Principles of Programming Languages

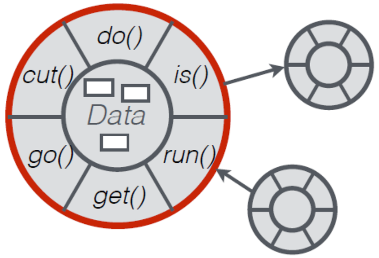
II.3 Object-oriented Programming Model

Dr. Herbert Prähofer
Institute for System Software
Johannes Kepler University Linz

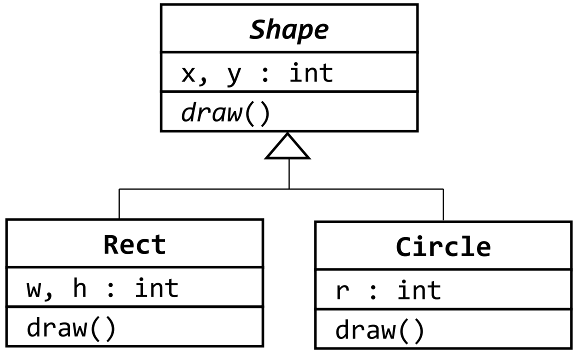
Object-oriented Programming Model

- Data encapsulation
  - Objects contain data
  - Methods for manipulating data

"objects = data + behavior"



- Inheritance hierarchies
  - Superclasses and subclasses
  - Inheritance of fields and methods



- Type hierarchies
  - Supertypes and subtypes

- Subtyping polymorphism
  - Variable of supertype can contain value of subtype

```
Shape s;
s = new Rect(x, y, w, h);
s = new Circle(x, y, r);
```

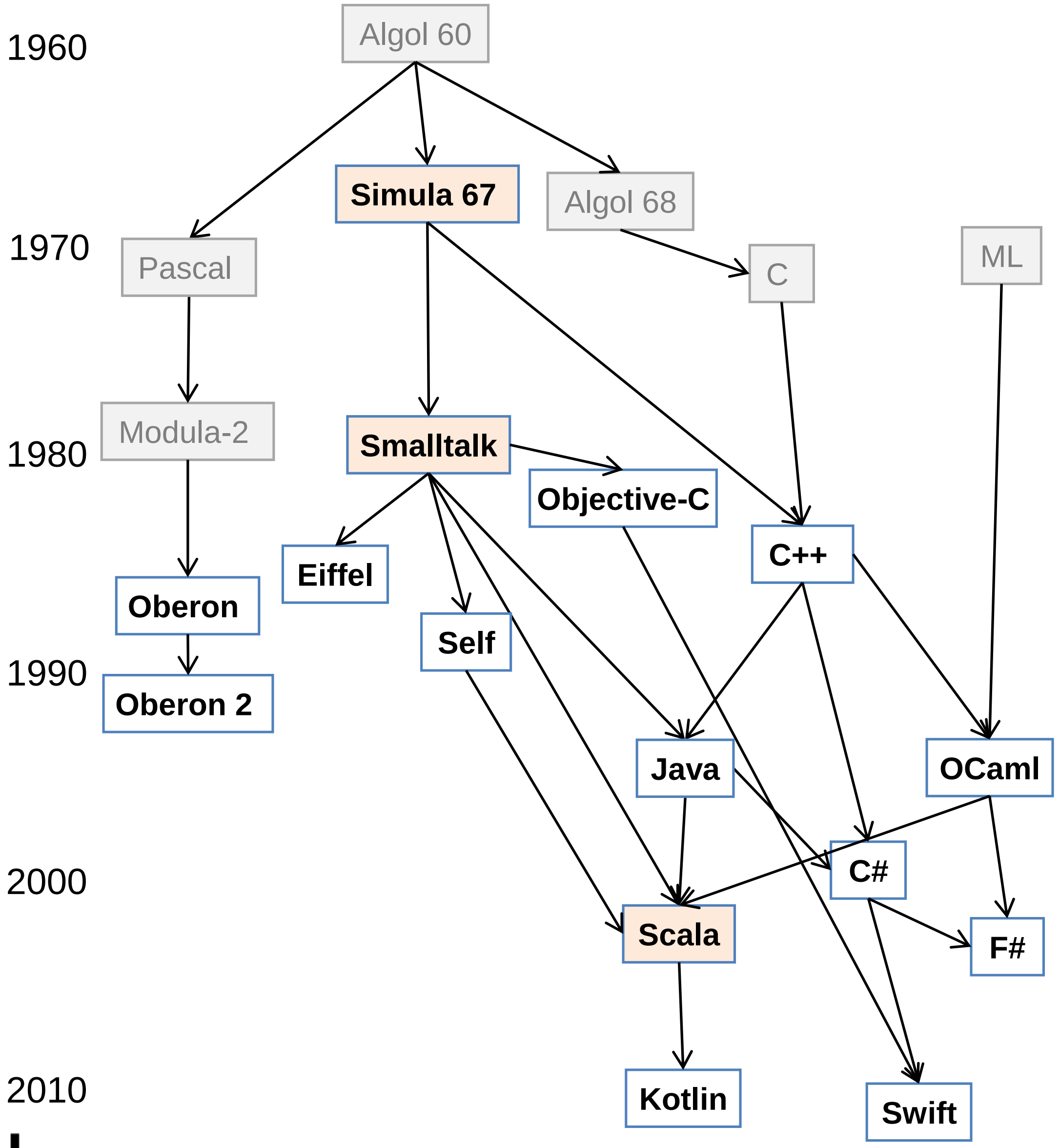
- Dynamic binding
  - Methods / messages are selected based on dynamic object type

```
s.draw()
```

II.3 Object-oriented Model

- History
- Smalltalk
- Introduction to Scala

History of Object-oriented Languages



Historical milestones

- ☐ 1967: Simula 67 released by Norvegian Ole-Johan Dahl and Kristen Nygaard as a simulation language
- ☐ 1980: Smalltalk as pure object-oriented language, developed at Xerox PARC by Allan Kay et al.; for programming GUI
- ☐ 1985: C++ as an object-oriented extension of the C programming language by Danish Bjarne Stroustrup
- ☐ 1986: Oberon system and language as a object-based language by Niklaus Wirth, as a successor of Pascal and Modula 2
- ☐ 1991: Oberon-2 by H. Mössenböck as a object-oriented extension of Oberon
- ☐ 1995: Java 1 released by Sun Microsystems
- ☐ 2000: C# released by Microsoft
- ☐ 2004: Scala released by Martin Odersky, EPFL Lausanne
- ☐ 2011: Kotlin released by Jetbrains
- ☐ 2014: Swift released by Apple Inc.

