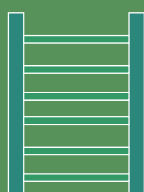
JJUG CCC 2009 Spring セッションC1

2009.4.21 国立オリンピック記念青少年総合センター

関数型xオブジェクト指向：マルチパラダイム言語*Scala*の可能性



羽生田 栄一

株式会社 豆蔵 取締役CTO

内容

●JVMの上で動きポストJava言語としても注目されるプログラミング言語Scalaを、本格的な関数型プログラミングとオブジェクト指向との融合の可能性という観点に立って紹介する。なぜ関数型が今後重要な計算パラダイムなのか。

- マルチパラダイム言語Scalaの生い立ち
- オブジェクト指向への道と関数型言語
- 関数型プログラミングの特徴
  - 関数、スコープとクロージャ、計算の合成、カリー化
  - 厳密な型理論、リストと高階関数
  - 強力で柔軟なパターンマッチング
- オブジェクト指向と関数型の融合の可能性
  - パターンマッチングとオブジェクト、for構文
  - DSLとしての可能性
- まとめ

Java言語のよさとは？

1. C/C++の言語スタイルの踏襲
2. 本格的なオブジェクト指向プログラミング
3. バイトコードとVMによる処理系（JVM）
4. ガベージ・コレクション
5. マルチスレッドによる並行処理
6. JDKをはじめとする潤沢なクラスライブラリ

ポストJava言語？の背景

1. Generics総称クラスとクロージャの機能
   - ある種の型理論、ある程度の高階関数（map等）機能
2. LL言語の隆盛：簡潔さと柔軟性
   - 動的言語・スクリプト言語: Python, Perl, Ruby
3. Webアプリの隆盛：手軽さと開発生産性
   - Ruby on Rails、アジャイル開発に向く言語
4. 関数型言語処理系の実用化・高速化
   - Scheme、Common Lisp、SML、Haskel、Ocaml
5. クラスタ、スケーラビリティ、分散・並列性へのニーズ
   1. Erlang、Google/MapReduce、アクターほか並列性理論
6. Java登場から10年以上経過…飽き?次世代へ！

ポストJava言語の条件？

1. C/C++/Javaの言語の伝統を踏まえていること
   - JVMを使うだけでなく、強い静的型付けまで入れるか？Yes！
2. 柔軟で簡潔な記述ができる言語文法であること
   - ある種の動的性が必要！➔ 動的言語だけを意味しない！＋*パタンマッチング*
3. スケーラビリティや並列性に対する工夫が見られること
   - クラスの継承＋Mixin＋関数＋安全でスケーラブルな並列性
4. 実用的なフレームワークやライブラリ、ツールが備わっていること
5. DSL（ドメイン記述言語）として拡張できること
   - 言語の柔軟な拡張性（ライブラリの追加が言語の拡張と見做せる）
6. 既存のソフトウェア資産との相互運用性があること
   - 既存のコードの柔軟な再利用、拡張による新規コードの記述
7. コンピュータサイエンスや言語処理系の理論的成果をうまく取り入れていること
   - 型理論、アクター等並列モデル、継続、アスペクト、Mix-in、言語理論
8. 既存のJavaプログラマの支持を受け十分な開発者が確保できること

Javaコード（37行、{}なしで21行）

```
1.  public class Person
2.  {
3.     private String lastName;
4.     private String firstName;
5.     private Person spouse;
6.
7.     public Person(String fn, String ln, Person s)
8.     {
9.        lastName = ln; firstName = fn; spouse = s;
10.    }
11.    public Person(String fn, String ln)
12.    {
13.       this(fn, ln, null);
14.    }
15.
16.    public String getFirstName()
17.    {
18.       return firstName;
19.    }
20.
21.    public String getLastName()
22.    {
23.       return lastName;
24.    }
25.
```

```
1.  public Person getSpouse()
2.     {
3.        return spouse;
4.     }
5.     public void setSpouse(Person p)
6.     {
7.        spouse = p;
8.           // 婚姻の対称性と姓の変更に関しては
9.           // 考慮していません
10.    }
11.
12.    public String introduction()
13.    {
14.       return "私の名前は," + firstName + " " + lastName +
15.          (spouse != null ?
16.  " 相方の名前は," + spouse.firstName + " " + spouse.lastName + " 。" :
17.  " 。");
18.    }
19. }
```

参考）http://blogs.tedneward.com/2006/03/02/Scala+Pt+2+Brevity.aspx

Rubyコード Ⅰ（16行、endなしで12行）

```
1.  class Person
2.    def initialize(firstname, lastname, spouse = nil)
3.      @firstName = firstname
4.      @lastName = lastname
5.      @spouse = spouse
6.    end

7.    attr_reader :lastName
8.    attr_accessor :firstName, :spouse
9.
10.   def introduction
11.     if spouse == nil
12.       "私の名前は, #{firstName} #{lastName}"
13.     else
14.       "私の名前は, #{firstName} #{lastName} 相方の名前は, #{spouse.firstName}
    #{spouse.lastName}"
15.     end
16.   end
17. end
```

Rubyコード Ⅱ（11行、endなしで8行）

```
1.  class Person
2.        def initialize(firstname, lastname, spouse = nil)
3.          @firstName, @lastName, @spouse = firstname, lastname, spouse
4.        end

5.        attr_reader :lastName
6.        attr_accessor :firstName, :spouse

7.        def introduction
8.            "私の名前は, #{firstName} #{lastName} " + (spouse ?
9.            " 相方の名前は, #{spouse.firstName} #{spouse.lastName}" : "")
10.       end
11. end
```

