Patterns Matching

© 2008 Haim Michael 20160119

Introduction

- The Patterns Matching is one of Scala's constructs that assist us when working with data structures.
- Patterns matching is the familiar case statement we know from C\C++\Java\C#\PHP. Unlike the case statement, it isn't limited to matching against specific values.

© 2008 Haim Michael 20160119

Match Statement

- The match expression functions similarly to the switch statement in Java.

```
selector match
{
   alternatives
}
```

- The selector is the expression we want to try to match with the alternatives.

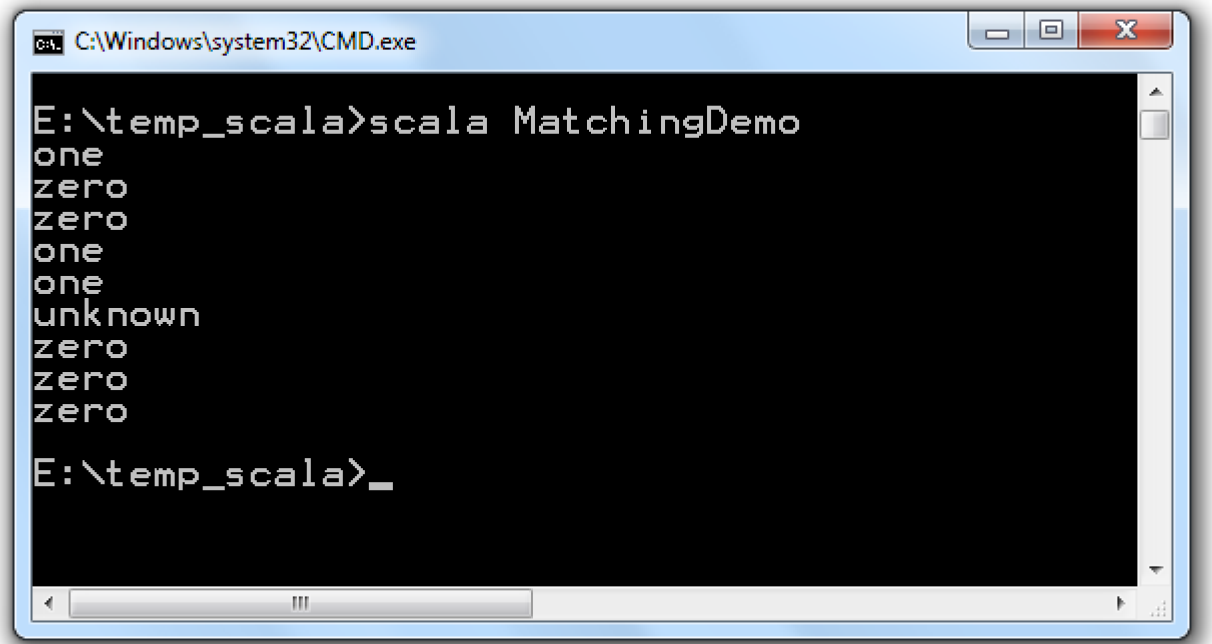
Match Statement

- Doing a patterns match we compare our expression with a sequence of alternatives.
- Each alternative starts with the keyword `case`. Each alternative includes a pattern and one or more expressions that will be evaluated when the pattern matches.
- The arrow symbol => separates the pattern from the expression.

Match Statement

```
object MatchingDemo
{
    def main(args: Array[String])
    {
        val numbers = List(1,0,0,1,1,7,0,0,0)
        for (num <- numbers)
        {
            num match
            {
                    case 1 => println("one")
                    case 0 => println("zero")
                    case _ => println("unknown")
            }
        }
    }
}
```

Match Statement



```
E:\temp_scala>scala MatchingDemo
one
zero
zero
one
one
unknown
zero
zero
zero

E:\temp_scala>_
```

