Scala -- from Adam and Eve (lots of Apples, a few Snakes)

Christian.Soderberg@cs.lth.se

19 mars 2012

Goals

- To give an overview of some of Scala's basic concepts.
- To illustrate some of the features you're likely to come across when using combinator parsers.

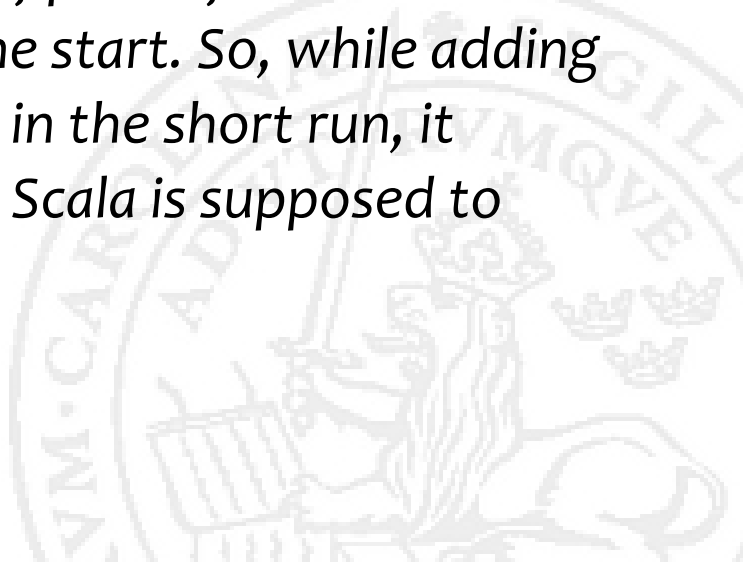
Background

- Scala can be understood from a few different (and partly orthogonal) perspectives:
  - It's *scalable* -- hence its name (it's meant to be expandable within itself).
  - It combines object-oriented features with functional programming features.
  - It makes a serious effort to 'simplify' things for the programmer.

Scalable

- The language specification is quite small.
- A flexible syntax enables designers to write embedded DSL:s (like the combinator parsers).
- Citation (Odersky, in a discussion thread last month):

> ``*Scala is emphatically not about adding syntax to remove boilerplate. That's not in its genes. It's about finding a small syntax kernel of powerful abstractions that avoid boilerplate from the start. So, while adding features might be an easy sell in the short run, it would dilute the idea of what Scala is supposed to be.*''



OOP and/or FP

- Odersky seems to be anxious not to take sides -- he pushes the community to pursue *both* OOP and FP.
- There is undeniably a bend towards using constants and immutability.

No `static`, and no primitive array

- We have no `static` keyword, instead:
  - A class describes objects.
  - An object accompanies the class -- it's called its *companion object*.
- There is an array class, but it's declared `Array[T]`, just as lists are declared `List[T]` -- the arrays are not 'built-ins' (as in Java).

Example

Exempel

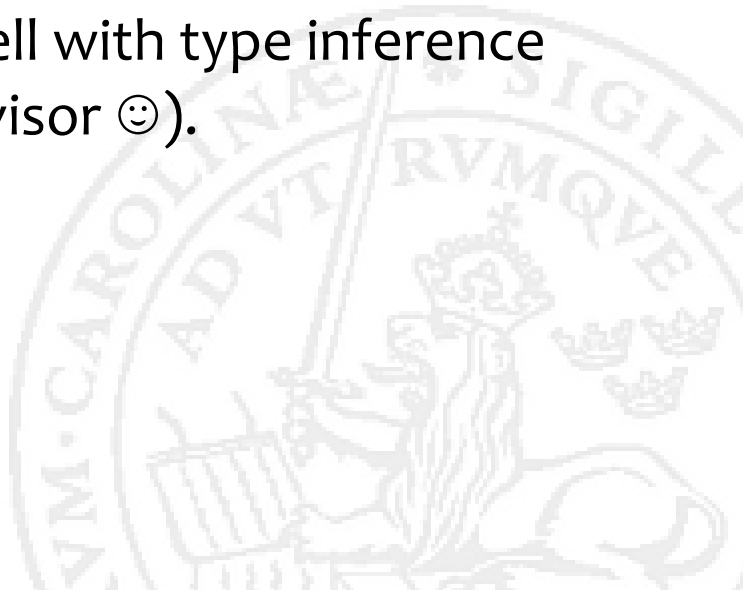
Implement the `"hello, world"`-program.

Declarations and type annotations

- Scala is statically typed, but the compiler often infers types for us.
- We use 'Pascal style' declarations:

```
name: type
```

- Postfix type annotations plays very well with type inference (also, Wirth was Odersky's PhD supervisor ☺).



val:s and var:s

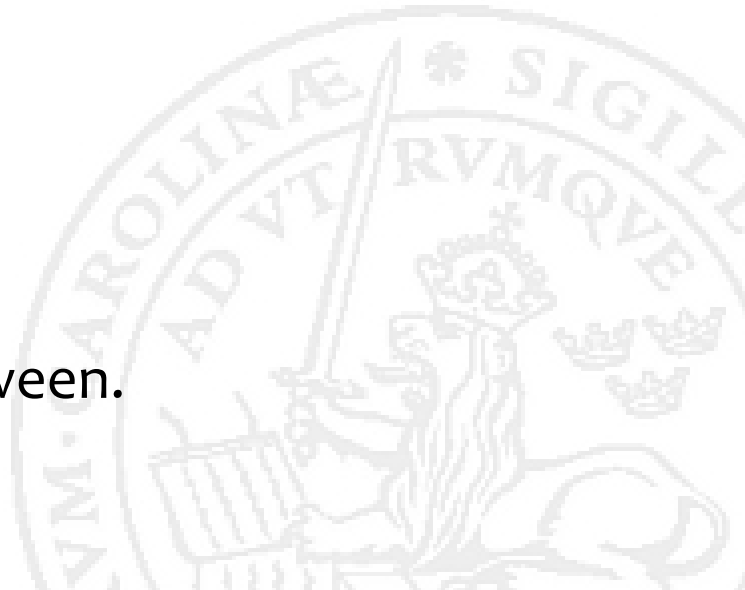
- We declare variables using var, and constants using val:

```
val n: Int = 10
var sum: Int = 0
```

- We don't have to declare the types when they can be inferred:

```
val n = 10
var sum = 0
for (term <- 1 to n) {
  sum += term
}
```

- var:s are surprisingly few and far between.