Scalaコード（10行、{}なしで8行）

```
1.  class Person(ln : String, fn : String, s : Person)
2.  {
3.     def lastName = ln;
4.     def firstName = fn;
5.     def spouse = s;
6.
7.     def this(ln : String, fn : String) = { this(ln, fn, null); }
8.     def introduction() : String =
9.        return "私の名前は, " + firstName + " " + lastName +
10.          (if (spouse != null) " 相方の名前は, " + spouse.firstName + " " +
    spouse.lastName + "。"  else "。");
11. }
```

<=型推論のおかげで型宣言が不要！

Rubyとほぼ同じ簡潔さ ＋ 強い型付き！

Scalaコード（5行、{}なしで4行）

```
1. case class Person(val lastName:String, val firstName:String, var spouse:Person) {
2.    def this(ln:String, fn:String) = this(ln, fn, null)
3.    def introduction = "私の名前は、" + firstName + " " + lastName +
4.        (if (spouse != null) "、相方の名前は、" + spouse.firstName + " " +
   spouse.lastName + "。" else "。")
5. }
```

クラスのコンストラクタ内の変数宣言から自動的に、

val x ➔ xの暗黙のゲッター追加

var y ➔ yの暗黙のセッター、ゲッター追加

Rubyとほぼ同じ簡潔さ ＋ 強い型付き！

新世代言語Scalaに関する事実1

- ScalaからすべてのJavaクラスを簡単に利用可
- JavaからScalaクラスを呼び出すことも自由
- 膨大なJava、J2EE、Java ME CLDCの資産が
  すべて、より合理的でコンパクトな形で利用可

新世代言語Scalaに関する事実2

- ScalaはJava VM上で実行され
- その実行性能はJavaコードとほぼ同等
- 結果として、
  ほとんどのスクリプト言語より一桁速い
  - Scala＞Ruby, Groovyほか…

新世代言語Scalaに関する事実3

- Scalaは**純粋なオブジェクト指向**言語
- しかも**本格的な関数型**言語

- Scalaのデータはすべてオブジェクト:
  - 文字列も配列も関数もすべて！
- 関数もオブジェクト:
  - データとして自在に操作可（高階関数、クロージャ、カリー化）

新世代言語Scalaに関する事実4

- Scala作者**Martin Odersky**教授
  - JavacやJava Genericsの開発貢献者
- **強力な開発体制**
  - 早いペースでリリース
  - ドキュメントも充実（ただし英語）
- **実用的な汎用プログラミング言語**
- Scala言語処理系 http://www.scala-lang.org/downloads
  - 2009年4月末時点でScala 2.7.4.RC1が最新版 (安定版は、Scala2.7.3.final)

まずは対話型インタープリタ

- Read-Eval-Printループ
- Scala基本サイトからダウンロード
- 日本語を使うには
  - *scala –Xnojline* でインタプリタ起動

インタプリタ利用例　1/2

1. scala> val msg = "こんにちは" // valで定数の宣言。型省略。リテラルに日本語OK
2. msg: java.lang.String = こんにちは　 // msgがStringだと型推論されている
3. scala> msg size　　　　　　　　// msg.sizeやmsg.size( )としても同じ
4. res2: Int = 5　　　　　　　　//結果の値がres<N>に、後で利用可。正しく5文字！
5. scala> (1 to res2) foreach print　//Intのメソッドtoで範囲。foreachの引数に関数print
6. 12345　　　// 1～5を関数printに順番に適用。改行したければ関数println
7. scala> msg = "さよなら"　　　// valで定義したmsgは定数なので代入不可
8. <console>:2: error: assignment to non-variable　　// エラー：定数への代入
9. scala> var msg = “挨拶”　　　// varで変数を宣言。
// 型指定するとvar msg: String = "挨拶"
10. msg: java.lang.String = 挨拶　// 型指定しなくてもStringと推論。

インタプリタ利用例　2/2

1. scala> msg = "やぁ"　　　// 変数なので新規に代入可能
2. msg: java.lang.String = やぁ
3. scala> msg.foreach(println)　　　// msgの各Charに関数printlnを適用
4. やぁ
5. scala> msg (0)　//StringはArray[Char]とみなせ、ゼロ始まりの(i)で要素iにアクセス
6. res3: Char = や　　　// msg (i)はmsg.apply(i)の省略形。
   // 更新はupdate(i, a)だが文字列には適用不可
7. scala> for (c <- 'あ' to 'ん')　　　// toでIterator[Char]の範囲指定。
   // <-は要素[集合記号]集合の記号を意味する
8. | print(c)　　　// for文の途中であえて改行。
   // インタープリタ中で縦棒で継続可能。
9. あぃいぅうぇえぉおかがきぎくぐけげこごさざしじすずせぜそぞただちぢっつづてでとどなにぬねのはばぱひびぴふぶぷへべぺほぼぽまみむめもゃやゅゆょよらりるれろゎわゐゑをん
10. scala> :quit　　　// Scalaインタープリタを終了

マルチパラダイム言語Scalaの特徴1

- 統一性 : *すべての値は、オブジェクト*
  - *Int*
  - *Array*
  - *Function<N>*
- *すべて*
  - *メソッドをもち*
  - *変数に代入し*
  - *引数に渡せる*

（注）関数はオブジェクトだがメソッドはオブジェクトではない
しかし、「 method _ 」として、いつでも関数に変換できる

マルチパラダイム言語Scalaの特徴2

- 統一性: *すべての操作(演算)は、メソッド適用*
  - 3 + 4 ・・・ 実は、次の式と同じ意味
  - 3 . +(4) ・・・ +というメソッドを引数4で適用
  - array (3) ・・・ 配列の要素の参照も
  - array. apply(3) ・・・ 実は、メソッドapply(i)の適用
  - array(3) = 0 ・・・ 配列の要素の更新も
  - array. update(3, 0) ・・・ 実は、メソッドupdate(i, v)の適用

