㈱バリューソース　　神崎　善司

JAVAプログラマーのための ＳＣＡＬＡ入門

かんざき　とは

- ㈱バリューソース　代表取締役
  - オブジェクト指向をキーワードにサービスを提供
- 普段は
  - 要件定義の支援
  - オブジェクト指向やUMLのセミナー開催
- Scalaとの関係は
  - Smalltalk～PrographCPX～Java～
  - 数年前から関数型の言語に着目
  - Haskellで挫折　　F#　ん～～～
  - OOと関数型を融合した言語を探していた
  - ３年ほど前にScalaを発見
  - 要件定義用のツールをScalaでサンデープログラマーとして作成している

Scalaとは1

- オブジェクト指向＋関数型言語＋LL言語の記述力を合わせ持つ言語
- Scalable language
  - 「オブジェクト指向と関数型言語のコンセプトを統合することがスケーラブルな言語には必要」という仮説に立っている
- 関数型言語のフレーバー
  - 不変クラスを作りやすい定義
  - 関数型として便利なクラスの提供List、Option....
- 型付き言語
  - 一貫性のあるGenerics
  - Javaよりも型チェックが厳しい
- JVMの上で動き　Javaと親和性の高い言語

Scalaとは2

- 開発環境
  - Scala SDK　　scalac.bat
  - Eclipse　　Plug-in
  - NetBeans6.5　Plug-in
  - インタープリター
    - コンソール実行　　scala
    - ソースファイルから実行　　scala ~~.scala

- 実行環境
  - Scala　SDKから実行
    - scala
  - JREから実行
    - 「scala-library.jar」をクラスパスに追加
    - 「 scala-compiler.jar 」をAntに組み込んでScalaタスクを組むことでコンパイル可能

Javaとどう違うの

	Java	Scala
変数	int i = 0;	var i:Int = 0; // var i = 0
変数参照用	final int i = 0;	val i:Int = 0; // val i = 0
クラス	class Foo {	class Foo {
メソッド	public int getName() { public void setAge(int i) {	def getAge() : Int = { def setAge(i:Int){

コンストラクター	class Foo{
	  public Foo(String name, int age){
	  public Foo(String name) {
		class Person(name: String, age: Int){
		  def this(name: String) {
		  this(name, 1);
		  }

継承	class Foo extends A{	class Foo extends A{
インターフェース	interface Foo {	trait Foo {
Generics	<String>List	List[String]

エントリーポイント	class A{
	  public void main(String arg[]){
		object A{
		  def main(arg:Array[String]){

パフォーマンス ベンチマーク

x64 Ubuntu :
Intel® Q6600®
Computer Language Benchmarks Game

x	language	mean
1.0	C++ GNU g++	1.47
1.1	ATS	1.55
1.1	C GNU gcc	1.68
1.2	Fortran Intel	1.81
1.3	Haskell GHC	1.96
1.3	Java 6 -server	1.98
1.4	Clean	2.01
1.5	OCaml	2.28
1.6	Lisp SBCL	2.36
2.0	Scala	2.99
2.2	Pascal Free Pascal	3.22
2.5	C# Mono	3.74
2.6	Ada 2005 GNAT	3.88
4.2	Erlang HiPE	6.18
5.1	Scheme PLT	7.48
8.6	Java 6 -Xint	12.70
8.8	Lua	12.96
9.7	Smalltalk VisualWorks	14.36
11	Python	16.03
19	Perl	27.55
19	PHP	27.97
21	Ruby JRuby	31.36
39	Ruby	56.78