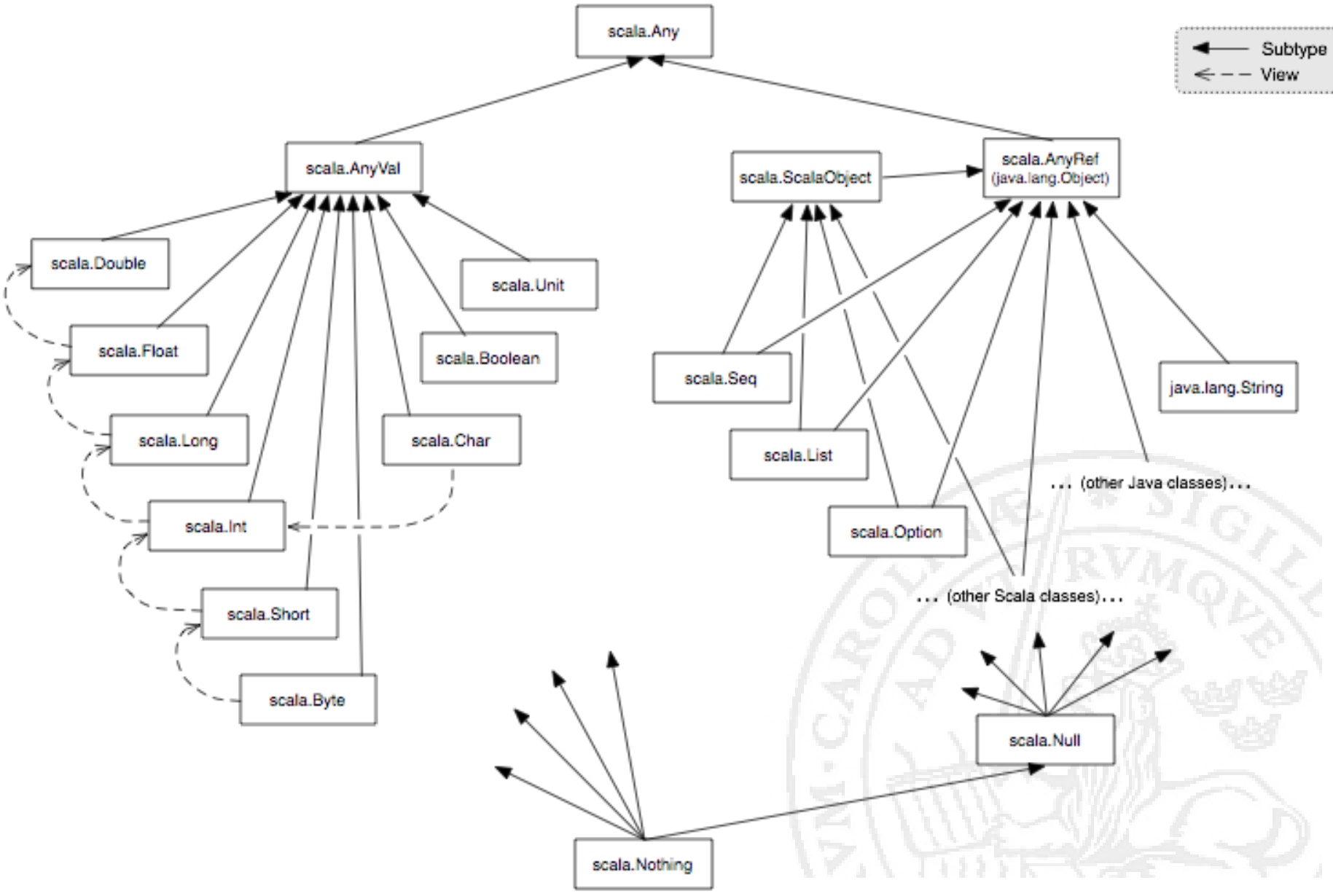


Every value is an object

- We don't have to differentiate between primitive types and reference types -- every value is an object, and `1 + 2` is just shorthand for `1.+(2)`
- The syntax rules allows us to skip dots, parentheses and semicolons in many places. We could also sometimes chose between using `{}` and `()`.
- There are quite a few methods defined on numbers, such as:

```
val indianaPi = math.Pi.toInt
val scale = 0 to 11
val biggest = a max b
println(42.toHexString)
```

Overview of classes



Values are all around

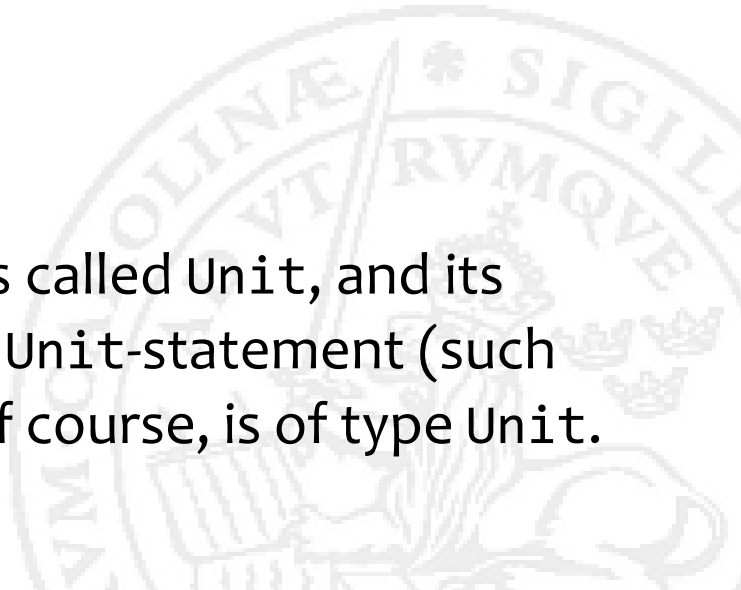
- The `if`-statement has a value:

```
val smallest = if (a < b) a else b
```

- Blocks have values (their last calculated value):

```
val gauss = {
  var sum = 0
  for (term <- 1 to 100)
    sum += term
  sum
}
```

- The Scala counterpart to Java's `void` is called `Unit`, and its only value is `()`. A block with a closing `Unit`-statement (such as `println`) has the value `()`, which, of course, is of type `Unit`.



