

Scala入門

渡辺 義則(a-san)

お品書き

- 1. 自己紹介
- 2. Scalaとは
- 3. リテラル(Literal)
- 4. 構文(Syntax)
- 5. コレクション(Collection)
- 6. 関数
- 7. クラス、オブジェクト、トレイト、パッケージ
- 8. 関数型プログラミングへ

自己紹介1



- 渡辺 義則(a-san)
- 株式会社CSK 90年度入社
- 担当プロジェクト数＝約40プロジェクト
- プログラマ歴＝約28年
- プログラム言語＝20言語ぐらい
- Scala歴＝3年弱ぐらい
- Scala勉強会2回参加
- 第1回Scala座のライトニングトークで発表

自己紹介2

■ 開発したオープンソース

- C言語関数ツリー(funcntree)
- JCom(Java-COMブリッジ) *1
- AsanDatabaseBrowser
- Sylbis(シューティングゲーム)
- クリップボードのビットマップをグレイに
- JavaScreenSaver
- etc...

■ Xyzyy のアイコンを作成



*1: 日経ソフトウェア2001年10月号執筆

お品書き

- 1. 自己紹介
- 2. Scalaとは
- 3. リテラル(Literal)
- 4. 構文(Syntax)
- 5. コレクション(Collection)
- 6. 関数
- 7. クラス、オブジェクト、トレイト、パッケージ
- 8. 関数型プログラミングへ

Scalaとは1

- スケーラブルな言語
 - 小さなスクリプトから大規模なシステムまで
- JavaVM上で実行
 - Javaの膨大なライブラリが利用可能
- オブジェクト指向 + 関数型プログラミング
- 静的型付け + 型推論
- better Java !



Scalaとは2

- スイス ローザンヌ工科大学 (EPFL) で開発
 - Martin Odersky 教授 (Java の Generics 設計者)
- 最新版は Scala 2.8.1
 - 書籍は 2.7 で書かれたものが多いが 2.8 で大きな変更がいくつかあったので注意。
- ライセンス
 - Scala License (BSD っぽい)

Scalaとは3

- Twitterなどで採用。
- Lift
 - いいとこ取りのWebフレームワーク(らしい)
- 日本の書籍
 - 現在7冊(ここ2・3年で急に増えてる)



Scalaのコマンド

■ scala

- メインクラスを指定すると実行(java.exe相当)
- 引数なしだとインタプリタ

■ scalac

- コンパイラ(javac.exe相当)
- *.classファイルができる

■ fsc(Fast Scala Compiler)

- scalacの高速版(scalacをデーモン化)

■ scaladoc

- APIリファレンス生成(javadoc相当)

Hello, Scala!

■ インタプリタ版

```
scala> "Hello, Scala !"  
res0: java.lang.String = Hello, Scala !
```

■ コンパイラ版

```
$cat HelloScala.scala  
object HelloScala {  
    def main (args:Array[String]) {  
        println("Hello, Scala !")  
    }  
}  
$ scalac HelloScala.scala  
$ scala HelloScala  
Hello, Scala !
```

開発環境

■ eclipseプラグイン

- イマイチと言われてきたが、最近よくなったらしい。
(使ったことがない)

■ SBT

- ビルドツールがあるらしい。(使ったことがない)

■ Antタスク

```
<taskdef name="scalac" classname="scala.tools.ant.Scalac"
  classpath="{scala-task-classpath}"/>
<taskdef name="scaladoc" classname="scala.tools.ant.Scaladoc"
  classpath="{scala-task-classpath}"/>
<target name="build">
  <scalac classpath="{classpath}" srcdir="{src}" destdir="{bin}"/>
</target>
```

予約語

- Javaでおなじみの予約語と、新しい予約語がある。
- 記号に見える予約語がある。

```
abstract  case    catch  class  def
do        else    extends false  final
finally   for     forSome if    implicit
import    lazy   match  new    null
object    override package private protected
requires  return  sealed super  this
throw     trait  try    true   type
val       var    while  with   yield
_        :      =      =>    <<:   <%    >:    #     @
```

