Updated: 2021-01-23

2.6 Scala in context

A little history, a little general description

Northeastern University

What is Scala?

- Scala
  - stands for *Scalable Language*
  - is a "blend of **object-oriented** and **functional programming** concepts in a **statically typed** language"
  - is a <u>general-purpose</u> language
  - runs on the **JVM** (Java Virtual Machine)* via "bytecode," just the same as Java, Groovy, Clojure, Kotlin, etc.
  - is bi-directionally compatible with Java (and other JVM languages). Although Scala has its own definitions for things such as ***List***, you can, if you like, use ***java.util.List***

* Note that ***Scala.js*** can also compile to Javascript in your browser

- *Happy birthday to you,*
- *Happy birthday to you,*
- *Happy birthday dear*

- *Happy birthday to you.*

20 candles for the cake.

Very quick History of Scala

- Design began in 2001 by Martin Odersky at EPFL
  - Many ideas from existing languages contributed
- First public release 2004
- Version 2.0 in 2006
- Typesafe (now Lightbend) launched in 2011
- Scala.js version 0.1 released November 2013
- 2.11 released April 2014
- 2.12 released November 2016 (mainly internal improvements)
- 2.13 released 2019.
- “Dotty” (aka Scala 3) to be released in early/mid 2021

Code Example – interoperability with Java

```
package edu.neu.coe.scala
case class Mailer(server: String) {
  import java.net._
  val s = new Socket(InetAddress.getByName(server), 587)
  val out = new java.io.PrintStream(s.getOutputStream)
  def doMail(message: String, filename: String) = {
    val src = scala.io.Source.fromFile(filename)
    for (entry <- src.getLines.map(_.split(","))) out.println(s"To: ${entry(0)}\nDear
${entry(1)},\n$message")
    src.close
    out.flush()
  }
  def close() = {
    out.close()
    s.close()
  }
}
object EmailApp {
  def main(args: Array[String]): Unit = {
    val mailer = new Mailer("smtp.google.com")
    mailer.doMail(args(0), "mailinglist.csv")
    mailer.close
  }
}
```

Notice that we can place imports wherever needed.

Notice how we are including some Java types here, although that is unusual.

Notice how easy it is to print strings with the values of variables.

Notice that there are no semi-colons

Notice that Scala uses [] for types, not <>

Here we create a main program which is just like one in Java, except here it is not marked “static”. There is a better way, however.

Another code example

```scala
object Ratios {
  /**
  * Method to calculate the Sharpe ratio, given some history and a constant risk-free rate
  * @param investmentHistory the history of prices of the investment, one entry per period, starting with the most recent price
  * @param periodsPerYear    the number of periods per year
  * @param riskFreeRate      the risk-free rate, each period (annualized)
  * @return the Sharpe ratio
  */
 def sharpeRatio(investmentHistory: Seq[Double], periodsPerYear: Int, riskFreeRate: Double): Double =
    sharpeRatio(investmentHistory, periodsPerYear, LazyList.continually(riskFreeRate))
  /**
  * Method to calculate the Sharpe ratio, given some history and a set of benchmark returns
  * @param investmentHistory the history of prices of the investment, one entry per period, starting with the most recent price
  * @param periodsPerYear    the number of periods per year
  * @param riskFreeReturns   the actual (annualized) returns on the risk-free benchmark
  * @return the Sharpe ratio
  */
 def sharpeRatio(investmentHistory: Seq[Double], periodsPerYear: Int, riskFreeReturns: Stream[Double]): Double = {
    // calculate the net gains of the investment
  val xs = for (x <- investmentHistory.sliding(2)) yield x.head - x.last
    // calculate the net returns on the investment
  val ys = (xs zip investmentHistory.iterator) map { case (x, p) => x / p }
    // calculate the annualized returns
  val rs = for (y <- ys) yield math.pow(1 + y, periodsPerYear) - 1
    // calculate the Sharpe ratio
  sharpeRatio(rs.toSeq, riskFreeReturns)
 }
  /**
  * Method to calculate the Sharpe ratio, given actual and benchmark returns
  * @param xs the actual historical returns (expressed as a fraction, plus or minus) of the given investment
  * @param rs the actual historical returns (expressed as a fraction, plus or minus) of the benchmark
  * @return the Sharpe ratio
  */
 def sharpeRatio(xs: Seq[Double], rs: Stream[Double]): Double = {
    val count = xs.size
    // calculate the excess returns
  val zs = (xs zip rs) map { case (r, f) => r - f }
    // calculate the Sharpe ratio
  zs.sum / count / math.sqrt((xs map (r => r * r)).sum / count)
 }
}
```