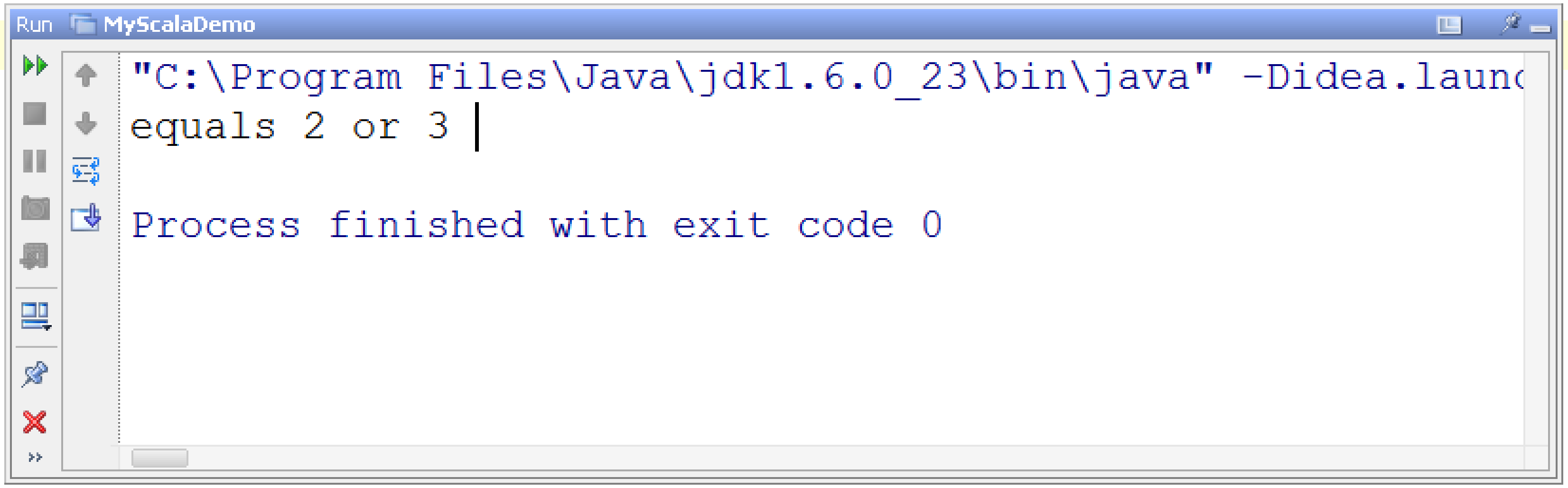
Logical Operators

- We can use comparison operators, such as | , in order to define multiple cases as one.

Logical Operators

```
package com.abelski.samples

object MyScalaDemo extends Application
{
  def myfunc(num:Int)
  {
    num match
    {
        case 2 | 3 => println("equals 2 or 3")
        case _ => println("all other cases")
    }
  }
  myfunc(2)
}
```

Logical Operators



```
"C:\Program Files\Java\jdk1.6.0_23\bin\java" -Didea.launc
equals 2 or 3

Process finished with exit code 0
```

Typed Pattern

- When dealing in Java with an object we don't know its type we need to use a series of if-else statements and instanceof casts in order to check the exact type of our object before moving forward with casting the type of the reference we hold in order to invoke the relevant method.
- Scala allows us to use patterns matching for processing different code segments in according with the type we are dealing with.

© 2008 Haim Michael 20160119

Typed Pattern

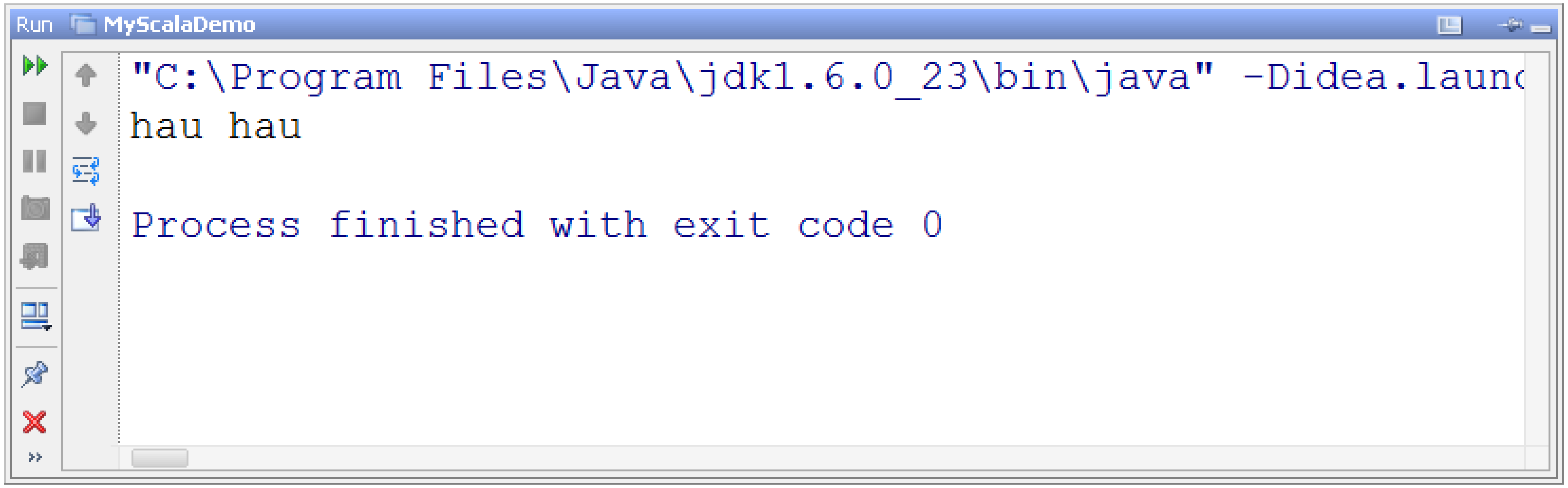
```
package com.abelski.samples

object MyScalaDemo extends Application
{
  def sayHello(ob:AnyRef) =
  {
    ob match
    {
      case ob:Cow => ob.moo()
      case ob:Dog => ob.hau()
      case ob:Cat => ob.miau()
      case _ => println("hello")
    }
  }
  sayHello(new Dog())
}
```