他言語との速度比較

- Javaより少し遅い程度。
- 他のスクリプトやC#に比べれば圧倒的に速い

	compare 2	-	-	25%	median	75%	---	-
<input type="checkbox"/> C GNU gcc	1.00	1.00	1.00	1.08	1.41	2.03	4.86	
<input checked="" type="checkbox"/> C++ GNU g++	1.00	1.00	1.00	1.14	1.41	1.51	1.51	
<input type="checkbox"/> ATS	1.00	1.00	1.09	1.32	2.80	5.36	7.17	
<input type="checkbox"/> Ada 2005 GNAT	1.00	1.00	1.04	1.33	1.98	3.39	4.71	
<input checked="" type="checkbox"/> Java 6 -server	1.37	1.37	1.47	1.75	2.28	3.50	3.58	
<input type="checkbox"/> Go	1.30	1.30	1.77	2.00	6.63	13.92	49.10	
<input type="checkbox"/> Scala	1.31	1.31	1.59	2.18	3.25	5.74	6.13	
<input checked="" type="checkbox"/> Haskell GHC	1.20	1.20	2.02	2.20	7.27	12.27	12.27	
<input checked="" type="checkbox"/> C# Mono	1.72	1.72	2.21	2.94	9.62	12.99	12.99	
<input type="checkbox"/> OCaml	1.31	1.31	2.03	3.46	4.46	7.82	7.82	
<input checked="" type="checkbox"/> Lisp SBCL	1.03	1.03	1.42	3.49	8.62	11.38	11.38	
<input type="checkbox"/> Fortran Intel	1.00	1.00	1.54	4.22	8.83	19.78	25.65	
<input type="checkbox"/> Pascal Free Pascal	1.54	1.54	2.14	4.70	7.87	16.46	23.74	
<input type="checkbox"/> Clojure	1.62	1.62	3.40	4.89	11.22	21.13	21.13	
<input type="checkbox"/> F# Mono	2.34	2.34	3.11	9.43	25.16	35.74	35.74	
<input checked="" type="checkbox"/> Erlang HiPE	4.87	4.87	6.62	13.05	20.61	41.59	52.01	
<input type="checkbox"/> Java 6 -Xint	2.27	2.27	14.21	18.90	39.29	59.19	59.19	
<input checked="" type="checkbox"/> Python 3	1.05	1.05	8.84	36.19	83.60	195.75	205.08	
<input checked="" type="checkbox"/> Ruby 1.9	3.60	3.60	11.33	78.14	113.83	267.57	333.65	

お品書き

- 1. 自己紹介
- 2. Scalaとは
- 3. リテラル(Literal)
- 4. 構文(Syntax)
- 5. コレクション(Collection)
- 6. 関数
- 7. クラス、オブジェクト、トレイト、パッケージ
- 8. 関数型プログラミングへ

リテラル1

- 論理値 (Boolean)
 - true, false
- 整数値 (Byte, Short, Int, Long)
 - 99, -100, 0xFF, 60L
- 実数値 (Float, Double)
 - 3.14, 2.2250738585072012e-308
- 文字 (Char)
 - 'A', 'あ', '¥n', '¥u0041'
- 文字列 (String)
 - "UTF-8", "Javaと同じです。"

リテラル2

- 値はJavaと同じ。ただし型は、Javaと違い、すべてクラス。
- Scalaは純粋なオブジェクト指向言語である。

リテラル3

■ Javaと違うところ(文字列)

```
// ヒア・ドキュメント。改行やエスケープ文字が可能
val sql = """update T_USER
set PATH = 'C:¥Program Files¥'
where USER_ID = '902211' """
```

```
// stripMarginを使えば、さらにキレイに
val sql = """|update T_USER
              |   set
              |     PATH = 'C:¥Program Files¥'
              |   where
              |     USER_ID = '902211' """ .stripMargin
```

リテラル4

■ シンボル

- 'Sunday '平成 'Center
- LISPのクォート、Rubyのシンボル、Cのenumみたいなモノ。あまり使わない。
- Symbol("Sunday")でも作成可能

■ Unit

- Javaのvoidとほぼ同じ。

```
def main(args:Array[String]) { ... }  
は  
def main(args:Array[String]):Unit = { ... }  
の省略形
```

お品書き

- 1. 自己紹介
- 2. Scalaとは
- 3. リテラル(Literal)
- 4. 構文(Syntax)
- 5. コレクション(Collection)
- 6. 関数
- 7. クラス、オブジェクト、トレイト、パッケージ
- 8. 関数型プログラミングへ

