# Scala
# - Some essential concepts -

Marcel Lüthi

Graphics and Vision Research Group
Department of Mathematics and Computer Science
University of Basel

# Programming – an important activity

Data Visualization

Implementing Systems

Producing charts

Data preprocessing

…

Automation of experiments

Implementing Algorithms

Exploration of theory

# We need good tools


# A powerful tool

# Scala – a scalable language

- Can start with a one liner
- Can experiment quickly
- Has structures in place to manage complexity in large systems

## Examples of huge, stable Systems written in Scala


# What enables scalability

- Functional programming (functions compose)
- Object oriented programming (Objects as modules with well defined interfaces)
- Static typing that does not come in the way

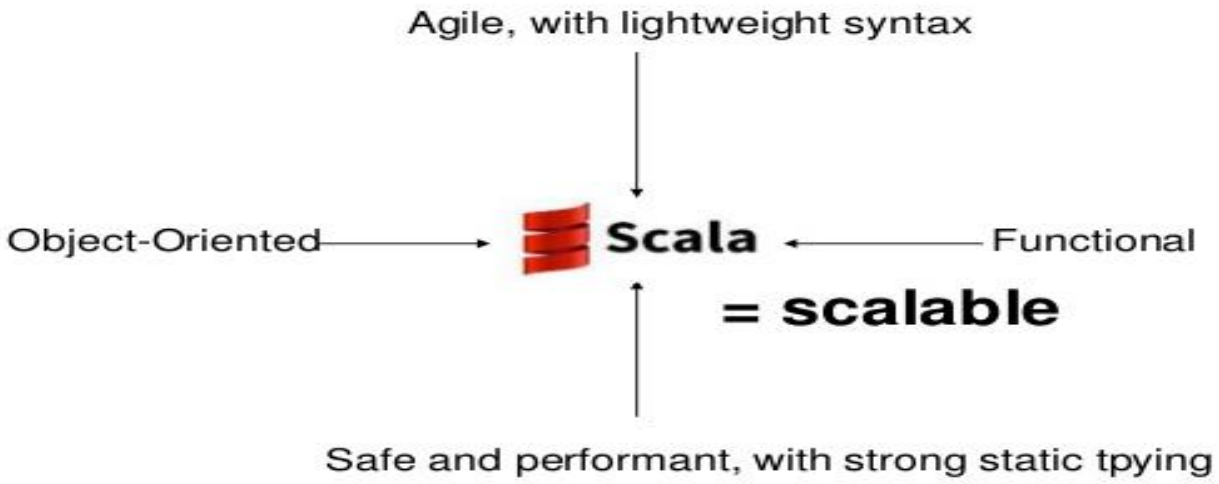



Figure: M. Odersky, Scala – the simple parts

# Aaaaaaagh : Scala is so complex

```
final override def map[B, That](f: A => B)(implicit
bf: CanBuildFrom[List[A], B, That]): That
```

# Goal: Discover the simplicity

Most of Scala is simple (not easy) – After this talk you should see its simplicity.

# Basic concepts and constructs

# Expressions

```
> (5 + 3) * 8

> List(1,2,3).toString

> GaussianProcess[_3D, Vector[_3D]](
>     DiagonalKernel(GaussianKernel[_3D](10), 3)
> )
```

- Expressions are program text
- Expression evaluate to values
- Expression have a type

# Values

```
> val result = (5 + 3) * 8
> val numberOneToThreeAsString = List(1,2,3).toString
> val gp = GaussianProcess[_3D, Vector[_3D]] (
    DiagonalKernel(GaussianKernel[_3D](1), 3)
)
```