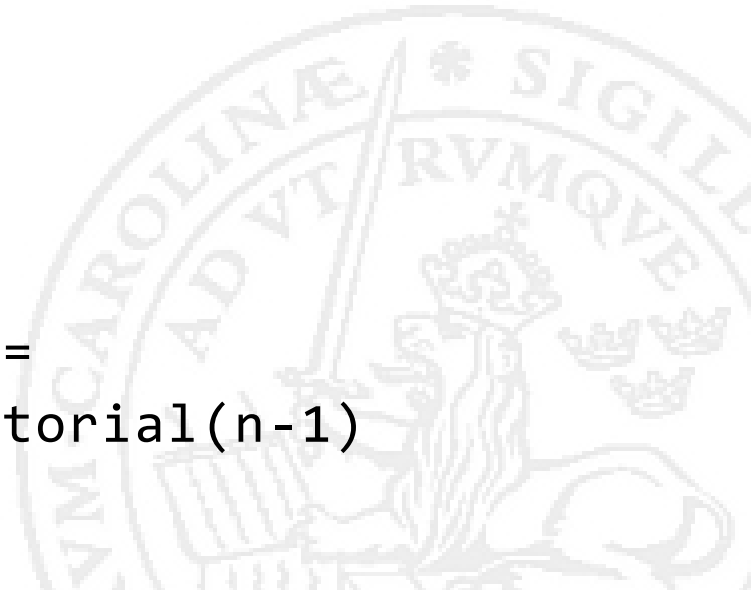
Defining methods

- We define methods using the `def` keyword:

```
def square(x: Double): Double = x * x
```

- We don't have to declare return types when they can be inferred (they usually can be, except for when dealing with recursive methods):

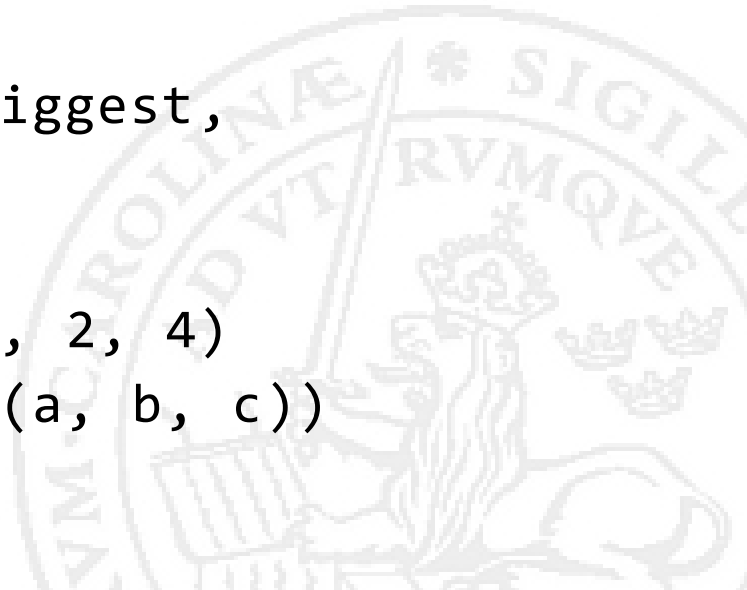
```
def arithmeticSum(n: Int) = {
  var sum = 0
  for (term <- 1 to n)
    sum += term
  sum
}
def factorial(n: Int): Int =
  if (n < 2) 1 else n * factorial(n-1)
```



More on methods

- Methods can be nested to any depth.
- We can access names bound in enclosing blocks.
- Parameters are read-only.
- We can return tuples (of any arity up to 22):

```
def minmidmax(a: Int, b: Int, c: Int) = {
  val smallest = a min b min c
  val biggest = a max b max c
  (smallest,
   a + b + c - smallest - biggest,
   biggest)
}
val (a, b, c) = minmidmax(5, 2, 4)
println("%d, %d, %d".format(a, b, c))
```



Functions

- We define lambda expressions using the =>-operator, eg. we can define $\lambda x.x^2$ as `(x: Double) => x * x`
- We can save such a *function* in a variable

```
val square = (x: Double) => x * x
```

  and can then use it as in:

```
val area = square(side)
```

- The type of `square` can be written:

```
(Double) => Double
```

  which can be simplified into:

```
Double => Double
```

Functions are values

- Functions are actually objects with an `apply`-method.
- Using some syntactic sugaring, the compiler allows us to write

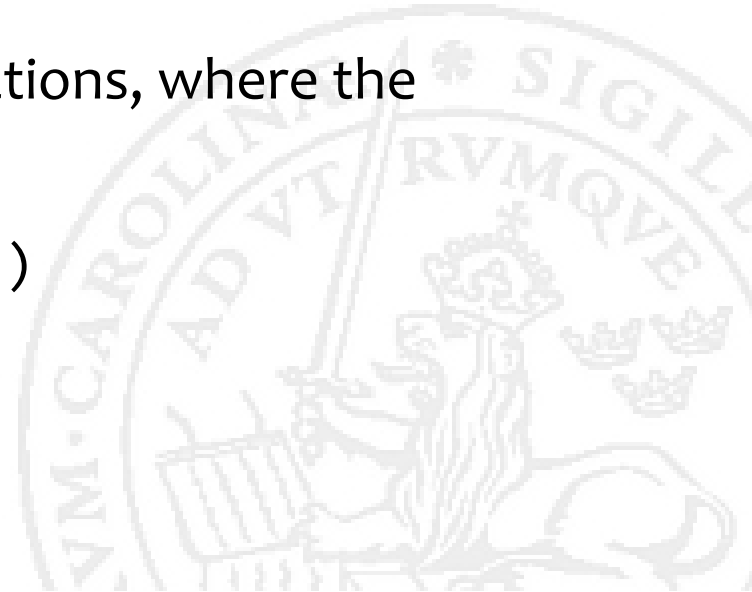
```
square(a) + square(b)
```

instead of

```
square.apply(a).+(square.apply(b))
```

- Collections can be seen as partial functions, where the `apply(Int)`-method gives us values:

```
for (k <- 0 until args.size)
  println(args(k))
```