変数(var と val)

- varは値が何度でも代入可能な変数。
 - いわゆるフツウの変数

```
var x: Int = 10
x += 1      // OK x = x + 1 と同じ
```

- valは関数型プログラミングで使われる変数。
 - 値は再代入できない。(Javaのfinal変数)

```
val x: Int = 10
x += 1      // エラー
```

- 慣れてきたらなるべくvalを使いましょう

if式1

■ if式

- else は省略可能。
- else if でつなげることも出来る。

```
def max(x:Int, y:Int):Int = {  
  if (x > y)  
    return x  
  else  
    return y  
}
```

if式2

- if式はif文と違って、それ自体が値を返す。
 - 3項演算子と同じ

```
def max(x:Int, y:Int):Int = {  
    return if (x > y) x else y  
}
```

- 引数にも使える

```
callProcedure(  
    userId,  
    if (lang == "ja") "日本語" else "English",  
    detail)
```

match式1

■ match式

- switch文に近い。
- break文は不要。
- これも式。

```
val nengou:Char = 'S'  
val text = nengou match {  
    case 'M' => "明治"  
    case 'T' => "大正"  
    case 'S' => "昭和"  
    case 'H' => "平成"  
    case _   => "??"  
}
```

match式2

■ パターンマッチが可能

```
any match {  
  case 0 => // 定数パターン  
    "零"  
  case somethingElse => // 変数パターン  
    "???" + somethingElse  
  case Member(code, name) => // コンストラクタパターン  
    code + ":" + name  
  case List(0, x1, x2) => // シーケンスパターン(固定長)  
    format("List(0, %d, %d)", x1, x2)  
  case List(2, _*) => // シーケンスパターン(可変長)  
    "List(2, ...)"  
  case (a, b, c) => // タプルパターン  
    "3つのタプル" + a + b + c  
  case str:String => // 型付きパターン  
    str  
  case _ => // ワイルドカードパターン  
    "その他"  
}
```

※説明の都合上、すべてのパターンを1つのmatch式に書いてますが、同居できないモノ、順番が正しくないモノがあります。

match式3

■ パターンガード

– caseの中に条件が付けれる。

```
any match {  
  case dir:File if dir.isDirectory => ...  
  case dir:File => ...  
  
  case _ =>  
}
```

■ 複数の場合は「|」で。

```
any match {  
  case 1 | 2 | 3 => ...  
  case _ =>  
}
```

match式4

- 型チェックに`isInstanceOf[型]`が、型変換に`asInstanceOf[型]`が使える。

```
if (any.isInstanceOf[String]) {  
    val text = any.asInstanceOf[String]  
    ...  
} else {  
    ...  
}
```

- しかし、match式を使うのが望ましい。

```
any match {  
    case text:String => ...  
    case _ => ...  
}
```

whileループ

■ whileループ

– ただし、break, continue不可。

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

– 関数型プログラミングではwhile式はおすすめしない

```
// 上記を関数型スタイルで
args.foreach(arg => println(arg))
args.foreach(println) // 上記と同じ
args foreach println // 上記と同じ
```

do-whileループ

■ do-whileループ

- これも、break,continue不可。
- これも関数型プログラミングではおすすりめしない。

```
var line = ""
do {
    println(">")
    line = readLine()
    println("command:+line)
} while (line != ":quit")
```

for式1

■ for式

– 同様に、break,continue不可。

```
for (i ← 1 to 10) {  
    println(i)      // 1から10までが表示される。  
}
```

```
for (i ← 1 until 10) {  
    println(i)      // 1から9までが表示される。  
}
```

```
for (i ← 1 until 10 by 2) {  
    println(i)      // 1から9まで増分2で表示される。  
}
```

for式2

■ for式

- 配列、リスト、マップのとき

```
for (arg <- args) {  
    println(arg)  
}
```

- 実は `for(i <- 1 to 10)` の中の”1 to 10”とは
“(1).to(10)”と同じ

```
scala> 1 to 10  
res0: scala.collection.immutable.Range.Inclusive with scala.collection.immutable  
.Range.ByOne = Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

```
scala> (1).to(10)  
res1: scala.collection.immutable.Range.Inclusive with scala.collection.immutable  
.Range.ByOne = Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

for式3

■ for式(条件付)

```
for (file ← files if file.isFile;  
      if file.getName.endsWith(".scala")) {  
  println(file) // ファイルで、拡張子が".scala"のもののみ  
}
```