Typed Pattern

```
class Dog
{
  def hau()= println("hau hau")
}

class Cat
{
  def miau() = println("miau miau")
}

class Cow
{
  def moo() = println("moooo moooo")
}
```

© 2008 Haim Michael 20160119

Typed Pattern



```
"C:\Program Files\Java\jdk1.6.0_23\bin\java" -Didea.launc
hau hau

Process finished with exit code 0
```

Functional Patterns Matching

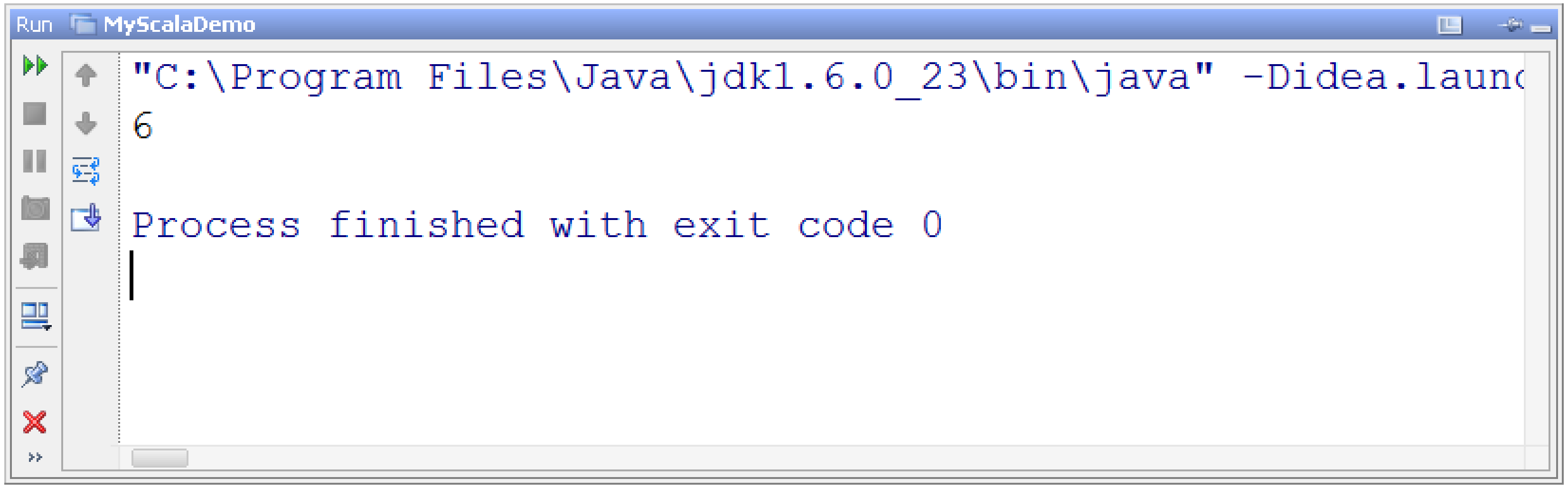
- We can define a function that uses patterns matching as a replacement for a series of `if..else` statements.

