meetup scala class

part 1

11 June 2014

Open a terminal and checkout the repo:

```
$ git clone https://github.com/meetup/meetup-scala-class.git
$ cd meetup-scala-class
```

This presentation is located at *pdf/class1.pdf*.

```
$ open pdf/class1.pdf
```

Launch the REPL:

```
$ ./sbt console
...
Welcome to Scala version 2.10.4 ...
Type in expressions to have them evaluated.
Type :help for more information.

scala> "Hello, world"
res0: String = Hello, world
```

Scala ignores lines beginning with //:

```scala
// four thousand years ago Babylonians knew
// that 22/7 was a good approximation of pi.
22.0 / 7.0
```

We've got a calculator:

```
1 + 2 + 3 + 4 // addition
1000 - 245    // subtraction
7.25 * 35     // multiplication
22980.0 / 52  // division [1]
44 % 7        // remainder

// [1] compare to 22980 / 52
```

Values:

- Immutable, i.e. they don't change.
- Types describe the possibilities/shape of data
- Values can either be:
  - simple (numbers, text, true/false, etc.)
  - complex (pairs, lists, dictionaries, etc.)

Naming values:

```scala
val minimumWage = 7.25
val hoursPerWeek = 35
val weeksPerYear = 52

minimumWage * hoursPerWeek * weeksPerYear
// res0: Double = 13195.0

15.0 * hoursPerWeek * weeksPerYear
// res1: Double = 27300.0
```

Functions:

```
def wages(wage: Double, weeklyHours: Int) =
  wage * weeklyHours * weeksPerYear

wages(7.25, 35)
wages(11.0, 35)
wages(13.0, 35)
wages(17.0, 35)
```

More functions:

```
def pythagoras(x: Double, y: Double): Double =
  math.sqrt(x * x + y * y)

pythagoras(1.0, 1.0)
pythagoras(3.0, 4.0)
pythagoras(1.0, 2.0)
```

How do we know what values are capable of?

Types!

(We've already seen types; they come "after the colon")

Each type provides methods which we can call. E.g.:

```
// + is a method on Double
// so is .abs
def f(x: Double, y: Double) = (x + y).abs
```

Let's look at some useful types and methods...

Number Types (`Int`, `Double`, etc.)

- arithmetic (`+ - * / %`)
  `x + y * z`
- comparisons (`== != > >= < <= max min`)
  `x <= y`
- misc
  (`.abs .ceil .floor .round .signum`, `.toString`)
  `(x / y).round`

Number types (continued)

- `Int` is useful for reasonably-sized whole values.
- `Double` is useful for most fractional values.

(The `math` package provides functions like `pow`)

(There are other more exotic number types as well!)

Text (String, Char, etc.)

```scala
val line = "Defines some text."
val char = 's'  // a single character

val fancy = s"We can interpolate: $line"
// fancy: String = We can interpolate: Defines some text.
```

(String is the same as `java.lang.String`)

Text (continued)

```
val s = "batman"

s.length                // res0: Int = 6
s.toUpperCase           // res1: String = BATMAN
s.startsWith("cat")     // res2: Boolean = false
s.replace("b", "w")     // res3: String = watman
s + " and robin"        // res4: String = batman and robin
```

Boolean

Contains only two values: `true` and `false`.

```
val x = true
val y = false
!x                 // not: since x is true, (!x) is false
x && y             // and: since y is false (x && y) is false
x || y             // or: since x is true (x || y) is true
if (y) 1 else 0 // if/else: since y is false, return 0
```

(technically, if/else is syntax, not a method call)

We can already do a lot with this!

```
// concatenate s to itself n times
def repeatConcat(s: String, n: Int) =
  if (n <= 0) "" else s + repeatConcat(n - 1)

// calculate interest on p compounded n times per year
def interest(p: Double, rate: Double, years: Double, n: Double) =
  p * math.pow(1 + (rate / n), num * years)
```

Let's look at `repeatConcat` more closely

```
repeatConcat("cat", 3)
"cat" + repeatConcat("cat", 2)
"cat" + "cat" + repeatConcat("cat", 1)
"cat" + "cat" + "cat"
"catcatcat"
```

This is an example of recursion.

We can write the method in a different way if we want

```
def repeatConcat2(s: String, n: Int): String = {

  def loop(sofar: String, i: Int): String =
    if (i < n) loop(s + sofar, i + 1) else sofar

  loop("", 0)
}
```

Let's dissect that a bit...

The `{ ... }` define a block.