Notice that wherever we define a variable or parameter, it's **always** of form *name : Type*

Look at these ***for*** loops. Each of them <u>returns a value</u> via the ***yield*** keyword

Notice that "variables" must be preceded by "val" or "var" but don't necessarily need a type annotation.

Notice that we don't need a ***return*** keyword.

Newton’s Approximation (FORTRAN)

```
program newton
   Print *, "Newton's Approximation"
   x = 1.0
   maxTries = 100
10 y = cos(x) - x
   if (abs(y)<1E-7) goto 20
   x = x + y / (sin(x) + 1)
   maxTries = maxTries - 1
   if (maxTries>0) goto 10
   Print *, "failed"
   goto 30
20 Print *, x
30 Print *, "Goodbye"
end program newton
```

Newton’s Approximation (Java8)

```
import java.util.function.DoubleFunction;
public class Newton {
  public Newton(final String equation, final DoubleFunction<Double> f, final DoubleFunction<Double> dfbydx) {
          this.equation = equation;
          this.f = f;
          this.dfbydx = dfbydx;
    }
  public Either<String, Double> solve(final double x0, final int maxTries, final double tolerance) {
          double x = x0;
          int tries = maxTries;
          for (; tries > 0; tries--)
               try {
                    final double y = f.apply(x);
                    if (Math.abs(y) < tolerance) return Either.right(x);
                    x = x - y / dfbydx.apply(x);
               } catch (Exception e) {
                    return Either.left("Exception thrown solving " + equation + "=0, given x0=" + x0 + ", maxTries=" + maxTries + ", and tolerance=" +
tolerance + " because " + e.getLocalizedMessage());
               }
          return Either.left(equation + "=0 did not converge given x0=" + x0 + ", maxTries=" + maxTries + ", and tolerance=" + tolerance);
    }
    public static void main(String[] args) {
   Newton newton = new Newton("cos(x) - x", (double x) -> Math.cos(x) - x, (double x) -> -Math.sin(x) - 1);
   Either<String, Double> result = newton.solve(1.0, 200, 1E-7);
   result.apply(
      System.err::println,
                    aDouble -> {
        System.out.println("Good news! " + newton.equation + " was solved: " + aDouble);
         });
    }
    private final String  equation;
    private final DoubleFunction<Double> f;
    private final DoubleFunction<Double> dfbydx;
}
```

Newton (Scala)

```scala
package edu.neu.coe.csye7200
import scala.annotation.tailrec
import scala.util._
case class Newton(f: Double => Double, dfbydx: Double => Double) {
  private def step(x: Double, y: Double) = x - y / dfbydx(x)
  def solve(tries: Int, threshold: Double, initial: Double): Try[Double] = {
    @tailrec def inner(r: Double, n: Int): Try[Double] = {
          val y = f(r)
          if (math.abs(y) < threshold) Success(r)
          else if (n == 0) Failure(new Exception("failed to converge"))
          else inner(step(r, y), n - 1)
      }
    inner(initial, tries)
    }
}
object Newton extends App {
   val newton = Newton( x => math.cos(x) - x, x => -math.sin(x) - 1 )
   newton.solve(10, 1E-10, 1.0) match {
      case Success(x) => println(s"the solution to math.cos(x) - x is $x")
      case Failure(t) => System.err.println(t.getLocalizedMessage)
   }
}
```

A case class is a class but with fields that are visible both for construction and extraction; it also comes with other useful methods.