Scalaの文法　統一性

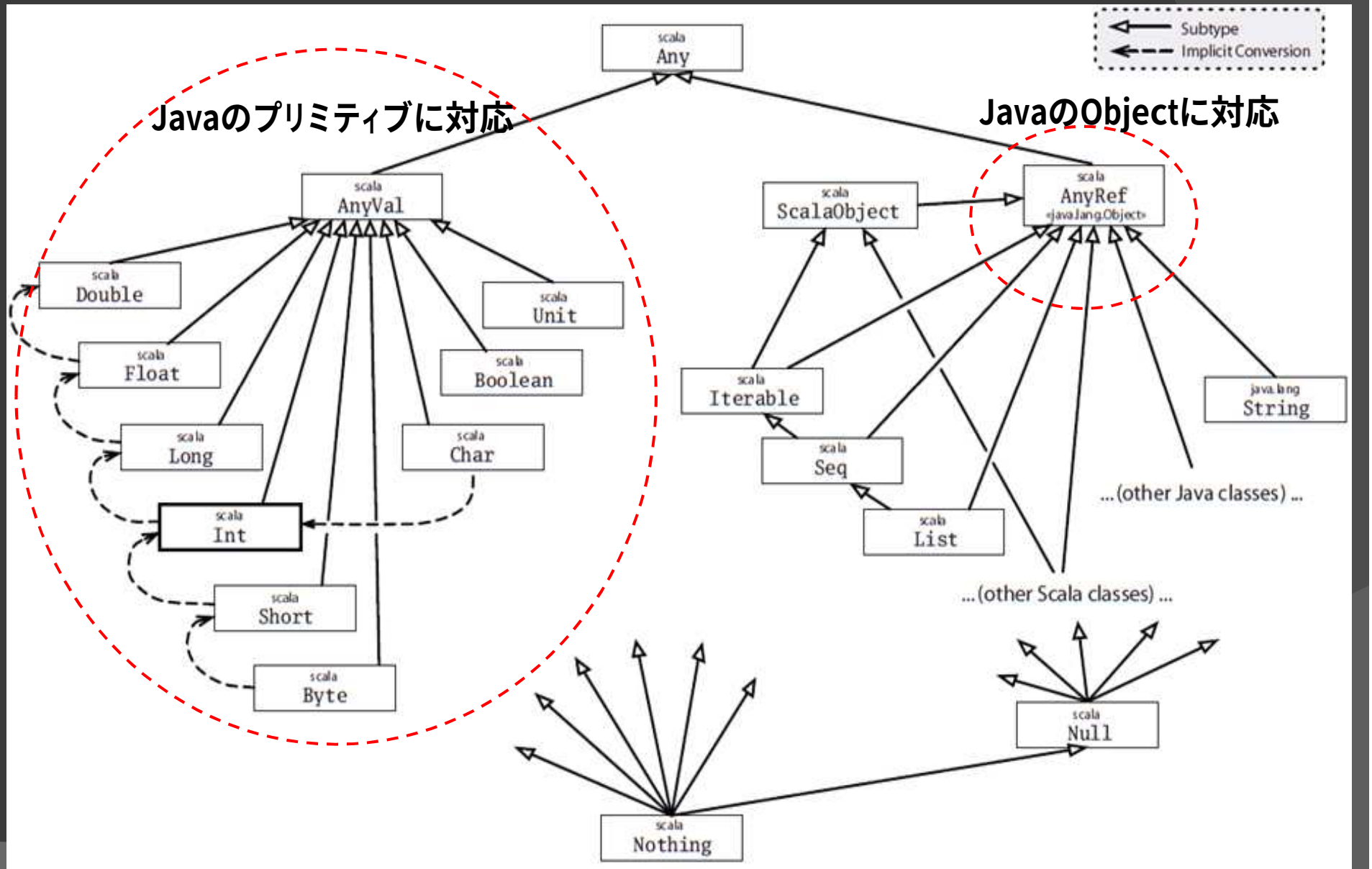
- 全てがオブジェクト
  - Javaのプリミティブ（int long boolean...）もオブジェクト
  - メソッドもオブジェクト
  - ScalaのライブラリはJavaのクラスライブラリを拡張
- 基本構文もメソッド
  - 3＋4
  - 3.+(4)
    - def +(i:Int){
  - val a = array(4)　　//配列の要素の参照
    - array. apply(i:Int)
  - array(4) = 0　　//配列への値のセット
    - array. update(i:Int, v:Any)
- staticが無い
  - staticをシングルトンオブジェクトとして実現
  - staticの代わりにobjectキーワードを利用

```
scala> 1 + 3
res0: Int = 4

scala> 1 .+(3)
res2: Int = 4
```

