

Scala Course

Basics 2

47 Deg

2021-02-01

Scala basics 2

In this session we'll deepen our knowledge of pattern matching & recursion!

Pattern matching

Pattern matching is a technique used in Scala (and other languages) to compare values against shapes and conditions. You can think of it like a more powerful `switch` statement.

Pattern matching

```
val a: Int = 3

a match {
  case 3 => "it's three!"
  case _ => "it's not three!"
}
```

Exhaustively

Scala's pattern matching has an exhaustively checker. This means that the compiler will warn if we forget to match against one of the cases.

```
sealed trait Color
case object Blue extends Color
case object Red extends Color
case object Green extends Color
```

Exhaustively

```
val color: Color = Blue
```

```
color match {  
  case Blue => println("it's blue!")  
  case Red => println("it's red!")  
}  
  
// warning: match may not be exhaustive.  
// It would fail on the following inputs: Green  
// color match {  
// ^^^^^
```

Exhaustively

```
color match {  
  case Blue => println("it's blue!")  
  case Red  => println("it's red!")  
  case Green => println("it's green!")  
}  
// it's blue!
```

Exhaustively

```
val newColor: Color = Green
```

```
newColor match {  
  case Blue => println("it's blue!")  
  case _ => println("if it's not blue, I don't mind")  
}  
// if it's not blue, I don't mind
```


Destructuring

Destructuring allows us to query inner parts of an ADT

```
sealed trait Vehicle
case class Car(
  brand: String, model: String, color: Color
) extends Vehicle
case class Plane(
  brand: String, model: String, wingSpan: Int
) extends Vehicle
```

Destructuring

```
val vehicle: Vehicle = Car("Honda", "Accord", Red)
```

```
vehicle match {  
  case Car(brand, model, Red) =>  
    s"it's a red $brand $model"  
  case Car(brand, model, Blue) =>  
    s"it's a blue $brand $model"  
  case Car(brand, model, _) =>  
    s"it's an unknown color $brand $model"  
  case Plane(brand, model, wingSpan) =>  
    s"it's a $brand $model with $wingSpan m wing span!"  
}  
  
// res4: String = "it's a red Honda Accord"
```

Destructuring

This is possible thanks to unapply method (extractor object)

```
object Car {  
  def unapply(car: Car): Option[(String, String, Color)]  
    Option(car.brand, car.model, car.color)  
}
```

We will see this afterwards

Destructuring

```
object FullName {  
  def unapply(fullName: String): Option[(String, String)]  
    = fullName.split(" ") match {  
      case Array(f, s) => Option((f, s))  
      case _ => None  
    }  
}  
  
def splitFullName(name: String): String = name match {  
  case FullName(f, s) => s"firstname: $f, surname: $s"  
  case _ => "more than two words"  
}  
  
splitFullName("John Doe")  
// res5: String = "firstname: John, surname: Doe"  
  
splitFullName("Jose García García")  
// res6: String = "more than two words"
```

Guards

Guards are boolean conditions we want to check while pattern matching.

```
val plane: Vehicle = Plane("Boeing", "747", 47)
```

```
plane match {  
  case Plane(brand, model, wSpan) if wSpan > 40 =>  
    s"it's a big $brand $model"  
  case Plane(brand, model, wSpan) if wSpan <= 40 =>  
    s"it's a small $brand $model"  
  case _ => s"it's not a plane..."  
}  
  
// res7: String = "it's a big Boeing 747"
```

Break: Desugaring Cases

Let's see all this with examples!

Recursion

Recursion happens when a function calls itself. It's the solution we use in functional programming to the problems for which OOP uses loops.

Notice: we will not deal with tail recursion in this section

Fibonacci sequence

Fibonacci sequence is an infinite in which every number is defined by summing the two previous numbers.

Fibonacci in Python (strawman :D)

```
def fib(num):  
    a, b, temp = (1, 0, 0)  
    while(num >= 0):  
        temp = a  
        a = a+b  
        b = temp  
        num = num - 1  
    return b
```

Fibonacci in Scala

```
def fib(num: Int): Int = num match {  
  case 0 => 1  
  case 1 => 1  
  case x => fib(x - 1) + fib(x - 2)  
}
```

Recursion

Recursion is tightly coupled to pattern matching and algebraic data types.