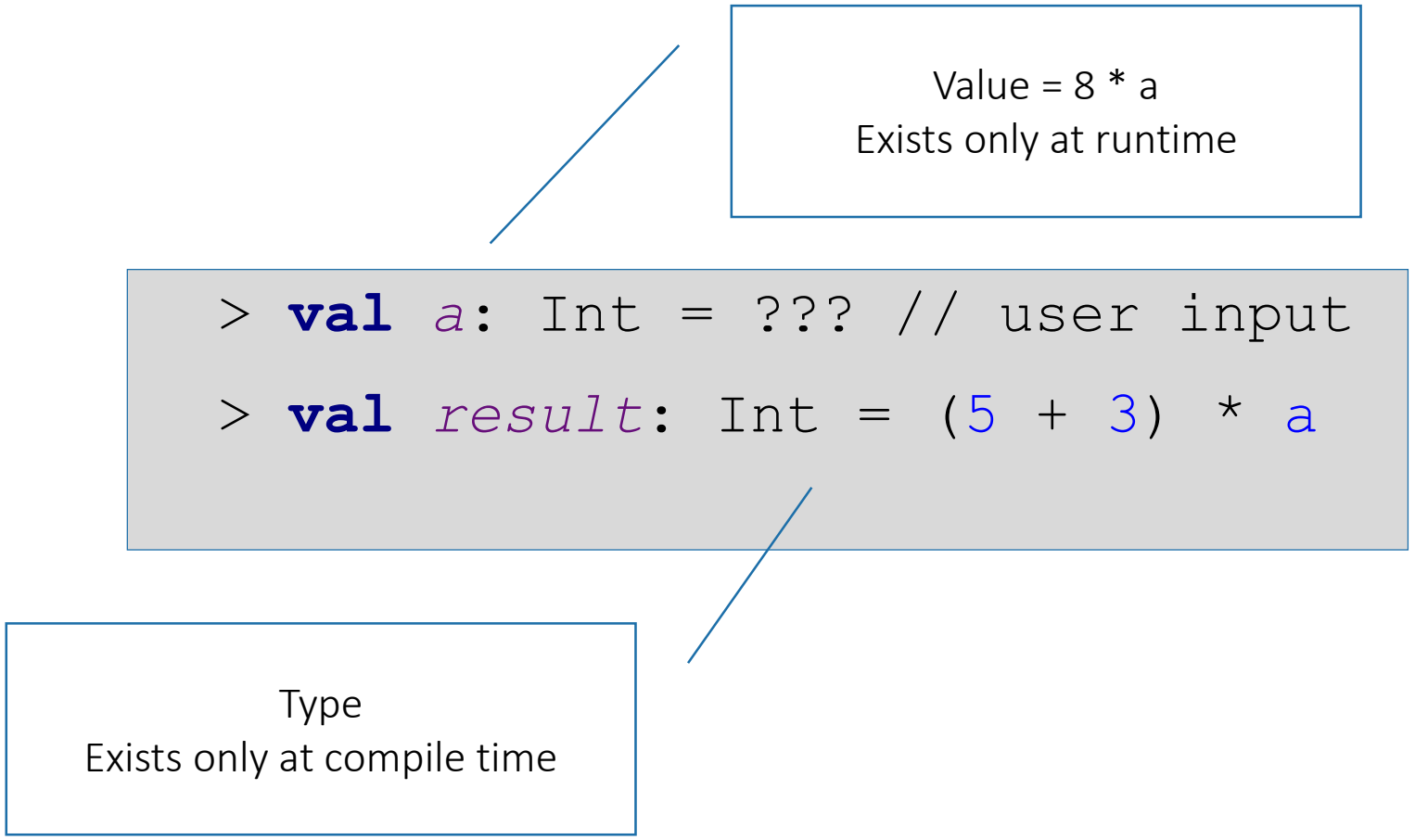
- Expressions can be named using **val**

# Types

```
val result: Int = (5 + 3) * 8
val numberOneToThreeAsString: String = List(1,2,3).toString
val gp: GaussianProcess[_3D, Vector[_3D]] =
  GaussianProcess[_3D, Vector[_3D]](
     DiagonalKernel(GaussianKernel[_3D](1), 3)
)
```