マルチパラダイム言語Scalaの特徴3

- 基本: 1引数のメソッド適用
  - オブジェクト . メソッド (引数リスト)
  - オブジェクト メソッド (引数リスト)
  - オブジェクト メソッド 1引数 ⇐ *DSLに便利*
- 文と文の区切り: ; か改行
- ブロック: {S; T; U} ⇐ *DSLに便利*
- 式を整理するための括弧: ()か{} ⇐ *DSLに便利*
  - (a .m(b)) * (c / (d + f(e)))
  - { a m(b)} * { c / { d + f(e) } }
  - (a m b) * (c / { d + f(e) })
  - { a m b } * (c / { d + f(e) })

ScalaのJava基本型相当のクラス群

- Scalaで利用できる基本的なデータ型は実はすべて普通のクラス
  - 7.toString()　7.toString　7 toString
  - 1.to(10)　1 to 10　1 until 11
- クラス管理　Any, AnyVal, AnyRef, ScalaObject, Null, Nothing
- 数　Int, Ling, Byte, Short, Float, Double
- 真偽値　Boolean
- 文字　Char, String, Symbol
- 不定　Unit

Scala vs. Java：文法上の相違点1

- 型の宣言は型名変数= 値ではなくて変数：型名= 値で指定
  - ただし、与えられた値から型推論できる場合には型名は省略可
- 代入不可の変数はvalで宣言
- 通常の変数はvarで宣言
- 任意のデータにdefを用いて名前を付けられる
  - valの代わりにもdefを使えるが振る舞いは若干異なる
  - 関数やメソッドの宣言にはdefを用いる
- 文の区切り目の;はオプション。通常は改行で表す
- 一連の複文は;で区切ってブロック{ }でまとめる。
  - 単文はブロックにしなくてよい。
  - forループ中のprint(c)は{ print(c) }としてもしなくても問題ない

Scala vs. Java：文法上の相違点2

- 数や文字列、配列も含めすべてのデータはオブジェクト。int、double、booleanなども含め**すべてのデータは特定の型を表すScalaクラス**に属する
- voidはUnitクラスとして扱い、Unit型の唯一のインスタンスは( )
- 配列のインデックスはarray[i]ではなくarray(i)と

アクセスする。配列の参照array(i)および更新array(i)=xもarray.apply(i)およびarray.update(i,x)という通常のメソッド適用と見なされる

Scala vs. Java：文法上の相違点3

- [ ]は型パラメータの指定に使われ、type IList =List[Int]のように宣言
- 型T へのcastはasInstanceOf [T]メソッドで
  - 型パラメータを使えばほとんど無用
- forループはfor-comprehensionとして定義
  - map、filterなどに変換される
- staticという概念はない
  - Singletonパターンで代用される。classではなくobjectを
    - object Singleton { def m1():Unit = println("Singleton example") }
  - として定義し、その特性にstaticメンバ代わりにアクセスできる
- import文でパッケージやクラスをインポートできる
  - *ではなく_を利用
    - import javax.swing.JFrame; import javax.swing.JFrame._ などとする。
  - 名前を付け替えられる
    - import javax.swing.{JFrame=>MyWindow}など

Scalaクラス階層の基本構造

- Any
- AnyVal
- Double, Float, Long, Int, Short, Char, Byte, Boolean, Unit
- AnyRef = java.lang.Object
- String = java.lang.String
- その他のすべてのJavaクラス群
- その他のすべてのScalaクラス群
- *Iterable* = Scalaコレクションクラスのベース
- *Seq*
- Array
- List
- ・・・
- Null = すべてのAnyRef系統サブクラスのボトムクラス
- Nothing = Anyすなわちすべてのクラスのボトムクラス

オブジェクト指向 vs 関数型

- **オブジェクト指向**
  - 概念：状態をカプセル化したオブジェクトとその相互作用
  - 材料：オブジェクト（状態を保持）
  - 方式：オブジェクトへメッセージ送信=メソッド起動
  - 結果：元のオブジェクトの状態が変化する（副作用）
- **関数型**
  - 概念：計算とはデータへの関数の適用のこと。関数もデータ。
  - 材料：データ、ないしデータの集合
  - 方式：データへの関数の適用＝データの変換
  - 結果：データは不変、変換により新たなデータを生成

関数型プログラミングの位置づけ？

- Martin Oderskyは2種類に分けている:
  - 排他的な関数型プログラミング：Haskell
    - 副作用なしの計算方式
  - 包括的な関数型プログラミング：Scheme, Ocaml, Scala
    - プログラミングスタイル, 関数を自由に利用した計算を許す
- Scalaは後者を目指し、オブジェクト指向と融合！
  - 全体的なアーキテクチャ、データ構造 の部品化
    ➔ オブジェクト指向 （純粋化、Trait）
  - 集合演算、機能の部品化、データ非共有並列化
    ➔ 関数型 （不変データ、map等の高階関数）
  - 共通: 強力な型理論、パターンマッチ、for構文

関数の定義

- 基本は、名前付きの関数

  def 関数名 （引数リスト）: 戻り型 = 関数本体
  - 関数の仮引数の型は絶対に省略できません
  - 戻り型は通常は型推論できるので省略可
    - ただし、関数定義本体が再帰的な場合は
      戻り型を省略できない
  - 戻り型voidは、Unit型として宣言:
    - ()はUnit型の唯一のリテラル
- Scala関数のシグニチャ