Simula 67

- by Kristen Nygaard and Ole-Johan Dahl from Oslo Computing Center
- 1962: Simula I, 1967 Simula 67
- Extension of Algol 60
- As a simulation language
  → objects representing simulated entities
- First object-oriented language


Kristen Nygaard


Ole-Johan Dahl

Features

- Classes and objects
- Inheritance
- Dynamic binding
- Coroutines (lightweight multitasking)
- Discrete event simulation

Simula Example

- Class **Glyph** with subclasses **Char** and **Line**

```
begin

  class Glyph;
    virtual: procedure print;
  begin
  end;

  Glyph class Char (c);
    character c;
  begin
    procedure print;
      OutChar(c);
  end;

  Glyph class Line (elements);
    ref (Glyph) array elements;
  begin

    procedure print;
    begin
      integer i;
      for i := 1 step 1 until UpperBound(elements, 1) do
        elements(i).print;
      end;
  end;
```

```
    ref (Glyph) line;
    ref (Glyph) array a(1:4);

    ! Main program;
    a(1) :- new Char('a');
    a(2) :- new Char('b');
    a(3) :- new Char('c');
    a(4) :- new Char('d');
    line :- new Line(a);
    line.print;
  end;
```

Base class

abstract method

Subclass

dynamically bounded methods

Array of Glyph polymorphic references

Dynamic binding

Reference assignment

II.3 Object-oriented Model

- History
- Smalltalk
- Introduction to Scala

Smalltalk

- Developed at Xerox PARC led by Alan Kay
- 1980 generally released as Smalltalk-80
- Influenced by Simula 67
- Language for the first graphical user interfaces

Alan Kay

Features

- Purely object-oriented
- Dynamically typed
- Message passing
- Code blocks as function objects
- Control structures as messages to code blocks

Smalltalk: Language Characteristics

Pure object-oriented language, everything is an object

- Values
- Classes
- Blocks = function objects

Dynamically typed

- Variables untyped
- Values have type

Execution

- Message passing
  - The only means of execution is **sending a message to an object**
  - The only **error** that can happen is **"message not understood"**
- Variable assignments
- Return values

Smalltalk: Example Shape

Base class Shape

- Subclass of **Object**
  - with instance variables **x** and **y**

```
Object subclass: #Shape
    instanceVariableNames: 'x y '
```

- Instance methods
  - for getting and setting x and y

```
x   ^x.

y   ^y.

x: aNumber
    ^x := aNumber.

y: aNumber
    ^y := aNumber.
```

read value of variable

: for messages with parameters

loction of variable

- for moving the shape

```
moveToX: newX moveToY: newY
    self x: newX. self y: newY. ^self.
```

message with multiple name parts and arguments

Subclass Rect

- subclass of **Shape**
  - with instance variables **w** and **h**

```
Shape subclass: #Rectangle
    instanceVariableNames: 'w h '
```

- message **new** calls new of meta class

```
new ^(super new) x: 0 y: 0 w: 10 h: 10
```

Allocation by constructor of meta class

- Instance methods
  - for getting and setting fields

```
w   ^w.
h   ^h.
w:  aNumber   ^w := aNumber.
h:  aNumber   ^h := aNumber.
```

- for drawing this Rect

```
draw
   Transcript show: 'Rect at:(' , self …
```

Standard out

this pointer

JKU

Smalltalk: Example Shape

- **Creating shapes** by sending **new** to class

variable declarations (untyped)

```
| shape x h |
shape := Rectangle new .
```

statement separator

- **Setting properties** by sending setter messages to object

```
shape x: 0 .
shape h: 20 .
```

- **Getting properties** by sending getter messages to object

```
x := shape x .
h := shape h .
```

- **Execute action** by sending message to object

```
shape draw .
shape moveToX: 100 moveToY: 100 .
```

Smalltalk: Block Objects

A block is an object which can be executed

- Block is statements in square brackets

```
[ statements ]
```

- A block can define a **parameter**

```
[ :p | statements ]
```

- message **value** executes the block

```
[ statements ] value
```

- message **value:** executes block with argument

```
[ :p | statemens] value: a
```

Examples:

```
[ shape draw ]
```

```
[ :s | s draw ]
```

```
rect := [ Rect new ] value
```

```
[ :s | s draw ] value: rect
```

➔ Blocks correspond to lambda-functions
➔ Messages value and value: mean function application

JKU

Smalltalk: Working with Blocks

Examples: Control structures as messages

- Message **ifTrue: ifFalse:** for Boolean values

Boolean value | message | block as argument for then | message | block as argument for else

```
(x = 0) ifTrue: ['Div by zero' print] ifFalse: [(1 / x) print]
```

- Implementation of **ifTrue: ifFalse:** in classes **True** and **False**
  - in class **True** execute block for then

```
Boolean subclass: #True
  ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
     ^trueAlternativeBlock value
```

  - in class **False** execute block for else

```
Boolean subclass: #False
  ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
     ^falseAlternativeBlock value
```

Class hierarchy for Boolean values

```
Object
  Boolean
    False
    True
```

II.3.a Introduction to Scala

Introduction to Scala

- Development and Installation
- Language Basics
- Characteristics
- Class definitions
- Case classes and pattern matching
- Functional data structures
- Miscellaneous

Scala - A SCALABLE LANGUAGE

www.scala-lang.org

- Development
  - by Martin Odersky and his team at the EPFL Lausanne
  - 1st release 2004
  - current version 3.2
    - Scala 3 represent major step compared to previous Scala 2 versions

Scala 2.0 (2006) – Scala 2.13 (2021)

- Combines object-oriented and functional features
  - functional influencer: Haskell
  - Object-oriented influencers: Smalltalk, Self, Java, OCaml, ...

Also allows imperative programming

- Platform integration
  - Scala for Java VM
    - translates to Java-Bytecode
    - can use Java APIs
  - Scala on Android
  - Scala on JavaScript VM

Scala Community

`www.scala-lang.org`

Key Players

- Programming Methods Laboratory, EPFL Lausanne
  - Research and development
  - Theory, compiler, libraries, applications
- Lightbend Inc. (https://www.lightbend.com/)
  - Industrial application of Scala technology
  - 2 key products
    - Reactive Platform: Akka Actors, Akka Reactive Streams
    - Play Framework: Web framework