- Every expression has a type
- Type is often automatically inferred

# Expression types and values



# Blocks

```
> {
    val x = 1 + 1
    x + 1
  }
```

- Sequence of expression
- Last line is result of block => expression

# Blocks

```
> val res = {
  val x = 1 + 1
  x + 1
 }
```

- Sequence of expression
- Last line is result of block => expression

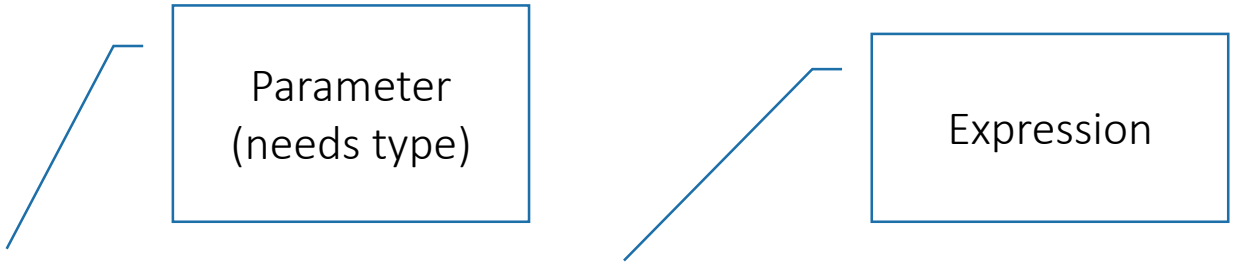
# Blocks

```
> println({ val x = 1 + 1
            x + 1 })
```

- Can be placed wherever an expression is required

# Functions

- Expressions that take parameters




```
(x : Int) => x + 3
```

# Functions

- Function body can be a block (which is an expression)

```
(x : Int) => {
    val y = 1
    x + y
  }
```