Blocks can be used to:

- allow "inner" method/value definitions
- support pattern matching
- allow writing "imperative" code

```scala
def repeatConcat2(s: String, n: Int): String = {
  def loop(sofar: String, i: Int): String =
    if (i < n) loop(s + sofar, i + 1) else sofar
  loop("", 0)
}

repeatConcat("dog", 2)
loop("", 0)
loop("dog", 1)
loop("dogdog", 2)
loop("dogdogdog", 3)
"dogdogdog"
```

The previous strategy is called "tail recursion"

- Compiles to a very efficient representation
- Often faster than "normal" recursion
- Less general (not all recursive methods can be tail-recursive)

Tuples

We can group several values together to create a tuple:

```
val nyc = (40.7127, -74.0059)
// nyc: (Double, Double) = (40.7127,-74.0059)

val poem = ("The Raven", Poe", 1845)
// poem: (String, String, Int) = (The Raven,Poe,1845)

poem._3
// res1: Int = 1845
```

- Any types can be combined in a tuple.
- `(Int, Int)` is a type (it holds two `Int` values).
- Access positions with `._1`, `._2`, `._3`, etc.
- We can also "destructure" tuples (take them apart).

```
val (title, author, year) = poem
// title: String = The Raven
// author: String = Poe
// year: Int = 1845
```

Case classes

Like tuples, but with fixed names/types.

```
case class Point(lat: Double, lon: Double)
case class Poem(title: String, author: String, year: Int)

val nyc = Point(40.7127, -74.0059)
val poem = Poem("The Raven", Poe", 1845)
```

Type-checking

Because our types are fixed, we can catch mistakes.

```
val wrong = Point("40.7127", "-74.0059")
// <console>:9: error: type mismatch;
//  found   : String("40.7127")
//  required: Double
//         val wrong = Point("40.7127", "-74.0059")
```

Destructuring

Case classes can be destructured just like tuples.

```
val Point(lat, lon) = nyc
val Poem(title, author, year) = poem
```

In fact, destructuring is a form of *pattern matching*

Pattern Matching

Here's a method we might choose to write:

```scala
def isModernist(poem: Poem): Boolean = {
  val Poem(title, author, year) = poem
  year >= 1890
}
```

We can use the `match` statement to do the same thing:

```
def isModernist(poem: Poem): Boolean =
  poem match {
    case Poem(_, _, year) =>
      year >= 1890
  }
```

(In this case _ avoids binding names.)

Unlike `val`, `match` supports conditional logic:

```
def score(poem: Poem): Double =
  poem match {
    case Poem("The Raven", _, _) => 0.9
    case Poem(_, "Eliot", _) => 0.7
    case Poem(_, "Poe", _) => 0.5
    case Poem(_, _, y) if y == 1923 => 0.3
    case _ => 0.2
  }
```

We can even use pattern matching on simple values:

```scala
def synesthesia(n: Int): String =
  n match {
    case 3 => "yellow"
    case 5 => "red"
    case 7 => "blue-green"
    case _ => "grey"
  }
```

Another example:

```
def ordinal(n: Int): String = {
  def suffix(n: Int): String = n match {
    case 1 => "st"
    case 2 => "nd"
    case 3 => "rd"
    case x if x <= 20 => "th"
    case x if x >= 100 => suffix(x % 100)
    case x => suffix(x % 10)
  }
  n.toString + suffix(n)
}
```

List

Often we want to talk about a list of values:

```scala
val poems: List[Poem] =
  Poem("The Raven", "Poe", 1845) ::
  Poem("Jabberwocky", "Carroll", 1871) ::
  Poem("The Waste Land", "Eliot", 1922) ::
  Nil
```

(The `Nil` value is an empty list.)

Lists can be prepend to with the `::` method.

```
val addedPoems: List[Poem] =
  Poem("Howl", "Ginsberg", 1955) ::
  Poem("Tulips", "Plath", 1966) ::
  poems // previously-defined poems
```

We can also take a list apart via pattern matching.

```
def isListEmpty(nums: List[Int]): Boolean =
  nums match {
    case Nil => true
    case _ => false
  }
```

Using a simple recursive method, we can sum a list.

```scala
def sumList(ns: List[Int]): Int = ns match {
  case Nil => 0
  case first :: rest => first + sumList(rest)
}

sumList(Nil)                // res0: Int = 0
sumList(1 :: 2 :: 3 :: Nil)  // res1: Int = 6
```