the result of *inner* and so of *solve* is a *Try[Double].* This is a bit like the *Either* of the Java version, but it has value which is either a Double (*Success*) or an Exception (*Failure*).

lambdas define f and dfbydx

Another Example: Ingest

```scala
package edu.neu.coe.scala.ingest

import scala.io.Source

class Ingest[T : Ingestible] extends (Source => Iterator[T]) {
  def apply(source: Source): Iterator[T] = source.getLines.toSeq.map(e =>
implicitly[Ingestible[T]].fromStrings(e.split(",").toSeq)).iterator
}

trait Ingestible[X] {
  def fromStrings(ws: Seq[String]): X
}

case class Movie(properties: Seq[String])

object Ingest extends App {
  trait IngestibleMovie extends Ingestible[Movie] {
    def fromStrings(ws: Seq[String]): Movie = Movie.apply(ws)
  }
  implicit object IngestibleMovie extends IngestibleMovie

   val ingester = new Ingest[Movie]()
   if (args.length>0) {
      val source = Source.fromFile(args.head)
      for (m <- ingester(source)) println(m.properties.mkString(","))
      source.close()
    }
}
```

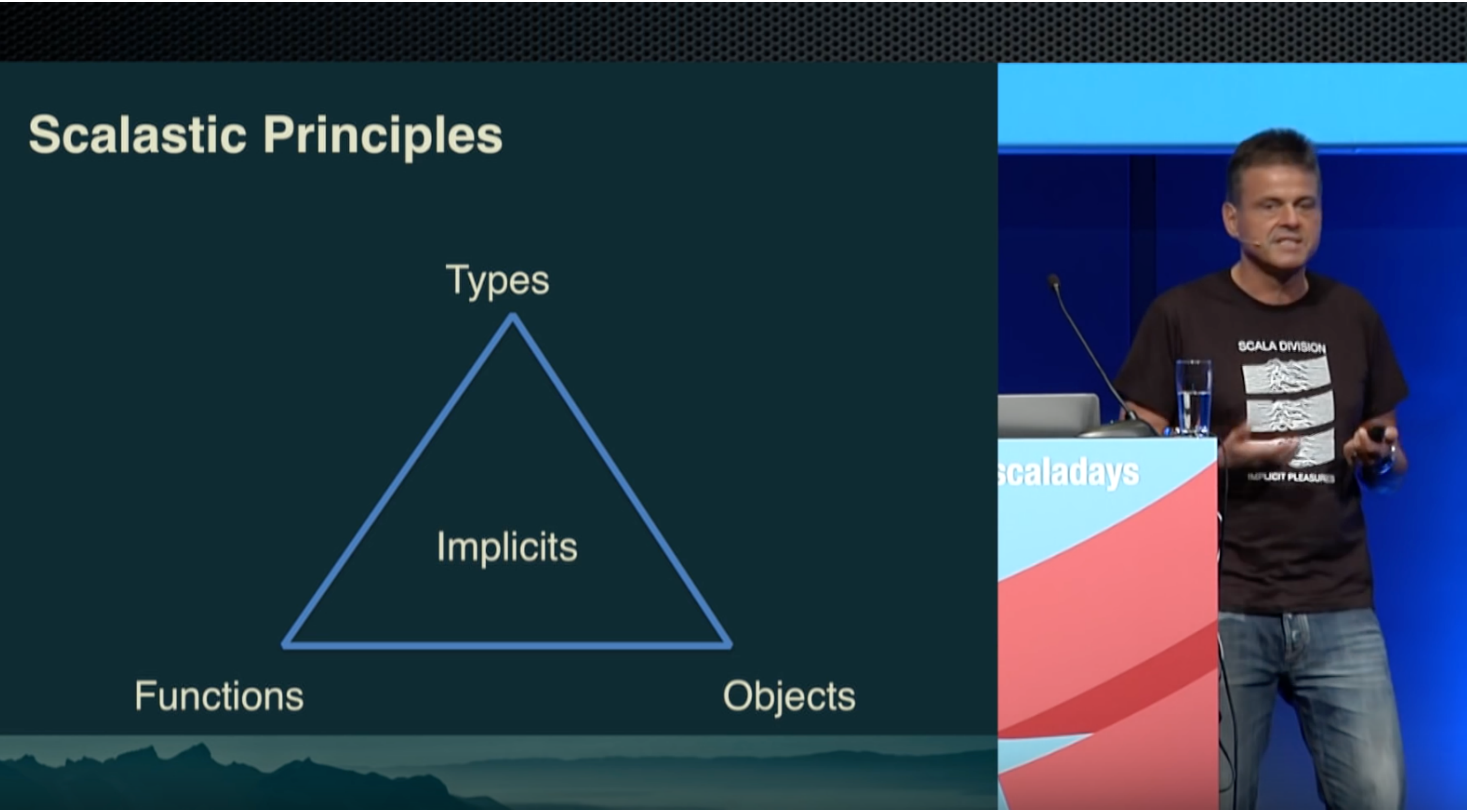
This example of Scala uses some of the advanced features of the language, including ***type classes*** and ***implicits*** to allow us a truly generic solution to ingestion. Don't expect to understand it just yet.