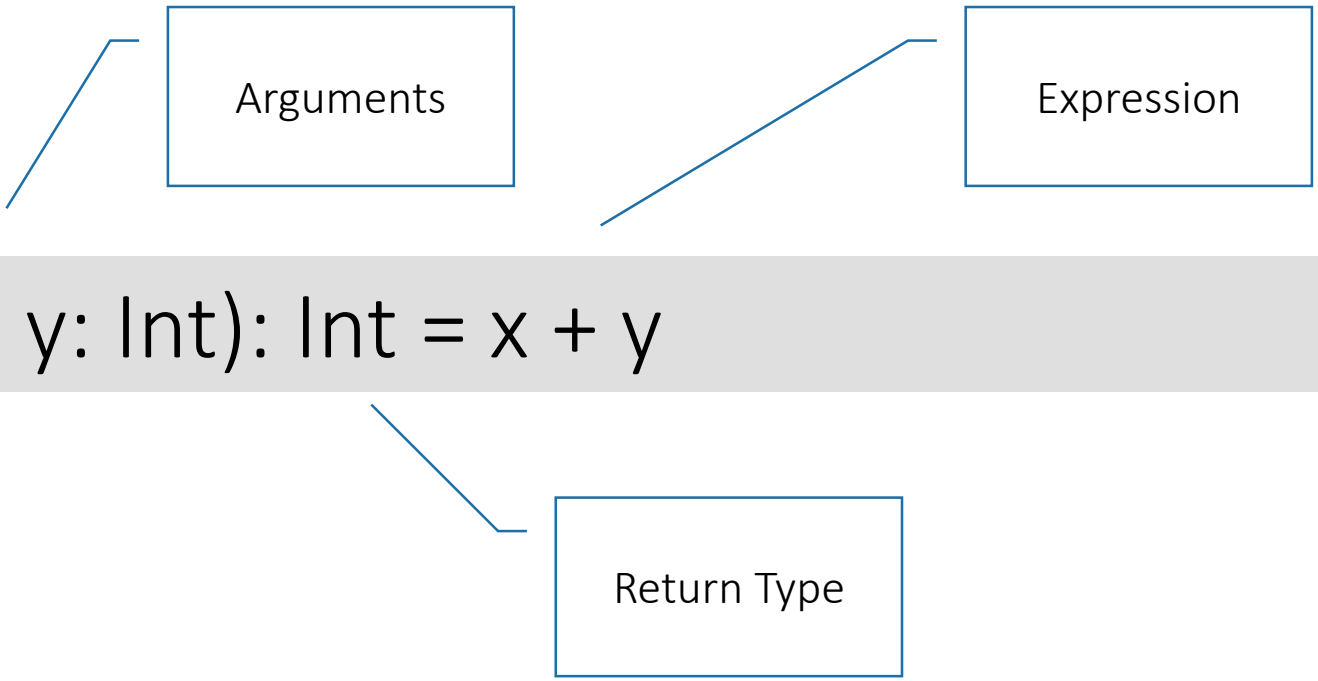
# Functions

- Functions are expression, hence values

```
val f =(x : Int) => {
      val y = 1
      x + y
    }
f(3) // 4
```

# Methods

- Similar to functions, but with special syntax



# Methods

- Can be turned to function by adding underscore

```
def add(x: Int, y: Int): Int = x + y

val adderFun = add _
adderFun(3, 4)
```

# Classes

```scala
class Greeter(prefix: String, suffix: String) {
    def greet(name: String): Unit =
    println(prefix + name + suffix)
}
```

```scala
val greeter = new Greeter("Hello, ", "!")
greeter.greet("Scala developer") // Hello, Scala developer!
```

# Case Classes

```
case class Point(x: Int, y: Int)
```

```
val p = Point(3,5)
```

- Super useful
- Ensures proper equality
- Ideal to structure data (records)
- Does not need new keyword

# Objects

- A singleton (only one instance exists)

```
object IdFactory {
  private var counter = 0
  def create(): Int = {
    counter += 1
    counter
  }
}
```

# Objects

- Often associated to a class – called the companion object

```
class PositiveNumber(num : Int) { // some methods}
object PositiveNumber {
  val MaxValue : Int = java.lang.Integer.MAX_VALUE
  val MinValue : Int = 0
}
```

# Traits

- Types containing certain fields
- Similar to interfaces – but can contain implementations

```
trait Greeter {
  def greet(name: String): Unit
}
```

```
class DefaultGreeter extends Greeter {
  override def greet(name: String): Unit
}
```

# Pattern matching

- Generalizes case statements from other programming languages

```
expression match {
  case pattern1 => expression1
  case pattern2 => expression2 // ...
}
```

# Pattern matching

```
def matchTest(x: Int): String = x match {
      case 1 => "one"
      case 2 => "two"
      case _ => "many"
}
```

```
def matchTest(x: Any): Any = x match {
      case 1 => "one"
      case "two" => 2
      case y: Int => "scala.Int"
}
```

# Scala - the simple parts

# (almost) Everything is an expression

- Everything can be composed with everything

```
println(if (a == 3) "abc" else "cde")

val c: Int = s match {
  case "abc" => try { userInput } catch { case _ => 0 }
  case _ => 2
}
```

# Everything is an object

- We always interact with any value by
  - Calling methods
  - Accessing fields

```
1 + 3
```

```
1 is an object
+ is a method
2 is an argument
1.+(2)
```

# Groups

- Everything can be grouped and nested
- Static scoping – uniform

```
def foo() : Unit = {
  case class KeyValue(key : String, value : Int)
  val list = List(KeyValue("A", 3), KeyValue("B", 2))
  def keyIsA(kv : KeyValue) : Boolean = { kv.key == "A" }
  list.count(keyIsA)
}
```

Allows naming of things. Leads to more clear code.