Scala Software

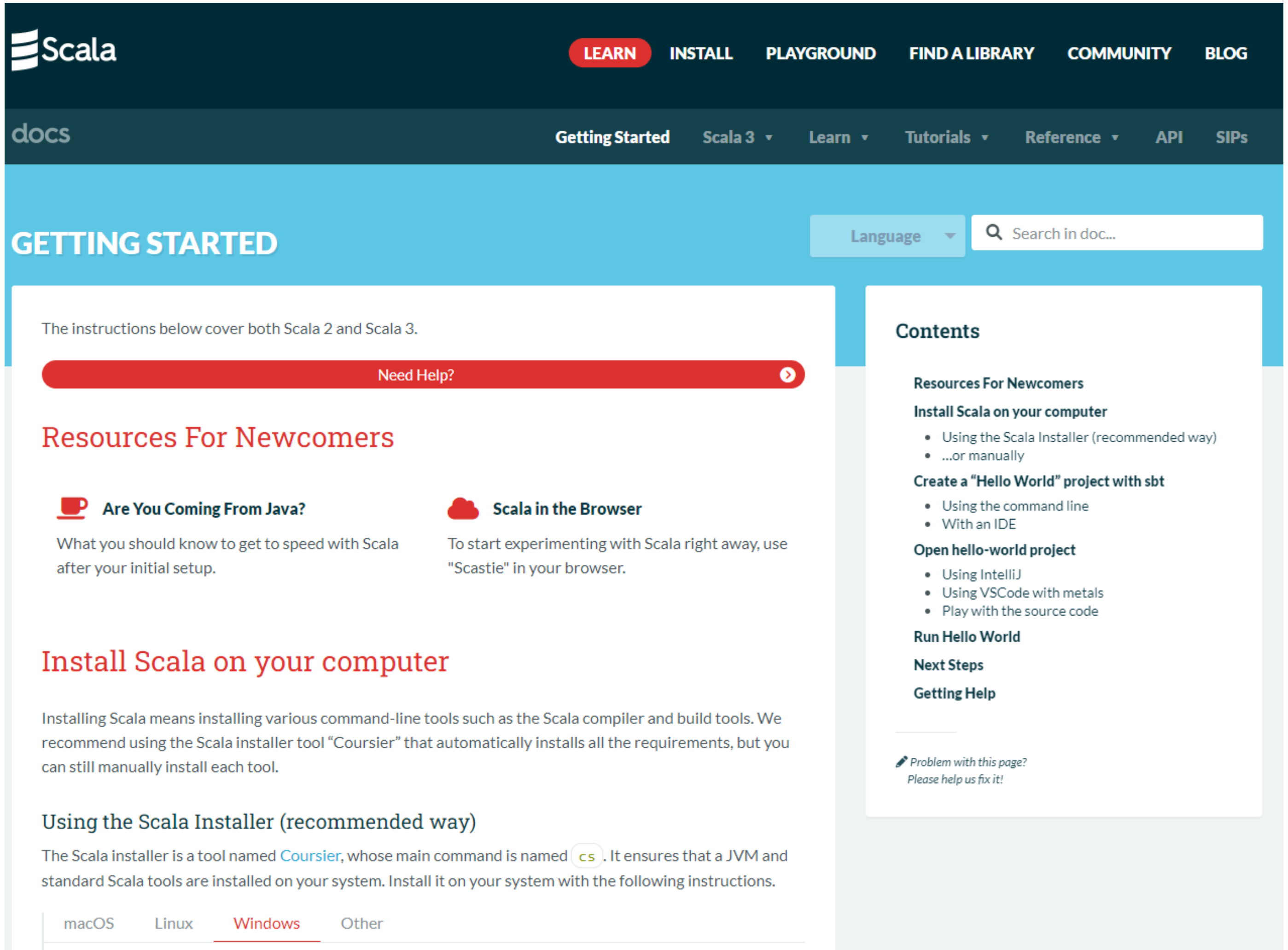
- Download from http://www.scala-lang.org/downloads

- Current stable release: 3.2 => major update to Scala 2 (partly not compatible)

- Packages
  - scala-devel — The Scala compiler
  - scala-library — The Scala library
  - scala-tool-support — Tool support files for various text editors like emacs, vim or gedit
  - scala-documentation — PDF documentation on the Scala programming language
  - scala-devel-docs — Contains the Scala API and code examples
  - scala-test — Test Suite we use to test the compiler and library
  - scala-msil — Tools required to develop Scala programs for .NET

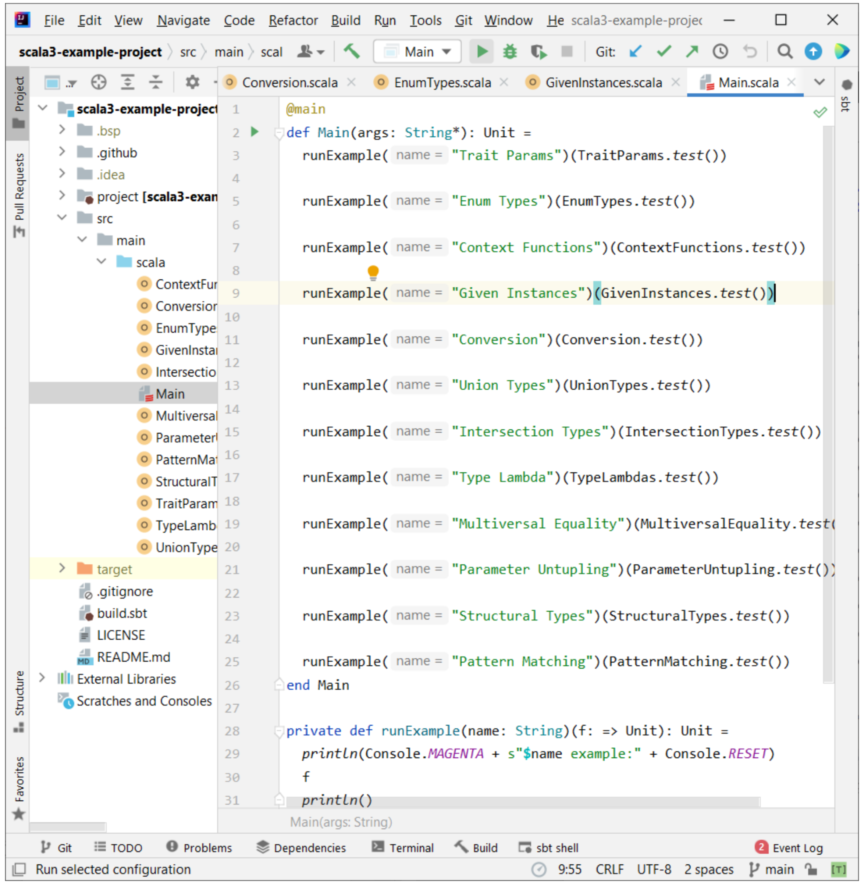
Scala Installation

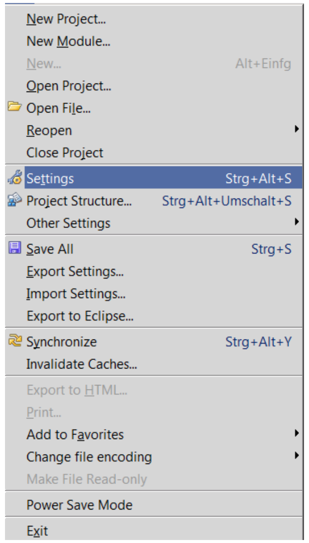
- Download from http://www.scala-lang.org/downloads