  関数名: (引数型リスト)戻り型

関数の定義例

1. scala> def ##(str: String) = str.size … 文字数カウント。戻り型は省略
2. $hash$hash: (String)Int ……..... 戻り型Intをメソッドsizeから型推論
3. scala> ##(“Scala大好き”) .... ##といった未使用記号も名前に可
4. res4: Int = 8
5. scala> def fact(n: BigInt): BigInt = ……..再帰なので戻り型宣言
6. | if (n == 0) 1 else n * fact(n - 1)
7. fact: (BigInt)BigInt …..................... 関数factのシグニチャ表示
8. scala> fact(100) 整数リテラル100から暗黙型変換でBigInt
9. res5:BigInt=9332621544394415268169923885626670049071596826
   43816214685929638952175999932299156089414639761565182862
   53697920827223758251185210916864000000000000000000000000

無名関数の定義

- 関数名を指定せずに関数が定義できる
  - (a: T1, b:T2,...)＝＞関数本体
  - ()＝＞関数本体
- 例
  - (n: Int) => n * 2
  - (a: Int, b: Int) => a * b
  - () => println(”言語はScala”)
- これらの関数はすべてオブジェクトなので、
  - 関数として他のオブジェクトにメソッド適用したり
  - 変数に代入したり
  - 後から名前を付けたりできる

無名関数とリスト、高階関数map, fold

1. scala> (n: Int) => n * 2 .... 2倍するという無名関数λn:Int. (n * 2)
2. res6: (Int) => Int = <function> ………. (Int)=>Intという型の関数
3. scala> res6(10) …………. 仮名res6で関数を参照し整数10に適用
4. res7: Int = 20 ………………………… 結果は整数20
5. scala> val double = (n: Int)=> n * 2 ...... 無名関数にdoubleと名前を
6. double: (Int) => Int = <function> ....
7. scala> double (100) ...... 関数バインド変数doubleに引数リスト適用
8. res4: Int = 200 .... 結果は整数200
9. scala> List(1, 2, 3, 4, 5) map double ….高階関数**map**でdouble適用
10. res14: List[Int] = List(2, 4, 6, 8, 10) 結果は2倍の値のリスト
11. scala> List(1,2,3,4,5).foldLeft(0){_ + _} ….高階関数**fold**で+適用
12. res28: Int = 15
13. scala> (0 /: (1 to 5))(_ + _) …. list.**foldLeft(0)(op)** ⇔ **(0 /: list)(op)**
14. res29 : Int = 15

スコープとimport

- object MyFunctions {
-   def foo(s: String, n: Int): Int = s.length * n
-   def bar(s: String, n: Int): Unit = for(i <- 1 to n) print(s)
- }
- MyFunctions.foo("豆蔵太郎", 3) ➔ 12
- MyFunctions.bar("豆蔵太郎", 3) ➔ 豆蔵太郎豆蔵太郎豆蔵太郎

必要ならimport MyFunctions._ とすれば、どこからでも使える

リスト操作と関数、ループ、クロージャ

- scala> val ssize = (s:String) => s.length　　//無名関数を変数に代入
- ssize: (String) => Int = <function>
- scala> ssize("豆蔵太郎左衛門")　//変数ssizeが関数として機能
- 7
- scala> List("豆", "豆蔵", "太郎左衛門").map((s:String) => s.length)
- List(1, 2, 5)　　//各要素に一律に無名関数を適用した結果のリスト
- scala> List("豆", "豆蔵", "太郎左衛門").map(ssize)
- List(1, 2, 5)　　//各要素に一律に変数ssizeを適用した結果のリスト
- scala> val f0 = (s:String, n:Int) => for(i <- 1 to n) print(s)
- f0: (String, Int) => Unit = <function>
- def bar(s: String, n: Int): Unit = 1 to n foreach {i => print(s)}

Scalaにおける関数の正体

- **実は関数は、Function0、Function1、Function2,…といったTraitを実現したオブジェクト**
  - 関数f: (ArgType1,...ArgTypeN)=>ReturnTypeはトレイトFunctionN[ArgType1,..., ArgTypeN, ReturnType]のインスタンス
  - **必ず、apply()を実装していなければならない！**

```
1. object double extends Function1[Int, Int] {
2.   def apply(n: Int): Int = n * 2
3. }
```

カリー化と関数部分適用

- カリーとは、多引数関数を1引数関数のリニアな関数適用の合成に変換すること
  - f:(X,Y)=>Z　=== f: X=>(Y=>Z)

def f0(x:Int, y:String, z:Int) = (x + y.length) * z をカリー化する記法

- def f1(x:Int)(y:String)(z:Int) = (x + y.length) * z
- f1: (Int)(String)(Int)Int
- f1(100)("test")(5) ➔ 520
- scala> val app1 = f1(100) _　関数の部分適用➔戻り値も関数
- ➔ app1: (String) => (Int) => Int = <function>
- scala>val app2 = app1("test")　関数の部分適用➔戻り値も関数
- ➔ (Int) => Int = <function>
- app2(5) ➔ 520

関数の部分適用と _（アンダースコア）

- 任意の仮引数を _ で無名パラメータ化できる
  - (_.isUpperCase) == (c:Char)=>c.isUpperCase
  - (_+_) == (a, b)=> a + b　…変数の型は推論
- カリー化記法の関数への部分適用
  - scala> def curryf(_:Int)("test")(_:Int)
  - curryf: (Int, Int) => Int = <function>
  - scala> curryf(_:Int)("test")(_:Int)
  - res2: (Int, Int) => Int = <function>
- 通常の関数記法への部分適用
  - scala> def f0(x:Int, y:String, z:Int):Int = (x + y.size) * z
  - f0: (Int,String,Int)Int
  - scala> val partf = f0(_:Int, "test", _:Int)
  - partf: (Int, Int) => Int = <function>
  - scala> partf(2, 100)
  - res3: Int = 600

Scalaにおけるクラス定義

- クラス宣言が基本コンストラクタの定義も兼ねる

```
1. class Person(na: String, ag: Int) {
2.   def name() = na
3.   def age() = ag
4.   def this(na: String) = this(na, 0) ……2次コンストラクタ
5. }

6. scala> val tanaka = new Person("田中", 25)
7.  tanaka: Person = Person@13c2797
8. scala> val hatena = new Person("奈々氏")
9.  hatena: Person = Person@1cfad77
10. scala> hatena.age
11.  res0: Int = 0
```