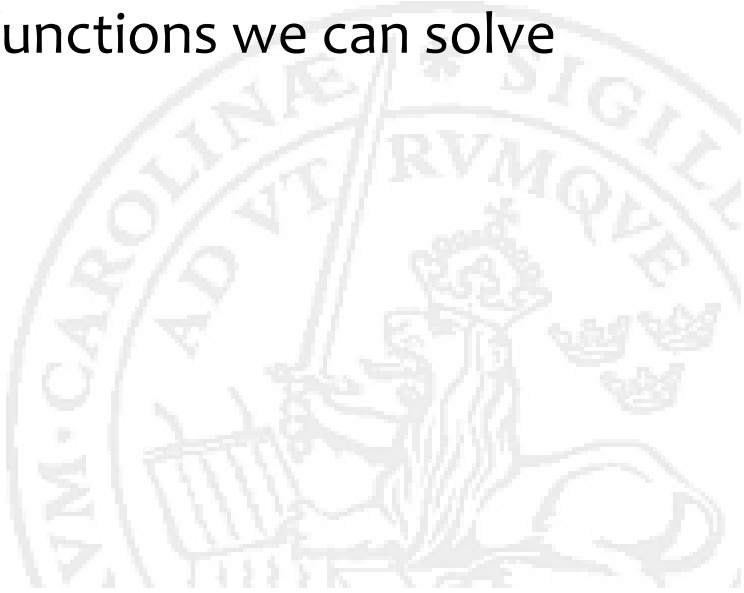
Functions/methods as parameters

- Using what's called η-expansion, Scala converts methods into functions, so we could use them almost interchangeably.
- We could send a function/method as a parameter to other functions:

```
def tabulate(f: Int => Int,
             min: Int, max: Int) = {
  for (k <- min to max)
    println("%3d: %5d".format(k, f(k)))
}
def cube(n: Int) = n * n * n
tabulate(cube, 1, 10)
```

Scala's collections

- Scala's standard collections come in three flavors:
  - immutable (default)
  - mutable
  - parallell
- Each flavor has sequences, sets, maps, etc.
- By using collections and higher order functions we can solve many problems suprisingly easily.



Collections and functions

- To map a function, such as `square`, over the elements of a sequence, we can write:

```
(1 to 10).map((p: Int) => square(p))
```

- The type of `p` can be inferred by the compiler):

```
(1 to 10).map(p => square(p))
```

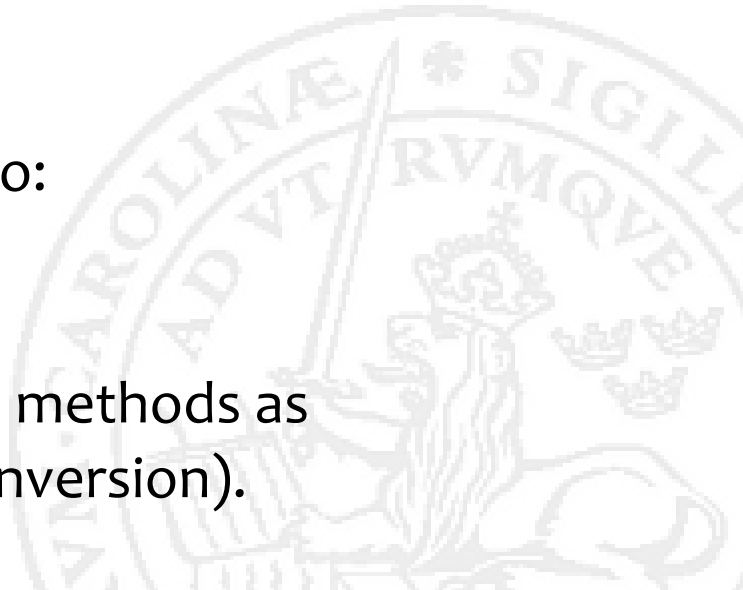
- We can replace the occurence of one parameter with _, as in:

```
(1 to 10).map(square(_))
```

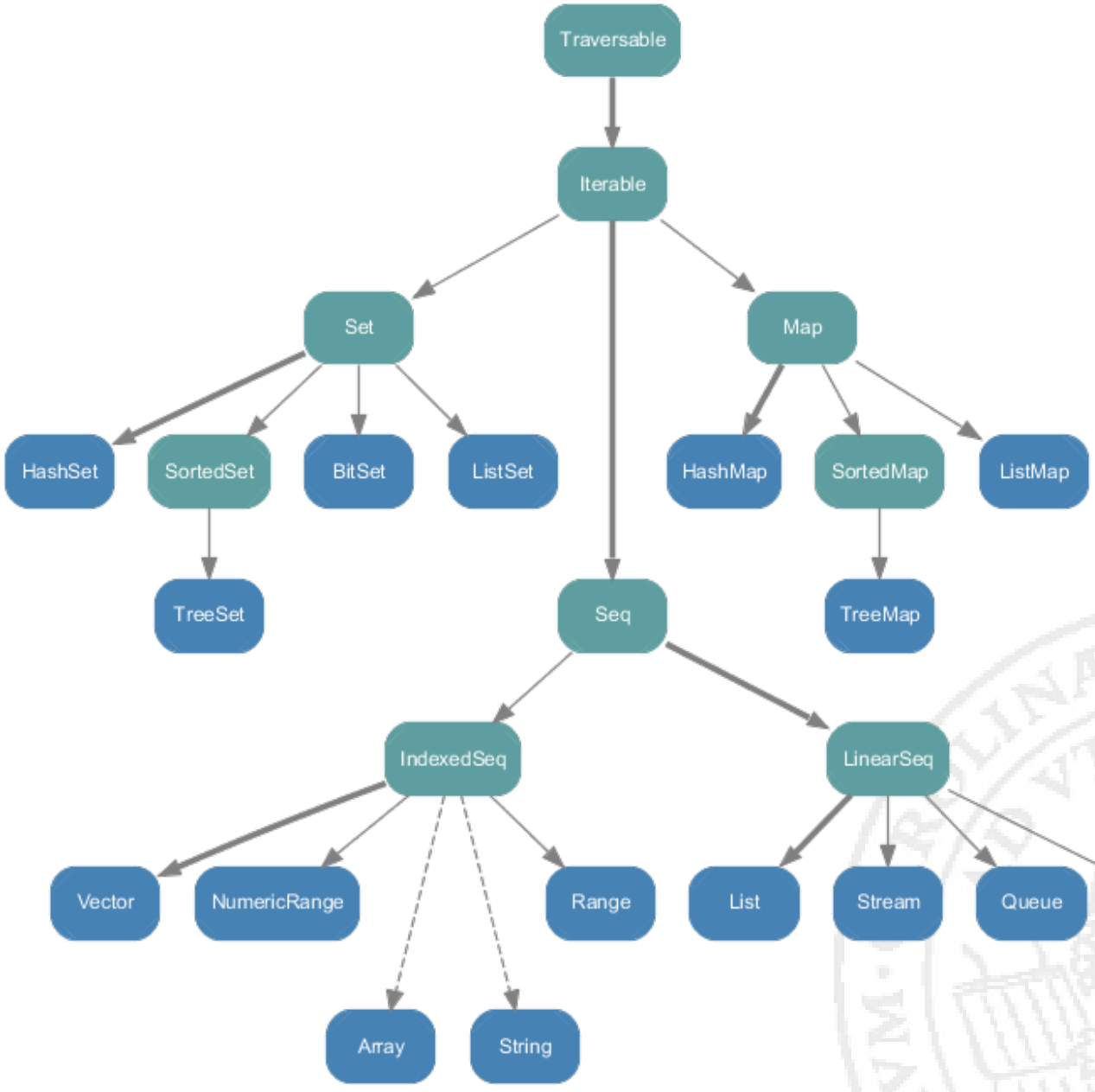
- This can be simplified even further, into:

```
(1 to 10).map(square)
```

