

# Scalaミニチュートリアル ~Introduction to Scala~

---

筑波大学大学院システム情報工学研究科  
水島 宏太

# はじめに

---

- Scalaの概要について構文ごとに説明します
  - Javaと特に異なる文法については  
Javaと比較して説明します
  - ディープなScalaの話はしません
  - スカラの話はしません
-

# Scala ?

---

- 最近話題の言語
    - Matzにつき など
    - Actorライブラリなどで注目を集めている
    - ポストJava ?
  - 静的型付けオブジェクト指向関数型言語
  - 作者: Martin Odersky
    - JavaへのGenerics導入の貢献者の一人
  - JVM上で動作する処理系
  - 近代的な言語仕様
-

# 関数型言語？

---

- 明確な定義は無い
    - 副作用無しスタイルを積極的にサポート？
    - OOPと同じくらい曖昧
  - 具体例
    - 純粋: Haskell
    - 非純粋: Scala, SML, OCaml, Scheme, ...
    - 関数型ではない: C, C++, Java, C#, Ruby, ...
-

# hello, world

---

```
object HelloWorld {  
  def main(args :Array[String]) :Unit = {  
    println("hello, world")  
  }  
}
```

- **object**でシングルトンオブジェクトを定義(後述)
  - **def**でメソッド定義
  - `Array[String] ≐ String[]`
  - `Unit ≐ void`
  - 以降、`main`の定義は省略
-

# 変数の宣言(varとval)

---

```
var x = 1 // ≡ var x : Int = 1
var y = 2 // ≡ var y : Int = 2
val z = x + y // ≡ val z : Int = x + y
println(z)
```

- varで変数(変更可能)を宣言
    - Javaの通常の変数と同じ
    - 変数の型は省略可能(右辺の型から推論される)
  - valで変数(変更不能)を宣言
    - Javaのfinalな変数
    - 変数の型は省略可能
-

# 条件分岐(if)

---

```
val num = args(0).toInt
println(
  if(num % 2 == 0) "偶数" else "奇数"
)
```

- Javaのifに類似
  - elseを書くと値を返すことができる
    - Javaの?:演算子と同じようにも使える
  - elseを書かない場合Unit型の値が返る
-

# forによる繰り返し(1)

---

```
var sum = 0
for (arg <- args) sum += arg.toInt
println(sum)
```

- コマンドライン引数の合計値を計算するプログラム
  - forでコレクションの要素について繰り返し
    - Javaの拡張for文に類似
      - 実際には拡張for文よりも強力
    - argの型は省略可能(推論される)
-

# forによる繰り返し(2)

---

```
for (i <- 1 to 100) {  
  println(if(i%15 == 0) "FizzBuzz" else if(i%3 == 0) "Fizz"  
    else if(i%5 == 0) "Buzz" else i.toString)  
}
```

- Fizz Buzzプログラム
  - `x to y`でxからyまでのRangeを生成
    - `x.to(y)`と同じ意味(.や())などを省略可能)
  - `if`が値を返す
    - `else`を省略した場合は、Unit型の値を返す
  - 整数もオブジェクト → メソッドが呼び出せる
-

# whileによる繰り返し

---

```
var sum = 0
var i = 0
while(i < args.length) {
    sum += args(i).toInt
    i += 1
}
println(sum)
```

- コマンドライン引数の合計値を計算
  - forよりも冗長だが、forよりも高速
    - 性能をチューニングしたいときに使う
-

# 高階関数(1)

---

```
println(args.foldLeft(0)((sum, n) => sum+n.toInt))
```

- コマンドライン引数の合計値を計算するプログラム
  - $(x, \dots) \Rightarrow \dots$  で無名関数を定義
    - Javaの無名クラスのようなもの
    - 無名関数の引数の型が推論可能なときは省略可能
  - foldLeft: 関数を引数に取るメソッド(高階関数)
    - $\dots f(f(f(0, \text{args}(0)), \text{args}(1)), \text{args}(2)) \dots$
    - 関数型言語でループを書くための一般的な方法
-

# 高階関数(2)

---

```
println((0/:args) (_+_ .toInt))
```

- コマンドライン引数の合計値を計算するプログラム
- $/:$   $\doteq$  foldLeft
  - 末尾に $:$ の付いた演算子は左右逆に適用
    - $(x /: y) (z) \rightarrow y. /: (x) (z)$
- $\_+ \_ .toInt \doteq (x, y) \Rightarrow x+y.toInt$ 
  - 式に「穴を開けた」無名関数を定義

# 高階関数(3)

---

```
import java.io._
def open(path :String) (block :InputStream => Unit) {
  val in = new FileInputStream(path)
  try { block(in) } finally { in.close; println("closed") }
}
```

```
open("hoge.txt") {in => ()} // inは自動的にcloseされる
```

- リソースが自動的にcloseされるopen
  - $X \Rightarrow Y$ :「Xを受け取ってYを返す関数」の型
    - $\text{InputStream} \Rightarrow \text{Unit}$   
「InputStreamを受け取り何も返さない関数」
    - 引数無しメソッドの呼び出しの()は省略可能
-

# クラス定義(1)

---

```
class HelloWorld {  
  def display() :Unit = println(toString())  
  override def toString() = "hello, world"  
}
```

```
(new HelloWorld).display // hello, world
```

- **class**に続けてクラス名を記述
    - Javaとほぼ同じ
  - 戻り値の型は省略可能(推論される)
  - **override**でメソッドをオーバーライド
    - Javaの`@Override`アノテーションと同じ
    - **override**を忘れた場合エラー
-