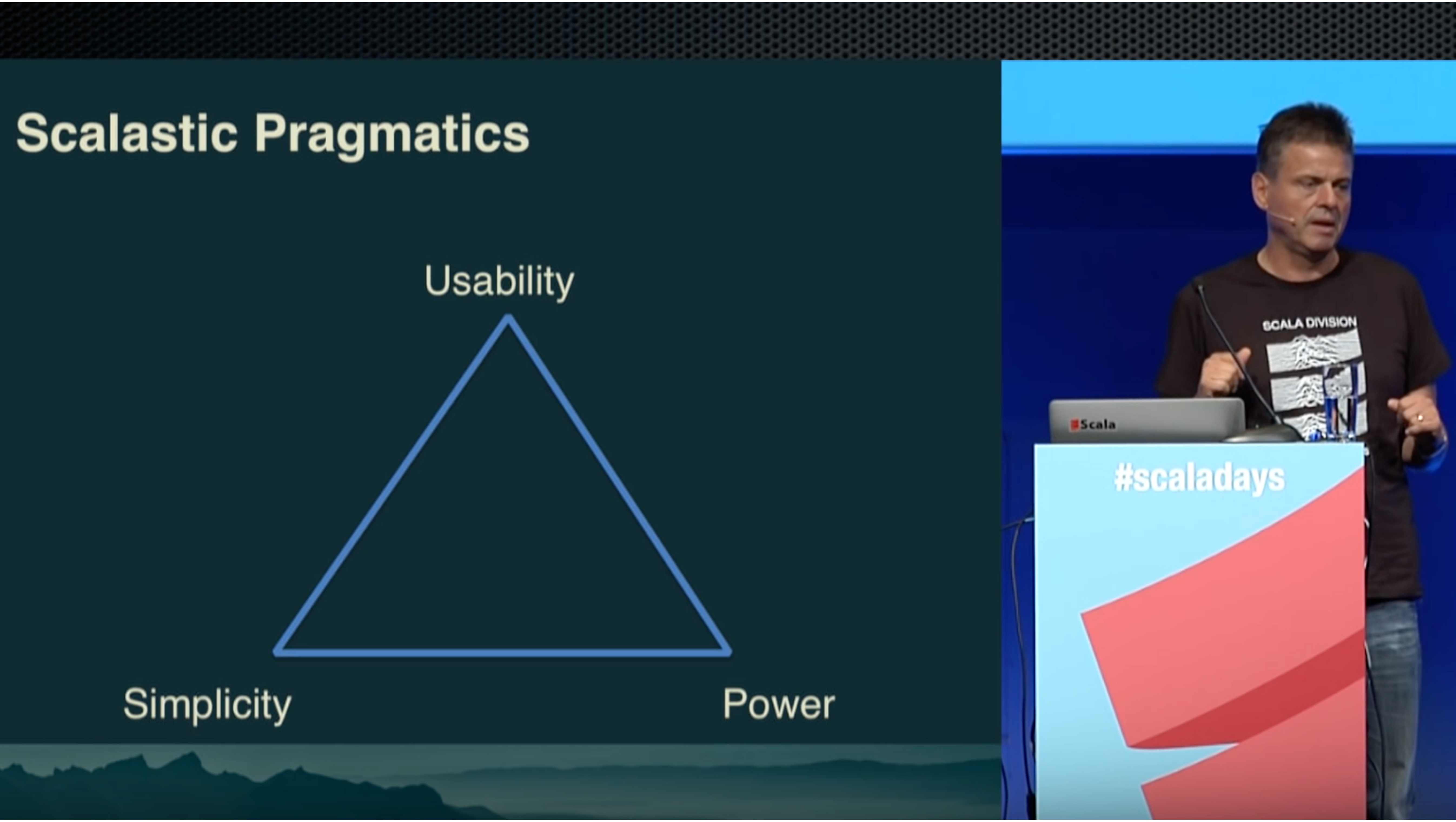
What Scala is not?

