Programs as Data

The Scala language, an overview

Peter Sestoft

2013-11-11*

Agenda

- Object-oriented programming in Scala
  - Classes
  - Singletons (object)
  - Traits
- Compiling and running Scala programs
- Functional programming in Scala
  - Type List[T], higher-order and anonymous functions
  - Case classes and pattern matching
  - The Option[T] type
  - For-expressions (comprehensions à la Linq)
- Type system
  - Generic types
  - Co- and contra-variance
  - Type members

Scala object-oriented programming

- Scala is designed to
  - work with the Java platform
  - be somewhat easy to pick up if you know Java
  - be much more concise and powerful
- Scala has classes, like Java and C#
- And abstract classes
- But no interfaces
- Instead, traits = partial classes

- By Martin Odersky and others, EPFL, CH
- Get Scala from http://www.scala-lang.org/
- You will also need a Java implementation

Java and Scala

Java

```java
class PrintOptions {
  public static void main(String[] args) {
    for (String arg : args)
      if (arg.startsWith("-"))
        System.out.println(arg.substring(1));
  }
}
```

Singleton class; no statics

Declaration syntax

Array[T] is generic type

Sca

for expression

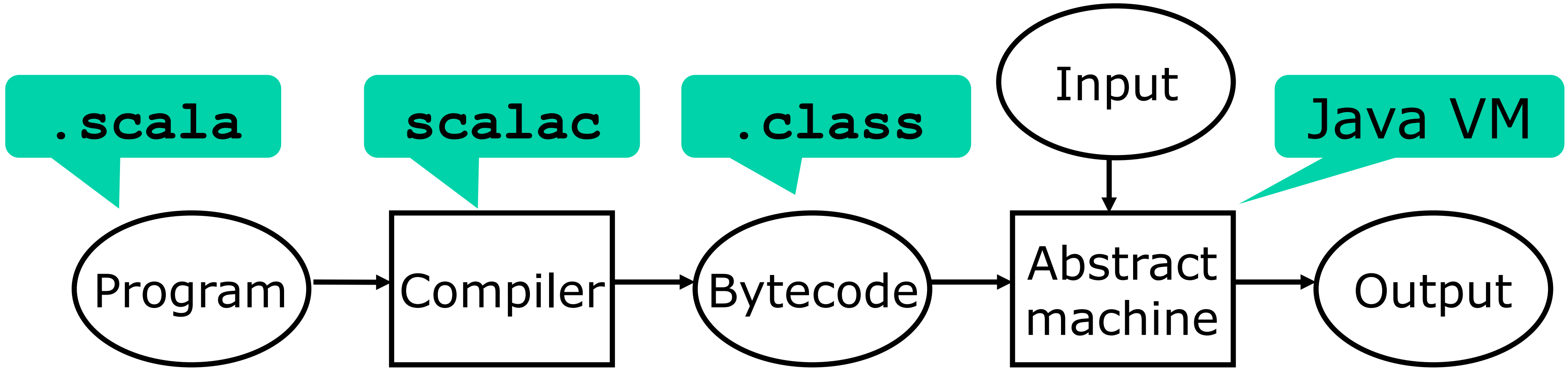
```scala
object PrintOptions {
  def main(args: Array[String]) = {
    for (arg <- args; if arg startsWith "-")
      println(arg substring 1)
  }
}
```

Can use Java class libraries

Compiling and running Scala

- Use **scalac** to compile *.scala files
- Use **scala** to run the object class file
  - uses **java** runtime with Scala's libraries

```
sestoft@mac$ scalac Example.scala
sestoft@mac$ scala PrintOptions -help foo -verbose bar baz
help
verbose
```



Interactive Scala

- Scala also has an interactive top-level
  - Like F#, Scheme, most functional languages

```
sestoft@mac ~/scala $ scala
Welcome to Scala version 2.10.3 (Java HotSpot(TM) 64-Bit...).

scala> def fac(n: Int): Int = if (n==0) 1 else n*fac(n-1)
fac: (n: Int)Int

scala> fac(10)
res0: Int = 3628800
```

java.util.BigInteger

```
scala> def fac(n: Int): BigInt = if (n==0) 1 else n*fac(n-1)
fac: (n: Int)BigInt

scala> fac(100)
res1: BigInt = 9332621544394415268169923885626670049071596
8264381621468592963895217599993229915608941463976156518286
253697920827223758251185210916864000000000000000000000000
```

