meetup scala class

part 2

11 June 2014

This morning we focused on:

- Concrete types and records
- Pattern matching
- Recursion and lists

This afternoon things will get a bit more general.

Say we have the following method:

```scala
def listLength(xs: List[Double]): Int =
  xs match {
    case Nil => 0
    case x :: rest => 1 + listLength(rest)
  }
```

Notice that we don't do anyting with `Double`.

It would be nice to be able to use that method for `List[Int]` (or lists of any type), not just `List[Double]`.

Enter… *type parameters*!

In the same way that `def foo(x: Int)` has a (value) parameter called `x`, methods can also take types as parameters.

```
def listLength[A](xs: List[A]): Int =
  xs match {
    case Nil => 0
    case x :: rest => 1 + listLength(rest)
  }
```

We can now call our method with lists of any type:

```
listLength[Boolean](true :: true :: false :: Nil)
listLength[Int](1 :: 2 :: 3 :: 4 :: Nil)
listLength[String]("foo" :: "bar" :: Nil)

// the [String] part can often be inferred
listLength("foo" :: "bar" :: Nil)
```

```
def foo[A](xs: List[A]): A = ...
```

Given this definition we would say that:

- `xs` is a *parameter*
- `A` is a *type parameter*
- `List` is a *parameterized type* (a *type constructor*)
- `List[A]` is a *concrete type*
- `foo` is a *generic* method

To be *generic* the method must:

— Handles different types
— Act the same no matter which type is used

If we can do both of these things, we can use a type paremeter instead of a concrete type.

```
def forgetful(x: Int): Int = 999

def moreForgetful[A](a: A): Int = 999
```

The `List` type we worked is generic, and supports methods like `.length` and `.isEmpty`:

```
Nil.isEmpty                // res0: Boolean = true
(1 :: Nil).isEmpty         // res1: Boolean = true

Nil.length                 // res2: Int = 0
(1 :: 2 :: Nil).length // res3: Int = 2
```

But notice that some methods (like `.sum`) can't be implemented directly in a generic way (since we are not allowed to use +):

```
def sum[A](xs: List[A]): A =
  xs match {
    case Nil =>
      0 // how to get a zero of type A ?
    case x :: rest =>
      x + rest.sum // how to call + on A ?
  }
```

So besides things like `length`, how can we do anything useful in a generic method?

Functions!

We've been talking about and calling functions as methods.

But functions have a type and are values in Scala.

```scala
def plusOne(x: Int): Int = x + 1

// we can bind the plusOne function to a value, with a type
val f = plusOne _
// f: Int => Int = <function1>

f(3)
// res1: Int = 4
```

So our function's type is `Int => Int`. This means that:

- It takes one parameter (an `Int`)
- It returns an `Int` value

Some similar function types would be:

- `List[Int] => Int`
- `Double => Double`
- `Int => Boolean`
- and so on...

More generally, given types `A` and `B`:

`A => B` is a function value which:

— Takes one `A` parameter
— Returns one `B` result

We can also define functions directly:

```scala
val plusTen = (x: Int) => x + 10

plusTen(20)
// res0: Int = 30

plusTen(plusTen(100))
// res1: Int = 120
```

What's the difference between these?

```scala
def g1(x: Int) = x + 2
// g1: (x: Int)Int

val g2 = (x: Int) => x + 2
// g2: Int => Int = <function1>
```

Very little (apart from syntax and Java compatibility).

And now for something completely different!

Collection Types!

- `List[E]`: the singly linked-list we know and love
- `Option[E]`: safe optional values
- `Set[E]`: unordered collection of unique elements
- `Map[K, V]`: dictionary mapping keys to values
- `Vector[E]`: array-like sequence of values

These types are all part of *Scala Collections*

- Large family of shared methods
- Overarching design principles
- Generic implementations
- Built-in conversions

List

Many of the recursive methods we wrote on List are simply mapping each value to a new value.

It turns out we can easily generalize this using the `.map` method!

```scala
val xs: List[String] =
  "batman" :: "robin" :: "joker" :: "penguin" :: "riddler" :: Nil

xs.map(s => s.length)
// res2: List[Int] = List(6, 5, 5, 7, 7)

xs.map(s => s.toUpperCase)
// res3: List[String] = List(BATMAN, ROBIN, JOKER, PENGUIN, RIDDLER)

opt.map(s => s.reverse)
// res4: List[String] = List(namtab, nibor, rekoj, niugnep, relddir)
```

`.map` is a higher-order function.