- Scala *is not*
  - a “scripting language,” although it can be and is used in many places scripts might be found, such as in the *build.sbt* file (used to build Scala applications)
  - a domain-specific-language (DSL) although it can be used to build DSLs
  - too esoteric for people to learn!

Functional vs. Procedural

	Functional	Procedural
Paradigm	Declarative (you describe/declare problem)	Imperative (you explicitly specify the steps to be taken to solve problem)
Functions	First-class objects, composable	“callbacks”, anonymous functions, etc.
Style	Recursive, mapping, pattern matching	Iterative, loops, if…
Examples	Lisp, Haskell, Scala…	Almost all other mainstream languages, including Java* *Java8 has many FP concepts


Scalastic Principles
Types
Implicits
Functions
Objects
scaladays
SCALA DIVISION
IMPLICIT PLEASURES


Scalastic Pragmatics
Usability
Simplicity
Power
SCALA DIVISION
Scala
#scaladays

Why Scala?

- Scala:
  - is fun (!) and very elegant;
  - is mathematically grounded;
  - is powerful (many built-in features);
  - has syntactic sugar: sometimes you can express something in different ways—however, usually, one form is “syntactic sugar” for the other form and the two expressions are entirely equivalent;
  - is predictable: once your program is compiling without warnings, it ***usually*** does what you expect;
  - is ideal for reactive programming (concurrency);
  - is ideal for parallel programming (map/reduce, etc.);
  - used by Twitter, LinkedIn, Foursquare, Netflix, Tumblr, Databricks (Spark), etc.

Is Scala really an important language today?

- Scala is presently in 34th position in the TIOBE index (http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html)
  - Down one place since September 2015
- In 17th place (more or less stable) on the PYPL index (http://pypl.github.io/PYPL.html)--up one places in five years.
- Ratios of Java : Scala
  - Google search
    - 10:1 in general (“xxx programming”)—was 20:1 five years ago.
  - “stackoverflow.com” in last two years
    - 15:1 (but Scala is growing relative to Java—it was 22 the first time I checked)

Why not Python?

- Python is a very popular language which is used by programmers in many disciplines, including Data Science.
- Python is interpreted, whereas Scala is compiled:
  - What difference does this make in practice? Compiled is better if you a running in a production environment; interpreter is better if you are in research/data science mode.
- Scala is strictly typed whereas Python is dynamically typed:
  - In practice, this means that you can probably write Python more quickly (less time spent trying to keep the compiler/interpreter happy) but strict typing helps ensure your code <u>does what you expect and won't crash</u>!

References: Quora; 6 points; Argument against Python

My thoughts on Python

- I can't take seriously a language that requires specific patterns of white space* in its source code!
- Python can't decide whether it's object-oriented or not.
  - For example, to get the length of a string ***s***, you write:
    - *len(s)*
  - But to find the index of the first occurrence of a substring ***t*** in string ***s***, you write:
    - *s.find(t)*
- I hate inconsistencies like that!