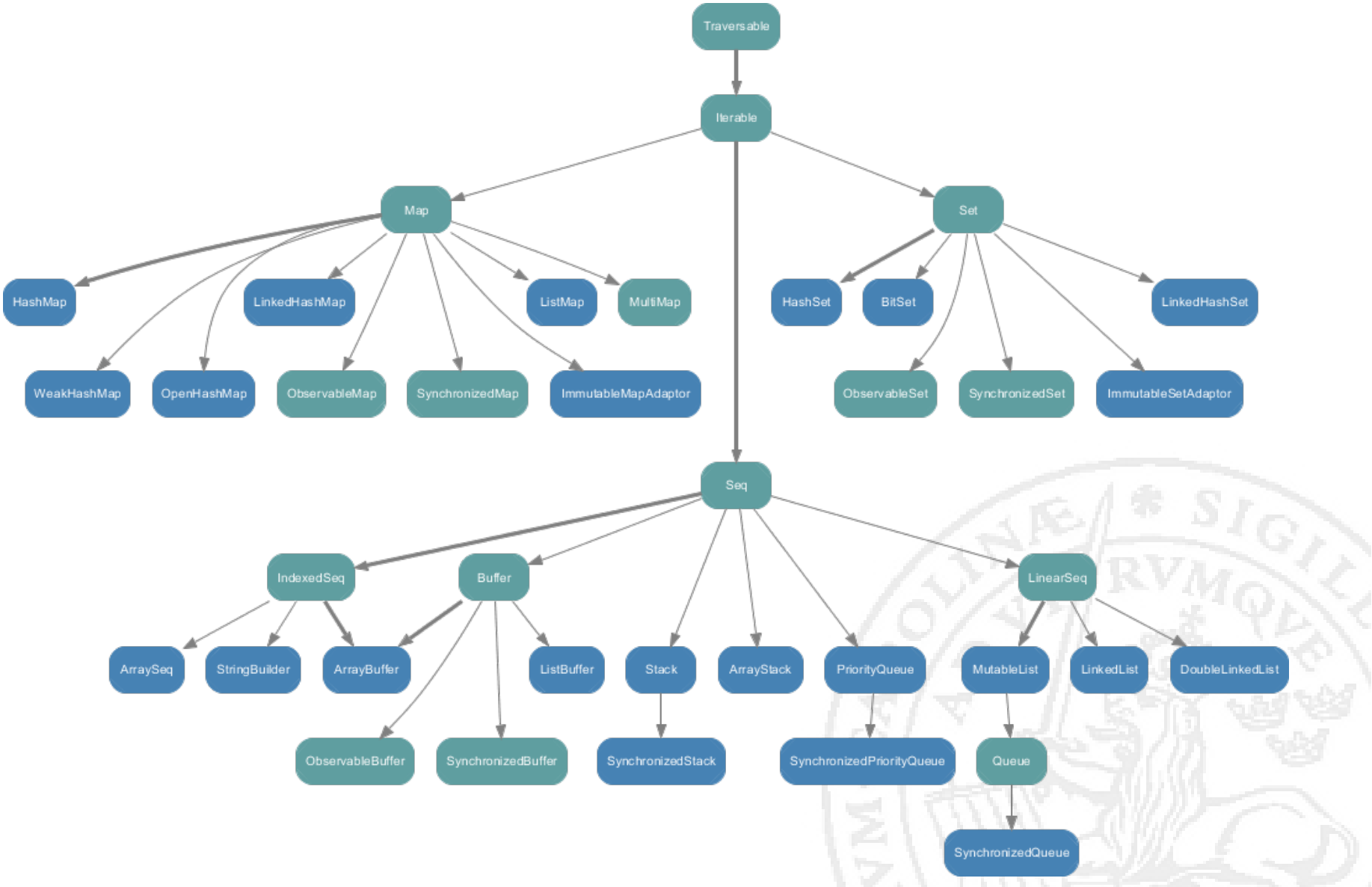
- We can use both proper functions and methods as parameters to `map` (thanks to the η-conversion).



scala.collection.immutable



scala.collection.mutable



Some methods on `Seq[A]`

- `map[B](f: (A) => B): Seq[B]`
- `filter(p: (A) => Boolean): Seq[A]`
- `foreach(f: (A) => Unit): Unit`
- `find(p: (A) => Boolean): Option[A]`
- `reduce[A1 >: A](op: (A1, A1) => B): B`
- `sum: A`

Creating and using collections

```
val smallPrimes = Seq(2, 3, 5, 7, 11)
val capitals = Map(
  "Sweden" -> "Stockholm",
  "Denmark" -> "Copenhagen",
  "Norway" -> "Oslo")
val squares = (1 to 10).map(v => (v, sq(v))).toMap
def isSmallPrime(n: Int) = smallPrimes.contains(n)
val letters = "Sequence of Chars".toSeq
```

Example

Exempel

Use the Scala REPL to calculate

$$\sum_{k=1}^{10} k^2$$

Implicit conversions

- Scala uses Java `Strings`.
- There is a collection class `StringOps` with many nice methods on strings -- if we want to use one of them on a `String`-object, Scala makes an *implicit conversion* into a `StringOps`-object.
- This way we can treat strings as sequences of `Chars`.
- Using implicit conversions we can seemingly add metods to any class.