© 2008 Haim Michael 20160119

Functional Patterns Matching

```
package com.abelski.samples

object MyScalaDemo extends Application
{
  def multiply(numA:Int,numB:Int):Int =
  {
    numB match
    {
      case 0 => 0
      case 1 => numA
      case _ => numB + multiply(numB,numA-1)
    }
  }
  println(multiply(3,2));
}
```

Functional Patterns Matching



```
"C:\Program Files\Java\jdk1.6.0_23\bin\java" -Didea.launc
6

Process finished with exit code 0
```

Case Classes

- Adding case to our class definition adds a factory method with the same name as the name of the class.
- Assuming we define the following classes:

```
case class Point(x:Double,y:Double) {}
case class Line(p1:Point,p2:Point) {}
```

We can now instantiate them without using the new keyword:

```
val ob = Line(Point(4,3),Point(2,2))
```

Case Classes

- Adding case to our class definition all arguments in the parameters list get a val prefix so we get them maintained as fields.
- Adding case to our class definition, the compiler adds natural implementations for the methods `toString`, `hashCode` and `equals`. These auto generated methods recursively print, hash and compare the entire tree of the class.

Case Classes

- Calling the `==` operator is forwarded to the `equals` method. Elements of case class compared using the `==` operator will be compared structurally.
- Case classes support patterns matching and this is their biggest advantage.

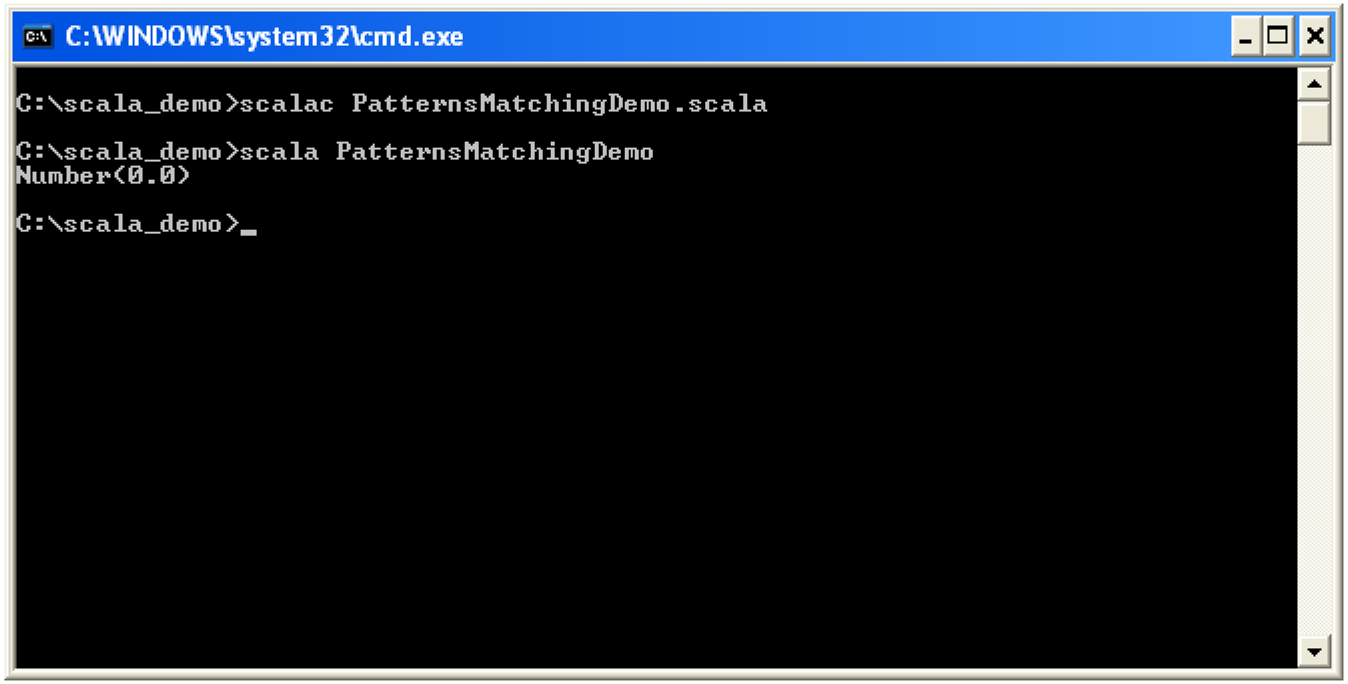
Patterns Matching

```
package com.lifemichael.samples

abstract class Expression

case class BinaryOperatorExpression(
                          operator:String,
                             rightArgument:Expression,
                             leftArgument:Expression) extends Expression

case class Number(num:Double) extends Expression

case class UnaryOperatorExpression(
                          operator:String,
                          argument:Expression) extends Expression
```