* it's true that Scala also pays attention to newlines but it doesn't ***rely*** on them.

How do we use Scala?

- Very briefly, there are at least *five** ways:
  - Create a module (file) called *xxx.scala* and compile it, build it, run it (using shell, *sbt*, or IDE such as IntelliJ);
  - The "REPL" (*read-execute-print-loop*) (using shell commands `scala` or `sbt console`)—or IDE—**the REPL is your friend;**
  - Compile it in-browser with, e.g. https://scalafiddle.io or https://scastie.scala-lang.org;
  - Create a "worksheet" called *xxx.sc* and save it (IDE only);
  - Use a *REPL-with-Notebook* such as Zeppelin.
- Two popular IDEs: *Scala-IDE* (based on *Eclipse)* and *IntelliJ-IDEA* (what I recommend).

* You can also dynamically inject code using Twitter's "eval" utility

Object-oriented programming

- The central concept of O-O is that methods (operations, functions, procedures, etc.) are invoked ***on objects*** (as opposed to local/global data structures in memory).
- A class is defined as the set of methods that can be invoked on its instances and the set of properties that these instances exhibit (aka fields).
- Objects are instances of a ***class*** and contain both methods and data (properties).
- Classes form an inheritance hierarchy such that sub-classes inherit methods and data from their super-classes (or may override).
- Examples: Smalltalk, Scala, Java, C++, C#, Objective-C, Python, PHP5, Ruby.

def vs. *val/var*

- The obvious way to think of the difference between ***def*** and ***val*** is to say: ***def*** declares a method (function) and ***val*** declares a variable.
- But there's a bit more to it:
  - ***def*** is (usually) parameterized and will have a different value for each combination of parameter(s), thus:
    - *def f(x: double) = ??? // something having to do with x*
  - but even with zero parameters, ***def*** is evaluated every time it is referenced, so
    - *def x = y is evaluated every time x is referenced, even though y may not have changed.*
  - *but a variable definition:*
    - *val x = y*
    - is evaluated once and once only
  - Later, we will learn about *lazy val* and *var*.

Functional Programming Cont'd

- Functional Programming:
  - Functions are first-class objects:
    - **Is a function *data* or *program*?**
    - Actually, it's both: it's all ***relative:***
      - If you are ***creating*** the function, think of it as ***program,*** but another function which receives it as a parameter sees it as an object (i.e. ***data.)***
      - functions, treated as data, can be ***composed*** and ***serialized.***

It's a bit like a photon: wave? particle? or both?

  - Avoids side-effects—variables, mutable collections, even I/O are the ***exception*** rather than the rule in the vast majority of Scala code:
    - functions **may not** change their input parameters; this, in turn:
    - leads to ***referential transparency*** (provable via substitution—Turing complete—"**λ**-calculus")
    - and to logic—i.e. predictability (and, thus, testability)
      - that's to say `f(x)==f(x)`
    - and so is inherently scalable and parallelizable

```
Source source=???
cal xs=source.getLines
```

```
: Iterator(String)
```

```
class FunctionalIterator[x](F: x=>Booleaan) extends Iterator[x]{}

     var x : x
     def hasNext : Boolean{
}
     def next : x=x
```

A brief diversion…

- What's so special about `f(x)==f(x)`?
  - Wouldn't we always expect that to be true?

```
Robins-MacBook-Pro:LaScala scalaprof$ date
Sat Jan  7 11:17:58 EST 2017
Robins-MacBook-Pro:LaScala scalaprof$ date
Sat Jan  7 11:18:01 EST 2017
```

  - Well, here we invoked the same function twice but it didn't give the same answer each time. WUWT? What's up with that?
  - In this case, the date function is not ***pure*.***
  - But, in general, functions defined in a functional programming language ***are*** pure.

* that's because it depends on something ***external;*** see https://stackoverflow.cor

A brief diversion contd.

