A Scala Tutorial

for Java programmers

Version 1.3
July 13,2010

Michel Schinz, Philipp Haller,宮本隆志(和訳), 水尾学文(文責)

PROGRAMMING METHODS LABORATORY
EPFL
SWITZERLAND

1 はじめに

この文書は Scala 言語とそのコンパイラについて簡単に紹介するものです。ある程度のプログラミング経験があり、Scala で何ができるか概要を知りたい人向けです。 オブジェクト指向プログラミングの基礎知識(とくに Java での)を前提としています。

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。これはあまり興味深いとは言えませんが、Scala 言語に関する知識をあまり必要とせずに、Scala 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１つのメソッドがあり、main は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。メソッドの本体で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、main メソッドを囲んでいる **object** という宣言でしょう。このような宣言によって、**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導入できます。この宣言はすなわち、HelloWorld と呼ばれるクラスと、そのクラスの同じく HelloWorld と呼ばれるインスタンスの両方をいっぺんに宣言します。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。というのは、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 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宣言します。

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 を使用します。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いくつか取り、オブジェクトファイルを１つあるいはいくつか生成します。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。上記プログラムを HelloWorld.scala というファイルに保存すれば、下記のコマンドでコンパイルできます。（不等号 '>' はシェルプロンプトなので、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いく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が、それは次節で示します。

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 コマンドにて実行できます。使い方は Java プログラムの実行で使う java コマンドと非常によく似ており、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でき、期待通りに出力されます。

```
> scala -classpath . HelloWorld

Hello, world!
```

3 Javaとの連携

Scala の長所の一つは Java コードとの連携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。その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を適用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 とシームレスに連動するので、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。対応する Java パッケージのクラスをそのままインポートできます。

```
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。同じパッケージの複数のクラスを、一行目のように波カッコ({ })で囲むことでいっぺんにインポートできます。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（_）を使うことです。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）だからです。

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 getDateInstance と静的フィールド LONG を直接見えるようにします。

main メソッドでは最初に、デフォルトで現在時刻を持つ Java の Date クラスのインスタンスを生成します。次に、先にインポートした静的メソッド getDateInstance 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 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には中置構文を使えます。つまりこの

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

```
df format now
```

という式は、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、実は重要な事です。詳しくは次節で述べます。

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 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できる、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、それは数や関数も含めて**全て**が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。というのは、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。それは前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。

数字の周りの括弧は必要です。というのは、Scala の字句解析はトークンに対して最長一致規則を使うからです。ですから、下記の式

```
1.+(2)
```

は、トークン 1.、+、2. に分解されます。このようにトークン化されるのは、1. が 1 よりも有効な長い一致となるからです。トークン 1. はリテラル 1.0 として解釈され、Int ではなく Double を形づくります。

```
(1).+(2)
```

と式を書けば、1 が Double として解釈されるのを防げます。

4.2 関数はオブジェクト

多分 Java プログラマがより驚くことは、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、他の関数からの戻り値にしたりできます。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ことが有用であるか、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、がきわめて論理的です。このように関数を渡す簡単な例は、多くのプログラマはよくご存知でしょう。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よく使うからです。

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。別の言い方をすれば、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)
  }
}
```

文字列を表示するために System.out のメソッドではなく、あらかじめ定義された println メソッドを使っている、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、もう少し洗練できます。まずはじめに、関数 timeFlies は、後で onecePerSecond 関数に渡すため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、一度きりしか使わないなら不必要と思われます。実のところ oncePerSecond に渡すためだけにこの関数を作成できればよいでしょう。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すと、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ことは、関数の引数リストと本体を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*2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 のクラスはパラメータを持てるということです。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いつも型を推論できるとは限りませんし、不運にも、どんな場合に推論できるかでき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。なぜなら明示的に与えられていない型を推論できない時は、コンパイラが苦情を言うからです。簡単な規則としては、Scala の初心者プログラマは、型を文脈から簡単に演繹でき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すべきでしょう。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、勘が働くようになるでしょう。

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、下記の例のように、呼び出すために名前の後に空の括弧の組を付けなくてはならないことです。

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

*2 補足しておくと、いくつかのオブジェクト指向言語はクラスの概念を持って いないことに注意すべきですが、Scalaはそういったものの１つではありません。

もし実部と虚部にあたかもフィールドのように、空の括弧の組なしでアクセスできたらよいでしょう。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。わたしたちの 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、例えば前節の 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できます。しかし、不用意にオーバーライドされることを避けるために、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