Example

Exempel

Write a program which prints all palindromes given on the command line (hint: `String`:s are collections).

Example

Exempel

Write a program which prints all perfect numbers less than 10 000.

Example

Exempel

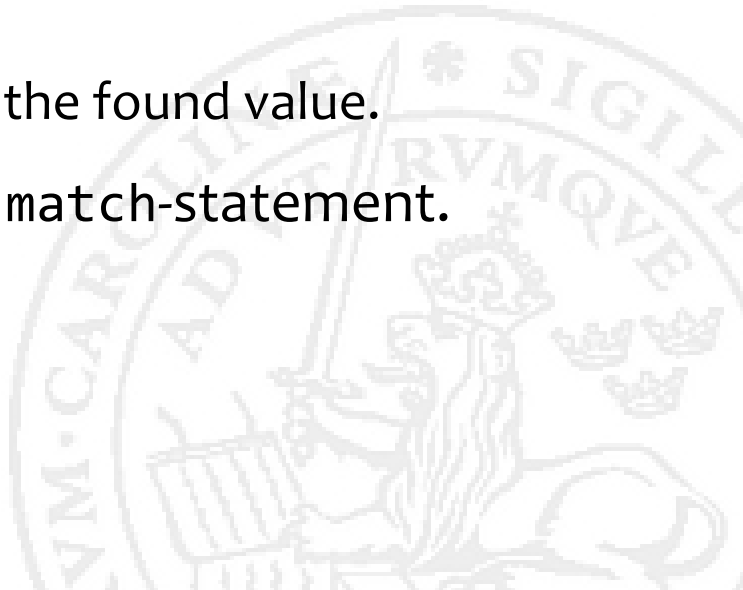
Write a program which prints the first four perfect numbers.

Avoiding `null`-pointers

- Some methods, such as

  ```
  find(p: (A) => Boolean)
  ```

  may or may not return a value -- such methods usually have the return type `Option[A]`.
- An `Option`-value may be either:
  - `None`: which is just nothing.
  - `Some(value)`: in which case `value` is the found value.
- We can check the return-value using a `match`-statement.



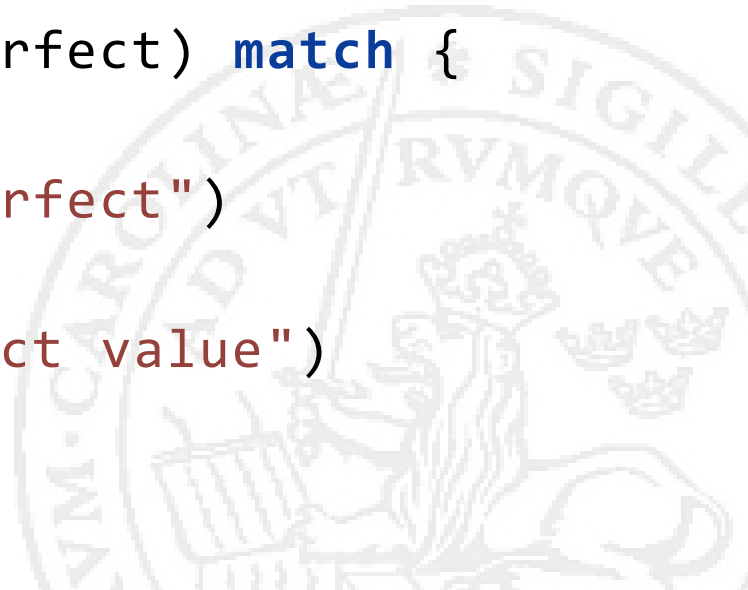
Some `match`-statements

- Matching `String`:s:

```
val capital = country match {
  case "Sweden" => "Stockholm"
  case "Denmark" => "Copenhagen"
  case "Norway" => "Oslo"
}
```

- Checking `Option`:s:

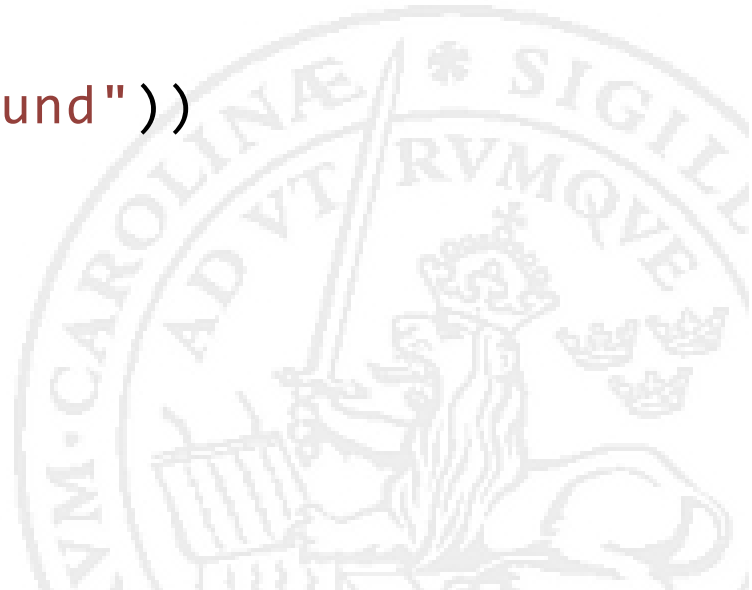
```
args.map(_.toInt).find(isPerfect) match {
  case Some(value) =>
    println(value + " is perfect")
  case None =>
    println("found no perfect value")
}
```