- Take a look at this Java program:

```
   public class Purity {
class MyInteger {
      int i;
      public MyInteger(int x) { i= x; }
      public int getInt() { return i; }
      public void setInt(int x) { i= x; }
}
public int getNext(MyInteger x) {
      int x1 = x.getInt() + 1;
      x.setInt(x1);
      return x1;
}
public int getNext(Integer x) throws Exception {
      int result = x.intValue()+1;
      Field f = x.getClass().getDeclaredField("value");
      f.set(x, result);
      return result;
}
public int getNext(int x) {
      x = x + 1;
      return x;
}
public static void main(String[] args) {
      Purity n = new Purity();
      int i1 = 0;
      testPurity (n.getNext(i1)==n.getNext(i1), "getNext(int)");
      MyInteger i2 = n.new MyInteger(0);
      testPurity (n.getNext(i2)==n.getNext(i2), "getNext(MyInteger)");
      Integer i3 = 0;
      try { testPurity (n.getNext(i3)==n.getNext(i3), "getNext(Integer)); } catch (Exception e) { e.printStackTrace();
}
}
public static void testPurity(boolean b, String message) {
      if (b) System.out.println(message+" is idempotent");
      else System.out.println(message+" is not idempotent");
}
}
```

What's happening here? is *x* updated or what?

Or here?

What about here?

It's awkward outputting the message in the exceptional case because Java doesn't support call-by-name

A brief diversion—result

- Here, we annotate the program with comments on the result:

```
  public class Purity {
class MyInteger {
    int i;
    public MyInteger(int x) { i= x; }
    public int getInt() { return i; }
    public void setInt(int x) { i= x; }
}
public int getNext(MyInteger x) {
    int x1 = x.getInt() + 1;
    x.setInt(x1);
    return x1;
}
public int getNext(Integer x) throws Exception {
    int result = x.intValue()+1;
    Field f = x.getClass().getDeclaredField("value");
    f.set(x, result);
    return result;
}
public int getNext(int x) {
    x = x + 1;
    return x;
}
public static void main(String[] args) {
    Purity n = new Purity();
    int i1 = 0;
    testPurity (n.getNext(i1)==n.getNext(i1), "getNext(int)");
    MyInteger i2 = n.new MyInteger(0);
    testPurity (n.getNext(i2)==n.getNext(i2), "getNext(MyInteger)");
    Integer i3 = 0;
    try { testPurity (n.getNext(i3)==n.getNext(i3), "getNext(Integer)); } catch (Exception e) { e.printStackTrace();   }
}
public static void testPurity(boolean b, String message) {
    if (b) System.out.println(message+" is pure");
    else System.out.println(message+" is not pure");
}
}
```

int does not get updated

MyInteger gets updated

This (using *Integer*) throws an exception

A brief diversion Contd. (2)

- We got three different results with this Java program:
  - *getNext(int)* is pure;
  - *getNext(MyInteger)* is not pure;
  - *getNext(Integer)* results in: *java.lang.IllegalAccessException*: Class *edu.neu.coe.scala.Purity* can not access a member of class *java.lang.Integer* with modifiers "private final" at *sun.reflect.Reflection.ensureMemberAccess*(*Reflection.java*:101)
- In other words, you cannot guarantee the truth of the test `f(x)==f(x)` in Java (or C++, etc.)
- **But in functional programming languages, such as *Scala*, you can!**

A brief diversion from the brief diversion

- Why does all this matter?
- What's the future of computers? Can we continue with Moore's law?
  - No, we can't! Because the speed of light is too slow, amongst other physical limitations.
  - On my computer, in one clock cycle (0.38 ns), light can travel about 115 mm — that's only about three times the size of the processor!
  - If we speed the processor up a bit, electric field fluctuations won't have time to get from one end of the chip to the other!