クラスライブラリ構成

Programing in Scalaより



Scalaの文法　型システム1

- Generics
  - **ほぼJavaと同じように使える**
    - Java　　<String>List
    - Scala　　List[String]
  - **型を限定する継承関係の表現**
    - class A[X <: ABC]{　　**XはABCクラスのサブクラス**
    - class B[X >: ABC]{　　**XはABCクラスのスーパークラス**
  - **Javaよりも一貫性がある**
    - **JavaではRuntimeエラーになる定義もScalaでは静的にチェック可能**
  - **型エラーを取るのが難しい場面もある**
    - **型パラメータまで含めた継承関係の考え方が難しい**
    - covariant　contravariant　nonvariant....

- **型推論**
  - **型を推論し型定義を省くことができる**
    - **変数定義、返り値の型、型パラメータ**
  - **メソッドのパラメータは型指定を省略できない**
  - **例えば**
    - val a = 0　　a**:Int**　Intの型指定が省略できる
    - def abc(i**:String**)=　**返り値の型指定が省略できる**　　def　abc(i:String):Int =
    - List(1,2,3)　　**型パラメータの省略**

Scalaの文法　型システム2

- typeキーワード
  - 型に別名（参照）を付けられる
  - 抽象型を定義できる

```
scala> class A{ type T; val l = List[T]()}
defined class A

scala> class B extends A{type T = Int}
defined class B
```

- ダッグタイピング
  - あるメソッド、属性を持ったクラスを任意の型として扱える

•dxという参照をもつクラスをdxとする
•abcメソッドのパラメータとして上記のクラスを指定する
•dxという参照をもつオブジェクトAdxを定義する
•abcメソッドにAdxを渡す

```
scala> type dx = {def dx:String}
defined type alias dx

scala> def  abc(f:dx) = println(f.dx)
abc: (dx)Unit

scala> object  Adx{def  dx = "Scala"}
defined module Adx

scala> abc(Adx)
Scala
```

Scalaの文法　メソッド1

- メソッド定義の幾つかの方法
  - def plus(i:int) = i + 1　　通常のメソッド定義
  - val plusX = plus _　　メソッドを変数に代入
  - val plus = (i:int) => i + 1　　クロージャーを定義し変数に代入
- メソッドはオブジェクト
  - メソッドはFunction 〜 FunctionNクラスとして実現される
  - 従ってメソッドをクラスとして拡張可能
- クロージャー
  - 入力値の型 => 返り値の型
  - Int => Int
  - List(1,2,3).map((a)=>a+100)

```
class  List[A]{
      def count (p : (A) => Boolean) : Int
```

```
scala> List(1,5,10,3).count(a=>a<5)
res2: Int = 2
```

- 内部関数（メソッドの中にメソッドを定義可能）
  - def total(l:List[Int])={
  - 　　def toal(

Scalaの文法　メソッド2

- **遅延評価（lazy）**
  - **呼び出された時に評価される**
  - lazy val a = abc(〜〜
  - def efg(a: ~){
  - 　if (~~) return
  - 　a.xxxx
- **特殊文字も使用可能**
  - @ - > < / - :: 〜〜
- **ブロックによるメソッドの定義**

```
scala> class A(var v:Int){def +(i:Int) = v += i}
defined class A

scala> val a = new A(10)
a: A = A@9c15d2

scala> a+100

scala> a.v
res6: Int = 110
```

```
val d = {new java.util.Date()}

scala> d
res6: java.util.Date = Sat Nov 22 07:44:06 GMT 2008
```

Scalaの文法　多重継承

- 多重継承の実現
  - traitを「with」キーワードを使って多重継承できる
  - class A extends B with C with D{　～
    - クラスAはクラスBを継承しtrait　C,Dを多重継承する
  - object X extends Y with Z{　～
    - オブジェクトXはクラスYを継承しtrait　Zを多重継承する
  - val a = new A with C
    - クラスAを生成するときにtrait　Cを継承
    - オブジェクト生成時に継承可能
  - Scalaのコレクションクラスは多重継承を多用している
- trait
  - Javaのinterfaceよりも強力
    - メソッドの実体を定義可能
    - 属性も定義可能
  - 機能はほぼ abstract class と同じ
  - コンストラクタを定義できない

```
trait A {
  def abc ~~
  val a =
}
```

