Pattern matching & functional composition

«Previous Next»

This lesson covers:

Function Composition

Let's make two aptly-named functions:

```
scala> def f(s: String) = "f(" + s + ")"
f: (String)java.lang.String

scala> def g(s: String) = "g(" + s + ")"
g: (String)java.lang.String
```

compose

`compose` makes a new function that composes other functions `f(g(x))`

```
scala> val fComposeG = f _ compose g _
fComposeG: (String) => java.lang.String = <function>

scala> fComposeG("yay")
res0: java.lang.String = f(g(yay))
```

andThen

`andThen` is like `compose`, but calls the first function and then the second, `g(f(x))`

```
scala> val fAndThenG = f _ andThen g _
fAndThenG: (String) => java.lang.String = <function>

scala> fAndThenG("yay")
res1: java.lang.String = g(f(yay))
```

Currying vs Partial Application

case statements

So just what are case statements?

It's a subclass of function called a PartialFunction.

What is a collection of multiple case statements?

They are multiple PartialFunctions composed together.

Understanding PartialFunction

A function works for every argument of the defined type. In other words, a function defined as (Int) => String takes any Int and returns a String.

A Partial Function is only defined for certain values of the defined type. A Partial Function (Int) => String might not accept every Int.

`isDefinedAt` is a method on PartialFunction that can be used to determine if the PartialFunction will accept a given argument.

Note `PartialFunction` is unrelated to a partially applied function that we talked about earlier.

See Also Effective Scala has opinions about PartialFunction.

```
scala> val one: PartialFunction[Int, String] = { case 1 => "one" }
one: PartialFunction[Int,String] = <function1>

scala> one.isDefinedAt(1)
res0: Boolean = true

scala> one.isDefinedAt(2)
res1: Boolean = false
```

You can apply a partial function.

```
scala> one(1)
res2: String = one
```

PartialFunctions can be composed with something new, called orElse, that reflects whether the PartialFunction is defined over the supplied argument.

```
scala> val two: PartialFunction[Int, String] = { case 2 => "two" }
two: PartialFunction[Int,String] = <function1>

scala> val three: PartialFunction[Int, String] = { case 3 => "three" }
three: PartialFunction[Int,String] = <function1>

scala> val wildcard: PartialFunction[Int, String] = { case _ => "something else" }
wildcard: PartialFunction[Int,String] = <function1>

scala> val partial = one orElse two orElse three orElse wildcard
partial: PartialFunction[Int,String] = <function1>

scala> partial(5)
res24: String = something else

scala> partial(3)
res25: String = three

scala> partial(2)
res26: String = two

scala> partial(1)
res27: String = one

scala> partial(0)
res28: String = something else
```

The mystery of case.

Last week we saw something curious. We saw a case statement used where a function is normally used.

```
scala> case class PhoneExt(name: String, ext: Int)
defined class PhoneExt

scala> val extensions = List(PhoneExt("steve", 100), PhoneExt("robey", 200))
extensions: List[PhoneExt] = List(PhoneExt(steve,100), PhoneExt(robey,200))

scala> extensions.filter { case PhoneExt(name, extension) => extension < 200 }
res0: List[PhoneExt] = List(PhoneExt(steve,100))
```

Why does this work?

filter takes a function. In this case a predicate function of (PhoneExt) => Boolean.

A PartialFunction is a subtype of Function so filter can also take a PartialFunction!

Built at @twitter by @stevej, @marius, and @lahosken with much help from @evanm, @sprsquish, @kevino, @zuercher, @timtrueman, @wickman, and @mccv; Russian translation by appigram; Chinese simple translation by jasonqu; Korean translation by enshahar;