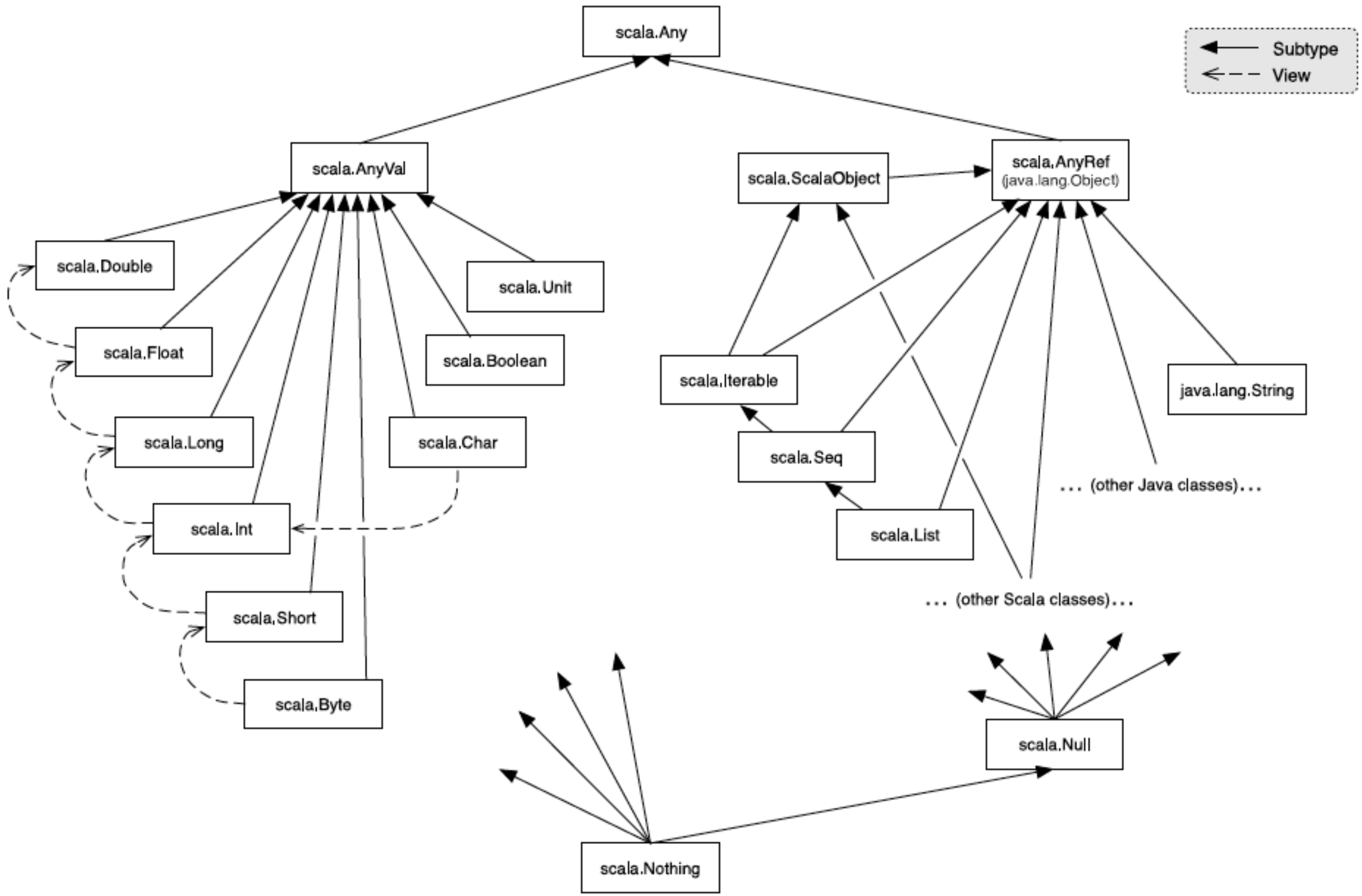
Much lighter syntax

- All declarations start with keyword (no **`int x`**)
- **`Unit`** and **`()`** and **`{}`** can often be left out
- All values are objects and have methods
  - So **`2.to(10)`** is a legal expression
- All operators are methods
  - So **`x+y`** same as **`x.+(y)`**
- Method calls can be written infix
  - So **`2.to(10)`** can be written **`2 to 10`**

```
for (x <- 2 to 10)
  println(x)
```