■ for式(入れ子)

```
for(i ← 1 to 9; j ← 1 to 9) print(i*j)  
  
12345678924681012141618369121518212...
```

for式4

■ for式(条件付)

```
for (file <- files if file.isFile;  
      if file.getName.endsWith(".scala")) {  
  println(file) // ファイルで、拡張子が".scala"のもののみ  
}
```

■ yieldをつけると、個々の値を置き換えたモノ (ArrayやListなど)を返す。

```
val files = new java.io.File(".").listFiles()  
val fileLengthList =  
  for (file <- files) yield {  
    file.length // ファイルサイズ  
  } // Array[Long]の値が返る
```

for式5

- 条件やループ、処理を重ねることができる。

```
def grep(pattern: String) =  
  for {  
    file ← files  
    if file.getName.endsWith(".scala")  
    line ← readLines(file)  
    trimmed = line.trim  
    if (trimmed.matches(pattern)  
  } println(file + ": " + trimmed)
```

for式6

- yieldをつけると、個々の値を置き換えたモノ (ArrayやListなど) を返す。

```
val files = new java.io.File(".").listFiles()
val fileLengthList =
  for (file <- files) yield {
    file.length // ファイルサイズ
  }             // Array[Long]の値が返る
```

例外

■ 例外

- Javaと同じ。
- catch句は match式のcaseと同じ構成
- Javaの様に関数にthrowsは書かない。
 - @throwsアノテーションで明記することは可能。

```
def writeTextFile(file:File, text:String, enc:String) {  
  var writer = null  
  try {  
    writer = new PrintWriter(  
      new OutputStreamWriter(  
        new FileOutputStream(file), enc))  
    writer.print(text)  
  } catch {  
    case ex:FileNotFoundException => // ファイルがない  
    case ex:IOException =>          // その他のIOエラー  
  } finally {  
    if (writer != null) writer.close  
  }  
}
```

お品書き

- 1. 自己紹介
- 2. Scalaとは
- 3. リテラル(Literal)
- 4. 構文(Syntax)
- 5. コレクション(Collection)
- 6. 関数
- 7. クラス、オブジェクト、トレイト、パッケージ
- 8. 関数型プログラミングへ

配列(Array)

■ Javaと同じ

```
def main(args:Array[String]) {  
    println("引数の数="+args.length)  
    println("最初の引数="+args(0))  
    val array1 = Array("1st", "2nd", "3rd")  
    ...  
}
```

– 配列よりもリストが使われる。

リスト(List) 1

■ リスト(List)

- 単方向リスト。
- LISPのCONSセルと同じ。

```
val bc = List('B', 'C')  
val abc = 'A' :: bc // List('A', 'B', 'C')  
val bcd = 'B' :: 'C' :: 'D' :: Nil // List('B', 'C', 'D')  
val abcde = abc ::: List('D', 'E') // リスト同士の結合
```

リスト(List)2

■ リストの使い方

```
abcde.head      // 'A' 先頭の要素を返す
abcde.tail      // List('B', 'C', 'D', 'E') 先頭以外のリストを返す。
abcde(1)        // 'B' 0-origin
abcde.count(e => e<'C')      // 2
abcde.drop(2)    // List('C', 'D', 'E')
abcde.dropRight(2) // List('A', 'B', 'C')
abcde.exists(e => e=='Z')    // false
abcde.forall(e => e<'Z')     // 要素すべてが条件を満たすか? true
abcde.filter(e => e<'C')    // List('A', 'B')
abcde.foreach(print)       // "ABCDE" 要素すべてに適用
abcde.isEmpty      // 空か? false
abcde.last         // 末尾の要素 'E'
abcde.length      // 要素数 5
abcde.map(e => s.toLowerCase) // List('a', 'b', 'c', 'd', 'e')
abcde.mkString(", ") // "A, B, C, D, E"
abcde.remove(s=>s<'C') // List('C', 'D', 'E')
abcde.reverse     // List('E', 'D', 'C', 'B', 'A')
abcde.sort((x, y)=>x>y) // List('E', 'D', 'C', 'B', 'A')
```

集合(Set)

■ 集合(Set)