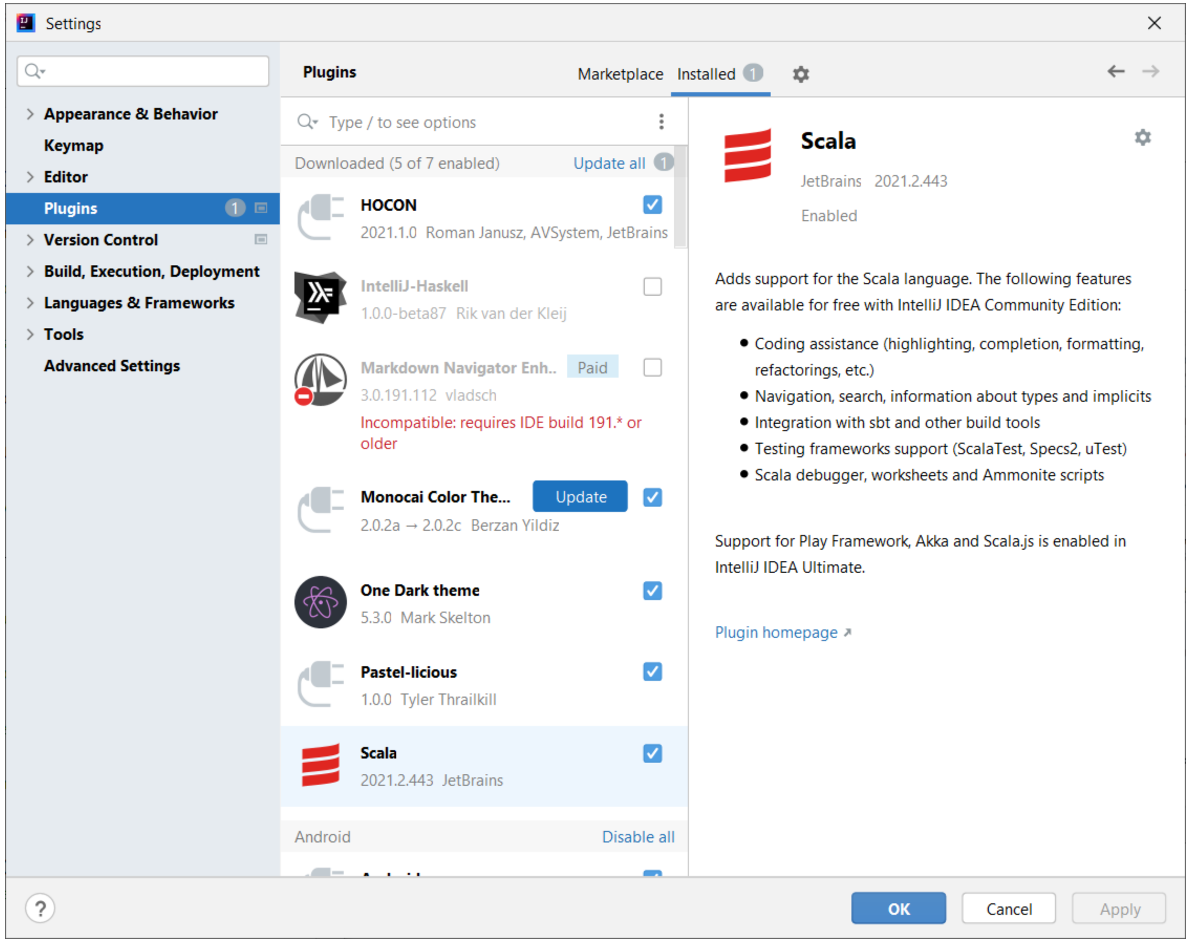


IntelliJ IDEA Plugin for Scala

- IntelliJ IDEA (http://www.jetbrains.com/idea/) + Scala Plugin

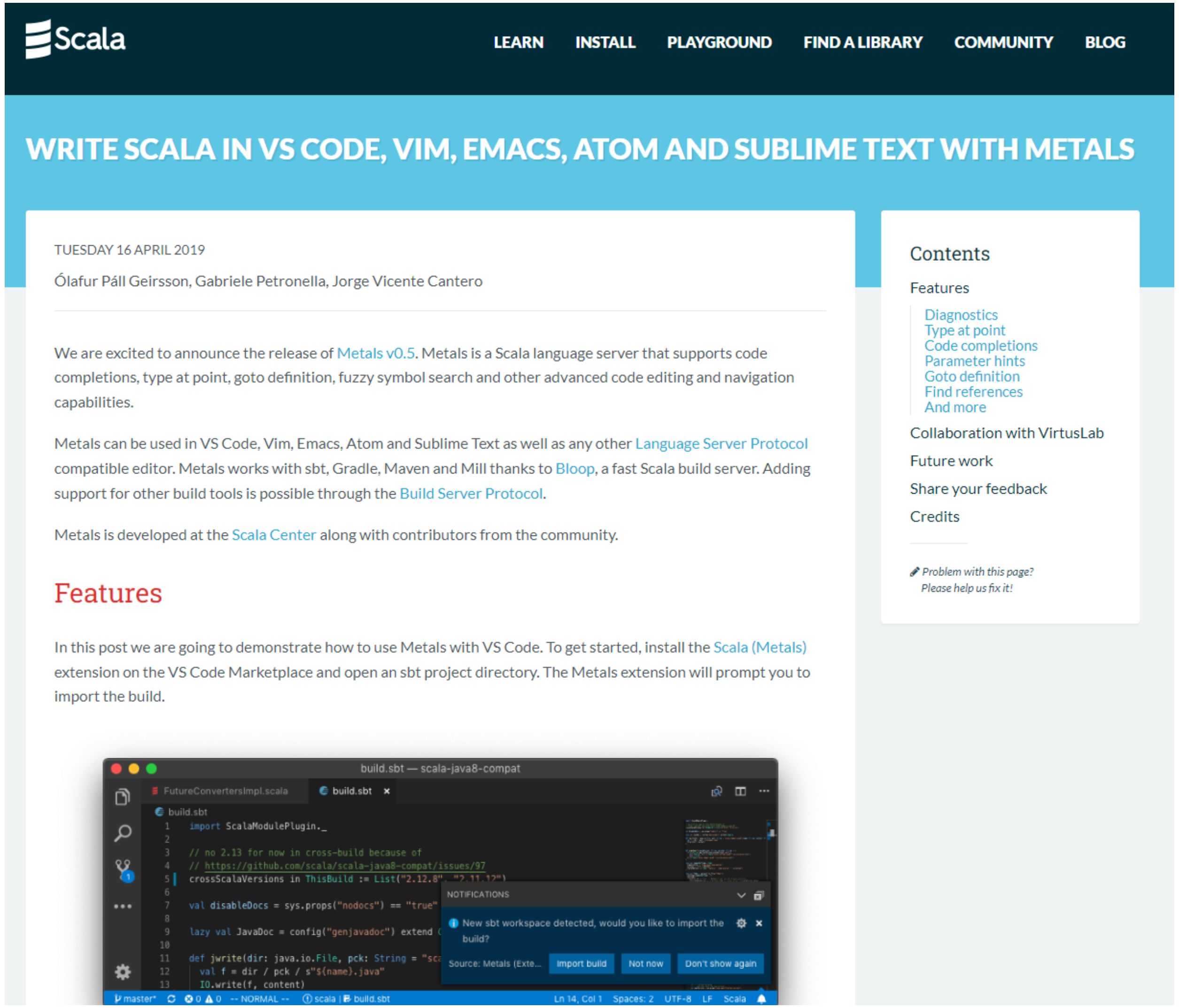






Visual Studio Code with Scala Plugin

- https://www.scala-lang.org/2019/04/16/metals.html



Scala

LEARN INSTALL PLAYGROUND FIND A LIBRARY COMMUNITY BLOG

WRITE SCALA IN VS CODE, VIM, EMACS, ATOM AND SUBLIME TEXT WITH METALS

TUESDAY 16 APRIL 2019

Ólafur Páll Geirsson, Gabriele Petronella, Jorge Vicente Cantero

We are excited to announce the release of Metals v0.5. Metals is a Scala language server that supports code completions, type at point, goto definition, fuzzy symbol search and other advanced code editing and navigation capabilities.