Method looks like infix "operator"

Uniform type system (like C#)



Singletons (object declaration)

- Scala has no static fields and methods
- An **`object`** is a singleton instance of a class

```
object PrintOptions {
  def main(args: Array[String]) = {
    ...
  }
}
```

- Can create an application as a singleton App

```
object ListForSum extends App {
  val xs = List(2,3,5,7,11,13)
  var sum = 0
  for (x <- xs)
    sum += x
  println(sum)
}
```

Immutable (final, readonly)

Mutable

Classes

Primary constructor

Field *and* parameter declaration

```
abstract class Person(val name: String) {
  def print()
}

class Student(override val name: String,
              val programme: String)
  extends Person(name)
{
  def print() {
    println(name + " studies " + programme)
  }
}
```

Abstract method

```
val p: Person = new Student("Ole", "SDT");
p.print()
~~p.print~~
println(p.name)
```

Method call

Same, bad style

Field access

Anonymous subclass and instance

```
val s = new Student("Kasper", "SDT") {
  override def print() {
    super.print()
    println("and does much else")
  }
}
```

Define anonymous subclass of Student, create an instance s

```
scala> s.print()
Kasper studies SDT
and does much else
```

- Similar to Java's anonymous inner classes:

Interface

```
pause.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    canvas.run(false);
  }
});
```

Define anonymous class implementing the interface & make instance

Traits: fragments of classes

- Can have fields and methods, but no instances

```
trait Counter {
  private var count = 0
  def increment() { count += 1 }
  def getCount = count
}
```

- Allows mixin: multiple “base classes”

```
class CountingPerson(override val name: String)
  extends Person(name) with Counter
{
  def print() {
    increment()
    println(name + " has been printed " + getCount + " times")
  }
}
```

Any number of traits can be added

```
val q1: Person = new CountingPerson("Hans")
val q2: Person = new CountingPerson("Laila")
q1.print(); q1.print();
q2.print(); q2.print(); q2.print()
```

Example: The Ordered trait (from package scala.math)

- A trait can define methods:

Abstract

```
trait Ordered[A] extends java.lang.Comparable[A] {
  def compare(that: A): Int
  def <  (that: A): Boolean = (this compare that) <  0
  def >  (that: A): Boolean = (this compare that) >  0
  def <= (that: A): Boolean = (this compare that) <= 0
  def >= (that: A): Boolean = (this compare that) >= 0
}
```

Concrete

```
class OrderedIntPair(val fst: Int, val snd: Int)
   extends Ordered[OrderedIntPair]
{
  def compare(that: OrderedIntPair): Int = { ... }
}
```

```
val pair1 = new OrderedIntPair(3, 4) ...
if (pair1 > pair2)
  System.out.println("Great");
```

Generic class List[T], much like F#

- A list
  - has form **`Nil`**, the empty list, or
  - has form **`x::xr`**, first element is **`x`**, rest is **`xr`**
- A list of integers, type List[Int]:

```
List(1,2,3)
```

```
1 :: 2 :: 3 :: Nil
```

- A list of Strings, type List[String]:

```
List("foo", "bar")
```

- A list of pairs, type List[(String, Int)]

```
List(("Peter", 1962), ("Lone", 1960))
```

Functional programming

- Supported just as well as object-oriented
  - Four ways to print the elements of a list

```
for (x <- xs)
  println(x)
```

```
xs foreach { x => println(x) }
```

Actual meaning of for-expression

```
xs.foreach(println)
```

```
xs foreach println
```

- Anonymous functions; three ways to sum

```
var sum = 0
for (x <- xs)
  sum += x
```

```
var sum = 0
xs foreach { x => sum += x }
```

As F#, ML, C#

```
xs foreach { sum += _ }
```

List functions, pattern matching

- Compute the sum of a list of integers

```
def sum(xs: List[Int]): Int =
  xs match {
    case Nil    => 0
    case x::xr => x + sum(xr)
  }
```

When **xs** has form **Nil**

When **xs** has form **x::xr**

Like F#

- A generic list function

Type parameter

```
def repeat[T](x: T, n: Int): List[T] =
  if (n==0)
    Nil
  else
    x :: repeat(x, n-1)
```

```
repeat("abc", 4)
```

Fold and foreach on lists, like F#

- Compute a list sum using a fold function

```
def sum1(xs: List[Int]) =
  xs.foldLeft(0)((res,x)=>res+x)
```

