メソッドの右辺を見て、両者の戻り値型が double であることを演繹します。

コンパイラはこの場合のように常に型を推論出来るとは限りませんし、不運にも、どんな場合に推論出来るか出来ないかを知る簡単な規則はありません。実際には、通常これは問題にはなりません。なぜならば明示的に与えられていない型を推論出来ない時は、コンパイラは苦情を述べるからです。簡単な規則としては、Scala の初心者プログラマは、型が文脈から簡単に演繹出来ると思った時は型宣言を省略して、コンパイラが同意するか試してみるべきでしょう。しばらくするとプログラマは、どんな時に型を省略し、どんな時に明示的に指定すべきかについての勘を養うことが出来るでしょう。

5.1 引数無しのメソッド

メソッド re と im のちょっとした問題は下記の例の様に、呼び出す為に名前の後に空の括弧の組を付けなくてはならないということです。

```
object ComplexNumbers {
  def main(args: Array[String]) {
    val c = new Complex(1.2, 3.4)
    println("imaginary part: " + c.im())
  }
}
```

もし実部と虚部にあたかもフィールドの様に、空の括弧の組なしでアクセスできたら良いでしょう。Scala ではこれは **引数無しメソッド** として定義することで可能です。そのようなメソッドは引数が 0 個のメソッドと異なり、宣言時でも使用時でも名前の後ろに括弧を必要としません。我々の Complex クラスは次のように書き換えられます。

```
class Complex(real: double, imaginary: double) {
  def re = real
  def im = imaginary
}
```

5.2 継承とオーバーライド

全ての Scala クラスはスーパークラスを継承しています。スーパークラスが指定されていない時、例えば先の節の Complex クラスの例では、scala.Object が暗黙的に使用されます。

Scala ではスーパークラスから継承したメソッドをオーバーライド出来ます。しかし、不用意にオーバーライドされることを避ける為に、メソッドをオーバーライドする際には **override** 修飾子が必須です。例では Complex クラスは Object から継承した toString メソッドを再定義して拡張しています。

```
class Complex(real: double, imaginary: double) {
  def re = real
  def im = imaginary
  override def toString() =
    "" + re + (if (im < 0) "" else "+") + im + "i"
}
```

6 ケースクラスとパターンマッチング

プログラムに良く出てくるデータ構造の一つにツリーがあります。例えばインタプリタやコンパイラは通常プログラムを内部的にツリーで表現しています。XML 文書はツリーです。ある種のコンテナは赤黒木のようなツリーに基づいています。

小さな計算機プログラムを通じてツリーが Scala でどのように表現され操作されるのか見てみましょう。このプログラムの目的は、加法と整定数と変数からなる非常に簡単な数式を操作することです。その例を 2 つ挙げると、$1+2$ や $(x+x)+(7+y)$ などです。

最初にそのような数式をどのように表現するか決めましょう。最も自然な方法はツリーです。ノードが演算（ここでは加法）で、リーフが値（ここでは定数か変数）です。

Java ではそういったツリーは、ツリーの為の抽象スーパークラスと、ノードやリーフ毎の 1 つの具象サブクラスを用いて表現されるでしょう。関数型プログラミング言語では同じ目的の為に代数的データ型を用います。Scala には両者の中間的なものである**ケースクラス**があります。それをどうやって私たちのツリーの型を定義するのに用いるかを示します。

```
abstract class Tree
case class Sum(l: Tree, r: Tree) extends Tree
case class Var(n: String) extends Tree
case class Const(v: int) extends Tree
```

`Sum`, `Var`, `Const` クラスがケースクラスとして宣言されていることは、幾つかの点で普通のクラスとは違うということを意味しています。

- クラスのインスタンスを作るのにキーワード **new** は必須ではありません。（すなわち **new** `Const(5)` の代わりに `Const(5)` と書けます。）
- コンストラクタのパラメータの為の getter 関数は自動的に定義されます。（すなわち `Const` クラスのインスタンス `c` のコンストラクタのパラメータ `v` の値を単に `c.v` と書けば取得出来ます。）
- メソッド `equals` と `hashCode` はデフォルトで定義され、それらは同一性ではなくインスタンスの構造に基づいています。
- `toString` メソッドはデフォルトで定義され、値を「ソース形式」で表示します。（例えば、式 $x+1$ の式のツリーは `Sum(Var(x),Const(1))` と表示されます）
- これらのクラスのインスタンスは以下で見る様に**パターンマッチング**を通じて分解されます。