- 集合もリストと同様に豊富なメソッドがあるが省略

```
var 動物 = Set("ねこ", "いぬ")
動物 += "にんげん"
動物.contains("イグアナ") // false
Set[String]() // 文字列の空集合
Set() ++ abcde // リストを集合に Set('A', 'B', 'C', 'D', 'E')
```

マップ(Map)

■ マップ(Map)

– マップも同様に豊富なメソッドがあるが省略

```
var nengou = Map('M' -> "明治", 'T' -> "大正",  
                'S' -> "昭和", 'H' -> "平成")  
nengou('S')    // "昭和"  
nengou('A')    // java.util.NoSuchElementException: key not found: ?  
nengou += ('J' -> "縄文")
```

タプル(Tuple)1

■ タプル(Tuple)

- 構造体やクラスを定義しなくても、複数の型の違うオブジェクトを手っ取り早く1つにまとめて扱うことができる。
- 複数の値を返すことができる。

```
val pair = (99, "Luftballons")  
println(pair._1) // 99 1オリジンなので注意
```

タプル(Tuple)2

■ タプル(Tuple)のサンプル

– パスを分解する (asanutilより)

```
/**
 パスを分割する。戻り値を結合すれば元のパスになる。
 ""C:¥dir¥dir¥dir¥filename.ext"" => ("C:", ""¥dir¥dir¥dir¥"", "filename", ".ext")
 ""C:/dir/dir/dir/filename.ext"" => ("C:", ""/dir/dir/dir/"", "filename", ".ext")
 "/dir/dir/dir/filename.ext" => ("", "", "filename", ".ext")
 "filename.ext" => ("", "", "filename", ".ext")
 "filename" => ("", "", "filename", "")
 */
def splitPath(fname:String):(String, String, String, String) = {
  省略
}

// 複数の値を返すことができる。
// 興味のない変数は_をつけるとリソースを消費しない。
val (_, _, f, e) = FileUtil.splitPath(pathname)
```

XML

- XMLが文法に取り入れられている！
 - 他の言語にはない。どうよ？

```
val xml = <hoge>ほげ</hoge>
```

- アリかも？

```
val html = <html>
  <body>
    <table>
      {
        for (rec <- reclist) yield
          <tr>
            <td><a href={rec.link}>{rec.title}</a></td>
            <td>{rec.desc}</td>
          </tr>
        }
    </table>
  </body>
</html>
```

お品書き

- 1. 自己紹介
- 2. Scalaとは
- 3. リテラル(Literal)
- 4. 構文(Syntax)
- 5. コレクション(Collection)
- 6. 関数
- 7. クラス、オブジェクト、トレイト、パッケージ
- 8. 関数型プログラミングへ

関数1

■ 可変長引数(連続パラメータ)

```
def sum(args: Int*): Int = {  
  var result = 0  
  for (arg <- args) {  
    result += arg  
  }  
  result  
}  
val answer = sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) // 55
```

関数2

■ 高階関数

- 関数を引数にしたり、あるいは関数を戻り値とするような関数のことである(Wikipediaより)。

```
def min(x:Int, y:Int) = if (x < y) x else y
def getFunction(f:String):(Int, Int)=>Int = {
  if (f == "MAX") max else min
}
val func = getFunction("MAX")    // maxを返す
func(5, 7)    // 7 を返す
```

関数3

■ カリー化(currying)

- 複数の引数をとる関数を、引数が「もとの関数の最初の引数」で、戻り値が「もとの関数の残りの引数を取り結果を返す関数」であるような関数にすること(Wikipediaより)。

```
// 引数 x を radix進数の文字列にする
def toNAdicString(radix: Int) (x: Int): String = {
  if (x < 0)
    "-" + toNAdicString(radix) (x. abs)
  else if (x < radix)
    "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ" (x). toString
  else
    toNAdicString(radix) (x / radix) + toNAdicString(radix) (x % radix)
}
toNAdicString(16) (255)          // "FF"
val toBinString = toNAdicString(2)_ // 部分適用
toBinString(255)                // "11111111"
```