Scalaにおけるクラス定義2 言語としての簡潔さ

- Javaの場合

```
1. class ClassA {
2.   public String name;
3.   private int age;
4.   public ClassA(String name, int age) {
5.     this.name = name;
6.     this.age = age;
7.   }
8. }
```

- Scalaの場合

```
1. scala> class ClassA(val name: String, private var age: Int) {}
2. scala> val x = new ClassA("佐々木", 25)
3.  x: ClassA = ClassA@968f9
4. scala> x.name
5.  res10: String = 佐々木
6. scala> x.age
7.  <console>:7: error: variable age cannot be accessed in ClassA
```

Traitによる多重継承（Mixin機能）

- 「Trait」という概念
  - 単一継承しか認められない通常クラスとは別に、Javaでいう Interfaceをより強力にし属性や操作も持たせられる
- Mixin的に多重継承が可能
  - Rubyのモジュールに相当
- しかも、インスタンス生成時にwith節を使ってMixinすることもできる

ScalaにおけるTrait（Mixinのこと）

```
1.  class Person
2.  trait Teacher {
3.    def teach:Unit {}    //バーチャルなメソッドであってもよいし
4.  }
5.  trait PianoPlayer {
6.    def playPiano =      //実装をともなう具象メソッドであってもよい
7.  }
8.  class PianoplayingTeacher extends Person with Teacher with PianoPlayer {
9.    def teach = {println("TEACHING")}
10. } ...................................................... .............クラスとしてTraitから継承

11. val katouSensei = new PianoplayingTeacher

12. val tanakaTaro = new Person with Teacher with PianoPlayer {
13.   def teach = {println("Teaching is Playing")}}
```

……………..田中太郎インスタンスに個別に性質を付与！

Traitを使ったアスペクト指向

- trait TAction { def doAction }
- trait TBeforeAfter extends TAction {// doAction前後処理をするアスペクト
  abstract override def doAction{
  println("/entry before-action")
  super.doAction
  println("/exit after-action")}}
- class RealAction extends TAction {
  def doAction = { println("** real action done!! **") }}

- scala> val act1 = new RealAction with TBeforeAfter
- act1: RealAction with TBeforeAfter = $anon$1@1bd06a0
- scala> act1.doAction
- /entry before-action
- ** real action done!! **
- /exit after-action

Traitによるアスペクト指向2

- trait TTwiceAction extends TAction {//元処理を繰り返すというアスペクト
- abstract override def doAction {
- for (i <- 1 to 2) {super.doAction // 元の処理doActionの呼び出し
- println(" ==> No." + i) }
- }}
- scala> val act2 = new RealAction with TBeforeAfter with TTwiceAction
- act2: RealAction with TBeforeAfter with TTwiceAction = $anon$1@1a28573

with節で定義した順番に適用されている

- scala> act2.doAction
- /entry before-action
- ** real action done!! **
- /exit after-action
- ==> No.1
- /entry before-action
- ** real action done!! **
- /exit after-action
- ==> No.2

ScalaからJavaを呼ぶ

- Scalaでは自由にかつ簡単にJavaクラスやJavaインタフェースを利用できます。
  - JavaからScalaを呼ぶのも簡単。
- 次の例は、JavaのSwingライブラリの利用例：

```
1. import javax.swing.{JFrame=>Mado, _}

2. val mameMado = new Mado("豆窓")
3. mameMado setSize(200, 150)
4. mameMado getContentPane() add(new JLabel("これからはScalaですから"))
5. mameMado setVisible(true)
```

パターンマッチング機能

- Scalaのcase文は非常に強力で**任意のオブジェクト対象（パターン）**

```
val value: Any = "文字列"
value match {
  case null => println("null!")
  case i: Int => println("Int: " + i)
  case s: String => println("String: " + s)
  case _ => println("その他: 何か")
  }
➔String: 文字列
```

- **Personクラス**のオブジェクトをマッチ

```
class Person(name:String)
val value: Any = new Person("豆蔵太郎")
value match {
  case null => println("null!")
  case i: Int => println("Int: " + i)
  case s: String => println("String: " + s)
  case _ => println("その他: 何か")
  }
➔その他: 何か
```

- パターンマッチのcase節は、「**部分関数（partial function）**」

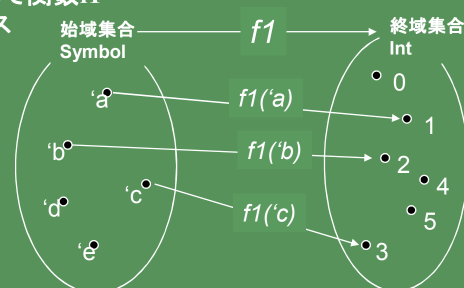
部分関数(partial function)

- f1:A=>B　型Aから型Bへの写像として関数f1
  - クラスFunction1[A, B]のインスタンス
    1. scala> class A
    2. defined class A
    3. scala> class B
    4. defined class B
    5. scala> type F = A=>B
    6. defined type alias F
    7. scala> val a1 = new A
    8. a1: A = A@742397
    9. scala> val b1 = new B
    10. b1: B = B@3c40f0
    11. scala> val f1:F = {case a1 => b1; case a2 => b2}
    12. f1: (A) => B = <function>
    13. scala> f1(a1)
    14. res1: B = B@3c40f0

scala> val (a2, b2) = (new A, new B)
a2: A = A@16a5d72
b2: B = B@1d07e4

部分関数(partial function) 2

- f1:A=>B　型Aから型Bへの写像として関数f1
  - クラスFunction1[A, B]のインスタンス
    - def f1: Symbol=>Int = {
    - case 'a => 1
    - case 'b => 2
    - case 'c => 3
    - }
    - scala> f1('c)
    - res0: Int = 3
    - scala> f1('d)
    - scala.MatchError:



- 部分関数としてf2:A=>Bを定義
  - trait PartialFunction[-A, +B] extends (A => B)
    {def isDefinedAt(x: A): Boolean }
  - 関数評価に先立ってisDefinedAt(x:A):Booleanを用いて定義域の確認ができる
    - def f2: PartialFunction[Symbol, Int] =
    - {case 'a => 1; case 'b => 2; case 'c => 3}
    - scala> for(s <- List('a, 'b, 'c, 'd)){ if(f2.isDefinedAt(s)) println(f2(s)) }
    - 1
    - 2
    - 3

便利なcase クラス

- パターンマッチ対象を「case class」として宣言
  - そのクラスのインスタンス生成時のコンストラクタ呼び出し時にnewを省略可
  - またtoString, hashCode, equalsが自動生成される

```
case class Person(name:String, var age:Int, var spouse:Person)
val p1 = Person(”Tom",49,null)
val p2 = Person(”Lucy",29, Person (”Ken",35,null))
```

パターンマッチング機能2

- Personクラスをcase classとして再定義

```
case class Person(name:String,  age:Int, var spouse:Person)

val value = Person("豆蔵太郎", 32, null)
val wife = Person ("豆蔵花子", 32, value)
value.spouse = wife

value match {
  case null => println("null!")
  case i: Int => println("Int: " + i)
  case s: String => println("String: " + s)
  case Person(_, _, null) => println("独身 : "+value.name)
  case Person(_, age, p) if age == p.age => println("同年齢どうしの結婚")
  case _ => println("その他: 何か")
  }
```

- ➔同年齢どうしの結婚

抽象データ型の宣言

- caseクラスを利用して、抽象データ型を定義

```
1. abstract class Term
2. case class Var(name: String) extends Term
3. case class Fun(arg: String, body: Term) extends Term
4. case class App(f: Term, v: Term) extends Term

5. Fun("x", Fun("y", App(Var("x"), Var("y"))))
```

パターンマッチによるパージング

```
1.  def print(term: Term): Unit = term match {
2.    case Var(n) => print(n)      //マッチ時の対応は個別にcase以下で定義
3.    case Fun(x, b) => print("/¥" + x + "." ); print(b)
4.    case App(f, v) => print("("); print(f); print(" "); print(v); print(")")
5.  }
6.  scala> val t1: Term = Fun("x", Fun("y", App(Var("x"), Var("y"))))
7.    t1: Term = Fun(x,Fun(y,App(Var(x),Var(y))))
8.  scala> print(t1)
9.    /¥x. /¥y. (x y)

10. def isIdFun(term: Term): Boolean = term match {
11.   case Fun(x, Var(y)) if x == y => true      // case中のif節で等値テスト
12.   case _ => false          //残りすべてのケース「 - 」
13. }

14. scala> isIdFun( Fun("a", Var("a")) )    // 項/¥a.aは明らかに恒等関数Id
15.  res33: Boolean = true
16. scala> isIdentityFun(t1)        // 項/¥x. /¥y. (x y)は恒等関数ではない
17.  res32: Boolean = false
```

for構文の利用例

* Scalaのfor構文はかなり強力（実はモナド）
  - map,flatMap,filterなどのメソッド呼出しのsyntax sugar

```
1. for (x <- 1 to 10; y <- 1 to 100
2.           if y == x * x) yield (x, y)
```

➔res0: Seq.Projection[(Int, Int)] = RangeG((1,1), (2,4), (3,9), (4,16), (5,25), (6,36), (7,49), (8,64), (9,81), (10,100))

```
3. val list = List((1, "a"), (2, "b"), (3, "c"), (1, "z"), (1, "a"))
4. for( (1, x) <- list ) yield (1, x)
```

➔List((1,a), (1,z), (1, a))

for構文の利用例2

```
1. val list = List(1, "two", Some(3), 4, "five", 6.0, 7)
2. for(x <- list){ x match{
3.   case x:Int => println("integer " + x)
4.   case x:String => println("string " + x)
5.   case Some(x) => println("some " + x)
6.   case _ => println("else " + x)
7. } }
```

➔ integer 1
string two
some 3
integer 4
string five
else 6.0
integer 7

高階関数・クロージャの利用

- Scalaでは、関数もオブジェクトなので自由に
  - 変数に代入できる
  - 他のオブジェクトに対し、メソッド適用できる
- 引数の名前呼び（call by name）で遅延評価
  - f1（ p1: T1） ・・・通常の値呼び（call by value）
  - f2（ q1: **=>T1**） ・・・**名前呼び（call by name）**

高階関数（クロージャ）の利用例

- ファイル入出力へクロージャの応用

```
1. import java.io._

2. def open(path :String)(block :InputStream => Unit) {
3.   val inSt = new FileInputStream(path)
4.   try { block(inSt) } finally { inSt.close; println("closed") }
5. }

6. open("test1.java") { in => () }
```

- リソース(in:InputStream)を自動的にcloseしてくれる

構造サブタイピング（Duck typing）

1. class A {def cry = println("a.a.a")}
2. class B {def cry = println ("b.b..b")}
3. class C {def whisper = println("…")}
4. type Crier = { def cry } ……構造型 {def cry}のままでも可
5. val criers = List[Crier](new A, new B, new C)
6. //これは型不一致でエラー
7. class C {def cry = println("c.c..c…")} と再定義
8. val criers = List[Crier](new A, new B, new C)
9. criers foreach(x => x.cry) ……… criers foreach(_ cry)もOK
10. //コンパイル時にこの式が実行可能だと保障される！！
11. a.a.a
 b.b..b
 c.c..c…

