Scala Course

Basics 2

Pepe García

2020-12-01

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

```scala
val a: Int = 3

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

Exhaustivity

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

```scala
sealed trait Color
case object Blue extends Color
case object Red extends Color
case object Green extends Color
case class Other(name: String) extends Color
```

Exhaustivity

```scala
val color: Color = Blue
// color: Color = Blue

color match {
  case Blue => println("it's blue!")
  case Other(x) => println(s"it's $x!")
}
// it's blue!
```

Destructuring

Destructuring allows us to query inner parts of an ADT

```scala
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

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

Destructuring

```scala
vehicle match {
  case Car(brand, model, Red) =>
    s"it's a red $brand $model"
  case Car(brand, model, Blue) =>
    s"it's a red $brand $model"
  case Car(brand, model, Other(colorName)) =>
    s"it's a $colorName $brand $model"
  case Plane(brand, model, wingSpan) =>
    s"it's a $brand $model with $wingSpan meter of wing span!"
}
```

Guards

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

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

plane match {
  case Plane(brand, model, wingSpan) if wingSpan > 40 =>
    s"it's a big $brand $model"
  case Plane(brand, model, wingSpan) if wingSpan <= 40 =>
    s"it's a small $brand $model"
  case _ => s"it's not a plane..."
}
```

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

Recursion

Fibonacci sequence

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

Recursion

Fibonacci in Python (strawman :D)

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

Recursion

Fibonacci in Scala

```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.

```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)
```

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

```scala
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

```scala
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

```scala
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

```scala
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

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

Postscript: Covariance

Let `CList` be a type constructor declared as:

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

If we have two types `Foo` and `Bar`,and `Foo` is a subtype of `Bar`, since `CList` is covariant, `CList[Foo]` is a subtype of `CList[Bar]`.

Postscript: Contravariance

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

```scala
trait Logger[-A]
```

We mean that, for two types `Foo` and `Bar` if `Foo` is a subtype of `Bar`, then `CList[Bar]` is a subtype of `CList[Foo]`

Postscript: Contravariance

```scala
class Fruit
class Banana extends Fruit

def bananaLogger: Logger[Banana] = new Logger[Banana] {}
def fruitLogger: Logger[Fruit] = new Logger[Fruit] {}

def logBanana(logger: Logger[Banana]): Int = 3

logBanana(bananaLogger)
logBanana(fruitLogger)
```

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.