Metals can be used in VS Code, Vim, Emacs, Atom and Sublime Text as well as any other Language Server Protocol compatible editor. Metals works with sbt, Gradle, Maven and Mill thanks to Bloop, a fast Scala build server. Adding support for other build tools is possible through the Build Server Protocol.

Metals is developed at the Scala Center along with contributors from the community.

Features

In this post we are going to demonstrate how to use Metals with VS Code. To get started, install the Scala (Metals) extension on the VS Code Marketplace and open an sbt project directory. The Metals extension will prompt you to import the build.

Contents

Problem with this page? Please help us fix it!

Introduction to Scala

- Development and Installation
- Language Basics
- Characteristics
- Class definitions
- Case classes and pattern matching
- Functional data structures
- Miscellaneous

Equal to Java

- Standard data types
  - but type names in upper case

```
Byte, Short, Int, Long, Float, Double, Char, Boolean
```

- Literals (some important)

```
1, 1S, 1289349348L, 3.14, 3.14F, 12.12E-12, 'a', '\n', '\u0044', ...
```

- java.lang.String for Strings

```
val s : String = "abc"
```

- Blockstruktur und Lexical Scoping

```
{
  var a = ...
  if (a == y) then {
    val s = "is y"
    println(s)
  } else {
    val s = "not is y"
    println(s)
  }
}
```

Syntactical Differences to Java

- **Source files**
  - file ending **`.scala`**
  - can contain arbitrary definitions: classes, traits, objects, methods, values
  - package structure same as in Java

File `imp/imperative.scala`

```
package imp

abstract class Expr
abstract class Val(val x: AnyVal) extends Expr
case class IntVal(i: Int) extends Val(i)
case class BoolVal(b: Boolean) extends Val(b)
case class Var(name: String) extends Expr
case class BinExpr(op: String, left: Expr, right: Expr) extends Expr
...

def eval(expr: Expr, bds: Map[String, Val]) : Option[Val] = ...


val expr1 = BinExpr("+", Var("x"), IntVal(2))

object Main :
  def main(args : Array[String]) : Unit = ...
```

class definitions

method definition

value definition

object definition

Syntactical Differences to Java

- Semicolons are optional

```
val x = 1
val str = "ABC"
println(x)
```

- No brackets for methods without parameters

```
println
x.toString
string.toLowerCase
```

- Methods also in infix notations

```
if (list contains x) then ...
```

→

```
if (list.contains(x)) then ...
```

- Type declarations
  - with **:** in postfix notation

```
var y : Double = 1.0
y = 2.0
```

Syntactical Differences to Java

- Access modifiers
  - no modifier ➔ *public*
  - **`protected`**
  - **`private`**

public

private

```
class Person :
  ...
  private var name : String = ...
  override def toString : String = ...
```

public

Basic Language Elements

- Variables
  - immutable variables with **val**

```
val x = 1.0
```

final!

  - mutable variables with **var**

```
var y = 1.0
y = 2.0
```

  - with type declarations

```
var y : Double
y = 2.0
```

  - often optional and inferred

```
var y = 2.0
```

type Double inferred from value 2.0

- Methods with **def**
  - with type declarations for parameter (mandadory) and return type (optional)

parameter type

return type

=

block

```
def max(list : List[Int]) : Int   =  {
  var m = Integer.MIN_VALUE
  for (x <- list) {
    if (x > m) then m = x
  }
  m
}
```

```
def sign(x : Int) : Int   =
  if (x < 0) then -1
  else if (x == 0) then 0
  else +1
```

or single expression

Basic Language Elements

Classes, traits and objects

- class definitions with **`class`**

colon !

```
class Person :
  ...
```

indentation is significant

- with inheritance

```
class Student extends Person :
  ...
  override def toString : String = ...
```

override mandatory

- traits similar to interfaces with default methods

```
trait Writeable :
  def write(out: PrintStream) : Unit
  def writeln(out: PrintStream) : Unit = {
    write(out)
    out.println()
  }
```

- objects definitions are singletons

```
object HelloWorld extends App :
  println("Hallo World")
```

Scala 2 Compatibility

Allows braces for colon

braces !

```
class Person {
  ...
}
```

```
class Student extends Person {
  ...
  override def toString : String = ...
}
```

```
trait Writeable {
  def write(out: PrintStream) : Unit
  def writeln(out: PrintStream) : Unit = {
    write(out)
    out.println()
  }
}
```

```
object HelloWorld extends App {
  println("Hallo World")
}
```

JKU

Introduction to Scala

- Basics
- Characteristics
- Class definitions
- Case classes and pattern matching
- Functional data structures
- Miscellaneous

Scala Characteristics

from: Martin Odersky, Keynote ScalaDays, Berlin 2018

Types

- statically typed
- generics
- type inference

Objects

- purely object-oriented
- type hierarchy including all types
- dynamic binding of methods and fields

Functions

not purely functional!

- expressions
- first-class functions
- higher-order functions
- immutable data
- pattern matching