Patterns Matching

```
package com.lifemichael.samples

object PatternsMatchingDemo
{
  def main(args: Array[String])
  {
    println( calc( BinaryOperatorExpression("x",Number(4),Number(0)) ) )
  }
  def calc(exp:Expression):Expression =
  {
    exp match
    {
      case UnaryOperatorExpression("+",Number(4)) => Number(4)
      case BinaryOperatorExpression("x",Number(4),Number(1)) => Number(4)
      case BinaryOperatorExpression("+",Number(4),Number(0)) => Number(4)
      case BinaryOperatorExpression("x",Number(4),Number(0)) => Number(0)
    }
  }
}
```

Patterns Matching



```
C:\scala_demo>scalac PatternsMatchingDemo.scala

C:\scala_demo>scala PatternsMatchingDemo
Number(0.0)

C:\scala_demo>_
```

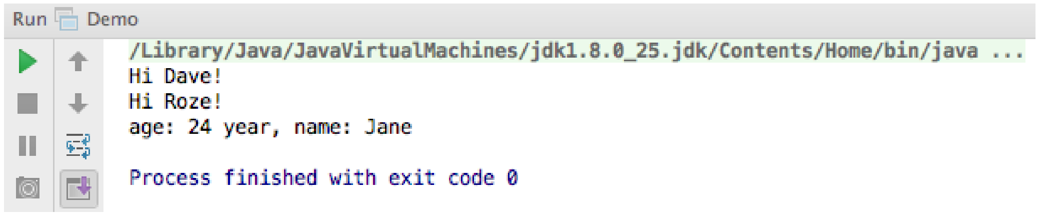
Case Classes

- The following code sample includes the definition of Student case class.

© 2008 Haim Michael 20160119

Case Classes

```
object Demo {
  def main(args: Array[String]) {
    val dave = Student("Dave", 25, 88)
    val roze = Student("Roze", 32, 82)
    val jane = Student("Jane", 24,74)
    for (person <- List(dave, roze, jane)) {
      person match {
        case Student("Dave", 25, 88) => println("Hi Dave!")
        case Student("Roze", 32, _) => println("Hi Roze!")
        case Student(name, age, average) =>
          println("age: " + age + " year, name: " + name)
      }
    }
  }
  case class Student(val name: String, val age: Int, val average: Double)
}
```



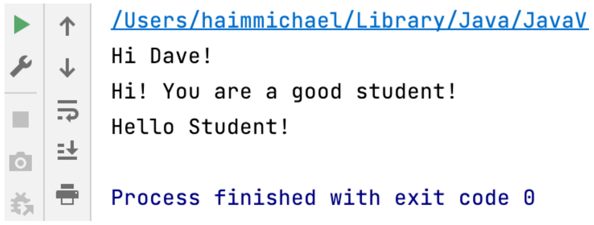
```
/Library/Java/JavaVirtualMachines/jdk1.8.0_25.jdk/Contents/Home/bin/java ...
Hi Dave!
Hi Roze!
age: 24 year, name: Jane

Process finished with exit code 0
```

Conditional Case Classes

- The following code sample includes the definition of Student case class, and the use of additional conditions in the code.

Conditional Case Classes

```
object Demo {

  case class Student(val name: String, val age: Int, val average: Double)

  def main(args: Array[String]) {
    val dave = Student("Dave", 25, 88)
    val roze = Student("Roze", 32, 82)
    val jane = Student("Jane", 24,74)
    for (person <- List(dave, roze, jane)) {
      person match {
        case Student("Dave", 25, 88)
             => println("Hi Dave!")
        case Student(_, _, avg) if avg>90
             => println("Hi! You are an excellent student!")
        case Student(_, _, avg) if avg<=90 && avg>80
             => println("Hi! You are a good student!")
        case Student(_, _, avg) if avg<=80
             => println("Hello Student!")
      }
    }
  }
}
```



```
/Users/haimmichael/Library/Java/JavaV
Hi Dave!
Hi! You are a good student!
Hello Student!

Process finished with exit code 0
```