Using `Option`

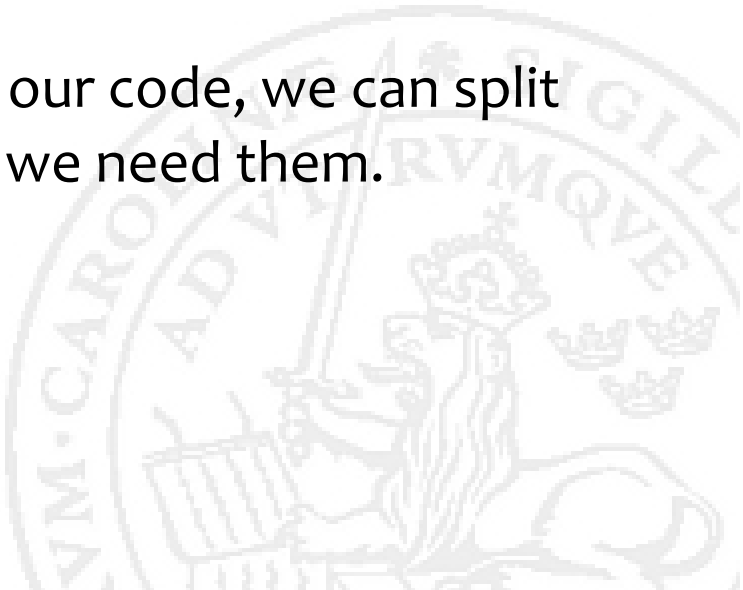
- We don't have to use `match`-statements to check our `Options`, instead we can use `map`, `filter`, etc:

```
println(args
        .find(isPalindrome)
        .filter(_.length < 5)
        .map(_.toUpperCase)
        .getOrElse("none found"))
```



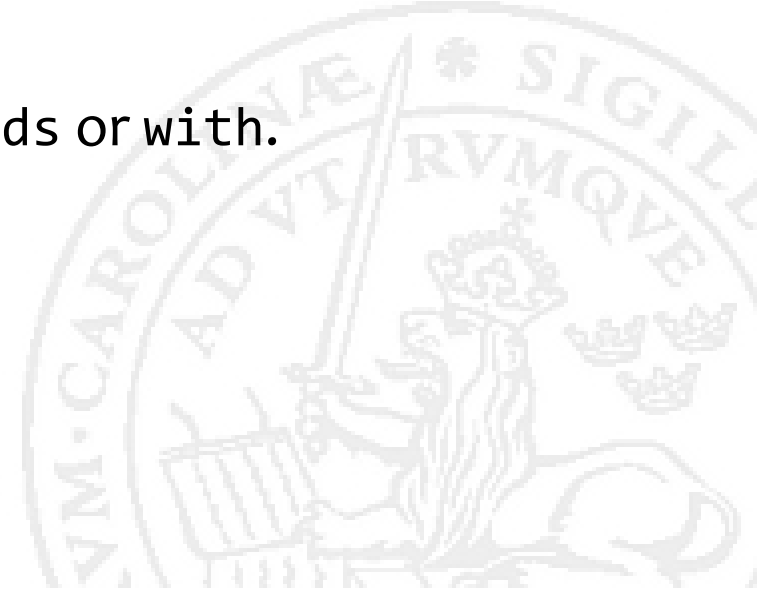
Traits

- A trait can be described as either:
  - a class without constructor, or
  - an interface, with implemented methods and/or attributes.
- Traits can be extended and mixed together in what looks like multiple inheritence -- using a technique called *linearization* Scala avoids the diamond problem.
- Traits are convenient for modularizing our code, we can split it into traits and mix them together as we need them.



Regular classes

- A Scala class looks a lot like a Java class, but its body can be seen as a constructor returning `this`.
- The parameters are not attributes, but they will be available inside the class.
- Parameters marked with `val` or `var` will be accesible from outside.
- We extend a class or trait using `extends` or `with`.

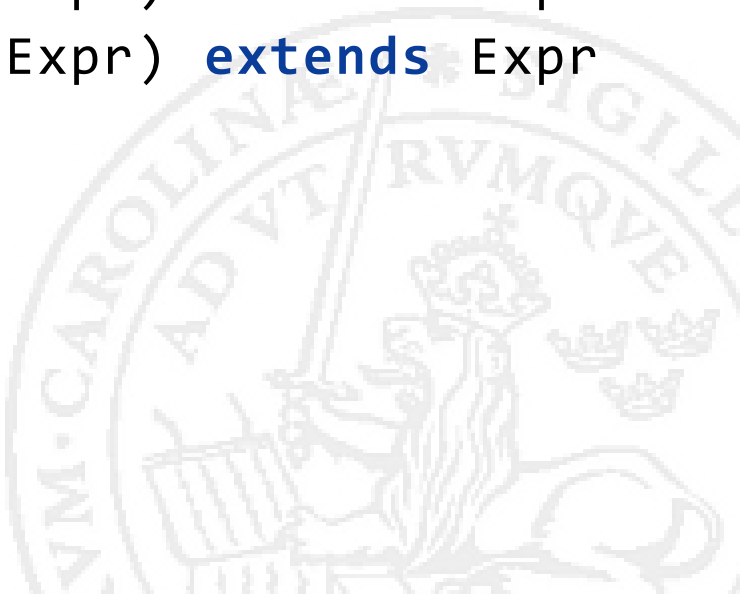


`case`-classes

- By marking our classes with the prefix `case`, we get:
  - A factory method for the class -- we can create a new instance without writing `new`.
  - All parameters gets a `val` prefix.
  - We get reasonable default definitions for `toString`, `hasCode` and `equals`.
- Case classes are great for matching.