数式を表現するデータ型を定義したので、それを操作する演算を定義することができます。ある**環境**で式を評価する関数から始めることにします。環境の目的は変数に値を与えることです。例えば式 $x+1$ の評価を変数 x を値 5 に関連づけるような環境 $\{x \to 5\}$ の元で行うと結果 6 を得ます。

ここで環境を表現する方法を見つける必要があります。もちろんハッシュ表のような連想データ構造を使うことも出来ますが、直接に関数を使うことも出来ます。環境はまさに値を（変数）名と関連付ける関数に他なりません。上で述べた環境 $\{x \to 5\}$ は Scala では下記の様に簡単に書くことが出来ます。

```
{ case "x" => 5 }
```

この書き方で、文字列 "x" が与えられたならば整数 5 を戻し、他の場合には例外で失敗させる関数を定義出来ます。

評価する関数を書く前に、環境の型に名前を付けましょう。もちろん型 String => int を環境の為に使うことも出来ますが、この型に名前を付ければプログラムがシンプルになり将来の変更も容易になります。Scala では下記のように書くことで行えます。

```
type Environment = String => int
```

以後 Environment 型は String から int への関数の型の別名として使えます。

では評価する関数の定義を行いましょう。概念としてはとても簡単です。２つの式の和の値は単にそれぞれの式の値の和です。変数の値は環境から直接得られます。そして定数の値は定数自身です。Scala でこれを表現するのは同じぐらい簡単です。

```
def eval(t: Tree, env: Environment): int = t match {
  case Sum(l, r) => eval(l, env) + eval(r, env)
  case Var(n)    => env(n)
  case Const(v)  => v
}
```

評価関数はツリー t へのパターンマッチングを行うことで行われます。直感的には上記の定義は明確なはずです。

1. まずツリー t が Sum であるかチェックし、もしそうならば左部分木を新しい変数 l に、右部分木を変数 r に束縛します。そして矢印に従って評価を進めます。矢印の左側のパターンによって束縛された変数 l と r を使用します。
2. もし最初のチェックが成功しなければ、すなわちツリーは Sum でなければ、続いて t は Var かチェックします。もしそうならば Var に含まれる名前を変数 n に束縛し、右辺の式を用います。
3. もし２番目のチェックにも失敗したならば、つまり t は Sum でも Var でもなければ、Const であるかチェックします。もしそうならば Const ノードに含まれる値を変数 v へと束縛し、右辺へと進みます。
4. 最後に全てのチェックに失敗したならば、式のパターンマッチングの失敗を伝える為に例外が上げられます。ここでは Tree のサブクラスが他に定義されない限り起きません。

ある値を一連のパターンに順に当て嵌め、マッチしたら直ちにその値の様々なパーツを取り出して名前をつけ、名付けられたパーツを用いてコードを評価する、というのがパターンマッチングの基本的なアイデアであることを見てきました。

年期の入ったオブジェクト指向プログラマは、なぜ eval を Tree クラスとそのサブクラスのメソッドにしなかったのか、不思議に思うかもしれません。実はそのようにも出来ます。Scala ではケースクラスのメソッド定義は普通のクラスのように出来るからです。ですからパターンマッチングとメソッドのどちらを使うかは趣味の問題ですが、拡張性にも密接に関係します。

- メソッドを用いるならば、新しい種類のノードを追加することは Tree のサブ

クラスを定義することで簡単に行えます。しかしツリーを操作する新しい演算を追加するのは、Tree の全てのサブクラスの修正が必要なため面倒です。

- パターンマッチングを用いるならば状況は逆転します。新しい種類のノードを追加する為にはツリーのパターンマッチングを行う全ての関数で新しいノードを考慮する為の修正が必要です。その一方で新しい演算を追加するのは簡単で、単に独立した関数を定義するだけです。

パターンマッチングをもっと調べる為に、別の数式への演算である記号微分を定義してみましょう。読者はこの演算に関する下記の規則を覚えているでしょうか。

1. 和の導関数は、導関数の和。
2. 変数 v の導関数は、v が微分を行う変数であるならば 1 、さもなければ 0。
3. 定数の微分は 0。

この規則はほとんど字句通り Scala のコードに変換出来、下記の定義を得ます。

```
def derive(t: Tree, v: String): Tree = t match {
  case Sum(l, r) => Sum(derive(l, v), derive(r, v))
  case Var(n) if (v == n) => Const(1)
  case _ => Const(0)
}
```