## クラス定義(2)

---

```
class Point(val x :Int, val y :Int) {  
  def +(p :Point) = new Point(x+p.x, y+p.y)  
  override def toString() = "Point(" + x + ", " + y + ")"  
}  
val p = new Point(1, 2) + new Point(2, 2)  
println(p) // Point(3, 4)  
println(p.x, p.y) //(3,4)
```

- 二次元座標上の点を表現するクラス
  - クラス名に続けてコンストラクタ引数を定義
    - `val` をつけると外部から読み取り可能
    - 引数のスコープはクラス定義全体
  - 演算子は通常の方法として定義
-

# クラス定義(3)

---

```
import javax.swing._
class MyFrame extends JFrame("フレーム 1") {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
    setSize(800, 600)
}
(new MyFrame).setVisible(true)
```

- **extends**に続けてスーパークラスを指定
    - Javaと同じ
  - スーパークラスのコンストラクタに渡す引数はスーパークラス名に続けて記述
  - クラス定義中の実行文はインスタンス生成時に実行される
-

# trait

---

```
trait Foo {  
  def foo() :String = "Foo"  
}  
trait Bar {  
  def foo() :String  
  def bar() :Unit = println(foo() + "Bar")  
}  
class FooBar extends Foo with Bar  
(new FooBar).bar() // FooBar
```

- traitに続けて名前と記述
  - インスタンスが生成できない一種のクラス
    - 実装を持つことができる
  - 多重継承が可能
-

# object

---

```
object MyRunnable extends Runnable {  
  def run() :Unit = {  
    for (i <- 1 to 100) print(i)  
  }  
}
```

```
new Thread(MyRunnable).start // 12345...100
```

- シングルトンオブジェクトを定義可能
    - **new**でインスタンスの生成不可能
  - クラスと同様にスーパークラスを指定可能
-

# Implicit Conversion(1)

---

```
implicit def int2String(i :Int) :String = i.toString
val n = 54321
println(n.substring(0, 3)) // 543
println(n.endsWith(1)) // true
```

- **implicit def**で暗黙の型変換を定義
    - Int(引数の型) → String(戻り値の型)
    - Int型に存在しないメソッドの呼び出しを検出  
→ Stringへの暗黙の変換をコンパイラが試みる
  - 危険な変換を定義しないようにするのは自己責任
    - String → Intなど
-

# Implicit Conversion(2)

---

```
class RangeExt(range :Range) {  
  def exclude(elem :Int) = range.filter(e => e != elem)  
}  
implicit def range2RangeExt(r :Range) :RangeExt = new RangeExt(r)  
for (i <- 1 to 5 exclude 3) print(i) // 1245
```

- **implicit def**で暗黙の型変換を定義
  - Range(引数の型) → RangeExt(返り値の型)
  - Rangeにexcludeメソッドが追加されたように見える
- 既存のクラスにメソッドを追加(したように見せかけることが)可能

# パターンマッチング(1)

---

```
println(args(0) match {  
  case "apple" => "りんご"  
  case "grape" => "ぶどう"  
  case "peach" => "桃"  
  case _       => "その他"  
})
```

- 式 `match { case パターン => ... }`
    - `switch(式) { case 定数: ... }`に相当
    - `_`はdefault相当
  - 値を返すことができる
  - `switch`文をより強力にしたようなもの
-

# パターンマッチング(2)

---

```
println(args match {  
  case Array("foo", "bar") => "FooBar"  
  case Array("bar", "foo") => "BarFoo"  
  case Array(name)         => name  
  case _                   => "default"  
})
```

- パターンにマッチした値を変数に束縛可能
    - 1要素の配列について、要素をnameに束縛
  - 構造を持ったデータをパターンマッチに使える
    - コレクションなど
-

# Case Class

---

```
abstract class Exp
case class Add(l :Exp, r :Exp) extends Exp
case class Num(v :Int) extends Exp
def eval(exp :Exp) :Int = exp match {
  case Add(l, r) => eval(l) + eval(r)
  case Num(v)    => v
}
println(eval(Add(Num(1), Num(2)))) // 3
```

- caseをclassの前に付加して定義
  - 名前(...)でインスタンスが生成可能
  - パターンマッチに使える
-

# Structural Typing

---

```
object Java { def name() :String = "じゃヴぁ" }
object CSharp { def name() :String = "しーしゃーぷ" }
object Scala { def name(): String = "すから" }
def printName(x : { def name() :String }) = println(x.name)
printName(Java) // じゃヴぁ
printName(CSharp) // しーしゃーぷ
printName(Scala) // すから
```

- 静的に型チェックされる duck typing
    - 特定のシグニチャのメソッド(の集まり)を持っているかのみがチェックされる
-

# 紹介しなかったScalaの機能

---

- Lazy Value
    - 遅延評価
  - Generics
    - Java Genericsと類似しているが、より強力
  - Abstract Type
  - Existential Type
    - Java Genericsのワイルドカードに似たもの
  - Extractor
  - XML関係の機能
    - リテラル
    - パターンマッチング
-

# まとめ

---

- Scalaの言語仕様の概要を駆け足で紹介
    - 制御構文
    - OOP機能
      - クラス, object, trait
    - 関数型の機能
      - 高階関数, パターンマッチング, case class
    - (おそらく)Scalaに特異な機能
      - implicit conversion
  - 是非、Scalaを一度お試しく下さい！
-