Scalaの文法　object

- objectの意味的な使い分け
  - デザインパターンのシングルトンの働き
  - モジュールとして扱う
    - sort(~~
    - xxx.sort(~~
  - 変数定義の代わり
  - Factory用のクラスとして使用
    - object　List　　ファクトリー用定義
      - List(1,2,3)

Singlton＆拡張

```
object A extends B with C{
  def abc ~~
}
```

モジュール

変数＆定義

```
object SwingApp extends SimpleGUIApplication {
  def top = new MainFrame {
    title = "SwingApp"
    var numclicks = 0
    object label extends Label {
      val prefix = "Number of button clicks: "
      text = prefix + "0  "
      listenTo(button)
      reactions += {
```

Scalaの文法　case class/case object

- Scala独特なキーワード[case]
  - 目的
    - 関数型のプログラミングに適したようにclassを最適化したもの
      - コンストラクタで定義したものがそのままval 定義と見なされる
      - オブジェクト内の変数の値によるequalsの判定メソッドが定義される
    - Classの記述を簡略化できる
  - 働き
    - パターンマッチングに必要なメソッドを暗黙のうちにもつ
      - apply　　unapply メソッド
    - コンストラクタで変数の定義が可能
    - 値によるイコール判定(==)

```
abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: int) extends Expr
case class BinOp(operator: String,left: Expr, right: Expr) extends Expr
```

case class の働きを試す

```
abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: int) extends Expr
case class BinOp(operator: String,left: Expr, right: Expr) extends Expr
```

- イコール判定を試す
- newのいらないオブジェクト生成

```
scala> val v = Var("x")
v: Var = Var(x)

scala> v == Var("x")
res0: Boolean = true
```

- 複合オブジェクトの作成とパターンマッチング

```
scala> val op = BinOp("+", v,Number(0))
op: BinOp = BinOp(+,Var(x),Number(0))

scala> op match {
  case BinOp("+", e, Number(0)) => e
  case BinOp("*", e, Number(1)) => e
  case _ => op
}
res2: Expr = Var(x)
```

case class が暗黙のうちに行っていること

```
abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: int) extends Expr
```

classとobjectを使ってcase classと同じ事をする

```
abstract class Expr
class Var(val  name: String) extends Expr{
    override def equals(obj:Any) = obj match{ //値によるイコール判定
      case o:Var => o.name == name
      case _=> false
    }
    override def toString() = “Var(”+name+“)”; //クラス名を含めた値の表示
}
class Number(   ～～
~~
object Var extends Expr{
    def apply(name: String) = new Var(name) //newの代わりとなるFactorメソッド
    def unapply(exp:Var) = Some(exp.name)  //パターンマッチング用の指定
}
```

ScalaのLL言語のような強力な記述力

記述力の向上

記述力の向上1

- **記述力**
  - Ruby、Python**のような**Light weight**な言語と同様な表現方法が可能**
- **省略記述**
  - val a:Int = 0
  - val a = 0;
  - List(1,2,3)
  - abc efg
  - object toString
  - List(1,2,3) drop 1
  - -> /: << >>

文末の「;」の省略
型推論　変数の型定義を省略
型推論　型パラメータの省略　List[A]
abc.efg()　**カンマを空白で代替え**
括弧の省略が可能
.drop(1) **括弧の省略**
特殊文字の利用

```
scala> 123->"jjj"
res2: (Int, String) = (123,jjj)
```

- **クロージャの簡易表現**
  - **アンダーバーでパラメータを表現**
  - “aaHHbb”.filter(_.isUpperCase)
    - →　filter((c:Char) => c.isUpperCase)

記述力の向上2

- パターンを使った値の割り当てる

```
scala> val Some(a) = Some(123)
a: Int = 123
```

- 高機能なfor文

```
scala> for(i <- 0 to 5) print(i+" ")
0 1 2 3 4 5

scala> for( i <- List(1,2,3); if i != 2) print(" "+i )
 1 3

scala> for( i <- List(1,2,3); j <- 0 to 3) print(" "+i+" "+j )
 1 0 1 1 1 2 1 3 2 0 2 1 2 2 2 3 3 0 3 1 3 2 3 3

scala> for( i <- 0 to 1; j <- List("AA","BB")) yield i -> j
res4: Seq.Projection[(Int, String)] = RangeG((0,AA), (0,BB), (1,AA), (1,BB))
```