Object-oriented

Everything is an Object

➔ No difference between built-in value types and reference types

- Methods for built-in types

```
1.toString
-1.abs
1.2.toInt
```

- Operators as methods

```
1.+(2)
```

➔ `1 + 2`

only conceptionally, compiled as in Java

- Methods as operators

```
val oneToN = 1 to n
```

➔ `1.to(n)`

- Comparison always by ==

```
x == 0
str == "abc"
this == that
```

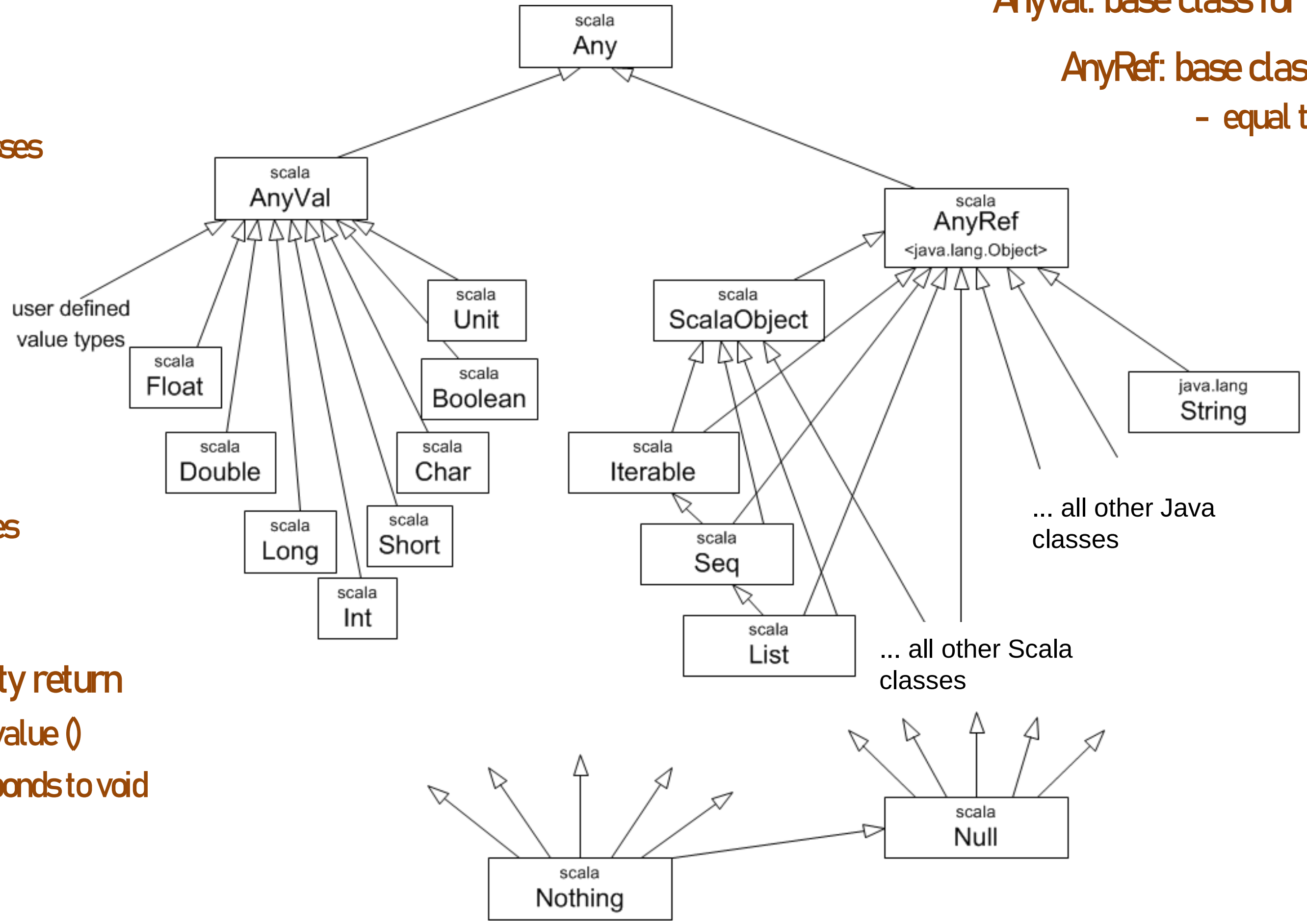
Object-oriented: Type Hierarchy

Any: base class for all types

AnyVal: base class for value types

AnyRef: base class for reference types
- equal to java.lang.Object

user-defined
value types: wrapper classes
with value semantics



Int, Double, ...
Java's built-in value types

Unit: for empty return
- single value ()
- corresponds to void

Null: subclass of all reference types
- single value `null`

Nothing: subclass of all types
- NO value!

Object-oriented: Class Any

- Abstract base class for all types

```
package scala

abstract class Any {
  final def == (that: Any): Boolean  =
    if (null eq this) then null eq that else this equals that

  final def != (that: Any): Boolean  =  !(this == that)

  def equals(that: Any): Boolean

  def hashCode: Int = …

  def toString: String = …

  def isInstanceOf[A]: Boolean

  def asInstanceOf[A]: A = this match {
    case x: A => x
    case _ => if (this eq null) then this else throw new ClassCastException()
  }
}
```

use == for all equality tests !

generic type parameter

equality

hashcode

toString

typtests and typcasts

Example: Typetests and Typcasts

```
if (x.isInstanceOf[Int]) then x.asInstanceOf[Int] + 1
```

Object-oriented: Classes AnyVal and AnyRef

- AnyVal: Base class for value types

```
class AnyVal extends Any
```

- AnyRef: Base class for reference types (= `java.lang.Object`)

```
class AnyRef extends Any {

  def equals(that: Any): Boolean      = this eq that
  def hashCode: Int = …

  def toString: String  = …


  final def eq(that: AnyRef): Boolean = …

  final def ne(that: AnyRef): Boolean = !(this eq that)

  def synchronized[T](body: => T): T
     // execute `body` while locking `this`.

}
```

equals and hashCode

toString

should be overridden

reference equality

synchronized as method (!?)

FUNCTIONAL

Working with expressions

- **`if`** is expression with returns value

```
val sign = if (x < 0) then -1 else if  (x > 0) then +1 else 0
```

- Blocks are expressions ➔ value of block is value of last expression

```
val max =
  {
    var m = Integer.MIN_VALUE
    for (x <- list) {
      if (x > m) m = x
    }
    m
  }
```

value returned

- All methods return a value (possibly `Unit` value **`()`**)

Unit ≅ void

```
def sign(x : Int) : Int =
  if  (x < 0) then -1
  else if (x > 0) then +1
       else 0
```

```
def write(list : List[Int]) : Unit = {
  for (x <- list) {
    System.out.print(x + "")
  }
}
```

Functional

Immutable data

- immutable variables and parameters

```
val xx = 1
```

```
def maxx(list : List[Int]) : Int  = { ... }
```

parameters always immutable

- immutable data structures
  - immutable pairs and tuples

```
val a1 = ('a', 1)
```

  - functional lists

```
val list21 : List[Int] = 2 :: 1 :: Nil
val list321 : List[Int] = 3 :: list21
```

creates new set with element 3 prepended

  - functional sets

```
val set12 : Set[Int] = Set(1, 2)
val set123 = set12 + 3
```

\+ creates new set with element 3 added

  - immutable maps
  - ...

Functional

Functions as first-class objects

- Lambdas

```
(x : Int) => x*2 + 1
```

- Generic function types

```
A => B
```

```
(A, B) => C
```

```
(A, B, C) => D
```

…

- Functions as parameters

```
def map[A, B](fn : A => B, xs : List[A]) : List[B] =
  for (x <- xs) yield fn(x)
```

- Functions as return values

```
def compose[A, B, C](f : A => B, g : B => C) : A => C =
  x => g(f(x))
```

Types

Static, strong typing

Type inference

```scala
val list = List(2, 1, 4, 2)
```

➔ l : List[Int]

Generics

- generic types

type parameters in square brackets (!)

```scala
class Buffer[A] {... }
```

- generic methods

type parameters after method name

```scala
def compose[A, B, C](f : A => B, g : B => C) : A => C =
  x => g(f(x))
```

Imperative

- Mutable variables

```
var i = 1
i = 2
```

- **`while`** loop

```
while (i < 10) {
  ...
  i = i + 1
}
```

returns Unit value ()

- **`for`** loop without return

```
for (j <- 1 to 10) {
    ...
}
```

returns Unit value ()

- Exceptions

```
try {
  val x = s.toInt
} catch {
  case ne : NumberFormatException => println("Not a number")
  case e : Exception => println("Exception")
}
```

returns Nothing

Scala Programming Style

Functional externally

- referential transparent functions

```
def fac(x: Int) : Int = {
  ...
}
```

- immutable data structures

```
class List[+T] extends Iterable[T] {
  def map[R](f : T => R) : List[R] = {
     ...
  }
}
```