遅延評価

- 必要になったときに一度だけ実行する。

```
/** 見えないカーソルを作り、登録する。*/  
lazy val NullCursor: Cursor = {  
    var image = new BufferedImage(16, 16, BufferedImage.TYPE_4BYTE_ABGR)  
    var g2 = image.createGraphics  
    g2.setColor(new Color(0, 0, 0, 0)) // 黒で透明  
    g2.fillRect(0, 0, 16, 16)  
    g2.dispose  
    Toolkit.getDefaultToolkit.createCustomCursor(image,  
        new Point(0, 0), "null_cursor")  
}
```

```
/** Javaでは実装不可能な関数 */  
def myAnd(arg1: => Boolean, arg2: => Boolean): Boolean = {  
    if (! arg1) return false  
    return arg2  
}  
myAnd(false, 1/0==0) // false. 0除算エラーにならない!  
myAnd(true, 1/0==0) // これは0除算エラーになる
```

お品書き

- 1. 自己紹介
- 2. Scalaとは
- 3. リテラル(Literal)
- 4. 構文(Syntax)
- 5. コレクション(Collection)
- 6. 関数
- 7. クラス、オブジェクト、トレイト、パッケージ
- 8. 関数型プログラミングへ

クラスとオブジェクト

■ クラスとオブジェクト

- Javaはstaticなメンバ(フィールド+メソッド)と、そうでないメンバがある。
- Scalaは非staticなメンバはclassに定義し、staticなメンバはobjectに定義する。
- Scalaの方がよりオブジェクト指向。
- 同じ名前のクラスとオブジェクトは作れる。

オブジェクト

■ オブジェクトのサンプル

- 定数やmain()、apply()はオブジェクトに定義。
- Javaでいうところのstaticなメンバを定義。

```
object Foo {  
  /* 定数. 先頭が大文字のキャメルスタイルで書く */  
  val Center = 0  
  
  /** List('A', 'B') は List.apply('A', 'B') と同じ */  
  def apply(...) {}  
  
  /** エントリ */  
  def main(args:Array[String]) {  
  }  
}
```

クラス1

■ クラスのサンプル1

- クラスに本体がなければ波括弧を省略可能
- 基本コンストラクタ

```
/** 有理数 n:分子, d:分母 */  
class Rational(n:Int, d:Int)
```

- クラス内部の処理=コンストラクタの処理

```
/** 有理数 n:分子, d:分母 */  
class Rational(n:Int, d:Int) {  
    println("コンストラクタ内:"n+"/"+d)  
    def this(n:Int) = this(n, 1)    // 補助コンストラクタ  
}
```

クラス2

■ クラスのサンプル2

```
/** 有理数 (numer:分子, denom:分母) */  
class Rational(n:Int, d:Int) {  
  require(d != 0)  
  private val g = gcd(n.abs, d.abs)  
  val numer = n / g  
  val denom = d / g  
  def this(n:Int) = this(n, 1) // 補助コンストラクタ  
  def +(that:Rational):Rational =  
    new Rational(  
      numer * that.denom + that.numer * denom,  
      denom * that.denom)  
  override def toString = numer + "/" + denom  
  private def gcd(a:Int, b:Int):Int =  
    if (b == 0) a else gcd(b, a % b) // 最大公約数  
}
```

トレイト(trait)1

■トレイトとは

- 特徴、特性、形質という意味。
- メソッドとフィールドの定義をカプセル化。
- クラスにミックスインできる。
- Javaのinterfaceよりも強力

トレイト2

■トレイトの使い方

```
package scala.math
trait Ordered[A] extends java.lang.Comparable[A] {
  def compare(that: A): Int
  def < (that: A): Boolean = (this compare that) < 0
  def > (that: A): Boolean = (this compare that) > 0
  def <= (that: A): Boolean = (this compare that) <= 0
  def >= (that: A): Boolean = (this compare that) >= 0
  def compareTo(that: A): Int = compare(that)
}
class Rational(n:Int, d:Int) extends AnyRef with Ordered[Rational] {
  // これ1つですべての比較演算子ができる。
  def compare(that:Rational) =
    this.numer * that.denom - that.numer * this.denom
}
```

トレイト3

■ その他のトレイト

– Map, Set, Seq, Iteratorなど

- Listクラスの大半のメソッドは trait LinearSeq で定義されている。

- トレイトを組み合わせることにより多機能なクラスが容易に作成可能。

– scala.swing.Component

- 前景色、背景色、minimumSize/ maximumSize/ preferredSize、フォント、位置、サイズ、カーソル、visibleなどが trait UIElementで定義されている。