記述力の向上3

- XML
  - Scalaの構文として直接XMLを記述できる
    - <タグ>{ Scalaの構文を記述 }</タグ>

```
scala> def  x(s:String) = <abc><efg>{s}</efg></abc>
x: (String)scala.xml.Elem

scala> x("hhhh")
res0: scala.xml.Elem = <abc><efg>hhhh</efg></abc>
```

  - XPathのような機能も提供
    - ＼　　バックスラッシュ

```
scala> val x = <a><b><c>ddddd</c></b></a>
x: scala.xml.Elem = <a><b><c>ddddd</c></b></a>

scala> x \ “b“          //  \  バックスラッシュ
res14: scala.xml.NodeSeq = <b><c>ddddd</c></b>
```

記述力の向上4　XMLパターンマッチング

```
def print(nd:Node){
    def abc(nds:Seq[Node]){ nds.foreach(_ match{
            case <a1>{node @ _*}</a1> => {println("A1 -> "); a(node)}
            case <a2>{node @ _*}</a2> => {println("A2 -> "); a(node)}
            case _ => nds})
    }
    def a(nds:Seq[Node]){ nds.foreach(_ match{
            case <b1>{node @ _*}</b1> => {println("B1 -> "); b(node)}
            case <b2>{node @ _*}</b2> => {println("B2 -> "); b(node)}
            case _ => nds})
    }
    def b(nd:Seq[Node]) =
             println("text -> "+nd.text)
    abc(nd.child)
}
print(xml)
```

```
val xml =
  <abc>
      <a1>
        <b1>A1_BBBB111</b1>
        <b2>A1_BBBB222</b2>
      </a1>
      <a2>
        <b1>A2_BBBB111</b1>
        <b2>A2_BBBB222</b2>
      </a2>
  </abc>
```

Javaとの親和性　Swingプログラム

```
class CheckNavigationPanel extends javax.swing.JPanel {
    val scrollPane = new javax.swing.JScrollPane();
    val checkTbl = getNewTable();
    initComponents();
```

JPanelの継承
内部コンポーネントの変数定義

```
    def initComponents():unit = {
        scrollPane.setViewportView(checkTbl);
        checkTbl.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        checkTbl.setRequestFocusEnabled(false);
        ~~
        for(i <- 0 to checkTbl.getModel().getRowCount-1){
            val cmp = checkTbl.getCellEditor(i,0).asInstanceOf[DefaultCellEditor].
                        getComponent().asInstanceOf[JCheckBox];
            val dig = checkTbl.getModel().asInstanceOf[CheckTblMdl].diagrams(i);
            val clm = checkTbl.getModel().asInstanceOf[CheckTblMdl].columns(i);
            cmp.addActionListener(
                new ActionListener(){
                    val clmNm = clm;
                    val row = i;
                    def actionPerformed(e:ActionEvent):unit =
                        MainView.getTab(dig).makeChack(isChaeck(row),clmNm)
        ~~
```

内部コンポーネントの初期化

イベントリスナーの登録

イベントのハンドリング

ほとんどJavaのSwingプログラムと一緒

関数型としての特徴1

- カリー化
  - パラメータの一部を解決して新たなメソッドを作る

```
scala> def plus( i:Int)( j:Int) = i+j
plus: (Int)(Int)Int

scala> plus(10)(100)
res10: Int = 110

scala> def plus1 = plus(1)_
plus1: (Int) => Int

scala> plus1(100)
res11: Int = 101
```

関数型としての特徴2

- 関数合成
  - メソッドを合成して新たなメソッドを作る
    - andThenメソッドによるメソッドの合成
    - 二つのメソッドを組み合わせて一つのメソッドを作る

```
scala> def  a = (x:int)=>x+100;
a: (int) => Int

scala> def  b = (x:int)=>x + "jjj";
b: (int) => java.lang.String

scala> b(a(100));
res0: java.lang.String = 200jjj

scala> def  c = a andThen b;
c: (int) => java.lang.String

scala> c(100);
res7: java.lang.String = 200jjj
```