この関数ではパターンマッチングに関する新しい概念を２つ導入します。最初に変数に関する **case** 式には**ガード**、すなわち **if** キーワードに続く式があります。ガードは式が真でなければパターンマッチングを失敗させます。ここでは微分される変数名が微分する変数 v と等しい場合のみ定数 1 を返す様に使われています。２つめのここで使われているパターンマッチングの特徴は _ で示されるワイルドカード（どんな値にもマッチするパターン）で、値に名前を与える必要はありません。

パターンマッチングの能力を全て調べてはいませんが、この文書が長くなり過ぎないようにここで止めるとしましょう。上記の２つの関数が実際の例でどう動くのか見たいと思います。この目的のため、簡単な main 関数を書いて式 $(x + x) + (7 + y)$ に対する演算を幾つか行ってみましょう。最初に環境 $\{x \to 5, y \to 7\}$ に対する値を計算し、次いで x と y とで微分した導関数を求めましょう。

```
def main(args: Array[String]) {
  val exp: Tree =
  Sum(Sum(Var("x"),Var("x")),Sum(Const(7),Var("y")))
  val env: Environment = { case "x" => 5 case "y" => 7 }
  println("Expression: " + exp)
  println("Evaluation with x=5, y=7: " + eval(exp, env))
  println("Derivative relative to x:\n " + derive(exp, "x"))
  println("Derivative relative to y:\n " + derive(exp, "y"))
}
```

プログラムを実行すると期待した結果が得られます。

```
Expression: Sum(Sum(Var(x),Var(x)),Sum(Const(7),Var(y)))
Evaluation with x=5, y=7: 24
Derivative relative to x:
 Sum(Sum(Const(1),Const(1)),Sum(Const(0),Const(0)))
Derivative relative to y:
 Sum(Sum(Const(0),Const(0)),Sum(Const(0),Const(1)))
```

結果を見ると、導関数の結果はユーザに見せる前に簡略化すべきであることが判ります。パターンマッチングを用いて基本的な簡約関数を定義することは興味深い（しかし驚く程に巧妙な）問題です。読者の練習問題としておきます。

7 ミックスイン

スーパークラスからコードを継承する以外にも、Scala では 1 つあるいは複数の**ミックスイン**でコードをインポートできます。

Java プログラマにとってミックスインが何かを理解する一番簡単な方法は、コードを含むことも可能なインターフェイスと看做すことでしょう。Scala ではあるクラスがミックスインから継承した場合、そのミックスインのインターフェイスを実装し、そのミックスインのコードも全て継承します。

ミックスインの有用性を見る為に、古典的な例の順序付きオブジェクトを見てみましょう。あるクラスのオブジェクト同士を比較することが出来ると、例えばソートしたりする場合など、しばしば役に立ちます。Java では比較可能なオブジェクトは `Comparable` インターフェイスを実装します。Scala では `Comparable` の同等品の `Ord` と呼ぶミックスインを定義することで、Java よりもう少しうまくやります。

オブジェクトを比較するには、6 つの異なる述語、小さい・小さいか等しい・等しい・等しくない・大きいか等しい・大きい、があると便利です。しかしそれらの全てを定義するのは冗長に過ぎます。なぜなら 6 つのうちの 2 つを使って残りの 4 つは表せるからです。例えば等しいと小さいの 2 つの述語があれば、他を表すことが出来ます。Scala ではこれらの事柄は次の様なミックスインを定義することで満たされます。

```
trait Ord {
  def < (that: Any): boolean
  def <=(that: Any): boolean =
    (this < that) || (this == that)
  def > (that: Any): boolean = !(this <= that)
  def >=(that: Any): boolean = !(this < that)
}
```

この定義によって、Java の `Comparable` インターフェイスと同じ役割をする `Ord` と呼ばれる型を作ると共に、抽象的な述語 1 つを用いて 3 つの述語のデフォルト実装が行われます。等しい・等しくないという述語は、全てのオブジェクトにデフォルトで存在するため、ここには現れません。

上の例で使われている `Any` 型は Scala の全ての型のスーパータイプの型です。Java の `Object` 型より更に一般的といえます。なぜならば `int` や `float` といった基本型のスーパータイプでもあるからです。

従って比較可能なクラスのオブジェクトを作る為には、等しいことと小さいことを判定する述語を定義し、上で述べた `Ord` クラスをミックスインすれば充分です。例としてグレゴリア暦の日付を表す `Date` クラスを定義してみましょう。そのクラスは全て整数値である日・月・年で構成されます。従って `Date` クラスの定義は次の様になります。

```
class Date(y: int, m: int, d: int) extends Ord {
  def year = y
  def month = m
  def day = d

  override def toString(): String =
    year + "-" + month + "-" + day
```