パッケージ(package)

■ Javaとの違い

- 1ファイルに複数のパッケージの定義が可能
- Javaの1ファイル=1クラスの制約がない

```
// Javaと同様に先頭で宣言する
package org.sourceforge.asanutil

// パッケージの範囲を指定することができる。入れ子も可能。
package org.sourceforge.asanutil {
    package util {
        ...
    }
    class Foo {
        ...
    }
}
```

インポート(import)1

■ インポートの使い方

```
import scala.collection.mutable.ListBuffer
import scala.swing._ // Javaの * と同じ
import java.io. {File, IOException}
import java.sql. {Date => SqlDate} // 別名を付けれる
import java.awt. {List => _, _} // java.awt.List以外をインポート

def getImage(file:File):Image = {
  // 使いたいときに使える
  import javax.imageio.ImageIO
  ImageIO.read(file)
}
```

インポート(import)2

■ 暗黙のインポート

```
import java.lang._ // java.langパッケージ
import scala._     // scalaパッケージ
import Predef._    // scala.Predefオブジェクト
```

■ scala.Predefには便利なメソッドがある

```
assert(式, 値) // AssertionError
require(式)   // IllegalArgumentException
char2int('A') // 65
classOf[Foo]  // Foo.classを返す
println()
print()
printf()
readLine()   // System.in を BufferedReaderにして readLine()
readInt()    // readLine()してIntに変換
```

お品書き

- 1. 自己紹介
- 2. Scalaとは
- 3. リテラル(Literal)
- 4. 構文(Syntax)
- 5. コレクション(Collection)
- 6. 関数
- 7. クラス、オブジェクト、トレイト、パッケージ
- 8. 関数型プログラミングへ

命令型プログラミング1

- 命令型プログラミングとは
 - 逐次実行(上から下へ時系列で順番に)
 - 条件分岐(if文, switch文)
 - 反復処理(while文、for文)

命令型プログラミング2

■ 命令型プログラミングの問題点

– よくある処理だが、部品化できない。

```
/** 命令型プログラミングでよくある処理 (Javaで記述) */  
Image findImage(File[] files) {  
    for (File file: files) {  
        if (file.getName().endsWith(".png")) {  
            return ImageIO.read(file);  
        }  
    }  
    return null;  
}
```

命令型プログラミング3

- 命令型ではうまく部品化できない。
 - 処理により変わる部分が「？」の箇所。
 - インターフェースや型引数を駆使すれば出来なくもないが、ものすごく冗長。

```
/** 命令型では部品化できない */  
??? find(??? args) {  
    for (??? arg: args) {  
        if (??? arg) {  
            return ???(arg);  
        }  
    }  
    return null;  
}
```

命令型プログラミング4

- 関数型ならうまく部品化できる。
 - Scalaではすでにメソッドとして用意されている。

```
/** 関数型なら部品化できる。Scalaで部品を利用した例 */  
files.find{ file => file.getName().endsWith(".png") } match {  
  case Some(file) => ImageIO.read(file)  
  case None => null  
}
```

関数型プログラミングとは

- 関数も整数や文字列と同格(FirstClass:一人前)である。
 - 関数を引数に渡したり、関数を戻り値として扱うことが出来る。
 - 関数の中でローカル関数を定義できる。
 - 整数や文字列が値だけで扱えるのと同様、変数に代入せずに関数が見える。
- 副作用を持つべきではない。
 - 引数だけを参照し、それ以外を参照・更新せずに、戻り値だけを返す。
 - 内部状態を持たない。
 - 画面、キー入力、ファイル入出力も副作用になる。

関数型プログラミングとは

■ 副作用クイズ

– Q: `java.lang.Math`クラスのメソッドはほとんどが関数型であるが、そうでないメソッドがある。何か？

■ ヒント1: 内部状態を持ち、引数と戻り値が呼ぶごとに一定でないモノ。

■ ヒント2: この中のどれかにある。

– `abs`, `cos`, `exp`, `floor`, `log`, `max`, `min`, `random`, `round`

お品書き

- 1. 自己紹介
- 2. Scalaとは
- 3. リテラル(Literal)
- 4. 構文(Syntax)
- 5. コレクション(Collection)
- 6. 関数
- 7. クラス、オブジェクト、トレイト、パッケージ
- 8. 関数型プログラミングへ