関数型としての特徴3

- 関数型言語の視点から
  - Scalaは宣言的ではない
    - 上から下に流れる制御構文を認める
  - Scalaは副作用を認める
    - 変更可能な変数をもつ
    - しかし不変オブジェクトを作りやすい構造を備えている
  - 参照透過性は保証しない
    - 参照透過性：パラメータの値が同じものは同じ値を返す
- Scalaライブラリにモナドを使ったクラスを用意
  - Option
  - for文

```
scala> List(1,5,10,11).find(_>5)
res18: Option[Int] = Some(10)

scala> List(1,5,10,11).find(_>5).getOrElse(0)*100
res19: Int = 1000
```

- パターンマッチング用構文　　次ページ以降

パターンマッチング１　case class

```
case class Elm
case class A(a:Int) extends Elm
case class B(b:String) extends Elm
case class Ent(a:Elm,b:Elm)
```

```
scala> val e @ Ent(A(f),B(g)) = Ent(A(1),B("kkkk"))
e: Ent = Ent(A(1),B(kkkk))
f: Int = 1
g: String = kkkk
```

分解してf,gに値を割り当てる
「@」を使って「e」には全体が割り当てる

```
def pm(m:Ent) = m match{
  case Ent(A(1),b) => println("Ent(A(1),b):"+b)
  case Ent(a,B("kkk")) => println("Ent(a,B(kkk))):"+a)
  case Ent(a,b) => println("Ent(a,b)"+a+" / "+b)
}
```

```
pm(e)
Ent(A(1),b):B(kkkk)
pm: (Ent)Unit
```

組み合わせのパターンマッチング

パターンマッチング2　変数定義

```
scala> val (p1,p2) = (11,22)
p1: Int = 11
p2: Int = 22

scala> p1 = 200
<console>:23: error: reassignment to val
       p1 = 200
          ^

scala> var (p3,p4) = Pair(10,"jjj")
p3: Int = 10
p4: java.lang.String = jjj

scala> p3 = 100
p3: Int = 100
```

分解してp1,p2に割り当てる
(11,22) はPair(11,22)と見なされる

valで定義されているので代入はエラー

varで定義されているので再代入可能

パターンマッチング3　for　Some

```
scala> val l = List((10,20),(100,200),(300,400))
l: List[(Int, Int)] = List((10,20), (100,200), (300,400))

scala> for( ( i , j ) <- l ) println( i+j )
30
300
700

scala> val sl = List( None, Some(10), Some(100), None)
sl: List[Option[Int]] = List(None, Some(10), Some(100), None)

scala> for( Some(o2) <- sl ) println(o2)
10
100

scala> val Some(oo) = sl(1);
oo: Int = 10
```

Some NoneはOption型
Collectionのfindの返値などに利用される

Someが選ばれる

Someの中身を取り出す

パターンマッチング４　Collection

Collectionのパターンマッチング

```
def m(lst:List[Int]) = lst match{
    case a::Nil    => println(a)
    case a::b::Nil => println(a+" / "+b)
    case a::b::c   => println(a+" / "+b+" / "+c)
    case _         => println("null list")
}
```

```
scala> m(List(1,2,3,4,5))
1 / 2 / List(3, 4, 5)        //３番目にヒット
m(List(1,2))
1 / 2                        //２番目にヒット
scala> m(List(1))
1                            //１番目にヒット
```

Collectionの分解

```
scala> val  a :: b :: c = List(1,2,3,4,5,6)
a: Int = 1
b: Int = 2
c: List[Int] = List(3, 4, 5, 6)
```