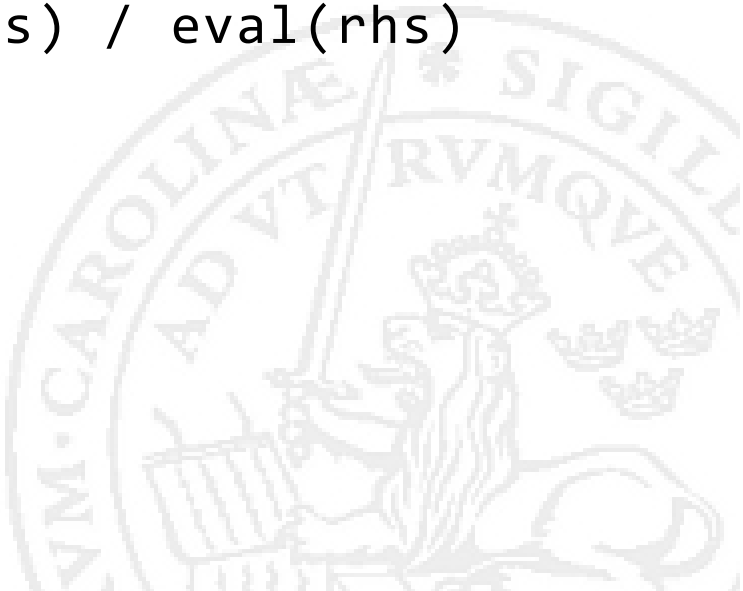
Arithmetic expressions

```
abstract class Expr
case class Number(value: Double) extends Expr
case class Add(lhs: Expr, rhs: Expr) extends Expr
case class Sub(lhs: Expr, rhs: Expr) extends Expr
case class Mul(lhs: Expr, rhs: Expr) extends Expr
case class Div(lhs: Expr, rhs: Expr) extends Expr
```



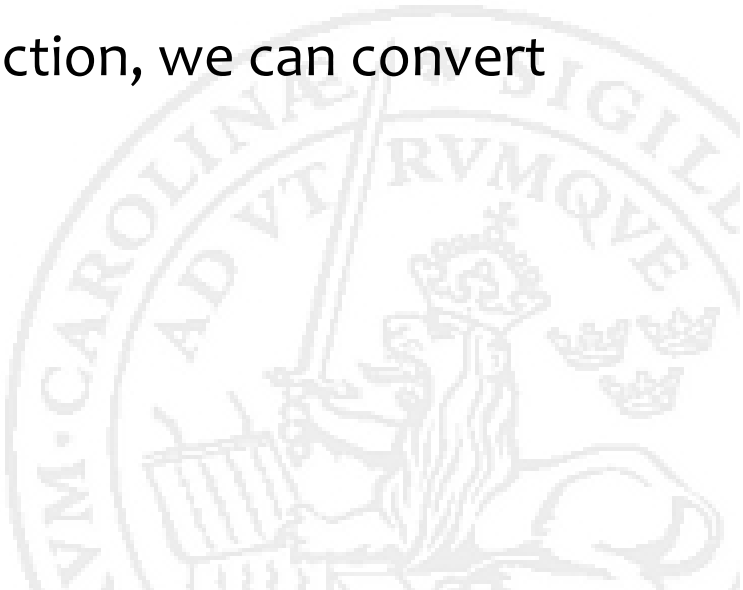
Evaluating arithmetic expressions

```
def eval(e: Expr): Double = e match {
  case Number(value) => value
  case Add(lhs, rhs) => eval(lhs) + eval(rhs)
  case Sub(lhs, rhs) => eval(lhs) - eval(rhs)
  case Mul(lhs, rhs) => eval(lhs) * eval(rhs)
  case Div(lhs, rhs) => eval(lhs) / eval(rhs)
}
```



Parsers

- A parser is an object extending `Input => ParseResult[T]`
- We define one method per production in our grammar, each method is its own parser.
- The parsers have methods like ~, |, etc. for composing and alternatives.
- If we call the ^^-method for any production, we can convert the parsed `String` into an AST-node.



Example -- a very simple calculator

⟨*expr*⟩ ::= <term> { '+' <term> }

⟨*term*⟩ ::= <factor> { '*' <factor> }

⟨*factor*⟩ ::= <floatingPointNumber> | '(' <expr> ')'

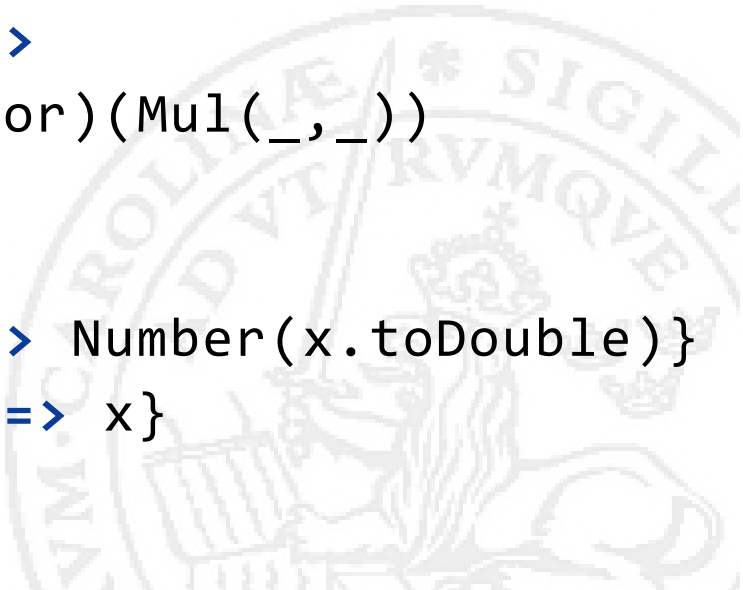
Just parsing

```
trait ArithmeticParser extends JavaTokenParsers {

  def expr: Parser[Any] =
    term ~ rep("+" ~ term)
  def term: Parser[Any] =
    factor ~ rep("*" ~> factor)
  def factor: Parser[Any] =
    floatingPointNumber | "(" ~ expr ~ ")"
}
```

Building an AST

```
def expr: Parser[Expr] =
  term ~ rep("+" ~ term) ^^ {
    case term ~ following =>
      following.map{case s~e => e}.
                foldLeft(term)(Add(_,_))
  }
def term: Parser[Expr] =
  factor ~ rep("*" ~> factor) ^^ {
    case factor ~ following =>
      following.foldLeft(factor)(Mul(_,_))
  }
def factor: Parser[Expr] =
  floatingPointNumber ^^ {x => Number(x.toDouble)}
  | "(" ~> expr <~ ")" ^^ {x => x}
```



Running our calculator

```
object Calculator
extends App with ArithmeticParser {

  parseAll(expr, args(0)) match {
    case Success(e, _)   =>
      println(eval(e))
    case Failure(msg, _) =>
      println("Syntax error: " + msg)
  }
}
```

The End