DSLとしてのScalaの可能性1：独自の制御構造の定義例

- 例：独自の制御構造Whileの定義
- Call by Name（字面渡し必要時評価）の利用

1. def MyWhile (p: => Boolean) (s: => Unit) {
2. if (p) { s ; MyWhile(p)(s) }
3. }

4. var i: Int = 0
5. MyWhile(i < 3) {i=i+1; print("がってん ")}
6. がってん がってん がってん
7. scala> MyWhile (true) (print("世界は無限だ"))

DSLとしてのScalaの可能性2：新しいデータ構造や型の追加

- 新しい型(例：BigInteger)を追加したいとき
  - > factorial(30)
  - 265252859812191058636308480000000
- 従来の言語（JavaやHaskelなど）だと…

```
1. import java.math.BigInteger
2. def factorial(x: BigInteger): BigInteger =
3.   if (x == BigInteger.ZERO)
4.     BigInteger.ONE
5.   else
6.     x.multiply(factorial(x.subtract(BigInteger.ONE)))
```

DSLとしてのScalaの可能性3：使い慣れたリテラルや演算子でOK

- Scalaであれば…

```
1. def factorial(x: BigInt): BigInt =
2.   if (x == 0) 1 else x * factorial(x - 1)
```

- クラスBigIntはJavaのBigIntegerをラップしているだけなのに通常の演算表記で使える!
  - 演算子（+,-,*,・・・）のオーバーローデングが簡単
  - 暗黙の型変換Implicit Conversionが強力！

```
1. implicit def intToBigInt(x: Int) = new BigInt(x)
```

DSLとしてのScalaの可能性4：
日本語で語彙（型・関数・変数）自由に表現！

1. case class `住所`(adr: (Int, String, String, String))
2. case class `利用者`(`登録id`:BigInt, `氏名`:String, `連絡先`:`住所`)
3. case class `図書`(`書名`:String, `図書id`:BigInt){
4. def `を貸し出す=>`(`借り手`:`利用者`) }
5. val `段田塊太` = `利用者`(9200800345, “段田塊太”, `住所`(101630434, “新宿”, “2-1-1”, “スカラ座ビル”))
6. val `本1`=`図書` (“はじめてのスカラ”, 1200700077)
7. `本1` `を貸し出す=>` `段田塊太` <= `本1`.`貸し出し`(`段田塊太`)

- Scalaの識別子の定義（以下の4種のいずれか）
  - *alphanumeric*：a letter or _ で開始 例） **HHanyuda _var _Max_Int**
  - *operator*：printable ASCII 文字 +, :, ?, or #等で開始 例） + ++ ::: <?> :->
  - *mixed*：*alphanumeric*＋*operator*
    - op_+ op_* power_** human_? toBeOrNotToBe_?
  - ***literal*：‘任意の文字列‘（2つのback-tick‘で囲った文字列）**
    - **‘x‘ ‘<clinit>‘ ‘yield‘ ‘人間‘ ‘貸し出す‘ ‘処理A‘**
- 要求⇔コード のトレーサビリティの確立
  - 日本語で書いた要求を日本語のクラスとメソッドで仕様化し、日本語の変数や文字列でプログラミングできる！

その他のScalaの可能性：注目機能

1. Lisp以来の関数型言語の伝統の柔軟なリスト構造の処理
2. 部分関数、カリー化や高階関数、クロージャの自由な扱い
3. 総称型（generics）および暗黙の型変換を含む高度な型処理
4. 文字列やリスト・クラス（case class）まで含めた強力なパターンマッチング
5. for構文（ 正確にはfor -comprehension）とOption[T]によるモナドの実現
6. 遅延評価lazy機能
7. メタプログラミングのためのアノテーション@機能
8. XML操作機能（XMLリテラル、Scala式の埋め込み、DOMツリー操作）
9. PEG（*Parsing Expression Grammar*：解析表現文法）とJSONパーサ
10. アクター理論にもとづく並列性ライブラリ（RubyのFiberに相当）
11. **Lift**（Scala＝階段、に対しエレベータの意。http://liftweb.net）というRuby on Rails風の強力なフルスタックのWebアプリ開発フレームワーク
12. 日本の西本氏「**Better CGI, Better PHP!**」を目標にシンプルで使いやすいフレームワーク**Web Flavor**(http://webflavor.sourceforge.net/index_ja.html)

lift *Web Framework* : http://liftweb.net
Rails風フルスタックF/W (現在v1.0)

- lift はJava Webコンテナ上で動く＋ Scala で簡単にコーディング.
  - lift アプリは、WAR files で任意の Servlet 2.4 engine (Tomcat 5.5.xx, Jetty 6.0, etc.)にデプロイ可能
  - lift は、オープンソースであり、Apache License V2.0で配布
- lift の特徴:
  - セキュリティ, アプリ開発の高生産性, 開発・保守の容易性, 高い実行性能, 既存J2EEシステムとの互換性
- 影響を受けた既存のフレームワーク
  - ***Seaside***の柔軟なセッションとセキュリティの取り扱い, ***Rails*** の超お手軽な高速アプリ開発,
  - ***Django***の単なるCRUD 以上の機能, ***Erlyweb***のCometスタイルのアプリのスケーラビリティ
- Lift vs. Rails
  - 同じくらいクリーンで簡潔なコード
    - lift と Rails でコード規模はほぼ同じ。ただし、テストコードサイズは、強い型のおかげで約60%
  - 厳密な型チェック
    1. User.find(By(User.email, "foo@bar.com")) // legal
    2. User.find(By(User.birthday, new Date("Jan 4, 1975"))) // legal
    3. User.find(By(User.birthday, "foo@bar.com")) // compiler error
  - 6倍高速、マルチスレッド対応
  - 簡潔なDSLの提供
    1. User.find(20) // find the user with primary key 20
    2. // find all the users that registered 4 days ago
    3. User.findAll(By(User.registered, 4.days.ago))
    4. // activate the account or suspend it after 4 days
    5. State(NewAcct, On({case Activate =>}, Activated),After(4 days, Suspended))
    6. State(Suspended) entry {sendSuspendedEmail}