Let's see an example in more detail:

```scala
def sumList(ns: List[Int]): Int = ns match {
  case Nil => 0
  case first :: rest => first + sumList(rest)
}

sumList(13 :: 45 :: 8 :: Nil)
13 + sumList(45 :: 8 :: Nil)
13 + 45 + sumList(8 :: Nil)
13 + 45 + 8 + sumList(Nil)
13 + 45 + 8 + 0
66
```

REPL tips:

- Add your solutions to `foo.scala`
- `:load foo.scala` will (re)load your code
- Test your solutions in the REPL
- You can also add test cases to your file

Exercises

```scala
// 1. Jane works 45 hours a week at $15.5/hour.
//    What are her yearly earnings?

// 2. Assuming overtime work gets paid at 1.5
//    times the normal rate, what are Jane's
//    yearly earnings?

// 3. Write a method that generalizes #2
def payWithOvertime(wage: Double, hours: Double): Double = ???

// 4. How many hours per week must Jane work
//    to earn $42,000 per year?
```

Exercises

```
// 5. Write a method that given a name (e.g. "Albert"),
//    produces a string of greeting (e.g. "Hello Albert").
def greet(name: String): String = ???

// 6. Modify the method to that produces a different greeting
//    for your team members' names (e.g. "Salutations Brian").

// 7. Write a method that returns the number of names in a list.
def numNames(names: List[String]): Int = ???

// 8. Write a method that determines if the given name exists
//    in a list of names.
def locate(given: String, names: List[String]): Boolean = ???
```

Exercises

```
// 9. Produce a single greeting for a list of names
def greet3(names: List[String]): String = ???

greet3(Nil)
// res0: String = "Hello!"

greet3("Alice" :: Nil)
// res1: String = "Hello Alice"

greet3("Alice" :: "Bob" :: Nil)
// res1: String = "Hello Alice and Bob"

greet3("Alice" :: "Bob" :: "Cate" :: Nil)
// res2: String = "Hello Alice, Bob, and Cate"
```

Exercises

```scala
// 10. Reimplement sumList but using Double instead of Int.

// 11. Rewrite sumList method to use tail recursion.
def sumList(ns: List[Double]): Double = {
  def loop(sofar: Double, xs: List[Double]): Double = ???
  loop(0.0, ns)
}

// 12. Implement a method to find the length of the list
def listLength(xs: List[Double]): Int = ???

// 13. Implement a way to reverse a list
def reverseList(xs: List[Double]): List[Double] = ???

// 14. Find the minimum value in a list
def minList(xs: List[Int], min: Int): Int = ???
```

```scala
// 15. Find the minimum and maximum values in a list
def minMax(xs: List[Int], min: Int, max: Int): (Int, Int) = ???

// 16. Return only multiples of 3 (use % operator)
def onlyDivisibleBy3(xs: List[Int]): List[Int] = ???

// 17. Return whether n is a multiple of any of
//     the given divisors.
def divides(n: Int, divisors: List[Int]): Boolean = ???
// divides(3, Nil)                // false
// divides(3, 3 :: Nil)           // true
// divides(3, 2 :: Nil)           // false
// divides(20, 3 :: 7 :: Nil) // false
// divides(20, 5 :: 7 :: Nil) // true

// 18. Return only multiples of the divisors.
def divBy(xs: List[Int], divisors: List[Int]): List[Int] = ???
```

Exercises

```
// 19. Compute the mean (average) of a list
//     (Hint: look at sumList and listLength)
def mean(xs: List[Double]): Double = ???

// 20. If you didn't already, solve #19 using
//     a single tail-recursive inner function.

// 21. (Extra credit) Standard deviation is defined
//     as the square root of the average distance
//     from the average.
def stdDev(xs: List[Double]): Double = ???
```

Exercises

```
// 22. The Fibonacci sequence is defined as:
//        fib(0) = 0
//        fib(1) = 1
//        fib(n) = f(n-1) + f(n-2)
//     Implement it using recursion (fib(20) is 10946).
def fib(n: Int): Int = ???

// 23. (Extra credit) The traditional recursive solution
//     to #10 will evaluate f(n-2) twice, once on the (n)
//     step and once on the (n-1) step.
//
//     Implement a better version using an inner method.
```

```scala
// 24. Your fib function probably only works for
//     values from 0 though 46 (for larger values
//     it likely produces negative numbers).
//
//     Implement fib10(n), a method that returns that
//     last digit of fib(n), and which does not have
//     this problem.
def fib10(n: Int): Int = ???

// fib10(51)    = 4
// fib10(503)   = 7
// fib10(5007)  = 8
// fib10(50003) = 7
```

You're done! Great!

You can:

- Ask someone to look over your work
- Compare notes with someone else
- Pair with someone who is still working