Value at **`Nil`**

Value at **`x::xr`**

- Same, expressed more compactly:

```
def sum2(xs: List[Int]) =
  xs.foldLeft(0)(_+_)
```

- Method **`foreach`** from trait Traversable[T] :

```
def foreach[T](xs: List[T], act: T=>Unit): Unit =
  xs match {
    case Nil   => { }
    case x::xr => { act(x); foreach(xr, act) }
  }
```

Case classes and pattern matching

- Good for representing tree data structures
- Abstract syntax example: An Expr is either
  - a constant integer
  - or a binary operator applied to two expressions

```
type expr =
   | CstI of int
   | Prim of string * expr * expr
```

F#

```
sealed abstract class Expr
case class CstI(value: Int)
  extends Expr
case class Prim(op: String,
                e1: Expr,
                e2: Expr)
  extends Expr
```

Scala

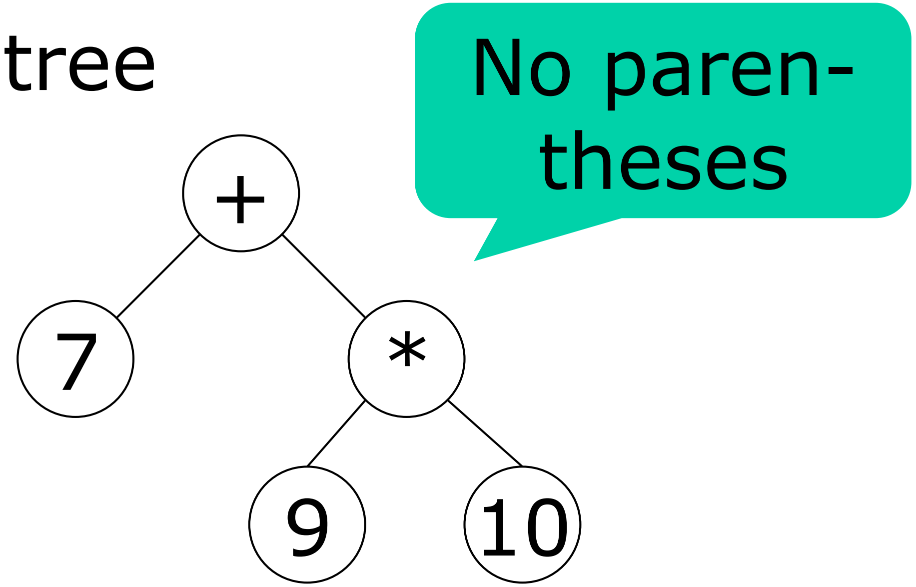
Also, case classes have:
- equality and hashcode
- copy method, keyword args
- public val fields
- no need for **new** keyword
- good print format (toString)

Representation of expressions

- An expression is a tree

```
7 + 9 * 10
```

```
7 + (9 * 10)
```



- Representing it with case class objects:

```
Prim("+",
     CstI(7),
     Prim("*",
          CstI(9),
          CstI(10)))
```

Plain evaluation of expressions

```
def eval(e: Expr): Int = {
  e match {
    case CstI(i) => i
    case Prim(op, e1, e2) =>
      val v1 = eval(e1)
      val v2 = eval(e2)
      op match {
        case "+" => v1 + v2
        case "*" => v1 * v2
        case "/" => v1 / v2
      }
}
```

```
eval(Prim("+", CstI(42), CstI(27)))
```

The built-in Option[T] case class

- Values **`None`** and **`Some(x)`** as in F#, or C# null:

```
def sqrt(x: Double): Option[Double] =
  if (x<0) None else Some(math.sqrt(x))
```

- Use pattern matching to distinguish them

```
def mul3(x: Option[Double]) =
  x match {
    case None    => None
    case Some(v) => Some(3*v)
  }
```

- Or, more subtly, use **`for`**-expressions:

```
def mul3(x: Option[Double]) =
  for ( v <- x )
    yield 3*v
```

Exercise!