Web Flavor フレームワーク

- 関西在住Scalaプログラマ：西本圭佑さん作
  - Web Flavorでは記述性に優れたScalaに特化し、Scalaの特徴を生かせるフレームワークを！
- 特徴
  - Java Servletコンテナ上で動作する
  - Java Servlet上でスクリプトとして動作する
  - 書いてすぐ実行できる (Flavorソースをコンパイルして実行)
  - スクリプト言語実行環境には, Java Servletインターフェースを用意
  - XMLリテラルで気軽に(X)HTML/XML出力
- 目標：「**Better CGI, Better PHP!**」
  - Liftほどかっちり枠に嵌めないので使いやすいハズ
- インストール
  - ${WEB_FLAVOR_HOME}/webflavor.warをServletコンテナに配置(デプロイ)
- http://webflavor.sourceforge.net/index_ja.html

Scala関連ツール

- 開発環境
  - Eclipse Scala plug-in
  - NetBeans Scala plug-in
  - maven-scala-plugin
- BDD仕様定義フレームワーク
  - Specs
    - http://code.google.com/p/specs/
- テストツール
  - Scala Check …….BDD仕様からのテスト自動生成
  - ScalaTest ………..テストツール

Scalaの使い方：4つの入り口

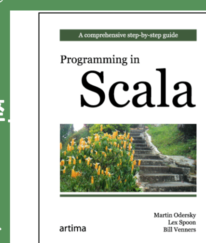
- **Entrance１： Better Java**として
  - エンタープライズApp開発の実用的汎用的手段
  - 強力なLiftフレームワーク
- Entrance２： **DSL**として
  - 要求やドメインの実行可能モデル記述手段
- Entrance3：ライブラリ・フレームワーク構築言語として
  - パタンマッチ、Mixin、高階関数、暗黙型変換等の高記述力
- Entrance4：教育研究用汎用言語として
  - 新Pascalとしての簡潔で美しい教育言語
  - 新しい型理論、分散並列処理、Mixin、アスペクト
  - 言語処理系のワークベンチ

新世代言語Scalaの基本情報

- Scalaの基本サイト
  - http://www.scala-lang.org/index.html
- 日本のScalaコミュニティ **ぜひMixiコミュへご参加を**
  - Mixi Scalaコミュ http://mixi.jp/view_community.pl?id=3111016
  - Scala-ja http://cappuccino.jp/scala-ja/ scala-sandboxコード共有サイトあり
- Scala言語チュートリアル（以下の2本がお勧め）
  - 「 Why Scala? 」 by Tim Dalton（Java Technical Insight of the Month）
    - http://ociweb.com/jnb/jnbDec2007.html
  - Scalazine「First Steps to Scala」（英語）
    - http://www.artima.com/scalazine/articles/steps.html
- より詳しく
  - 仕様 Scala Language Specification
    - http://www.scala-lang.org/docu/files/ScalaReference.pdf
  - 解説 Scala By Example
    - http://www.scala-lang.org/docu/files/ScalaByExample.pdf

言語Scalaの基本情報

- Scalaの基本サイト
  - http://www.scala-lang.org/index.html
- 日本のScalaコミュニティ **ぜひMixiコミュへご参加を**
  - Mixi Scalaコミュ http://mixi.jp/view_community.pl?id=3111016
  - Scala-ja http://cappuccino.jp/scala-ja/ scala-sandboxコード共有サイトあり
- Scala言語チュートリアル
  - 「 Why Scala? 」 by Tim Dalton（Java Technical Insight of the Month）
    - http://ociweb.com/jnb/jnbDec2007.html
  - Scalazine「First Steps to Scala」（英語）
    - http://www.artima.com/scalazine/articles/steps.html
  - **IT Pro連載「刺激を求める技術者に捧げるScala講座」**
    - http://itpro.nikkeibp.co.jp/article/COLUMN/20080613/308019/
- より詳しく
  - 仕様 Scala Language Specification
    - http://www.scala-lang.org/docu/files/ScalaReference.pdf
  - 書籍 Programming in Scala (現在翻訳中 **インプレス**より出版予定)
    - http://www.scala-lang.org/docu/files/ScalaByExample.pdf



まとめ：Scalaとにかく使うべし！

- Scalaは単純で統一性のある文法を目指している
  - 関数型と純粋オブジェクト指向の完全な融合
- さまざまな機能は言語としてでなく、ライブラリとして
- 性能を意識した最適化された処理系

Ruby等の動的スクリプト言語と比較して、Scalaは強い静的型付け
  - 動的な性質が弱い分だけ柔軟性に欠けると予想されがちだが
  - 統一性、型推論や高階関数、パターンマッチングに助けられて、
- 関数とオブジェクトの両立による表現力において
- 記述のコンパクトさにおいて

十分互角に戦える言語に仕上がっている

ぜひ食わず嫌いを止めて、まずは味見を！

ご案内： Scalaユーザ会

- 5/22(金)19：00-21：00
- 豆蔵トレーニングルーム@新宿三井ビル34階
- Scalaに興味のある方は誰でもご参加ください。
- 内容（予定）：
  - Scala勉強会（短かめの講演x数人）
  - Scalaに関するライトニングトークス
  - 日本にScalaを普及させるための悪巧み（予定）
  - 新宿西口喫茶 スカラ座前で記念撮影（予定）