Imperative internally

- with imperative internal implementations

```
def fac(x: Int) : Int = {
  var r = 1
  for (i <- 2 to x)  r = r * i
  r
}
```

more efficient compared to recursive solution

- with mutable internal implementation

```
class List[+T] extends Iterable[T] {
  def map[R](f : T => R) : List[R] = {
    val builder : Builder[T] =  new Builder[T]
    for (t <- this) builder.add((f(t))
      builder.build
  }
}
```

mutable builder

Rules

- **Use imperative** programming **internally** for efficiency reasons
- **Avoid non-local side effects** and **public access to mutable data** structures

Introduction to Scala

- Basics
- Characteristics
- Class definitions
- Case classes and pattern matching
- Functional data structures
- Miscellaneous

Class Definitions

- Class parameters
  - **parameters** of primary constructor
  - plus **private fields**
- Class body
  - **field and methods** declarations
  - plus **code of constructor**

Classes have no static members!

class parameters

colon (!)

```
class Car(model: String, year: Int, initial : Int) :

  private var miles: Int = initial

  def getModel = model
  def getYear = year

  //...
  println("Car " + model + " year " + year + " created ")
```

field and method declarations

constructor code

- Instantiation with arguments for class parameters

```
val bmw = new Car("BMW", 2019, 0)
```

- alternatively without new

```
val bmw = Car("BMW", 2019, 0)
```

new omitted !

Class Definitions

Compatibility with Scala 2 Syntax

- **braces** instead of **colon**

opening brace

```
class Car(model: String, year: Int, initial : Int)  {

  private var miles: Int = initial

  def getModel = model
  def getYear = year

  //...
  println("Car " + model + " year " + year + " created ")
}
```

closing brace

Remark:
I will often use Scala 2 variant of class definitions

Class Definitions

Overloaded Constructors

- Definition with name this
- must call primary constructor

```
class Car(model: String, year: Int, initial : Int)  :

  private var miles: Int = initial

  def this(model: String, year: Int) = {
     this(model, year, 0)
  }

  def this(model: String) = {
     this(model, 2021, 0)
  }

  ...
```

Class Definitions

Inheritance

- **extends** with **call to constructor** of superclass
- **override** mandatory for **overriding concrete** members
- **abstract** classes and members supported

```
abstract class Vehicle(model: String, initial: Int) :
  protected var miles = initial
  def getMiles = miles
  override def toString : String = model + " with miles " + miles
  def drive(distance : Int) = miles += distance
```

call superclass constructor

override!

```
class Car(model: String, year: Int, initial : Int) extends Vehicle(model, initial) :
  private val FULL = 20.0
  private val MILAGE = 50.0
  private var fuelLevel: Double = FULL;

  override def toString : String = super.toString + " fuel " + fuelLevel
  override def drive(distance: Int) = {
    super.drive(distance)
    fuelLevel = fuelLevel - distance / MILAGE
  }
  def refill() = { fuelLevel = FULL }
```

Singleton Objects

Definition of singleton objects

- with keyword **`object`**
- with **`extends`** from superclass

```
object MyCar extends Car("Qasqai", 2011, 113409)
```

- possibly with class body

```
object MyCar extends Car("Qasqai", 2011, 141200) :
  val owner : String = "Me"
  override def toString : String = "This is my car with " + getMiles + " miles"
```

- Accessing singleton by object name

```
MyCar.drive(125)
println(MyCar.toString)
```

Specific constraints and properties of objects

- **cannot** have **class parameters**
- **cannot be extended**
- **same name as class** allowed (= ***companion object*** for the class)

Main is Object

- Object with **main** method

```
object MyApp :
  def main(args : Array[String]) : Unit = {
    println("Hallo World")
  }
```

- Object extending App

```
object MyApp2 extends App :
  println("Hallo World")
```

Class Members

- Class members can be
  - **val** – immutable variable
  - **var** – mutable variable
  - **def** – method
- All members are **dynamically bound**
  - also **val** and **var** variables

in distinction to Java

- All members **can be abstract**

```
abstract class AbstractClass :
  def abstractMethod : ReturnType
  var abstractVar : VarType
  val abstractVal : ValType
```

abstract because no definition

- All members **can be overridden**

```
abstract class Shape :
  val pos : Point
  def draw : Unit
  ...
```

```
class Group(elems : Shape*) extends Shape :
  override val pos = new Point(minX(elems), minY(elems))
  override def draw = { /*...*/ }
```

override

Traits

- abstract types similar to interfaces in Java with default implementations

```
trait Writeable {
  def write(out: PrintStream) : Unit
  def writeln(out: PrintStream) : Unit = {
    write(out)
    out.println()
  }
}
```

- inheriting from traits

use **with** for multiple supertypes

```
class Group(elems : Shape*) extends Shape with Writeable :
  ...
  override def write : Unit = {
    System.out.println("Group: " + elems)
  }
```

➔ **more on traits later**

Introduction to Scala

- Basics
- Characteristics
- Class definitions
- Case classes and pattern matching
- Functional data structures
- Miscellaneous

Case Classes

Case classes are special classes

- **class parameters** are **public final** fields
- **equal** and **hashCode** defined based on class parameters
- **toString** based on class parameters

```
abstract class Expr
case class Var(name: String) extends Expr
case class Lit(value: Double) extends Expr
case class BinOp(operator: String, left: Expr, right: Expr) extends Expr
```

- instantiation

```
val x = Var("x")
```

```
val expr = BinOp("*", BinOp("+", Var("x"), Var("y")), Lit(2))
```

$(x + y) * 2$

- access to class parameters

```
println ( x.name )
```

```
val left  = expr.left
val right = expr.right
```

- allow **pattern matching**

Pattern Matching

Pattern matching analogous to Haskell

- Syntax
  - keyword **`match`**
  - keyword **`case`** with patterns

data element

cases

expressions

```
x match {
  case 1   => 1
  case n   => n * fact(n - 1)
}
```

patterns

Haskell:

```
case x of
      1 ->   1
      n ->   n * fact (n - 1)
```

Pattern Matching

Pattern matching on case classes

- Patterns built by **class name** plus **class parameters**

```
abstract class Expr
case class Var(name: String) extends Expr
case class Lit(value: Double) extends Expr
case class BinOp(operator: String, left: Expr, right: Expr) extends Expr
```

```
expr match {
  case Var(n)                   => println("Variable: " ++ n)
  case Lit(1.0)                 => println("Value one")
  case Lit(x)                   => println("Value = " + x)
  case BinOp("*", l, r)         => println("Addition")
  case BinOp(op, Lit(0.0), r)   => println("A binary operation with zero")
  case _                        => println("Something unknown")
}
```

Patterns can be arbitrarily nested !