パターンマッチング5　型特定

- 特定のオブジェクトの型を識別する
  - 特定の型のメソッドを呼び出すため

```
class D{def d = 100}
object Do1{def d = "DDD"}
object Do2{def dx = "Do2"}
case class Dcc(d1:Int)
type T = {def dx:String}

def d_match(a:Any) = a match{
  case v:D => println("d_match"+v.d)              //class
  case v:Do1.type => println("d_match"+v.d)       //object
  case v @ Dcc(a) => println("d_match"+v.d1)      //case
  case v:T => println("d_match"+v.dx)             //type T
}
```

implicit conversion

- implicit　conversionとは
  - Scalaの型変換のためのメカニズム
  - このメカニズムを使うことで既存のクラスにメソッドを追加するような働きをさせることができる

- 働き
  - 既存のクラスにメソッドを追加する
    - クラスに定義していないメソッドを呼び出した場合に、型変換用のメソッドを呼び出し、そのメソッドが定義しているクラスに型変換する
  - パラメータの型変換
    - 仮引数と実引数の型が合わない時に型変換用のメソッドを呼び出す
  - パラメータのデフォルト設定
    - パラメータが省略された場合に、型変換用のメソッドを呼び出し、その返り値を実引数として使用する

implicitの使い方

intにメソッドを追加する

```
class WrapI(i:int){def happyString = "happy number:"+i}

implicit def int2wrapi(i:int):WrapI = new WrapI(i)

1 happyString;
```

happyStringメソッドを定義したクラス

IntからWrapIクラスに型変換するメソッド

パラメータの型変換

```
def hap(wp:WrapI) = wp.happyString

hap(10)
```

暗黙のうちにWrapIクラスに型変換され
happyStringメソッドが呼ばれる

暗黙のうちにintからWrapIクラスに型
変換され、そのオブジェクトが渡される

パラメータのデフォルト設定

```
implicit val wrapi:WrapI = new WrapI(100)
def hap2(implicit wp:WrapI) = wp.happyString

hap2
```

パラメータが省力されたのでデフォルトに置き換えられた

implicitを安全に使うには？

- import文を使って安全にimplicitを使う

```
object ImplicitTest {
    class WrapI(i:int){def happyString = "happy number:"+i}
    implicit def int2wrapi(i:int):WrapI = new WrapI(i)
}

object Main extends Application{
    def happyInt = {
      import ImplicitTest._;
      val a = 1 happyString;
      val b = 1 to 100;
    }
    def normalInt = 1 to 100;
//  val aa = 1 happyString;
}
```

WrapIクラスにラップされる

この範囲でimportが有効になる

WrapIクラスにラップされない
のでコンパイルエラーになる

コレクション

- **組み込み型コレクション**
  - List(1,2,3)
  - Array(2,3,4)　　⇒　JavaのArray（[]）と互換性がある
  - (1,”jjj”,23.3)　　⇒　異なる型を格納可能なタプル
  - Map("US" ->”Washington","Japan" ->"tokyo")

- **便利な操作**

```
scala> List(1,2,3).foldLeft(0)(_+_)
res5: Int = 6

scala> "AAbbbCCCfffdd".map(_.toUpperCase)
res6: Seq[Char] = ArrayBufferRO(A, A, B, B, B, C, C, C, F, F, F, D, D)

scala> 1 :: 2 :: 3 :: 4 :: List(5,6,7)
res15: List[Int] = List(1, 2, 3, 4, 5, 6, 7)

scala> List(1,2,3) ::: List(1,2,3)
res18: List[Int] = List(1, 2, 3, 1, 2, 3)
```

情報源

- Webサイト
  - Scalaの本家サイト
    - http://www.scala-lang.org/
  - 日本のScalaコミュニティ　ぜひMiixiiコミュへご参加を
    - Mixi Scalaコミュ
      http://mixi.jp/view_community.pl?id=3111016
  - お勧め：A Tour of Scala
    - http://www.scala-lang.org/node/104
  - 仕様Scala Language Specification
    - http://www.scala-lang.org/docu/files/ScalaReference.pdf
- Book
  - Programming in Scala
    - http://www.artima.com/shop/programming_in_scala

まとめ

- Scalaの魅力
  - 厳密な型付き言語
    - 一貫性のあるGenerics
  - ＬＬ言語のような簡易表現が可能
  - Javaとの高い親和性
    - Javaの豊富なライブラリが活用できる
  - オブジェクト指向にほどよく統合された関数型言語機能
- 未来のJavaの姿を今体験することができます。