ここで重要なことはクラス名とパラメタに続く **extends** Ord 宣言です。これによって Date クラスが Ord クラスをミックスインとして継承することが宣言されます。

そして Object から継承している equals メソッドを再定義し、個々のフィールドを比較することで正しく日付を比較する様にします。equals の Java でのデフォルト実装はオブジェクトを物理的に比較するため使えません。下記の様な定義になります。

```
override def equals(that: Any): boolean =
  that.isInstanceOf[Date] && {
    val o = that.asInstanceOf[Date]
    o.day == day && o.month == month && o.year == year
  }
```

このメソッドでは予め定義された isInstanceOf と asInstanceOf というメソッドを使用しています。最初の isInstanceOf は、Java の instanceof 演算子に対応し、適用されたオブジェクトが与えられた型のインスタンスである場合にのみ真を返します。2つめの asInstanceOf は、Java のキャスト演算子に対応します。もしオブジェクトが与えられた型のインスタンスであるならばそのように観られるようになり、さもなければ ClasscastException が投げられます。

最後に下記のように小さいことを判定する述語を定義します。別の予め定義されたメソッドである error を使います。error は与えられたエラーメッセージ付きの例外を投げる定義済みメソッドです。

```
def <(that: Any): boolean = {
  if (!that.isInstanceOf[Date])
    error("cannot compare " + that + " and a Date")

  val o = that.asInstanceOf[Date]
  (year < o.year) ||
    (year == o.year && (month < o.month ||
      (month == o.month && day < o.day)))
}
```

これで Date クラスの定義が完了しました。このクラスのインスタンスは日付であり比較可能オブジェクトでもあります。更に、上で述べた 6 つの比較用の述語が定義されています。equals と < は Date クラスの定義の中に直接現れており、他は Ord ミックスインから継承しています。

ミックスインがここで示したよりも有用な状況があるのは勿論ですが、そういった例を長々と議論するのはこの文書の目的外から外れています。

8 ジェネリシティ

このチュートリアルで調べる Scala の最後の特徴はジェネリシティです。Java のプログラマは、Java にジェネリシティが無いことによる問題、Java 1.5 で検討される欠点、について良く知っているに違い有りません。

ジェネリシティとは型でパラメタ化されたコードを書ける能力です。例えば連結リストのライブラリを書こうとしたプログラマは、リストの要素にどの型を使うか決めるという問題に直面します。このリストは色々な場面で使うつもりですから、要素型を例えば int とか決めることは不可能です。それでは全く恣意的かつ制限が強すぎるでしょう。

Java プログラマは、全てのオブジェクトのスーパータイプである Object の助けを借ります。しかしこの解決方法は理想的とはほど遠いものです。なぜなら基本型 (int,

long, float など) には使えませんし、たくさんの動的な型キャストがプログラマによって挿入されることを意味するからです。

Scala ではこの問題を解決する為にジェネリック型（とメソッド）を定義出来ます。もっとも簡単なコンテナクラスである参照の例を見てみましょう。参照は空であるか何かの型のオブジェクトを指しています。

```
class Reference[a] {
  private var contents: a = _

  def set(value: a) { contents = value }
  def get: a = contents
}
```

Reference 型はその要素の型である型 a でパラメタ化されます。この型はクラス本体の中で、変数 content 、set メソッドの引数、get メソッドの戻り値、の型として使われます。

上記のサンプルコードで Scala の変数が紹介されていますが、それ以上の説明は不要でしょう。但しその変数の初期値が _ として与えられていることは興味深いでしょう。_ はデフォルト値を表します。デフォルト値は、数値型に対しては 0 、論理型に対しては **false**、unit 型に対しては () 、全てのオブジェクト型に対しては **null** です。

この Reference 型を使う為には型パラメタ a 、すなわち cell に格納される要素型を指定する必要があります。例えば整数を格納する cell を作るには下記の様にします。

```
object IntegerReference {
  def main(args: Array[String]) {
    val cell = new Reference[Int]
    cell.set(13)
    println("Reference contains the half of " + (cell.get * 2))
  }
}
```

この例で見た様に、get メソッドの戻り値を整数として使う前に値をキャストする必要はありません。また整数でない何かをその cell に代入することは出来ません。なぜなら整数を格納すると宣言されているからです。

9 結論

この文書では Scala の簡単な例をざっと概観してきました。興味を持った読者は *Scala By Example* へ進むと良いでしょう。Scala 言語仕様に関するずっと多くの進んだ事柄を含んでいます。そして必要に応じて *Scala Language Specification* を参照すると良いでしょう。