Scala for-expressions

```
for (x <- primes; if x*x < 100) yield 3*x
```

generator | filter | transformer

- Just like C#/Linq:

```
from x in primes where x*x < 100 select 3*x
```

- Operations: `Traversable.groupBy`, `Seq.sortWith`
- Aggregates (sum...) definable with `foldLeft`

More for-expression examples

- Example sum

```
(for (x <- 1 to 200; if x%5!=0 && x%7!=0)
  yield 1.0/x).foldLeft (0.0) (_+_)
```

```
(from x in Enumerable.Range(1, 200)
 where x%5!=0 && x%7!=0
 select 1.0/x).Sum()
```

C# Linq

- All pairs (i,j) where i>=j and i=1..10

```
for (i <- 1 to 10; j <- 1 to i)
  yield (i,j)
```

Co-variance and contra-variance (as C#, with "+"=out and "-"=in)

- If generic class C[T] only outputs T's it may be made co-variant in T:

```
class C[+T](x: T) {
  def outputT: T = x
}
```

- If generic class C[T] only inputs T's it may be made contra-variant in T:

```
class C[-T](x: T) {
  def inputT(y: T) { }
}
```

- Scala's *immutable* collections are co-variant

Scala co/contra-variance examples

```
trait Iterable[+A] extends ... {
  def iterator: Iterator[A]
}
trait Iterator[+A] extends ... {
  def hasNext: Boolean
  def next(): A
}
```

As for C#
IEnumerable<A>
IEnumerator<A>

```
trait MyComparer[-T] {
  def compare(x: T, y: T) : Boolean = ...
}
```

Scala's actual Comparator is from Java and is not contravariant

Type members in classes

- May be abstract; may be further-bound

```
class Food
abstract class Animal {
  type SuitableFood <: Food
  def eat(food: SuitableFood)
}
```

Abstract type member

```
class Grass extends Food
class Cow extends Animal {
  type SuitableFood = Grass
  override def eat(food : SuitableFood) { }
}
```

Final-binding

```
class DogFood extends Food
class Dog extends Animal {
  type SuitableFood = DogFood
  override def eat(food : SuitableFood) { }
}
```

Simple Scala Swing example

- Scala interface to Java Swing

```
import scala.swing._

object FirstSwingApp extends SimpleSwingApplication {
  def top = new MainFrame {
    title = "First Swing App"
    contents = new Button {
      text = "Click me"
    }
  }
}
```

```
reactions += {
  case scala.swing.event.ButtonClicked(_) =>
    println("Button clicked")
}
```

Other Scala features

- Implicit arguments
- Pattern matching on user-defined types, non-case classes
- Actors for concurrency, the Akka library
- Simple build tool **`sbt`**
- Developer and language design community
- Limited tail call optimization (Java platform)
- EU project on domain-specific languages for parallel programming

Revealing Scala internals

- Useful because of
  - Syntactic abbreviations
  - Compile-time type inference
- To see possibilities, run **`scalac -X`**

```
sestoft@mac $ scalac -Xprint:typer Example.scala
[[syntax trees at end of typer]] // Example.scala
package <empty> {
  object PrintOptions extends scala.AnyRef {
    def <init>(): PrintOptions.type = {
      PrintOptions.super.<init>();
      ()
    };
    def main(args: Array[String]): Unit =
      scala.this.Predef.refArrayOps[String](args)
      .withFilter(((arg: String) => arg.startsWith("-")))
      .foreach[Unit](((arg: String) =>
                       scala.this.Predef.println(arg.substring(1))))
  }
}
```

Commercial use of Scala

- Twitter, LinkedIn, FourSquare, ... use Scala
- Also some Copenhagen companies
  - Because it works with Java libraries
  - And Scala code is shorter and often much clearer
- Several ITU students and PhD students use Scala, eg.
  - David, for embedded domain-specific languages
  - Hannes, for Eclipse plugins

Java compatible

References

- *A Scala tutorial for Java programmers*, 2011
- *An overview of the Scala programming language*, 2006
- Odersky: *Scala by Example*, 2011.
- Find the above at: http://www.scala-lang.org
- Documentation: http://docs.scala-lang.org
- Odersky, Spoon, Venners: *Programming in Scala*, 2nd ed, 2011 (book)
- http://www.scala-lang.org/docu/files/collections-api/collections.html
- Traits in Scala: http://stackoverflow.com/questions/1992532/monad-trait-in-scala
- Odersky's Coursera course on Scala: https://www.coursera.org/course/progfun

What's next

- Monday 18 November
  - Advanced Scala features: implicits, kinds, ...
  - Monads
- Monday 25 November
  - Haskell, a lazy functional language

- Friday 20 December at 1000: spørgetime
- Thursday 2 Jan 0900 to Friday 3 Jan 1400: 29-hour take-home exam