プログラムによ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。

小さな電卓プログラムを通して、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、加法と整数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、ツリーのための抽象スーパークラスと、ノードやリーフ毎に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、同じ目的のため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、いく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。

- それらの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用の 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 の値は、 単に c.v と書けば取得できます。）
- 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ように**パターンマッチング**を通して分解されます

数式を表現するデータ型を定義したので、つぎにそれを操作する演算を定義しましょう。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→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できますが、直接に関数を使うこともできます！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→5} は Scala では下記のように簡単に書けます。

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

この書き方で、引数として文字列 "x" が与えられたなら整数 5 を返し、他の場合には例外で失敗させる関数を定義できます。

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 を環境のために使うこともできますが、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けばよいのです。

```
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
}
```

この評価関数はツリー t に対して**パターンマッチング**を実行します。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 **var** の含まれる 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、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 Const ノードに 含まれる値を変数 v に束縛し、右辺へ進みます。
4. 最後に、全てのチェックに失敗したなら、式のパターンマッチングの失敗を 伝えるために例外が上げられます。ここで Treeのサブクラスが他に宣言されない限り、 それは起きません。

ある値を一連のパターンに順に当てはめ、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、というパターンマッチングの基本的なアイデアを見てきました。

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。実はそのようにもできます。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できるからです。それゆえ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。

- メソッドを用いれば、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れば状況は逆転します。新しい種類のノードを 追加するには、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 ノードを考慮するための修正が必要です。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ために、別の数式への演算である微分シンボル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 が微分を行う変数なら 1 、さもなければ 0。
3. 定数の微分は 0。

これらの規則はほとんど字句通り Scala のコードに変換でき、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 を返すように使われています。２つめに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、値に名前をつけ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すぎないようにここで止めておきましょう。上記２つの関数が実際の例でどう動く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) に対する演算をいくつか行っ

てみましょう。最初に環境 {x→5, y→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 トレイト

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**トレイト**からコードをインポートできます。

Javaプログラマがトレイトとは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みなすことでしょう。Scalaでは、あるクラスがトレイトから継承した場合、そのトレイトのインターフェイスを実装し、そのトレイトに含まれるコードも全て継承します。

トレイトの有用性を見るために、古典的な例である、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できると、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 と同等品の、Ordと呼ぶことにするトレイトを定義することで、Javaよりもう少しうまくやれます。

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つは表せるのですから。例えば、等しいと小さいの２つの述語があれば、他を表すことができます。Scalaでは、それらは次のようなトレイト宣言でうまく表せます。

```
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つを使って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、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ためには、等しいことと小さいことを判定する述語を定義し、上記 Ord トレイトをミックスインすれば充分です。例として、グレゴリオ暦の日付を表す Date クラスを定義してみましょう。そのような日付は、全て整数値である日・月・年から成ります。従って Date クラスの定義は次のようになります。

```
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、クラス名とパラメータに続く **extends** Ord 宣言です。これは Date クラスが Ord トレイトを継承することを宣言しています。

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ようにします。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。下記のような定義になります。

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

このメソッドではあらかじめ定義された、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 演算子に対応しており、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ならばそのように観られるようになり、そうでなければ 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。別のあらかじ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な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 トレイトから継承しています。

トレイト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から外れます。

8 ジェネリシティ

このチュートリアルで調べる最後の Scala の特徴はジェネリシティです。Java 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、についてよく知っているに違いありません。

ジェネリシティとは型でパラメー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、要素の型を例えば 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、プログラマはたくさんの動的な型キャストを挿入しなければならないからです。

Scala ではこの問題を解決するためにジェネリッククラス（とメソッド）を定義できます。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[T] {
  private var contents: T = _
  def set(value: T) { contents = value }
  def get: T = contents
}
```

Reference クラスはその要素の型である型 T でパラメータ化されます。この型はクラスの本体で、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 の変数が導入されていますが、これ以上の説明は不要でしょう。ただ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 、Boolean 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 クラスを使うためには、型パラメータ T 、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 を作るには下記のようにします。

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

この例で見たように 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 に代入することはできません。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 言語を概観し、基本的な例をいくつか見てきました。興味を持った読者は **Scala By Example** へ進むとよいでしょう。ずっと多くの進んだ例があります。そして必要に応じて **Scala Language Specification** を参照するとよいでしょう。