Continuing the brief diversion from the brief diversion

- So, to make computers run faster is going to cost many $$. But there's another way:
  - Instead of speeding up ***one*** computer by 10x, why not have 10 computers? or 100? or 1000?
  - But we ***must have*** the following:
    - the ability to pass a function over the network to a different computer, treating the function itself as ***data*** (requires serialization);
    - the computing environment on the other computer must be identical (that's where the JVM comes in);
    - and, of course, we need the result of invoking that function multiple times (on any computer) to be identical each time, i.e. `f(x)==f(x)`.
- Now, I want you to just think about this and its implications.

Functional Programming Continued (2)

- Functional Programming:
  - “lazy” (or deferred) evaluation is a major, significant aspect of functional programming:
    - for example, a lazy ***List*** is called a ***LazyList*** [formerly, thru 2.12, ***Stream***]***:*** the tail of the stream is lazily evaluated which is to say only when needed: thus you can define “infinite” streams of things.
  - uses recursion when that is the appropriate mathematical definition rather than iteration
    - for example, if you want to count the number of elements in a list, you split the list into its head and its tail and return 1 + the count of the tail.

Typed functional programming

- Typing adds a new dimension to FP
  - ML, Miranda, Haskell, Scala, Clojure, etc. are all typed
  - Variables have types; types have kinds; kinds have…?
- Typing eliminates* "casting" errors at run-time

* well, Scala does allow a certain amount of circumventing strict typing—mainly for compatibility with Java—so it's still possible (but rare) to run into class cast errors.

Quick introduction to tuples

- Language types:
  - All languages have types such as Int, Double, String, etc.
  - And languages such as Java allow you to create your own types where you can have any number of fields, each potentially of a different type. However, that requires a lot of “boiler-plate” code.
  - But this sort of thing is so common that Scala (and Python) allow you to create these types on-the-fly with no special boilerplate code. You don’t even have to declare the type. It’s called a “tuple” and it corresponds precisely to the kind of information contained in a row of a database table (where it’s also called a tuple).
  - Examples:
    - (1, “One”, true)
    - 1 -> “One”

Combining O-O and FP

- In a pure functional language we ***only*** have functions:

  val g = f(x)

  - But, recall that ***x*** can be anything, including a tuple (a group of disparate things). So, if ***x*** happened to be a tuple of ***a***, ***b***, and ***c***. Then *val g = f((a, b, c))* which we can rewrite as *val g = f(a, b, c)*.
  - We call these parameters ***a***, ***b***, and ***c*** a "parameter set" although as you can see, they really are just one parameter, functionally. However, most languages allow these comma-separated parameters so Scala does too.
- We can have more than one parameter (in the functional sense), eg. *val z = f(x)(y)*
  - The interpretation of this is that *f(x)* yields a function ***g*** and therefore

    z = g(y).

Combining O-O and FP (contd.)

- So, a Scala function invocation can have any number of these “parameter sets.”

    val z = f(x)(a,b,c)(d)(e)…

    - Which we can re-write as:

        val z = g(a,b,c)(d)(e)…

    - where *g* is the function yielded by *f(x)*

Combining O-O and FP (contd.)

- Methods
  - If we write, in O-O notation, the following expression:
    val z = x.m(a,b,c)
- We know that this means invoke method ***m*** on object ***x*** and pass parameters ***a, b,*** and ***c***.
- But if we simply rewrite ***x.m*** as ***f(x)***
- Then we can interpret ***f(x)*** as the function that we obtain when we apply the method ***m*** to object ***x***.
- So that:
  val z = f(x)(a,b,c)
- In the simple case where the right-hand parameter set is just one parameter we can write it either as:
  x.m(a) or
  x m a *
- Now, you can see how methods and functions are, more-or-less, the same thing. More on this later.

* This is called "infix" notation

One more thing…

- Remember the slide with Martin Odersky? He mentioned ***implicits***. Is this a big deal? Is it as important as: FP+OO+Types+JVM?
- Not quite. But it's really important:
  - Configuration: ***implicits*** enable library classes to behave differently without having to pile on extra parameters all the time;
  - generalizes the concept of "widening";
  - is as powerful and sweeping as allowing objects as the (implicit) first parameter of a function. In fact, any implicits are always the *last* parameter of a function.