Recursion

Let's declare a linked list in scala.

```
sealed trait MyList[A]
case class Nil[A]() extends MyList[A]
case class Cons[A](
  head: A,
  tail: MyList[A]
) extends MyList[A]
```

Recursion

This is how we could create instances of this list.

```
val three = Cons(  
  1,  
  Cons(  
    2,  
    Cons(  
      3,  
      Nil()))))
```

Recursion

length

```
def length[A](l: MyList[A]): Int =  
  l match {  
    case Nil() => 0  
    case Cons(x, xs) => 1 + length(xs)  
  }
```

```
length(three)  
// res8: Int = 3
```

Recursion

sum

```
def sum(list: MyList[Int]): Int =  
  list match {  
    case Nil() => 0  
    case Cons(x, xs) => x + sum(xs)  
  }
```

Exercise 2.1

Implement a generic binary tree data structure. There are **two possible cases** for binary trees:

- Empty binary trees
- Binary trees with a value and pointers to left and right

Exercise 2.1

Implement a generic binary tree data structure. There are **two possible cases** for binary trees:

- Empty binary trees
- Binary trees with a value and pointers to left and right

Solution

```
sealed trait Tree[A]  
case class Empty[A]() extends Tree[A]  
case class Node[A] (  
  l: Tree[A],  
  a: A,  
  r: Tree[A]  
) extends Tree[A]
```

Exercise 2.2

create a function to calculate the height of a tree.

Exercise 2.2

create a function to calculate the height of a tree.

Solution

```
def height[A](tree: Tree[A]): Int = tree match {  
  case Empty() => 0  
  case Node(l, _, r) => 1 + (height(l).max(height(r)))  
}
```

Exercise 2.3

Create a function that sums all the leaves of an Int tree.

Exercise 2.3

Create a function that sums all the leaves of an Int tree.

Solution

```
def sum(tree: Tree[Int]): Int = tree match {  
  case Empty() => 0  
  case Node(l, x, r) => x + sum(l) + sum(r)  
}
```

Exercise 2.4

Create a function that counts all the leaves in a tree

Exercise 2.4

Create a function that counts all the leaves in a tree

Solution

```
def count[A](tree: Tree[A]): Int = tree match {  
  case Empty() => 0  
  case Node(l, _, r) => 1 + count(l) + count(r)  
}
```

Exercise 2.5

Create a function that transforms each element in a tree into it's string representation

Exercise 2.5

Create a function that transforms each element in a tree into it's string representation

Solution

```
def toStringNodes(tree: Tree[Int]): Tree[String] = tree match {  
  case Empty() => Empty()  
  case Node(l, x, r) => Node(  
    toStringNodes(l),  
    x.toString,  
    toStringNodes(r))  
}
```

Exercise 2.6

Create a function that squares all elements in an Int tree

Exercise 2.6

Create a function that squares all elements in an Int tree

Solution

```
def squared(tree: Tree[Int]): Tree[Int] = tree match {  
  case Empty() => Empty()  
  case Node(l, x, r) => Node(  
    squared(l),  
    x * x,  
    squared(r))  
}
```

Postscript: variance

Scala allows us to express the variance of generic types. They can either be invariant (all the generics we've seen are invariant), covariant, or contravariant.

Postscript: Covariance

Let CList be a type constructor declared as:

```
trait CList[A]
```

And the types:

```
trait Foo  
trait Bar extends Foo
```

Postscript: Covariance

Let CList be a type constructor declared as:

```
trait CList[A]
```

And the types:

```
trait Foo  
trait Bar extends Foo
```

 Foo ---> Bar
CList[Foo] ??? CList[Bar]

Postscript: Covariance

Let CList be a type constructor declared as:

```
trait CList[A]
```

And the types:

```
trait Foo  
trait Bar extends Foo
```

 Foo ---> Bar
CList[Foo] -X-> CList[Bar]

There's no relationship between them. This is called Invariant.

Postscript: Covariance

We express Covariance adding a + sign before the generic parameter name.

```
trait CList[+A]
```

Let's see this with the following example:

```
trait Entertainment
trait Music extends Entertainment
trait Metal extends Music
```


Postscript: Contravariance

Contravariance is similar to covariance, but **inverting** the hierarchy.
If we declare a type constructor as contravariant:

$$\begin{array}{ccc} \text{Foo} & \text{--->} & \text{Bar} \\ \text{Consumer}[\text{Foo}] & \text{<---} & \text{Consumer}[\text{Bar}] \end{array}$$

Postscript: Contravariance

To see this, let's do it by the following example:

```
trait Consumer[-A] {  
  def consume(value: A): String  
}  
  
trait Food  
trait VegetarianFood extends Food  
trait VeganFood extends VegetarianFood
```

Postscript: Variance

As a final note, try to be very careful of when you use variance. It might get out of hand quickly, and when you get to the functional libraries such as cats or scalaz, it's difficult to make it fit.

More info:

<https://www.signifytechnology.com/blog/2018/12/variances-in-scala-by-wiem-zine-el-abidine>