- Called on a `List[A]`
- Takes a function `A => B` as an argument
- Returns a `List[B]`

Sometimes `.map` is not enough.

Say we want a single list of all the letters in our names:

```
def letters(name: String): List[Char] = name.toList

xs.map(s => letters(s))
// res5: List[List[Char]] = List(List(b, a, t, m, a, n),
//   List(r, o, b, i, n), List(j, o, k, e, r),
//   List(p, e, n, g, u, i, n), List(r, i, d, d, l, e, r))
```

But that gives us a `List[List[String]]` which isn't what we wanted.

We need to build one mega-list using *flatMap*!

```
xs.flatMap(s => letters(s))
// res6: List[Char] = List(b, a, t, m, a, n, r, o, b, i, n,
//   j, o, k, e, r, p, e, n, g, u, i, n, r, i, d, d, l, e, r)
```

FlatMap is awesome!

It's sort of the grand-daddy of many useful functions:

```
// mapping
val f: Int => Int = ...
xs.flatMap(x => f(x) :: Nil) == xs.map(x => f(x))

// filtering
val g: Int => Boolean = ...
xs.flatMap(x => if (g(x)) x :: Nil else Nil) xs.filter(g)
```

Also, see foldLeft!

```
def sumList(xs: List[Int]): Int = xs.foldLeft(0) {
  (currentTotal, x) => currentTotal + x
}
```

Option

Represents a possibly-missing value.

- `Some(x)`: the value `x` is present
- `None`: there is no value present

Create options via the `Option(...)` constructor:

```
val x = Option("hi")
// x: Option[String] = Some("hi")

def ugh: String = null
// ugh: String

val y = Option(ugh)
// y: Option[String] = None

val z = Option(123)
// z: Option[Int] = Some(123)
```

After receiving an optional value, one basic way to handle it is pattern-matching.

```
def path(opt: Option[String], default: String) =
  opt match {
    case Some(path) => path
    case None => default
  }
```

But we can use `.map` here too.

```scala
val opt: Option[String] = Option("batman")

opt.map(s => s.length)
// res0: Option[Int] = Some(6)

opt.map(s => s.toUpperCase)
// res1: Option[String] = Some(BATMAN)

opt.map(s => s.reverse)
// res2: Option[String] = Some(namtab)
```

What about this?

```
def calculate(x: Double): Double =
  if (x == 0.0) None else Some(1.0 / x)

def handle(o: Option[Double]): Option[Double] =
  o.map { n => calculate(n.toDouble) }
// found   : Option[Double]
// required: Double
```

Ugh, OK

```scala
def calculate(x: Double): Double =
  if (x == 0.0) None else Some(1.0 / x)

def handle(o: Option[Double]): Option[Double] =
  o match {
    case Some(x) => calculate(x)
    case None => None
  }
```

But again... *there is a better way!*

`.flatMap` allows us to chain Option-producing calls.

```
def calculate(x: Double): Option[Double] =
  if (x == 0.0) None else Some(1.0 / x)

def handle(o: Option[Double]): Option[Double] =
  o.flatMap(x => calculate(x))

def handleTwice(o: Option[Double]): Option[Double] =
  o.flatMap(x => calculate(x)).
    flatMap(x => calculate(x))

def handleThrice(o: Option[Double]): Option[Double] =
  o.flatMap(x => calculate(x)).
    flatMap(x => calculate(x)).
    flatMap(x => calculate(x))
```

In fact, there's a handy syntax for that:

```scala
def handleThrice(o: Option[Double]): Option[Double] =
  for {
    x <- o
    x1 <- calculate(x)
    x2 <- calculate(x1)
    x3 <- calculate(x2)
  } yield x3

// is the same as:

def handleThrice(o: Option[Double]): Option[Double] =
  o.flatMap(x => calculate(x)).
    flatMap(x => calculate(x)).
    flatMap(x => calculate(x))
```

WARNING

You *must* include the `yield` statement when using `for`.

Otherwise, it's a totally different thing.

(If `yield` is absent `for` behaves more like a Java loop.)

You have been warned!