# Aggregate (Collections)

- Collections aggregate data
  - Transform, don't update
  - Uniform -> Learn once, use everywhere
- Essence of functional programming

```
val people =
    List("bob martin", "john doe", "william tell")
people.map(name => name.toUpperCase)
people.filter(name => name.startsWith("b"))
people.flatMap(name => name.split(" "))
```

# Not your father's for loops

- For loops are syntactic sugar
- The following 2 expressions are the same

```
numbers.map(number => number * 2)
```

```
for (number <- numbers) yield number * 2
```

# Not your father's for loops

- For loops are syntactic sugar
- The following 2 expressions are the same

```
numbers.filter(number => number % 2 == 0)
```

```
for (number <- numbers if (number % 2 ) == 0)
  yield number
```

# Not your father's for loops

- For loops are syntactic sugar
- The following 2 expressions are the same

```
numbers.flatMap(number =>(number until number + 2))
```

```
for (number <- numbers;
     numberSeq <- number until number + 2)
  yield numberSeq
```

# Not your father's for loops

- Makes writing complex things look easy

```
for (i <- 0 until 10;
     j <- 0 until 10;
     if (i + j) == 7)
  yield (i , j)
```

# Algebraic Data Types

- A is a B or C
- Are called sum types

```
trait A
case class B() extends A
case class C() extends A
```

# Algebraic Data Types

- Products are modelled with case classes
  - A and B
- Are called product types

```
case class B()
case class C()
case class A(b : B, c : C)
```

# Example: Algebraic Data Types

```
trait Expression
case class Plus(a : Int, b : Int) extends Expression
case class Minus(a : Int, b : Int) extends Expression
```

# Taking things apart

- Sum types are destructed by pattern matching

```
val expr : Expression = ???
val res : Int = expr match {
  case Plus(a, b) => a + b
  case Minus(a, b) => a - b
}
```

# Parametric types (not that simple)

- Sometimes a single type is not enough

```
trait IntList {
  def head : Int
  def tail : IntList
  def push(a : Int) : Unit
  def isEmpty : Boolean
}
```

# Parametric types (not that simple)

- Scala's type system is very rich
- Among other things, it supports parametric types

```
trait List[A] {
  def head : A
  def tail : List[A]
  def add(a : A) : Unit
  def isEmpty : Boolean
}
```

# Parametric types (not that simple)

- Also methods can have parametric types
- Called parametric polymorphsim


```
def reverseList[A](l : List[A]) : List[A] = {
  // implementation holds for all types A
  ???
}
```

# Type classes

# Type Classes

- What can we do when we need to know something about the type

```
def sumElementsOfStack[A](s : Stack[A]): A = {
    // cannot sum type A !
}
```

# Type Classes

- Solution: Define the right capability and pass an object

```
trait SupportsAdding[A] {
  def zero : A
  def plus(a : A, b : A) : A
}

def sumElementsOfStack[A](s : Stack[A])(adder : SupportsAdding[A]) : A =
{
  var sum : A = adder.zero
  while (s.isEmpty) {
    val element = s.pop
    sum = adder.plus(sum, element)
  }
  sum
}
```

# Type Classes

```
object StringAdder extends SupportsAdding[String] {
  override def zero: String = ""
  override def plus(a: String, b: String): Str
}

val s : Stack[String] = ???
sumElementsOfStack(s)(StringAdder)
```

Problem: Still ugly!

- We can then define class that supports adding for any type we want

# Implicits

- Solution: Implicit arguments
- The compiler checks if it finds an object somewhere in scope
  - The Scope in which the method is called
  - Companion Object of SupportsAdding
  - Companion Object of type A (String in this case)

```
def sumElementsOfStack[A](s : Stack[A])
      (implicit adder : SupportsAdding[A]) : A =
{ /// implementation goes here }

val s : Stack[String] = ???
sumElementsOfStack(s)
```

# Case study in Scalismo

## Building your own Gaussian Process