Pattern Matching

Example: Symbolic differentiation

```scala
abstract class Expr
case class Var(name: String) extends Expr
case class Lit(value: Double) extends Expr
case class Plus(left: Expr, right: Expr) extends Expr
case class Times(left: Expr, right: Expr) extends Expr
```

```scala
object Expr :
  def deriv(expr : Expr, dx : Var) : Expr =
    expr match {
      case Var(n) if dx.name == n => Lit(1.0)

      case Var(_)                 => Lit(0.0)

      case Lit(_)                 => Lit(0.0)

      case Plus(u, v)             => Plus(deriv(u, dx), deriv(v, dx))

      case Times(u, v)            => Plus(
                                       Times(u, deriv(v, dx)),
                                       Times(v , deriv(u, dx))
                                     )
    }
```

$$\frac{dx}{dx} = 1$$

$$\frac{dc}{dx} = 0$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$

$$\frac{d(u*v)}{dx} = u*\frac{dv}{dx} + v*\frac{du}{dx}$$

Patterns Summary

- Values

```
case 1
case "Hans"
```

- Variables

```
case x
```

- Default

```
case _
```

- Typetests

```
case p : Person
```

- Guards

```
case (x, y) if x == y
```

- Case classes

```
case Var(n)
case Some(x)
case None
```

- Lists

```
case List(1, 2, 3, xs @ _*)
```

- Additional variable bindings with @

```
case add0@BinOp("+", zero@Lit(0), r)
```

- ... *some more patterns later* ...

Introduction to Scala

- Basics
- Characteristics
- Class definitions
- Case classes and pattern matching
- Functional data structures
- Miscellaneous

Tuples

Generic tuple data types with multiple elements

- generic types

```
(A, B)
```
```
(A, B, C)
```
```
(A, B, C, D)
```
…

- values

```
val personInfo = ("Franz", "Kafka", 1883, "male")
```
```
: (String, String, Int, String)
```

- access operations: _1, _2, ...

```
val first = personInfo._1
val born = personInfo._3
```

- pattern matching
  - in assignments

```
val (first2, last, born2, sex) = personInfo
```

  - in match-expressions

```
personInfo match {
  case ("Franz", "Kafka", year, _) => println("Kafka is born " + year)
  case (first, last, year, _)      => println(last ++ " is born " + year)
}
```

List

Generic list data type List[T]

- with two variants
  - ☐ empty list
    ```
    Nil
    ```
  - ☐ cons operator
    ```
    first :: rest
    ```

<u>Haskell:</u> analogous to Haskell list data type

```
[]
a : [a]
```

- construction with `::`

```
val list123 : List[Int] = 1 :: 2 :: 3 :: Nil
```

- construction with List constructors

```
val list123 = List(1, 2, 3)
```

```
val empty : List[Int] = List()
```

List() = Nil

- access operations: head, tail ...

```
val first = list123.head
val rest = list123.tail
```

Pattern Matching with Lists

Pattern matching with lists

- in assignments
  - with `::` patterns

```
val (first :: rest2) = list123
```

matches lists with at least 1 element

  - with **List** patterns

```
val List(first, second, third) = list123
```

matches lists with exactly 3 elements

```
val List(first, second, _*) = list123
```

matches lists with at least 2 elements

- in match expressions

binding rest to xs

```
list123 match {
  case List(1, 2, xs @ _*)  => println("first elements are 1, 2, rest is" + xs)
  case (1 :: xs)            => println("first element is 1")
  case List()               => println("empty list")
  case _                    => println("something else")
}
```

Pattern Matching with Lists

Example: equalLists

```scala
def equalLists[A](xs : List[A], ys : List[A]) : Boolean =
  (xs, ys) match {
    case (List(), List())                => true
    case (_     , List())                => false
    case (List(), _)                     => false
    case (x::xs , y::ys)  if x == y     => equalLists(xs, ys)
    case (x::xs , y::ys)                 => false
  }
```

Haskell:

```haskell
equalLists :: Eq a => [a] -> [a] -> Bool
equalLists []     []                = True
equalLists _      []                = False
equalLists []     _                 = False
equalLists (x:xs) (y:ys) | x == y = equalLists xs ys
equalLists (x:xs) (y:ys)           = False
```

OPTION

Analoguous to Haskell's **Maybe** type!
Analoguous to Java's **Optional** type!

Option[A] for expressing possibly empty values

- Defined as case classes with two variants **Some** and **None**

Details of class definition later!

```
sealed abstract class Option[+A]
case final class Some[+A](x : A) extends Option[A]
case object None extends Option[Nothing]
```

Some has value **x**
None has no value

Haskell:

```
data Maybe a = Just a
             | Nothing
```

- **Option** as return value
  - Example: find for lists

```
val optPrime : Option[Int] = list123.find(x => isPrime(x))
```

- Pattern matching with **Option**

```
optPrime match {
  case Some(p)   => println("The prime found is " + p)
  case None      => println("No prime found")
}
```

Maps

Map[K, V] is immutable hashmap

- Construction

immutable!

mutable!

```
var mappings : Map[String, Int] = Map(("x", 1), ("y", 2))
```

- Updating

```
mappings = mappings.updated("x", 7)
```

```
mappings = mappings - "y"
```

```
mappings = mappings ++ List(("z", 3), ("u", 0))
```

- Access

```
val yVal : Int = mappings("y")
```

Exception if not contained!

```
val optYVal : Option[Int] = mappings.get("y")
```

None if not contained!

- Iteration

```
for (k <- mappings.keys) println(k + " = " + mappings(k))
```

```
for ((k, v) <- mappings) println(k + " = " + v)
```

Introduction to Scala

- Basics
- Characteristics
- Class definitions
- Case classes and pattern matching
- Functional data structures
- Miscellaneous

Positional and Named Arguments

```
def speed(distance: Float, time: Float) : Float = {
  distance / time
}
```

Method calls

- Position of arguments

```
speed(1200, 10)
```

- With name of parameters

```
speed(distance = 1200, time = 10)
```

```
speed(time = 10, distance = 1200)
```

- Mixed : Zuerst aufgrund Positionen, dann benannt

```
speed(1200, time = 10)
```

Default Values for Parameters

- Definition of default values in method declarations

```
def printTime(out: java.io.PrintStream = Console.out, divisor : Int = 1) = {
  out.println("time = "+ System.currentTimeMillis() / divisor)
}
```

  - with new values

```
printTime(System.err, 1000)
```

  - with default values

```
printTime()
```

  - mixed

```
printTime(System.err)
printTime(divisor = 1000, out = System.err)
printTime(divisor = 1000)
printTime(System.err, divisor = 1000)
```

Vararg Parameter

- Varargs: last parameter can be repeated
  - type declaration with **<*Type*>***
  - within method represented as array **Array[<*Type*>]**

a number of string values!

```
def printLines(lines : String*) = {
  for (line <- lines) {
    println(line)
  }
}
```

a number of string values!

```
printLines( "This is the first line",
  "and this the second",
  "…")
```

String Interpolator

Insertion of computed values in string

- string with **s** prefix: **`s"..."`**
- ***`$expression`*** for insertions
- with **`${`*`expression`*`}`** for complex expressions
- inserts **toString** of value of expression

```
s"$name = ${eval(expr, bdgs)}"
```